Les microcontrôleurs MSP430 pour les applications faibles consommations – asservissement d'un oscillateur sur le GPS.

J.-M. Friedt, A. Masse, F. Bassignot, 13 septembre 2007

Association Projet Aurore
36A, avenue de l'Observatoire, 25030 Besançon Cedex
Email : friedtj@free.fr, web : http://jmfriedt.free.fr

Nous présentons les outils de développement sous GNU/Linux pour le microcontrôleur Texas Instruments MSP430. Ce microcontrôleur a été développé spécialement pour les applications embarquées consommant un minimum d'énergie. Il présente néanmoins une puissance de calcul et une architecture intéressante avec 16 registres de 16 bits et de nombreux périphériques. Nous présenterons dans ce document les méthodes de développement sur ce processeur en assembleur et en C grâce à la cross-compilation de code pour MSP430 par le compilateur `gcc`, en insistant sur l'interaction de ces deux langages afin de tirer le meilleur parti de chacun selon les situations. Les applications présentées s'orientent autour de la métrologie du temps, en implémentant d'abord une horloge, puis en asservissant un oscillateur sur une référence précise fournie par le GPS, pour finalement ajouter la prévision des horaires de lever et coucher du soleil. Ce dernier calcul illustre l'utilisation des librairies mathématiques émulant le calcul flottant sur un processeur capable de ne travailler que sur des entiers (absence d'unité matérielle de calcul flottant). Ces applications nous fournissent le prétexte de mettre en pratique des sujets aussi divers que quelques principes de base de l'asservissement d'un système par contrôleur PI ou l'estimation de l'énergie consommée par une opération de calcul.

Ce document est rédigé en deux parties distinctes : une première partie rappelle les notions de base du développement sur microcontrôleurs sous GNU/Linux et présente en particulier les outils pour le Texas Instruments MSP430. La seconde partie concrétise les acquis de la première partie dans une application concrète, à savoir la synchronisation de l'oscillateur à quartz du microcontrôleur sur une référence de temps excessivement précise issue d'une récepteur GPS.

Les tendances actuelles dans les développements informatiques divergent dans deux directions : les processeurs puissants et gourmands en énergie d'une part, et d'autre part les applications autonomes fonctionnant sur batteries et fournissant une puissance de calcul modeste pour une consommation électrique faible. Les applications embarquées qui vont nous intéresser ici se placent dans la seconde catégorie. Nous désirons nous familiariser avec un processeur ne consommant en moyenne qu'une fraction de microwatt pour une autonomie de plusieurs mois lors d'un fonctionnement sur piles : nous avons pour cela sélectionné un microcontrôleur dédié à cette tâche produit par Texas Instruments, le MSP430.

Nous proposons de présenter les outils nécessaires au développement sur cette plateforme, d'un point de vue matériel (cartes de démonstration et outils de programmation) et logiciel (outils de développement libres fonctionnant sous GNU/Linux).

1 Le microcontrôleur MSP430F149

La classe des microcontrôleurs MSP430 produite par Texas Instruments [1] est étonnante par ses performances et la consommation associée : pour le MSP430F149, nous disposons ainsi de 60 KB de mémoire non volatile flash, 2 KB de RAM, des converstisseurs analogique-numérique avec une résolution de 12 bits, une programmation par interface JTAG émulée sur le port parallèle du PC et ne nécessitant donc pas d'outils de programmation dédiés. La consommation maximale annoncée

[1] http://focus.ti.com/mcu/docs/mcuprodoverview.tsp?sectionId=95&tabId=140&familyId=342

est inférieure au milliampère, pour n'atteindre que quelques microampères dans le mode de veille le plus profond. L'architecture interne du microcontrôleur contient notamment 12 registes généraux de 16 bits nommés R4 à R15 qui limitent le besoin d'accéder à la RAM, ainsi que deux timers sur 16 bits dont nous ferons usage dans les applications que nous présenterons dans ce document. Bien que Texas Instruments distribue gratuitement les composants comme échantillon, nous proposons de nous focaliser sur les aspects logiciels de l'utilisation de cette classe de microcontrôleur en acquérant un circuit d'évaluation disponible auprès de Lextronic [2].

La caractéristique principale de ce microcontrôleur est la finesse avec laquelle il permet d'activer chaque périphérique et la multitude des oscillateurs associés à chaque périphérique. Rappelons ici que la consommation électrique d'un circuit numérique basé sur une technologie CMOS est principalement déterminée par la cadence des transitions d'un état à un autre des portes logiques : plus la fréquence est élevée, plus la consommation est élevée. Pouvoir associer un oscillateur de fréquence appropriée aux besoins de chaque périphérique est donc une stratégie très intéressante pour économiser de l'énergie. Le MSP430 fournit 3 sources d'oscillateurs possibles : un oscillateur interne peu stable mais ne nécessitant aucun composant externe, un oscillateur basse fréquence (typiquement 32768 Hz, nommé ACLK) et un oscillateur haute fréquence (inférieur à 8 MHz, connecté aux broches XT2 du composant, que nous associerons à une source d'horloge nommée SMCLK d'un point de vue logiciel) [1].

Ce microcontrôleur est suffisamment puissant pour justifier l'utilisation de `gcc`, qui a été adapté à ce type d'architecture [3]. Afin de profiter au maximum des performances du microcontrôleur (programmation en assembleur) tout en nous autorisant le développement d'algorithmes complexes (utilisation d'un langage évolué tel que le C), nous nous focaliserons sur la cohabitation de ces deux langages dans nos codes sources.

1.1 Installation des outils de développement

Nous nous sommes contentés, pour installer l'ensemble des outils de développement pour MSP430 sous GNU/Linux, de suivre la procédure proposée dans [2], après avoir converti les paquets RPM obtenus à `http://rpmfind.net/` en un format compatible avec la distribution Debian utilisée par les auteurs (utilisation de `alien`). Nous avons donc utilisé les paquets suivants :

```
cdk-msp-base-0.2-20031111.i386.rpm
cdk-msp-binutils-2.14-20031106.i386.rpm
cdk-msp-examples-libc-20031101cvs-20031102.noarch.rpm
cdk-msp-examples-mspgcc-20031101cvs-20031102.noarch.rpm
cdk-msp-gcc-3.3.2-20031106.i386.rpm
cdk-msp-gdb-5.1.1-20031106.i386.rpm
cdk-msp-gdb-proxy-5.1.1-20031106.i386.rpm
cdk-msp-isp-20031101cvs-20031104.noarch.rpm
cdk-msp-isp-bsl-20031101cvs-20031104.noarch.rpm
cdk-msp-isp-parjtag-20031101cvs-20031104.noarch.rpm
cdk-msp-isp-serjtag-20031101cvs-20031104.noarch.rpm
cdk-msp-jtag-lib-20031101cvs-20031103.i386.rpm
cdk-msp-libc-20031101cvs-20031102.noarch.rpm
```

Le résultat est une arborescence `/opt/cdk4msp` qui contient les librairies et entêtes nécessaires à la compilation, les outils de compilation (compilateur, assembleur, éditeur de liens et convertisseur entre les divers formats d'objets binaires dans `/opt/cdk4msp/bin` que nous pouvons éventuellement ajouter au `PATH` pour se simplifier la vie) et de programmation *via* le port parallèle par `msp430-parjtag` qui nécessite le language interprété Python (il s'agit en fait du programme `jtag.py`). Dans la suite de ce document, nous nous réfèrerons aux outils de développement en supposant qu'ils ont été installés dans cette arborescence.

[2] `http://www.lextronic.fr/elekladen/msp1.htm` et `http://www.lextronic.fr/elekladen/jtag.htm` pour un coût total de 56 euros

[3] `mspgcc.sourceforge.net/`

La compilation se fait au moyen d'une ligne de commande du type
`msp430-gcc -g -O3 -mmcu=msp430x149 -D_GNU_ASSEMBLER_ prog.S -o prog` si l'on veut générer, depuis un programme en assembleur `prog.S`, un binaire au format ELF `prog` qui pourra être flashé dans le microcontrôleur. Il est fondamentale de bien préciser le type de microcontrôleur utilisé – ici `msp430x149` – puisque c'est ainsi que le compilateur connaît les périphériques disponibles et leur emplacement, ou la table de vecteurs d'interruptions [4] à utiliser.

Une fois le fichier ELF disponible, la programmation *via* le port parallèle auquel est connecté l'adaptateur JTAG se fait au moyen du programme `msp430-jtag` : nous effaçons la mémoire flash avant d'y placer notre programme, et ce en imposant l'interface du port parallèle dans un environnement utilisant udev :
`msp430-jtag -e -l /dev/.static/dev/parport0 prog`. De nombreuses erreurs que l'on pourrait croire associées à l'accès au périphérique `parport0` traduisent en fait des erreurs d'accès au microcontrôleur (par exemple si le câble JTAG n'a pas été connecté) : les messages sont donc trompeurs et peuvent nécessiter une recherche aussi bien au niveau logiciel que matériel pour résoudre un éventuel problème.

La figure 1 présente quelques exemples de circuits sur lesquels exécuter les programmes présentés dans ce document : un circuit dédié à nos applications, dessiné sous Eagle (`www.cadsoft.de`) et inspiré du circuit fourni à `http://users.tkk.fi/~jwagner/electr/dc-rx/`, et d'autre part un support pour le circuit commercialement disponible auprès de Lextronic (MSP430-H149 de Olimex). Attention cependant au premier cas au port JTAG non standard qui nécessite de faire un adaptateur entre ce port et le cordon commercialisé par Lextronic. Ce schéma de principe ne présente pas les résonateurs qui sont déjà inclus sur le montage de démonstration de Olimex : il nous suffit d'ajouter les alimentations et les diodes pour faire nos premiers pas. Nous prévoyons dès maintenant de câbler les liaisons de communications asynchrones (entrée et sortie de l'UART) et deux broches de timers pour les applications qui vont suivre.

1.2 Premier exemple en C

`mspgcc` est fourni avec quelques exemples installés, suivant la procédure citée plus haut, dans `/opt/cdk4msp/examples/mspgcc` (table 1). L'exemple classique d'introduction de la diode qui clignote est développé dans le sous répertoire `leds`. Bien que cet exemple `main.c` illustre l'utilisation d'un langage évolué sur ce microcontrôleur, il propose une démonstration flagrante que pour une application aussi simple, le C ne présente aucun intérêt par rapport à l'assembleur puisque la majorité du code consiste en l'assignation de valeurs à des registres associés à des fonctions matérielles. La partie la plus fastidieuse du travail consiste donc à s'imprégner des diverses options associées aux divers registres telles que décrites dans le manuel de l'utilisateur [3]. Nous conserverons donc la capacité à travailler en C pour les cas exceptionnels où un algorithme complexe doit être implémenté, pour immédiatement nous focaliser sur l'assembleur, plus efficace et ne nous soumettant pas aux aléas des optimisations d'un compilateur.

Notez que par défaut les exemples fournis avec `mspgcc` compilent pour le MSP430F1121 : il faut penser, pour compiler pour le MSP430F149 qui nous intéresse, à modifier le Makefile et y remplacer `CPU=msp430x1121` par `CPU=msp430x149` pour des raisons que nous détaillerons plus loin. L'entête `msp430/include/msp430/gpio.h` contient les adresses des registres de configuration des ports, ici le port 1 auquel nous avons connecté les LEDs. L'initialisation d'un port pour une utilisation d'entrée sortie passe par la définition de la fonction de la broche puisque chaque broche est assignée soit à un périphérique, soit à une entrée-sortie générale (GPIO) : le registre `PiSEL` contient un bit à 0 pour le GPIO ou 1 pour une utilisation en entrée/sortie de périphérique. Puis la direction de communication est définie dans `PiDIR`, et enfin la valeur de la broche est définie en sortie dans

[4] une interruption est un évènement – impulsion sur une broche, fin de conversion analogique-numérique, compteur atteignant une valeur prédéfinie – qui provoque l'arrêt de l'exécution séquentielle du programme en le faisant sauter à une adresse programmée dans un emplacement prédéfinie nommé le vecteur d'interruption. Une fois la routine associée à l'interruption exécutée (*ISR : Interrupt Service Routine*), le programme retourne au point où il se trouvait lorsque l'interruption est intervenue. Certaines interruptions ont la faculté de réveiller un processeur placé en mode veille pour économiser l'énergie qu'il consomme.

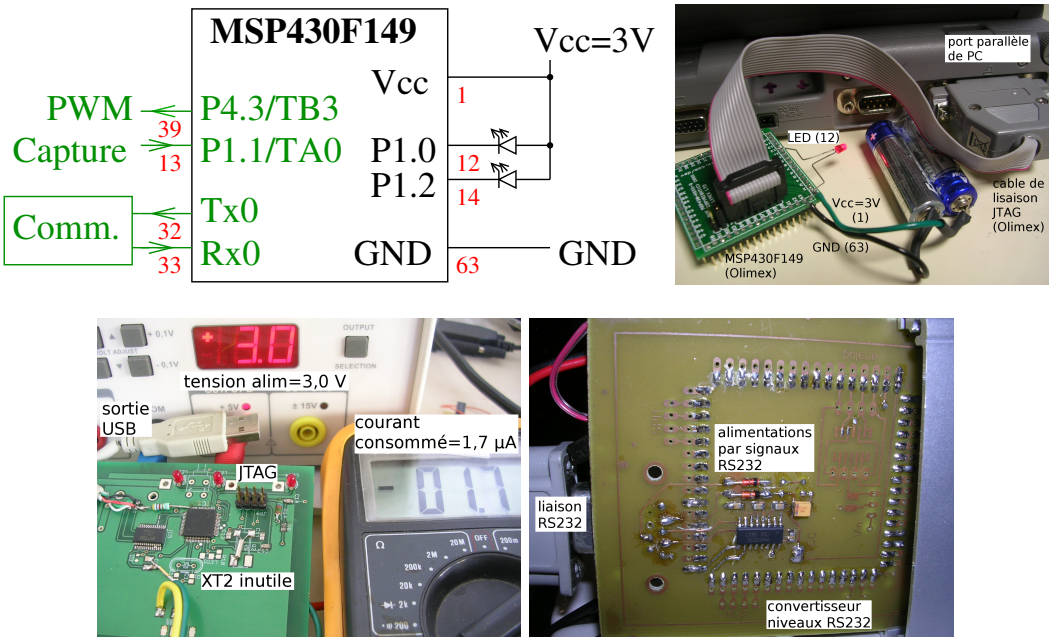


FIG. 1 – Circuits électroniques réalisés pour ces développements. En haut : en noir sur le schéma les connexions pour nos premiers essais consistant à faire clignoter des LEDs connectées au port 1. En vert nous prévoyons de connecter le broches qui vont nous servir dans la suite de ce document. Un montage rapide pour un premier essai, directement sur la plaque MSP430-H149 réalisée par Olimex et commercialisée en France par Lextronic, permet d'immédiatement se mettre au travail. Les nombres entre paranthèses sont les numéros de broches auxquelles sont connectés les composants. En bas, à gauche, une carte spécialement développée pour l'application horloge temps réel dans laquelle un MSP430 est associé à un FT232RL pour la communication USB. L'exécution du programme d'horloge temps réel avec alarme consomme en moyenne 1,7 μA et fonctionne au moins dans la gamme 3,0-3,6 V, compatible avec une alimentation directe par piles. En bas à droite, une carte supportant la MSP430-H149 avec ici le MAX3232 qui convertit les signaux issus de l'UART du microcontrôleur en niveaux compatibles avec la norme RS232. La carte MSP430-H149 de Lextronic vue de dessus est aussi visible sur la Fig. 2.

```
#include "hardware.h"    // originally written by chris <cliechti@gmx.net>

void delay(unsigned int d) {int i;for (i = 0; i<d; i++) {nop();nop();}}

int main(void) {
    WDTCTL = WDTCTL_INIT;               //Init watchdog timer
    P1OUT  = P1OUT_INIT;                //Init output data
    P1SEL  = P1SEL_INIT;                //Select GPIO or module function
    P1DIR  = P1DIR_INIT;                //Init port direction register
    P1IES  = P1IES_INIT;                //init port interrupts
    P1IE   = P1IE_INIT;
    while (1) {P1OUT ^= 0xff));delay(0x4fff);}
}
```

TAB. 1 – Exemple issu de `/opt/cdk4msp/examples/mspgcc/leds` proposant une application en C faisait clignoter les diodes connectées au port 1 du MSP430. On constate que mis à part la fonction `delay()` qui consiste en du C "classique", la majorité des instructions dans un cas aussi simple se résument à des assignations de ports dont la description se trouve dans les fichiers d'entête chargés par `hardware.h`.

`PiOUT`, ou en lecture dans `PiIN`. Dans le cas du MSP430F149, le numéro de port `i` va de 1 à 6 tel que défini dans la configuration matérielle du microcontrôleur `msp430/include/msp430x14x.h`. C'est ici que nous constatons l'importance de définir le bon type de processeur dans l'option `-mmcu` de `gcc` car de ce paramètre dépendra la définition des bonnes valeurs dans les bonnes variables.

Ce premier exemple ne nécessite donc qu'un composant – une LED – une pile, le circuit de

démonstration et le câble JTAG associé pour commencer à appréhender le fonctionnement du microcontrôleur. Essayons d'approfondir désormais notre compréhension de l'architecture par un exemple en assembleur.

1.3 Premier exemple en assembleur

Contrairement à un programme C qui repose souvent sur des librairies et des entêtes, avec une initialisation du processeur fournie et généralement cachée au programmeur, les programmes que nous proposerons ici seront autonomes. Ils doivent donc respecter une structure précise pour être compatible avec `gcc`.

Un programme assembleur compilé par `gcc` doit nécessairement contenir une fonction `.main`, qui correspond à la fonction `main()` du source C correspondant. Cette fonction est rendue globalement visible par `.global main` : elle sert de point d'entrée à l'exécution du programme. Habituellement, cette fonction contient les initialisations des périphériques du processeur, des horloges si nécessaires, puis une boucle infinie faisant appel aux fonctions que doit réaliser le microcontrôleur. Nous profiterons du préprocesseur de `gcc` pour définir des macros par `#define` afin de rendre l'utilisation des registres plus explicites à la relecture du programme. De la même façon, nous pouvons utiliser les instructions de compilation conditionnelle `#ifdef ... #endif` pour tester différentes versions d'un même programme, par exemple pour communiquer avec un des deux ports série selon le circuit sur lequel le programme doit être exécuté. Ces quelques notions esthétiques introduites, passons au vif du sujet de l'assembleur MSP430.

Comme tout processeur, les instructions se classent principalement en opérations logiques, arithmétiques, de déplacement de données et de contrôle du séquençage du processeur (interruptions par exemple). Nous n'allons pas ici passer en revue toutes ces instructions qui sont nommées par des mnémoniques simples à comprendre, mais nous allons nous focaliser sur les déplacements de données pour introduire les conventions de syntaxe.

Les assignations en assembleur MSP430 se font avec la source comme premier argument et la destination comme second argument : afin de placer une constante dans le registre R15, on notera `MOV #0x123,R15`. Les registres sont sur 16 bits donc toute instructions se décline pour un travail soit sur un octet (8 bits) par l'ajout du suffixe `.b`, soit sur un mot (16 bits) par l'ajout du suffixe `.w`. Il peut être utile de ne pas laisser l'aléatoire déterminer pour nous le bon fonctionnement du programme en demandant à `gcc` de choisir la bonne instruction en fonction de la taille des variables mais d'expliciter nous même le nombre de bits sur lesquels nous voulons travailler : dans l'exemple qui précède, on notera donc `MOV.W #0x123,R15`. Cette remarque est particulièrement importante pour les instructions de rotation où le dernier bit sortant est réinjecté dans le registre, soit au huitième bit, soit au seizième bit, selon l'instruction utilisée. Nous n'allons pas ici passer en revue toutes les instructions : les mnémoniques sont intuitifs et les plus utiles seront visibles dans les divers exemples que nous avons inclus dans ce document. Une constante est notée par le préfixe `#`, le contenu d'un registre est indiqué par le préfixe `&`, et une assignation indexée par un registre `Ri` se note `0(Ri)` si par exemple ici l'indice est nul. Donc `MOV.B #0x12,R4` place la valeur hexadécimale 0x12 dans le registre R4, tandis que `MOV.B &0x12,R4` place le contenu de l'emplacement mémoire 0x12 dans R4. Enfin, `MOV.B #0x12,0(R4)` placera la valeur 18 dans l'emplacement mémoire d'adresse contenue dans R4.

12 registres de 16 bits sont disponibles pour nos utilisations : R4 à R15. R1 est le pointeur de pile et doit être manipulé avec soin. Dans nos exemples, nous nous contenterons de l'initialiser au lancement du programme et le laisserons évoluer librement au gré des empilements de variables et d'appels de routines. R1 doit pointer vers un emplacement en RAM (puisqu'il empile et dépile des variables), vers une zone suffisamment élevée pour que le bas de la pile n'atteigne jamais la limite de la RAM en 0x200 [5], sans toutefois occuper toute la RAM qui peut avoir d'autres utilités. Nous avons choisi arbitrairement la valeur initiale de 0x280 qui alloue 128 octets de RAM à la pile. Le choix de la taille de la pile est un problème complexe qui nécessite de convenablement comprendre les appels aux fonctions et les paramètres qui sont conservés lors d'une fonction `CALL`. En C, `gcc`

[5] rappelons qu'empiler consiste à décrémenter R1 et dépiler consiste à incrémenter R1

conserve sur la pile tous les registres qui ne sont pas utilisés pour les passages de paramètres, soit dans le pire cas (R4 à R15) 12 registres de 16 bits ou 24 octets. Il nous faut de plus mémoriser l'adresse d'appel pour retourner (instruction `ret`) au point d'appel de la fonction à la fin de son exécution. Au rythme de 26 octets par appel de fonction (sans compter les variables que `gcc` place sur la pile en l'absence de registre disponible), nous n'avons droit qu'à 4 imbrications de subroutines avant de saturer la pile. Le contrôle de la pile est plus simple en assembleur puisque le programmeur choisit manuellement toutes les opérations qu'il effectue sur la RAM, sur les registres ou sur la pile.

L'exemple précédent consistant à faire clignoter les diodes sur le port 1 s'écrit alors en assembleur :

```
#include <signal.h>
#include <io.h>

#define         tmp     R15

.global main
main:
RESET:          MOV.W   #0x280,R1
                MOV.B   #WDTPW+WDTHOLD,&WDTCTL  ; Stop Watchdog Timer
                MOV.B   #0,&P1OUT
                MOV.B   #0,&P1SEL
                MOV.B   #0xff,&P1DIR
                MOV.B   #0,&P1IES
                MOV.B   #0,&P1IE

Mainloop:       xor.b   #0xff,&P1OUT    ; clignote
                call    #delai          ; attend
                JMP     Mainloop        ; Endless Loop

delai:          mov.w   #0x4fff,tmp
attend:         dec.w   tmp
                jnz     attend
                ret
```

Un registre, R15, est assigné à une fonction de variable temporaire, `tmp`, qui nous servira de compteur d'attente. Après l'initialisation du port 1, la diode clignote par opération de OU exclusif sur la valeur précédente du port.

Cet exemple fonctionne, mais ne met pas en valeur l'intérêt de travailler en assembleur avec le contrôle précis du déroulement de l'exécution du programme, ni l'intérêt d'un processeur faible consommation avec ses multiples modes de veille, puisque les phases d'attente consistent à faire travailler le microcontrôleur dans une boucle vide.

1.4 Économie d'énergie et gestion des interruptions

Notre objectif est ici double : d'une part remplacer un délai dont la durée est difficile à déterminer par une définition de la période du clignotement de la diode précise, et d'autre part placer le processeur en mode veille lorsqu'il est inactif.

Le programme suivant produit le même résultat que l'exemple précédent, mais avec une consommation considérablement réduite.

```
// ici on conserve le gcrt0.S avec ses initialisation et on *modifie*
// la table des vecteurs d'interruption plutot que de vouloir la
// redefinir
// une des differences entre les differents MSP430 est la taille et le
// contenu de la table des vecteurs d'interruption => NE PAS les definir
// par des .word car ce n'est pas portable
// /opt/cdk4msp/bin/msp430-jtag -l /dev/.static/dev/parport1 -e jmf

#include <signal.h>
#include <io.h>

.global main
main:           MOV.W     #0x280,R1

                CALL    #Setup          ; Prepare LCD and basic timmer
Mainloop:       BIS     #LPM3,R2        ; Set SR bits for LPM3
                xor.b   #0xff,&P1OUT
                JMP     Mainloop        ; Endless Loop

Setup:          MOV     #WDTPW+WDTHOLD,&WDTCTL  ; Stop Watchdog Timer

setupTA:        MOV     #TASSEL0+TACLR, &TACTL  ; ACLK for Timer_A.
                BIS     #CCIE,&CCTL0            ; Enable CCR0 interrupt.
                MOV     #0x7FFF,&CCR0           ; load CCR0 with 32,767.
                BIS     #MC0,&TACTL             ; start TA in "up to CCR0" mode

                MOV      #0,&P1OUT
                MOV      #0,&P1SEL
                MOV      #0xff,&P1DIR
                MOV      #0,&P1IES
                MOV      #0,&P1IE

                EINT                            ; Enable interrupts
                RET                     ; Done with setup.

;-------------------------------------------------------------------------
;       Timer_A ISR:
;       CPU is simply returned to active state on RETI by manipulated the SR
;       bits in the SR value that was pushed onto the stack.
;       CCR0 IFG is reset automatically.
;-------------------------------------------------------------------------
interrupt(TIMERA0_VECTOR)               ;register interrupt vector
.global CCR0INT                         ;place a label afterwards so
CCR0INT:        BIC     #LPM3,0(R1)     ; Clear SR LPM3 Bits, on top of stack
                RETI                    ;
```

Plusieurs nouveautés apparaissent dans ce programme : l'utilisation d'un timer sur 16 bits (TA) pour régler l'intervalle de temps entre deux changements d'états de la LED, la mise en mode veille du processeur par la manipulation du bit LPM3, et la création d'une interruption qui fait sortir le processeur de son sommeil à chaque fois que le compteur atteint une condition prédéfinie.

Contrairement aux exemples proposés par Texas Instruments (et distribués dans les exemples de mspgcc à http://mspgcc.cvs.sourceforge.net/mspgcc/examples) qui redéfinissent une table d'interruption fixe dans le programme principal, il vaut mieux partir de la table d'interruption

proposée par défaut par gcc dans la séquence de démarrage telle que décrite à `/opt/cdk4msp/msp430/include/msp430x14x.h` et adaptée à chaque modèle, et en modifier les assignations pour les interruptions qui seront utilisées. En effet, la position des vecteurs d'interruption dépend du modèle de microcontrôleur et définir une nouvelle table induit une perte de portabilité du code. Nous trouvons dans le fichier cité plus haut les noms des diverses interruptions dont nous redéfinissons les opérations par un code du type

```
interrupt(TIMERA0_VECTOR)
.global FONCTION
FONCTION: // diverses op\'erations
          RETI
```

où dans cet exemple `TIMERA0_VECTOR` est une constante définissant l'emplacement du vecteur d'interruption associé à une des horloges du microcontrôleur, `FONCTION` est un nom quelconque qui doit être visible au moment de l'édition de lien, et donc la séquence s'achève par l'instruction `RETI` dont le rôle est de retrouver sur la pile l'adresse où se trouvait le programme au moment de l'interruption et de poursuivre l'exécution à cet emplacement. Par défaut, toutes les interruptions sont associées à une unique instruction `reti` telle que visible dans le `gcrt0.s` des sources de `msp430-libc`.

Nous constatons donc une fois de plus que choisir le bon processeur lors de la phase de compilation (option `CPU=` dans le Makefile mentionné plus haut – section 1.2 – ou option `-mmcu` de `gcc`) est fondamentale afin d'obtenir les bonnes adresses de périphériques et une table de vecteur d'interruption convenablement initialisée.

1.5 Application à l'horloge temps-réel

Une des raisons qui nous ont poussé à nous intéresser au MSP430 est comme solution de remplacement de l'horloge temps réel DS1305 qui fournit un calendrier, une horloge et deux alarmes pour une consommation totale de l'ordre de la centaine de μA. Nous désirons travailler sur une application qui réveille quotidiennement un appareil : la majorité de la consommation électrique de cette application sera déterminée par l'horloge qui réveille l'application. L'implémentation d'une telle horloge sur MSP430 fournit donc à la fois la perspective d'une réduction de consommation et la souplesse d'une solution logicielle facilement adaptable au problème étudié. Nous allons donc ici proposer une implémentation d'une horloge temps réel [4] avec alarme et en évaluer la consommation électrique.

Dans l'exemple qui précède, nous faisions clignoter une diode avec une période déterminée par le Timer A dont la fréquence de comptage est déterminée par le quartz basse fréquence ACLK. En réglant la période de réveil sur 0x7fff=32765 pulsations, nous obtenons une impulsion par seconde, et ce avec une précision déterminée par la qualité du quartz. Plutôt que de changer l'état d'une diode, nous pouvons faire appel à une fonction qui se charge d'incrémenter les secondes, minutes et heure afin de réaliser une horloge. Les trois registres mobilisés pour cette tâche sont disponibles et ne voient pas leur état changer par le passage en mode veille : la fonction d'incrément de l'horloge est donc très simple. Son appel par `CALL #CLOCK` remplace la ligne `XOR #0xff,...` de l'exemple précédent.

```
#define         SEC     R13
#define         MIN     R14
#define         HR      R15
...
Mainloop:       BIS     #LPM3,R2        ; Set SR bits for LPM3
                CALL    #Clock          ; Update Clockifdef
                JMP     Mainloop        ; Endless Loop

;-------------------------------------------------------------------------
;    Clock: Update clock SEC and MIN and HR in BCD format
;    Originally written by Lutz Bierl.
;-------------------------------------------------------------------------
Clock:          SETC                    ; Set Carry bit.
                DADC.b  SEC             ; Increment seconds decimally
                CMP.b   #0x60,SEC       ; One minute elapsed?
                JLO     Clockend        ; No, return
                CLR.b   SEC             ; Yes, clear seconds
                DADC.b  MIN             ; Increment minutes decimally
                CMP.b   #0x60,MIN       ; Sixty minutes elapsed?
                JLO     Clockend        ; No, return
                CLR.b   MIN             ; yes, clear minutes
                DADC.b  HR              ; Increment Hours decimally
                CMP.b   #0x24,HR        ; 24 hours elapsed?
                JLO     Clockend        ; No, return
                CLR.b   HR              ; yes, clear hours
Clockend:       RET                     ;

Setup:
...
                MOV.b   #0,SEC          ; Clear SEC
                MOV.b   #0,MIN          ; Clear MIN
                MOV.b   #0,HR           ; Clear HR
                EINT                    ; Enable interrupts
```

Supposons que nous nous trouvions dans le cas le plus défavorable d'une consommation de l'ordre de 100 μA (nous avons en pratique mesuré – avec une incertitude de l'ordre de 100% – une consommation proche de la valeur théorique de 3 μA puisque le microcontrôleur passe la majeur partie du temps en mode de veille LPM3), et que les piles utilisées fournissent une tension supérieure à 3,9 V pour une énergie emmagasinée de 2100 mA.h (cas de 4 accumulateurs NiMH en série par exemple). Alors la durée de vie de notre montage est de l'ordre de 2100/0,1=21000 h ou 875 jours, soit plus de deux ans. Un montage capable de fonctionner en autonomie pendant plus d'un an ouvre des perspectives intéressantes d'automatisation du fonctionnement d'instruments dont la mise en marche par appui sur des interrupteurs par un opérateur humain est simulé par un interrupteur analogique de type 4066.

Nous avons vu une exemple simple de programme en C et son équivalent en assembleur pour le clignotement d'une LED. Ces programmes simples ne présentent pas de différence majeure quelquesoit le langage utilisé. Il arrive cependant qu'un langage présente un avantage sur l'autre : lisibilité et portabilité pour le C, granularité plus fine dans le cas de l'assembleur. Nous allons donc voir comment faire dialoguer ces deux langages.

1.6 Appel d'une fonction en C depuis l'assembleur

Il s'agit ici de la fonctionnalité la plus intéressante : le cœur du séquençage du programme avec sa gestion des interruptions se fait en assembleur, et parfois une fonction écrite en C est appelée afin d'effectuer un calcul plus complexe.

Le passage de paramètres aux fonctions C depuis l'assembleur sont décrites au chapitre 5 de [5] : les arguments sont passés dans l'ordre par R15, R14, R13 et R12 pour des paramètres codés sur 8 ou 16 bits. Les arguments codés sur plus de 16 bits seront passés par des concaténations de registres de la forme R15 :R14. De la même façon, l'argument retourné par une fonction C sera contenu dans R15 (8 ou 16 bits) ou R15 :R14 (32 bits).

Bien que `gcc` prenne soin de protéger en les empilant (instruction assembleur `push`) les registres qui ne sont pas utilisés en passage de paramètre, il est prudent d'analyser grossièrement l'utilisation de la pile et des registres par la fonction C après assemblage. La meilleure lisibilité du code assemblé est obtenue en désassemblant le fichier ELF issu de la compilation : `msp430-gcc -g -mmcu=msp430x149 -D_GNU_ASSEMBLER_ asm_in_c.c -o asm_in_c && msp430-objdump -dSt asm_in_c`. Cette méthode a le bon goût de fournir un code assembleur *commenté* plus simple à lire que l'interruption à la phase de génération du code assembleur (option `-S` de `msp-gcc`). Un exemple d'un tel code est proposé en bas de la table 3.

Nous verrons plus loin (section 3) une application concrète de ces principes, mais illustrons ces concepts sur un exemple simple issu de l'horloge temps réel : nous voulons y ajouter une fonction d'alarme en C qui se déclenche chaue fois que les secondes sont un multiple de 5. Par souci de généralité, nous passerons trois arguments que sont les heures, minutes et secondes stockées au format BCD dans les registres R15 à R13 respectivement. La fonction C renverra une valeur signifiant si l'alarme est atteinte ou non.

Soit une fonction C présentée dans le tableau 2 : tous les paramètres sont des valeurs sur 8 bits. Cette fonction recevra ses 3 arguments dans les registres R15 à R13 et renverra le résultat de son calcul dans R15 (qu'il aura fallu prendre soin de protéger en l'empilant dans le code assembleur si on ne voulait pas en perdre la valeur). Du point de vue assembleur, cette fonction s'appelle par

```
    push    r15
    push    r14
    push    r13
    CALL    #jmf_alarme     ;
    cmp.b   #1,R15
    jne     pas1
    ...  ; action si R15=1 ie alarme activee
pas1:
     pop     r13
     pop     r14
     pop     r15
```

À l'issue de la fonction C, nous testons la valeur contenue dans R15, qui sera égale à 1 si les secondes sont multiples de 5, faute de quoi aucune action n'est prise et les registres contenant la date retournent à leurs valeurs initiales que nous avions pris soin de conserver sur la pile.

```
// params en entree sont R15, R14 (puis R13, R12) et reponse dans R15
unsigned char jmf_alarme(unsigned char heure, unsigned char min, unsigned char sec)
{unsigned char reste;
 reste=((seconde&0xF0)>>4)*10+(seconde&0x0F);
 reste=sec%5;
 if (reste==0)
     {*((volatile unsigned unsigned char*)0x21) = 0x02;
      return(1);}
  else return(0);
 }
```

Tab. 2 – Exemple de programme C interagissant avec le programme assembleur présenté dans le texte : le programme assembleur gère une horloge temps réel avec incrément des secondes (R13), minutes (R14) et heures (R15) au format BCD, tandis que le programme C gère une alarme. Les variables sont échangées entre les deux programmes mais le C permet d'implémenter de façon plus lisible des algorithmes complexes. Ici nous nous contentons d'effectuer des comparaisons pour déclencher une alarme en cas d'égalité.

Une autre méthode de passage de paramètre entre l'assembleur et le C ne passe plus par l'API de `gcc` (convention d'utilisation des registres en entrée et en sortie des fonctions) mais par la RAM. Nous allons placer une variable dans un emplacement de la mémoire vive que nous choisissons par convention, et ce même placement sera appelé depuis le C qui peut éventuellement décider d'y stocker le résultat de son calcul. Cela revient donc à effectuer un passage de paramètre par pointeur.

Cette méthode est particulièrement efficace lorsque le C doit se souvenir de la valeur d'une variable entre deux appels. Nous ne savons en effet pas *a priori* quel usage est fait des registres lors de la compilation d'une fonction C. Nous ne pouvons pas non plus faire d'hypothèse quand à la capacité de ces registres à conserver leur valeur entre deux appels de la fonction (une fonction C ne doit pas se rappeler d'un appel à l'autre du contenu de ses variables). C'est là que nous ferons appel à la mémoire volatile du processeur : en prenant soin de ne pas occuper toute la RAM pour la pile mais de conserver un espace pour notre propre usage, nous pourrons y placer des variables dont nous aurons le contrôle des modifications. Rappelons la syntaxe peu usuelle pour allouer une valeur `v` à un emplacement prédéfini `e` en RAM : `*(type *)(e)=v`, avec `type` soit `unsigned char` si la variable `v` est définie sur 8 bits, `unsigned short` ou `unsigned long` pour 16 et 32 bits respectivement. Éviter l'utilisation du type `int` dont la taille n'est pas standard (un `int` est contenu dans 16 bits sur MSP430 mais sur 32 bits sur processeur Intel x86, d'où des problèmes de portabilité que nous illustrerons plus tard dans ce document, section 3).

Ces deux possibilités doivent être convenablement comprises pour une utilisation efficace des fonctions C dans l'assembleur : par convention, `gcc` utilisera les registres autant que possible car sur la majorité des processeur, leur accès est plus rapide que pour des données stockées en mémoire. Tandis que sur des processeurs équipés de peu de registre l'utilisation de la RAM est fréquente, les 12 registres du MSP430 sont largement suffisant pour la majorité des applications et l'accès à la mémoire n'est *a priori* pas nécessaire.

1.7 Utilisation de l'assembleur dans un programme C

La documentation de mspgcc [5] décrit en détail au chapitre 6 l'utilisation de l'assembleur dans un programme C. La principale difficulté consiste à passer des variables définies en C au bout de code assembleur, et réciproquement de renvoyer le résultat disponible dans un registre à l'issue de la routine en assembleur vers le C. Nous avons expérimenté (Tab. 3) avec deux types de passages de paramètres : par la mémoire et par un registre.

La commande `asm()` permet d'inclure un bout de code en assembleur dans un programme C compilé par `gcc`. Plusieurs lignes d'assembleur peuvent être insérées dans une même commande `asm`, séparées par des retours chariot \n.

Nous constatons dans cet exemple que

– les variables sont définies sur la pile. En l'absence d'optimisation, la pile (R1) est temporairement copiée dans R4 et les variables sont placées en des emplacements relatifs à cette pile,

```
// #include "hardware.h"

int main()
{int i,j,k,l[3];
 i=1;
 j=2;
 k=i+j;
 l[1]=k;

 asm("mov #1,r12");
 asm("mov %1,r12\n"
     "adc r12\n"
     "mov r12,%0"
 : "=m" (j) : "m" (i));
 asm("mov %1,%[iasm]": [iasm]  "=r" (i) :"r" (j));
 asm("mov %1,%[iasm]": [iasm]  "=m" (i) :"r" (j));
 asm("mov %1,%[iasm]": [iasm]  "=r" (i) :"m" (j));
 asm("mov %1,%[iasm]": [iasm]  "=m" (i) :"m" (j));
 return(0);
}
```

```
int main()
{int i,j,k,l[3];
    1140: 31 40 f4 09  mov #2548,r1 ;#0x09f4
    1144: 04 41        mov r1,r4 ;
 i=1;
    1146: 94 43 00 00  mov #1,0(r4) ;r3 As==01
 j=2;
    114a: a4 43 02 00  mov #2,2(r4) ;r3 As==10
 k=i+j;
    114e: 2f 44        mov @r4,r15 ;
    1150: 1f 54 02 00  add 2(r4),r15 ;
    1154: 84 4f 04 00  mov r15,4(r4) ;
 l[1]=k;
    1158: 94 44 04 00  mov 4(r4),8(r4) ;
    115c: 08 00

 asm("mov #1,r12");
    115e: 1c 43        mov #1,r12 ;r3 As==01
 asm("mov %1,r12\n"
    1160: 2c 44        mov @r4,r12 ;
    1162: 0c 63        adc r12 ;
    1164: 84 4c 02 00  mov r12,2(r4) ;
     "adc r12\n"
     "mov r12,%0"
 : "=m" (j) : "m" (i));
 asm("mov %1,%[iasm]": [iasm]  "=r" (i) :"r" (j));
    1168: 1f 44 02 00  mov 2(r4),r15 ;
    116c: 0f 4f        mov r15,r15 ;
    116e: 84 4f 00 00  mov r15,0(r4) ;
 asm("mov %1,%[iasm]": [iasm]  "=m" (i) :"r" (j));
    1172: 1f 44 02 00  mov 2(r4),r15 ;
    1176: 84 4f 00 00  mov r15,0(r4) ;
 asm("mov %1,%[iasm]": [iasm]  "=r" (i) :"m" (j));
    117a: 1f 44 02 00  mov 2(r4),r15 ;
    117e: 84 4f 00 00  mov r15,0(r4) ;
 asm("mov %1,%[iasm]": [iasm]  "=m" (i) :"m" (j));
    1182: 94 44 02 00  mov 2(r4),0(r4) ;
    1186: 00 00
 return(0);
    1188: 0f 43        clr r15 ;
}
```

TAB. 3 – Exemple de code C incluant des morceaux d'assembleur et illustrant l'échange de variables définies dans le programme C avec l'assembleur. En bas, résultat de la compilation du code C qui illustre le passage de paramètre par registre (=r) ou par la pile (=m).

– le passage de paramètre entre le C et l'assembleur se fait en recherchant la valeur de la variable sur la pile : dans les 4 dernières lignes, nous prenons comme paramètre d'entrée la variable `j` située à l'offset 2 de la pile (`2(r4)`) et nous renvoyons le résultat dans `i` situé à l'offset 0 de la pile. De plus nous avons déclaré un alias pour la variable nommée `i` dans le programme C : cette variable s'appellera `iasm` dans le programme assembleur. Nous constatons que le passage par mémoire ou par registre fonctionne bien, et que dans ce cas le passage de paramètres par registre nécessite des instructions supplémentaires inutiles.

Les lignes d'assembleur ne sont pas compilées si une option d'optimisation est fournie au compilateur car ce dernier a identifié que nous affectons des registres qui ne sont jamais utilisés au cours de l'exécution. Afin de forcer la compilation, nous devons informer le compilateur de ne faire aucune hypothèse sur l'initialisation ou l'utilisation des registres inclus dans le code assembleur. Ce résultat s'obtient en remplaçant `asm();` par `__asm__ __volatile__();`. La lecture du code résultant en fonction des niveaux d'optimisations est alors très instructive et illustre bien la subtilité des optimisations.

1.8 Communication asynchrone : le port RS232

La communication asynchone – dans laquelle les deux interlocuteurs n'échangent pas explicitement de signal d'horloge mais supposent connaître par convention le débit de communication – nécessite une base de temps précise pour générer les transitions d'état de la ligne d'émission et échantillonner la ligne de réception avec la bonne cadence.

Nous avons vu que chaque périphérique peut être cadencé par diverses sources de fréquence, et plus cette fréquence est basse, plus la consommation est faible. Dans le cas de la transmission asynchrone RS232, la source basse fréquence ACLK à 32,768 kHz limite le débit à 9600 bauds. Au delà, il faut sélectionner une source haute fréquence (quartz), XT2 associé à SMCLK.

La principale contrainte de l'utilisation de l'UART est l'assignation de deux broches du port 3 qui ne peuvent plus être utilisées pour diriger l'allumage de diodes. L'assignation des broches au périphérique à la place d'un GPIO comme nous l'avons vu jusqu'ici s'obtient par :

```
SetupUART0:     bis.b #0xF0,&P3SEL ; P3.4,5,6,7 = USART select (0 and 1)
                bis.b #0x50,&P3DIR ; P3.4,6 = output direction

                mov.b #SWRST,&UCTL0     ; cf p.13-4
                bis.b #CHAR,&UCTL0      ; 8-bit characters: 13-22
                mov.b #SSEL0,&UTCTL0    ; UCLK = ACLK @ 32768 Hz
                mov.b #0x03,&UBR00      ;
                mov.b #0x00,&UBR10      ; cf p.13-16: 9600 bauds +/- 15%
                mov.b #0x4A,&UMCTL0     ;
                bic.b #SWRST,&UCTL0
                bis.b #UTXE0+URXE0,&ME1 ; Enable USART0 TXD/RXD
                ret
```

Le registre `UTCTL0` détermine la source de l'horloge : l'oscillateur RC interne est trop imprécis pour générer les signaux de communication nécessaires à la communication asynchrone, mais l'utilisation de l'horloge haute freǫuence XT2 est possible pour atteindre des débits plus élevés. Dans le cas d'un asservissement sur un quartz à haute fréquence, nous devons remplacer `mov.b #SSEL0,&UTCTL0` par `mov.b #SSEL1,&UTCTL0` afin de fournir le signal SMCLK à l'UART, et surtout sélectionner le bon facteur de division dans `UBR00` et `UBR10` en fonction du quartz choisi.

Une fois ces initialisations effectuées, la transmission d'un octet s'obtient par

```
rs_tx:          bit.b   #UTXIFG0,&IFG1  ; p.13-29: UTXIFG0=1 when U0TXBUF empty
                jz      rs_tx
                mov.b   tmp,&TXBUF0     ;
                ret
```

avec `tmp` un alias vers un registre disponible dont nous n'utiliserons que 8 bits (par exemple, `#define tmp R15`). Cette fonction s'insère aisément dans la fonction `CLOCK` vue auparavant pour transmettre sur le port RS232 vers un PC l'heure telle que la compte le MSP430 et ainsi valider le bon fonctionnement de l'horloge. La même opération permet de transmettre l'octet de poids fort en échangeant octets de poids faible et fort par `swpb R13`. Nous allons utiliser la capacité à transférer un mot de 16 bits pour réaliser un thermomètre numérique très simple.

1.9 Conversion analogique numérique : la mesure de température

Le MSP430F149 est équipé d'un convertisseur analogique-numérique de 12 bits de résolution, précédé d'un multiplexeur pour permettre la mesure de plusieurs signaux analogiques ou une diode interne à fort coefficient de température servant de thermomètre. Nous allons nous servir de cette dernière option, associée au canal 10 du multiplexeur, pour transférer sur le port série la température du microcontrôleur [6].

La conversion analogique-numérique s'obtient par la routine suivant :

```
SetupADC12:     ; book C. Nagy p.90
                mov.w   #SHT0_6+SHT1_6+REF0N+ADC12ON,&ADC12CTL0
                mov.w   #SHP,&ADC12CTL1
                mov.b   #INCH_10+SREF_1,&ADC12MCTL0 ; p.17-11: single conversion
                bis.w   #ENC+ADC12SC,&ADC12CTL0         ; MUST be .w
adc:            bit.b   #1,&ADC12CTL1   ; wait while ADC busy==1
                jnz     adc
                mov.w   &ADC12MEM0,R13  ; conversion result
```

Le résultat stocké dans le registre R13 est alors transmis par la routine de communication sur port RS232 vue dans le paragraphe précédent. La documentation du MSP430 [3, p.17-16] précise que tandis que la pente de la relation tension-température est déterminée par la physique de la diode et est constante à 3,55 mV/^{o}C, l'offset de la tension (*a priori* 0,986 V) peut varier d'un processeur à l'autre car dépendante des conditions de fabrication. Nous avons effectivement constaté dans notre cas un offset de 13^{o}C entre la température prédite par la formule fournie dans la documentation et la mesure expérimentale en étuve. Par la suite nous ne fournirons que des températures convenablement corrigées de cet offset.

1.10 La datation d'évènements

Nous avons déjà présenté l'utilisation d'un Timer pour générer périodiquement une interruption : un compteur s'incrémente jusqu'à atteindre une valeur prédéfinie puis repasse à 0 en déclenchant un évènement. Au contraire, ce même timer peut transférer sa valeur courante au moment où un évènement extérieur déclenche une interruption sur une broche sur laquelle est connectée un signal. Le cas classique est la mesure d'une période entre deux évènements : si nous déclenchons la capture de la valeur du timer sur le front montant d'une impulsion, alors l'application d'un créneau sur la broche associée à l'interruption d'Input Capture permet de mesurer la période de ce signal. Concrètement, le timer s'initialise alors de la façon suivante :

```
setupTA:MOV.w  #TASSEL0+TACLR, &TACTL  ; ACLK for Timer_A.
       BIS    #MC1,&TACTL              ; start TA in "up to FFFF" mode
       mov.w   #CM0+CAP+CCIE+SCS,&TACCTL0  ; capture sur front montant
//  /opt/cdk4msp/msp430/include/msp430/timera.h
```

Le compteur tourne ici continuellement de 0 à 0xFFFF, et transfère dans le registre `CCR0` la valeur du timer au moment du front montant du signal connecté sur la broche TA0 (numéro 13 pour le MSP430F149). Nous verrons plus loin comment utiliser cette méthode avec une impulsion générée à intervalles de temps bien connus pour mesurer la fréquence de l'oscillateur qui dirige le compteur.

1.11 La sortie modulée en largeur d'impulsion

Nous avons vu comment utiliser le timer A pour dater des évènements associés à une transition de niveaux sur une broche. Nous allons ici au contraire nous intéresser à la génération d'une impulsion lorsqu'une condition prédéfinie sur un timer est vérifiée. Nous allons utiliser le timer B afin de combiner cette fonctionnalité avec la précédente ([7]).

Concrètement, nous désirons une transition d'un état bas à haut lorsqu'une condition du timer est vérifiée, et de haut à bas lorsqu'une autre condition est vérifiée. L'application classique est la modulation en largeur d'impulsion (PWM, *Pulse Width Modulation*), classiquement utilisée pour contrôler les servos moteurs tels que ceux utilisés en modélisme (période des impulsions : environ 20 ms ; largeur des impulsions encodant l'angle de l'arbre du moteur : 1 à 2 ms), varier la vitesse d'un moteur à courant continu (en variant la puissance moyenne qui lui est appliquée), ou dans le cas qui nous intéresse ici, générer un signal analogique continu après passage dans un filtre passe-bas de constante de temps beaucoup plus longue que l'intervalle entre deux impulsions [8, 9]. En effet, le filtre passe bas lisse la sortie oscillante de la PWM pour générer un signal constant de valeur moyenne égal à la fraction de la largeur d'impulsion à la période. L'architecture classique choisie est dite de Sallen-Key, ne nécessitant que peu de composants passifs additionels. Nous utiliserons le signal ainsi généré pour modifier la valeur d'un condensateur contrôlé en tension (varicap) et ainsi décaler la fréquence de fonctionnement de l'oscillateur déterminant la cadence de fonctionnement du timer A. Noter que l'utilisation d'une PWM pour fabriquer par filtrage passe bas un convertisseur numérique-analogique (*Digital to Analog Converter – DAC*) nous permet d'insérer un transformateur entre le signal alternatif et le filtre passe bas et ainsi de générer une tension de sortie *dépassant* la tension d'alimentation du microcontrôleur selon un principe classiquement utilisé dans les pompes de charge.

Un tel DAC a cependant l'inconvénient d'être bruyant car un filtre basse bas ne peut pas "parfaitement" lisser le signal issu de la PWM. On préfèrera donc, pour les applications nécessitant un tension variable contrôlée numériquement, un montage de type R-2R, qui a cependant l'inconvénient de mobiliser plus de broches de sortie du microcontrôleur.

2 Application à l'asservissement d'un oscillateur sur le GPS

Nous proposons, comme application concrète des notions vues plus haut, l'asservissement du quartz contrôlant la cadence de fonctionnement du cœur et des périphériques du microcontrôleur sur une source de fréquence fournie par un récepteur GPS. Cet exemple sera une opportunité d'accéder à de nombreux périphériques et d'en étudier en détail le fonctionnement.

2.1 Mesure de la fréquence d'oscillation

Ayant affirmé que nous pouvons remplacer une horloge temps réel par notre circuit asservi sur un quartz annoncé à 32768 Hz, nous désirons valider la précision et la stabilité de l'oscillateur. Pour ce faire nous avons besoin d'une référence de la seconde *a priori* plus précise et plus stable que l'oscillateur à mesurer.

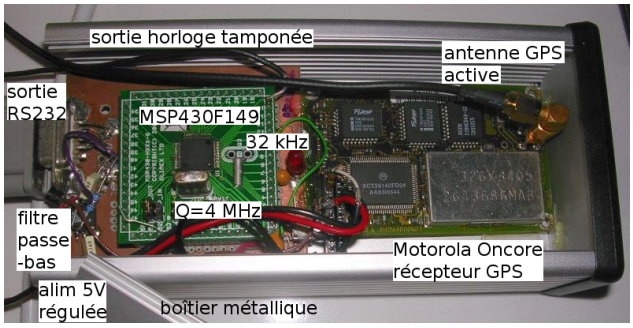


FIG. 2 – Le circuit à droite de la photo est un récepteur GPS Motorola Oncore GT, dont la sortie 1 PPS est connectée à l'entrée Input Capture A du MSP430, ici monté sur la carte commercialisée par Lextronic. Ce montage supporte le programme présenté dans la section 2.

Une des meilleures références de temps disponible en l'absence de matériel spécialisé est fournie par le système de géolocalisation GPS [10, 11, 12]. En effet, la position du récepteur est obtenue par mesure du temps de vol de signaux issus de la constellation de satellites dont la position est connue. Ces signaux sont générés par divers types d'horloges atomiques embarquées dans les satellites dont la précision est partiellement transférée pour finalement se traduire au sol, dans le cas d'un récepteur grand publique tel que le Motorola Oncore VP [6], par une impulsion de période 1 s±65 ns selon la documentation [13] : ce signal se nomme le 1 PPS (*1 Pulse Per Second*). Nous nous servirons de cette impulsion pour fabriquer un compteur basé sur la fonction Input Capture du Timer A : nous comptons le nombre d'oscillations de l'oscillateur basse fréquence ACLK (32 kHz) dans l'intervalle de temps d'une seconde (Fig. 2). Tout écart de la valeur 32768 (0x8000 en hexadécimal) résultera en une imprécision de la date fournie par notre horloge. Nous désirons de plus que la fréquence soit stable sur la gamme de température que peut rencontrer notre circuit – typiquement -20 à 60°C – et nous allons donc tenter de mesurer la stabilité de l'oscillateur sur cette plage de températures.

La fonction Input Capture place de façon matérielle (donc sans latence associée à un traitement logiciel) la valeur du compteur A dans le registre `TACCR0`. Notre logiciel se charge de transférer par liaison RS232 les valeurs successives de `TACCR0` lues chaque seconde ainsi que la température mesurée par la sonde interne de température du MSP430F149. Le logiciel de réception des données sur le PC soustrait les valeurs consécutives du timer pour en déduire le nombre d'oscillation dans l'intervalle de temps et permettre le tracé de la fréquence de quartz basse fréquence en fonction de la température (Fig. 3).

Lors de l'utilisation de l'oscillateur basse fréquence ACLK, le nombre d'oscillations par seconde (théoriquement 32768) est inférieur à la taille du compteur (16 bits) et le décompte est directement représentatif de la fréquence d'oscillation de l'oscillateur par lequel le compteur est cadencé (Fig. 3). La mesure de le fréquence en fonction de la température donne une parabole avec des coefficients cohérents avec ceux fournis dans les datasheets des résonateurs à quartz (le coefficient quadratique obtenu dans notre expérience est $1{,}3\times10^{-3}/32768 = 0{,}039$ ppm$/(\Delta^oC)^2$ comparable aux 0,042 ppm$/(\Delta^oC)^2$ proposés à `www.hosonic.com/pdf/frequency/f04.pdf`). Cependant, nous constatons que la mesure sur un intervalle d'une seconde sera probablement trop

[6] le récepteur GPS Globalsat ET-312 commercialisé en France par Lextronic est annoncé avec un point de test fournissant le 1 PPS mais nous n'avons pas expérimenté avec ce composant.

imprécise pour permettre un traitement additionel tel qu'un asservissement de la fréquence autour d'une valeur de consigne.

Nous nous orientons donc vers l'utilisation d'un résonateur haute fréquence qui fournira une bien meilleure résolution de l'estimation de la fréquence lors d'un décompte au cours d'une seconde du timer. En effet, un compteur connecté à une horloge de fréquence f compte f impulsions en 1 seconde : si l'incertitude Δf du compteur est constante et déterminée par l'architecture matérielle, alors la précision relative de la mesure $\frac{\Delta f}{f}$ sera d'autant meilleure que f est élevée.

2.2 Asservissement d'un oscillateur haute fréquence

La souplesse d'associer chacun des trois oscillateurs (les oscillateurs à quartz basse et haute fréquence ainsi que le circuit RC interne) signifie que nous pouvons jouer avec la source de fréquence ACLK sans par exemple affecter MCLK ou SMCLK qui seront asservis sur l'oscillateur haute fréquence. Ainsi, le passage d'une mesure de ACLK à SMCLK se fait sans modification majeure du logiciel, si ce n'est que de contrôler Timer A par `MOV.w #TASSEL1+TACLR,&TACTL` au lieu de `MOV.w #TASSEL0+TACLR,&TACTL`.

Dans ce cas, pour une fréquence nominale de XT2 de 4 MHz, le nombre d'oscillation par seconde est supérieur à 65536 et nous obtenons, par soustraction des deux valeurs enregistrées successivement par le timer A lors de l'impulsion 1 PPS, la fréquence de l'oscillateur modulo 65536. Étant donné que nous avons un fort *a priori* sur la valeur approximative de résonance du quartz, une telle mesure est suffisante pour ne donner qu'une estimation de la dérive par rapport à la fréquence nominale du résonateur à quartz. Dans le cas qui nous intéresse ici d'un résonateur à fréquence nominale 4,000 MHz ($4 \times 10^6 = 61 \times 65536 + 2304$), nous nous attendons à obtenir des différences entre captures successives du compteur de l'ordre de 2304. Toute dérive par rapport à cette valeur donne directement l'écart par rapport à 4 MHz de l'oscillateur en hertz (puisque le 1 PPS intègre l'erreur sur une seconde) (Fig. 4).

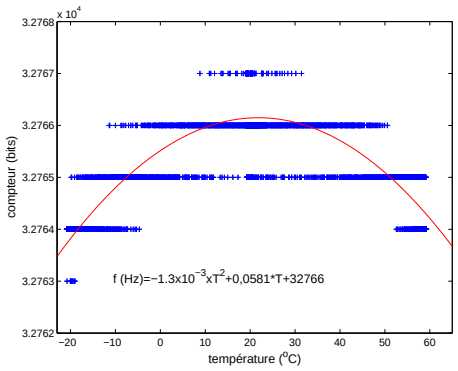


FIG. 3 – Nombre d'impulsions mesurées par le compteur Timer A, synchronisé sur l'oscillateur basse fréquence ACLK, entre deux impulsions 1 PPS du GPS, en fonction de la température. L'ajustement de la parabole montre que la fréquence proche de la température ambiante n'est pas 32768 Hz. La dépendance parabolique de la fréquence avec la température semble convenablement vérifiée et illustre l'excellente stabilité du résonateur à quartz utilisée avec la température, justifiant ainsi l'utilisation de ce matériau dans la réalisation d'oscillateurs stables.

L'asservissement d'un quartz haute fréquence sur le 1 PPS donne lieu à de nombreux développements commerciaux [14, 15]. En effet, la synchronisation de bases de temps distantes est un problème fondamental, qu'il s'agisse de synchroniser les liaisons haut débit asynchrones sur de longues distances [16] ou de mesures synchrones par des capteurs distants (par exemple le projet Auger [7] d'imagerie de cascades de particules générées par l'entrée dans l'atmosphère terrestre de rayons cosmiques

[7] http://www.obs-besancon.fr/auger/tfauger.html

très énergétiques, ou Zeus [8] pour la cartographie d'impacts de foudre). Le quartz fournit une excellente stabilité de la base de temps à court terme mais tend à dériver sur le long terme, dérive qui est alors compensée par une boucle de rétroaction visant à maintenir la fréquence moyenne mesurée sur une longue durée proche de la fréquence définie par un étalon : dans le cas qui nous intéresse ici, cet étalon est le 1 PPS du GPS qui est lui même issu de la moyenne des signaux issus d'horloges atomiques. Cette stratégie est celle utilisée sur tous les dispositifs visant à fournir un signal stable en fréquence : un résonateur haute fréquence fournit la stabilité à court terme tandis que l'asservissement sur un principe physique stable (par définition de la seconde, une transition énergétique de l'atome de césium mesurée de façon optique ou par ses effets magnétiques – méthodes de mesures lentes nécessitant des intégrations de signaux sur plusieurs secondes) fournit la stabilité à long terme. La recherche de phénomènes stables à long terme (rotation des pulsars, transitions d'états atomiques, ...) afin de compléter la stabilité à court terme des oscillateurs à quartz est à la base de la métrologie dite du temps-fréquence.

Nous allons donc désormais aborder la problématique de l'asservissement du quartz sur le 1 PPS – justifiant pleinement l'utilisation du C pour implémenter un algorithme d'asservissement trop complexe pour être écrit en assembleur. Nous proposons d'implémenter le contrôleur le plus simple qui soit, le contrôleur proportionnel-intégrale, nommé désormais PI [17]. Une description détaillée de la théorie sous jacente dépasse largement le cadre de cette présentation : mentionnons simplement que le principe d'un tel asservissement est basé sur la mesure d'une quantité – ici la fréquence d'un oscillateur – et la comparaison de cette valeur avec une consigne. Le contrôle P a pour vocation de générer une action d'autant plus importante que l'erreur entre la mesure et la consigne est grande. Afin de ne pas uniquement dépendre de la dernière valeur mesurée mais de tenir compte de l'historique de l'évolution de la fréquence, la composante I intègre un certain nombre de valeurs et ajoute une composante à l'action, composante d'autant plus importante que l'écart des valeurs passées de la mesure est important par rapport à la consigne. Un certain nombre de propriétés mathématiques de tels contrôleurs peuvent être démontrées théoriquement, et sans entrer dans le détail de ces calculs nous allons ici nous efforcer de présenter une implémentation de ce contrôleur et quelques subtilités expérimentales quand aux choix des paramètres de l'asservissement.

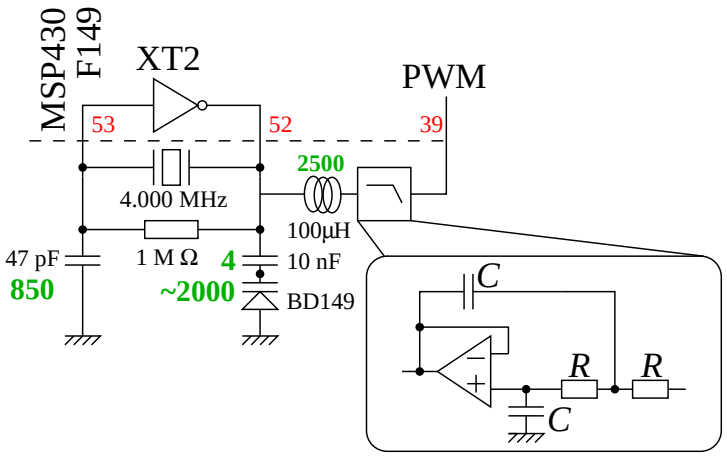
Nous devons identifier :

1. si nous sommes capables d'ajuster la fréquence de sortie par une commande en tension (sortie PWM suivie du filtre passe bas : Figs. 4 et 6)
2. ayant mesuré la plage de correction disponible, estimer la plage des fluctuations de fréquence associées aux variations de l'environnement du quartz (tension d'alimentation de l'oscillateur, température) et déterminer si notre correction sera capable de compenser ces fluctuations (Fig. 8).

La figure 5 présente une mesure rapide de la variation de fréquence du quartz en fonction de la température : nous observons que sur une gamme de -20 à +60 ^{o}C, la fréquence varie d'environ 450 Hz, soit 110 ppm. L'hystérésis de la courbe est associée à un balayage trop rapide de la température : la mesure de température se fait avec la sonde interne au MSP430 soudé sur son circuit imprimé (grande inertie thermique) tandis que la fréquence traduit la température du quartz (faible inertie thermique). Un balayage trop rapide de la température se traduit par une différence entre la température de la sonde et du quartz et donc en une hystérésis. Cette courbe nous donne une idée de la dynamique de correction que nous devons pouvoir fournir pour compenser les effets de température sur l'oscillateur.

La figure 6 présente la variation de fréquence du quartz en fonction de la commande issue du DAC : un compteur commençant à 0 est incrémenté toutes les 20 secondes et sa valeur est placée dans TBCCR3 pour incrémenter le rapport cyclique de la PWM. Nous avons choisi de faire tourner le Timer B de 0 à 0x0FFF pour fournir une sortie définie sur 12 bits. Nous y constatons que la correction permet de faire varier la fréquence de l'oscillateur d'environ 1 kHz, suffisamment pour corriger les fluctuations observées auparavant sur la gamme de températures industrielle.

[8] http://www.zeus.iag.usp.br/index.html



Tension appliquée	Mesure de fréquence (sans R, 27 pF)
0 V	3514 Hz
0,24 V	3642 Hz
0,50 V	3830 Hz
0,75 V	4119 Hz
1,00 V	4358 Hz
1,25 V	4541 Hz
1,40 V	4651 Hz

FIG. 4 – Circuit utilisé pour ajuster la fréquence de l'oscillateur XT2=4 MHz sous le contrôle d'une tension issue de la PWM générée par le port B et filtrée par une un filtre passe bas. Le composant clé du montage est une varicap BD149 dont la capacité varie avec la tension qui y est appliquée. Les valeurs des composants ont été choisies de façon à ce que l'impédance du condensateur empêchant la composante DC de remonter dans l'amplificateur (impédence du condensateur de 10 nF à 4 MHz : 4 Ω) soit négligeable devant l'impédance de l'inductance à cette même fréquence de travail (2500 Ω). Le filtre passe-bas qui convertit la sortie PWM en DAC est composé de condensateurs $C = 100$ nF et de résistances $R = 220$ kΩ autour d'un amplificateur opérationnel OP113 alimenté entre 0 et 5 V (unipolaire), pour une fréquence de coupure $f_c = \frac{1}{2\pi\sqrt{RC}} = 1$ Hz. Le tableau fournit les mesures expérimentales de la fréquence du quartz (modulo 65536) en fonction de la tension appliquée à la varicap : la fréquence de fonctionnement de l'oscillateur peut ainsi facilement être déplacée de près de 1 kHz, suffisamment pour compenser tout dérive thermique et biais lors du fonctionnement du montage.

Dans un premier temps, ayant observé la plage sur laquelle nous pouvons contrôler la fréquence du quartz, nous allons implémenter une fonction C relativement simple, `pid.c`, présentée dans la table 5. Nous y retrouvons le passage de paramètres, le retour d'un paramètre sur 16 bits qui transitera donc par R15, et des fonctions classiques de calcul sur les entiers en C. Deux subtilités sont cependant mises en évidence par cette fonction :

1. le choix des types des entiers. Le MSP430 est équipé de registres de 16 bits : nous essayerons autant que possible de limiter nos calculs à des `short`. Il faut prendre soin de gérer les types signés ou non : nos fonctions en assembleur attendent *a priori* des valeurs non signées, alors qu'un asservissement a toutes les chances de parfois passer dans le négatif et donc de devoir gérer des valeurs signées. Nous prenons donc soin de renvoyer à la fin dans R15 un `unsigned short` alors que les valeurs en entrées sont converties en `short` signés pour les calculs de l'asservissement.
2. certaines fonctions doivent être mémorisées entre deux appels de la fonction C. Une façon considérée comme sale en C serait la création de variables globales, une autre façon étant la création de variables dans le `main` et le passage de paramètres dans les diverses fonctions.

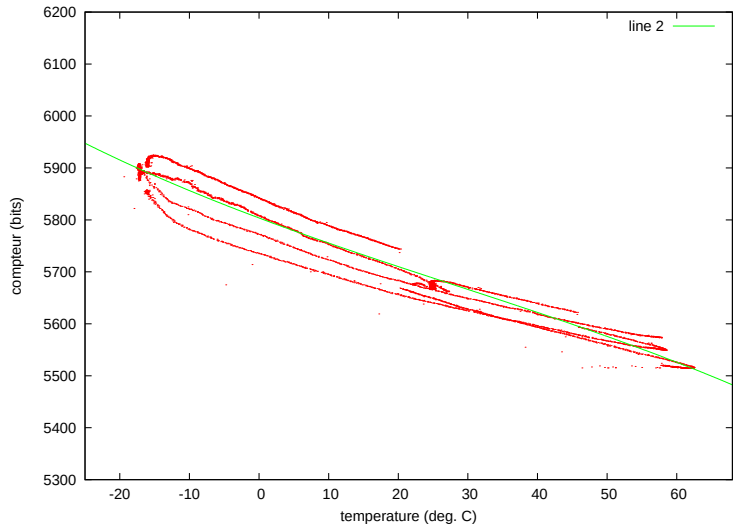


FIG. 5 – Mesure de la variation de fréquence d'un quartz de fréquence nominale 4 MHz en fonction de la température. L'hystérésis est due à un balayage trop rapide de la température entre les phases ascendantes et descendantes de l'enceinte climatique. La courbe verte est un ajustement polynomial de degré 3.

> Étant donné que le contrôle de notre programme se fait dans la partie assembleur, nous allons utiliser la première méthode en convenant d'un certain nombre d'emplacements en RAM – en dehors de la zone allouée à la pile – pour y stocker des variables entre deux appels de la fonction C, variables qui seront ainsi aussi accessibles depuis l'assembleur sans mobiliser de registre. Cette approche est d'autant plus nécessaire que nous devons conserver les valeurs passées de la quantité que nous voulons asservir afin d'implémenter la composante I de l'asservissement. Nous avons pour cela implémenté un tampon rotatif, donc la gestion de l'adresse se fait dans la zone assembleur de façon à ce que le C ne voit qu'une donnée obsolète dans un emplacement en RAM à soustraire de la moyenne, et une nouvelle valeur à ajouter à la moyenne et à mémoriser à chaque mise à jour du contrôleur.

Rappelons ici, pour l'accès en RAM, la syntaxe quelque peu inhabituelle pour le programmeur habitué à travailler sur un processeur équipé d'un coprocesseur de gestion de mémoire (MMU) : ici nous sommes seuls sur notre processeur et avons le droit, en l'absence de contrôle d'un superviseur ou d'un système d'exploitation, d'allouer comme bon nous semble un morceau de RAM par `*(type *)(adresse)=valeur ;`. Sur un système équipé d'une MMU, cette instruction se solderait invariablement par une erreur de segmentation (*segmentation fault*) si `adresse` ne se trouve pas dans le segment de mémoire alloué à notre programme par le superviseur. Nous pouvons nous permettre de telles libertés ici, au détriment de la lisibilité puisque le code C ressemble cruellement à de l'assembleur : même la programmation en C sur microcontrôleur n'affranchit pas d'une lecture détaillée des description du cœur du processeur, et notamment la carte de la mémoire, ou l'organisation, la nomenclature et la taille des registres. À côté de ces contraintes du développement sur des systèmes embarqués, nous avons la liberté décrire des fonctions de haut niveau telle que `DAC-=((somme)/(14*I)) ;` qui se traduit par des appels aux multiples routines associées à l'implémentation logicielle de la division.

Les résultats des mesures sont présentés dans les figures 7 et 8. La première figure illuste les subtilités du choix des coefficients de l'asservissement et surtout le résultat *a priori* remarquable si on ne s'est pas penché sur la théorie sous jacente de la génération d'oscillations de la quantité à asservir avec un code aussi simple que celui de la table 5. Ces résultats expérimentaux présentent divers temps de convergence (ou de divergence) du contrôleur en fonction du choix des coefficients :

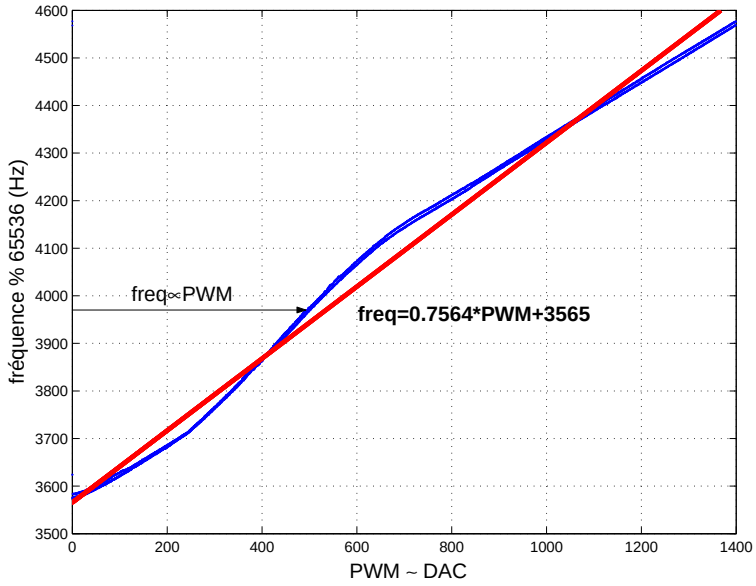


FIG. 6 – Plusieurs mesures de la variation de fréquence d'un quartz de fréquence nominale 4 MHz en fonction de la tension appliquée sur la varicap : l'abscisse correspond à la valeur du compteur TBCCR3 de la PWM, alors que la période de la PWM est de 2^{12}=4096. Nous avons ainsi pu aller de 0 à 1400 (correspondant à un tension de 0 à environ 1,1 V) avant que l'oscillateur ne cesse d'osciller du fait d'un déséquilibre trop important entre la capacité fixe et la varicap. La courbe est bijective et permet donc un asservissement. Noter que la fréquence du quartz est de 61×65536=3997696 Hz auxquels on ajoute l'ordonnée de cette courbe : nous sommes incapables, avec le choix des condensateurs de pieds du résonateur d'atteindre 4,000000 MHz et asservirons notre oscillateur autour de 4001668 Hz. Autour du point de fonctionnement du DAC qui nous intéresse – DAC≃500 – la pente de la relation fréquence-DAC est environ 1.

il s'agit exactement du même principle – ici dans sa version numérique – qui permet à un oscillateur analogique d'osciller lorsque le gain de la boucle compense les pertes dans le résonateur. La seconde figure illustre l'efficacité de l'asservissement de de l'oscillateur et sa stabilité face aux fluctuations de températures qui auraient sinon induit des variations de la fréquence de plusieurs dizaines de hertz. Nous avons ici focalisé notre attention sur la température comme source dominante de perturbation, mais notons que la tension de polarisation du processeur, l'accélération, le champ magnétique, les rayonnements ionisants ou plus généralement l'environnement du résonateur à quartz seront source de fluctuations de fréquence qui seront ici automatiquement compensées par l'asservissement sur le 1 PPS du GPS. Réciproquement, la perte de ce signal de référence induit un comportement bruyant de l'oscillateur qui tente de s'asservir sur un signal instable : ce phénomène est illustré par la déconnexion volontaire pendant quelque jours (milieu du graphique de la Fig 8) de l'antenne GPS afin de rendre le récepteur sourd aux signaux de référence issus des satellites de la constellation GPS.

Cette source de fréquence stable peut désormais être disséminée afin de fournir une source de fréquence commune à des systèmes distant, répondant ainsi à notre objectif initial d'obtenir une référence de temps commune pour des systèmes en vue des satellites GPS mais incapables de communiquer entre eux. La stabilité de ±2 Hz des oscillateurs cadencés près de 4 MHz sur plus d'une semaine nous permet d'espérer de dater des évènements à mieux que la microseconde et d'ainsi atteindre par triangulation une localisation avec une précision de l'ordre du kilomètre.

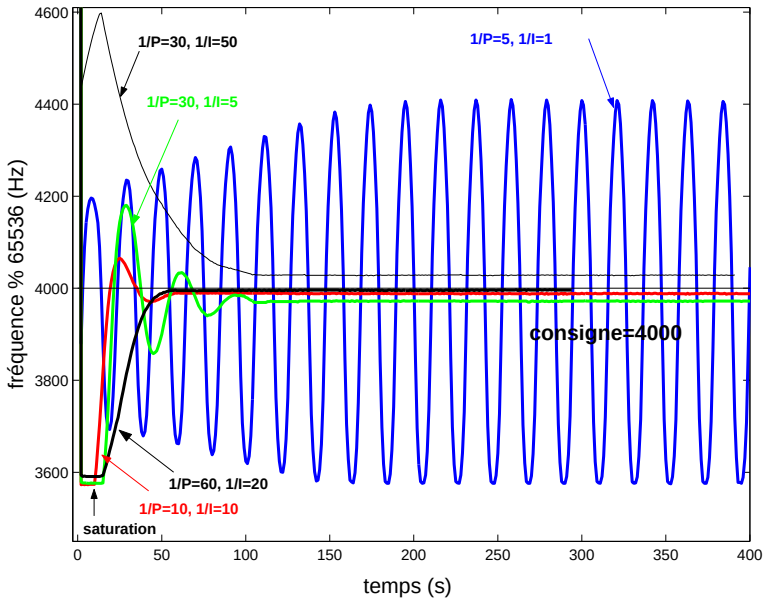


FIG. 7 – Effets sur l'asservissement de la fréquence de l'oscillateur de quelques paramètres P et I, choisis afin d'illustrer l'oscillation du contrôleur lorsque les coefficients de rétroaction sont trop élevés. Le graphique est annoté avec des coefficients P et I cohérents avec la littérature (coefficients multiplicatifs des erreurs) mais qui ne sont par conséquent pas cohérents avec le code source de la table 5 dans lequel l'erreur est *divisée* par P et I afin de n'effectuer que des calculs sur des entiers.

3 Exemple d'utilisation de la librairie mathématique

Le MSP430 ne possède pas d'unité de calcul flottant : toute opération sur des variables décimales devra donc se faire par une émulation logicielle. Nous allons donc exploiter au maximum ici la capacité à programmer en C au moyen de `gcc` en utilisant la librairie `libm` qui fournit ces routines. Ce sera enfin l'occasion de comprendre l'utilité de tant de mémoire sur un processeur dans lequel nous avons déjà bien du mal à injecter 1 KB de code issu d'une programmation en assembleur.

Comme dans l'exemple précédent, nous allons conserver le cœur du programme en assembleur, avec ses initialisations de registres et sa gestion des interruptions que nous maîtrisons correctement. Il devrait *a priori* être possible d'appliquer l'exemple qui suit à un programme entièrement en C, commençant par l'exécution de la fonction `main` après initialisation par `/opt/cdk4msp/msp430/lib/crt430x149.o` qui est automatiquement linké à la compilation tel que le démontre la compilation avec l'option `-v` de `gcc` au moment de linker le fichier ELF. Notre principale préoccupation sera donc de valider le passage de paramètre entre l'assembleur et le C et de vérifier le résultat d'un calcul non-trivial nécessitant des calculs sur des valeurs décimales.

Nous nous proposons de calculer sur le MSP430 l'heure de lever et de coucher du soleil à toute date de l'année. Cet exemple complète donc l'implémentation d'une horloge temps-réel vue auparavant, et nous verrons qu'un tel calcul nécessite un grand nombre d'appel aux fonctions trigonométriques que nous implémenterons "bêtement" par appel à la `libm` (plutôt qu'en programmant une table de correspondance – LUT – comme on le ferait classiquement en "vrai" assembleur, solution gourmande en mémoire si nous travaillons sur des nombres codés sur 16 bits). Une fois ce calcul validé, il nous sera par exemple possible de programmer le déclenchement d'appareils photo numériques non pas à horaires prédéfinis, mais à des dates équidistantes du lever et du coucher du soleil et donc avec des positions du soleil proches d'un jour à l'autre.

L'algorithme que nous allons implémenter est issu de [18] : il déduit de la position du soleil à une date connue (01/01/2000), la position à toute date antérieure ou postérieure à cette référence.

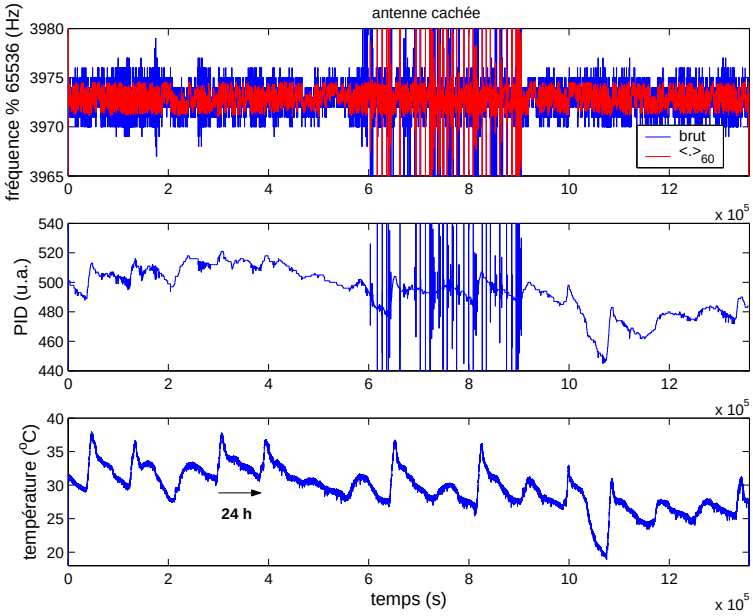


FIG. 8 – Résultat de l'asservissement d'un oscillateur de fréquence nominale 4 MHz sur le 1 PPS issu d'un récepteur Motorola Oncore VP pendant plus d'une semaine. Noter que sur une semaine, la fréquence (courbe du haut) n'a bougé que de ± 2 Hz alors que la température a varié de plus de 10 ^{o}C (courbe du bas). La correction (courbe du milieu) a bien agit puisque la commande de la PID (*i.e.* la valeur fournie à la PWM) reproduit bien les fluctuations de température – principale cause de fluctuation de fréquence de l'oscillateur dans ce montage. La PID a agit sur une plage de 40 unités, soit d'après la figure 6, a compensé une variation de fréquence de 40 Hz autour de la fréquence nominale. Ce résultat est cohérent avec la figure 5 qui annonce une dérive d'environ 320 Hz sur 80^{o}C soit 4 Hz/^{o}C. L'écart type de la fréquence sur 1 semaine d'enregistrement est de 7 Hz sur les données brutes issues directement des acquisitions espacées de 1 seconde, et de 0,8 Hz après application d'une moyenne glissante sur 60 secondes. Le niveau de bruit élevé au milieu de la courbe est dû à la déconnexion volontaire de l'antenne GPS : alors qu'un asservissement efficace permet de corriger les fluctuations de l'oscillateur, la perte du signal de référence dégrade considérablement les performances de l'oscillateur.

Il nous faudra donc calculer le nombre de jours entre cette référence et la date qui nous intéresse. Connaissant la latitude et la longitude de l'observateur, l'algorithme en déduit les horaires de lever et de coucher du soleil. Nous ne voulons connaître ces quantités qu'à la minute près (une précision plus importante nécessite de déterminer des conditions environnementales de la position de l'observateur et réduit la généralité de notre approche), donc renvoyer au programme assembleur 4 valeurs comprises entre 0 et 24 ou 0 et 60, soit 4 valeurs codées sur 8 bits. Ces 4 valeurs seront enfin transmises par RS232 pour comparaison avec le calcul effectué sur un PC et les valeurs tabulées issues du web (`http://aa.usno.navy.mil/data/docs/RS_OneYear.html`).

Le choix de la position de la pile pose déjà un problème. Alors qu'un programmeur humain n'imbriquera que quelques appels à des procédures et empilera quelques variables, ces hypothèses ne sont plus valables lors de l'utilisation intensive d'un compilateur de langage de haut niveau tel que le C. Les 0x80 octets que nous nous étions réservés auparavant sont insuffisant pour le calcul qui nous intéresse : nous devons initialiser la pile à une valeur plus élevée pour que le calcul aboutisse. Nous aovns choisi "arbitrairement" `MOV.W #0x880,R1` afin d'avoir 0x680 octets disponibles sur la pile et garder un peu de RAM pour notre propre usage (0x80 octets sont insuffisants). Rappelons que l'utilisation de la pile `décrémente` R1 qui doit être supérieur à 0x200 (adresse minimum de la RAM). Le choix de la taille de la pile lors de l'utilisation d'un compilateur

est un problème complexe qui ne peut être validé que par le test de tous les cas possibles d'exécution du programme. Nous avons constaté dans notre cas que l'exécution du programme pour le calcul sur 2 années consécutives s'achève convenablement : nous en déduisons que cette taille de pile est suffisante.

La boucle principale du programme assembleur consiste alors à donner soit une date sous la forme jour, mois, année, soit à fournir un nombre de jours depuis le 01/01/2007 : nous avons utilisé cette seconde option qui se rapproche le plus d'une utilisation en combinaison avec une horloge temps réel. Ces trois paramètres sont passés au programme C dans les registres R15 à R13 selon la convention de passage de paramètre aux fonctions C que nous avons décrit auparavant (section 1.6). Nous récupérerons nos 4 octets dans une variable de type `unsigned long` dans laquelle nous aurons concaténé les minutes et heures de lever et coucher du soleil. La convention de `msp-gcc` définit le passage de valeurs sur 32 bits par R15 :R14 (deux registres 16 bits successifs) : c'est donc dans ces deux registres que nous récupérerons les quantités qui nous intéressent. Nous faisons l'hypothèse que le programme C perturbera tous nos registres, que nous prenons donc soin d'empiler avant l'appel à la fonction C `unsigned long sunriseset()` afin de les dépiler après exécution du programme. Nous validons toutes ces informations issues de la documentation de `msp-gcc` [5] par une étude du code assembleur issu du désassemblage du fichier ELF généré par la compilation de notre application. Sans savoir comment fonctionne la fonction de calcul des horaires, en ne connaissant que le prototype de la fonction `unsigned long sunriseset(unsigned short year,unsigned short month,unsigned short day)`, nous vérifions par `msp430-objdump -dSt sunriseset > sunriseset.lst` (avec `sunriseset` le fichier ELF issu de la compilation de `sunriseset.c` et `sunriseset.S`) que

- `msp-gcc` s'est chargé d'empiler les registre qui ne lui servent pas au passage de paramètres, les protégeant ainsi de changements lors de l'exécution des fonctions C et restituant le microcontrôleur au programme assembleur dans un état connu
- les premières lignes de la fonction `sunriseset` font bien appel aux registres R15 à R12 : sous réserve de ne pas s'être trompé dans l'ordre des paramètres, il est probable que le passage se fasse convenablement
- les dernières lignes consistent dans un premier temps à la définition des contenus de R15 et R14 puis au dépilement des registres qui avaient été protégés par empilement lors de l'appel à la fonction C. Il est donc probable que nous récupérions des résultats pertinents du calcul.

Cette rapide analyse peut paraître triviale au lecteur qui reçoit ici le résultat d'un long travail de debuggage, mais le dernier point nous a notamment permis d'identifier diverses erreurs de taille de variable et d'assignation des paramètres retournés. Ainsi, l'utilisation du type `int` est à proscrire au profit des types `short` (16 bits) ou `long` (32 bits). En effet, la taille de l'`int` change selon la plateforme (32 bits sur Intel x86, 16 bits sur MSP430) et résulte en une erreur de passage de paramètre si nous désirons concaténer 4 valeurs sur 8 bits dans une unique variable retournée par la fonction `sunriseset`.

Du point de vue du C, tous nos calculs se font en `float`. Les paramètres de date (jour, mois, année) sont fournis à la fonction qui en déduit le nombre de jours depuis une date de référence à laquelle la position du soleil est connue et précalculée, puis un certain nombre de calculs flottants sont effectués impliquant des fonctions trigonométriques aussi variées que le sinus, consinus, et tangente ainsi que leurs inverses. Finalement, les 4 variables sur 8 bits sont concaténées en un `unsigned long` par

```
unsigned long res ;
...
res=(unsigned long)(rintf(floorf(GSTr)))*256*256*256+(unsigned long)
(rintf((GSTr-floor(GSTr))*60.))*256*256+(unsigned long)(floor(GSTs))*256+(unsigned
long)rintf((GSTs-floor(GSTs))*60.) ;
```

En effet, toutes les heures résultant du calcul sont fournies sous forme d'heures décimales, donc nous déduisons les composantes entières heures et minute pour concaténation.

Le transfert de ce programme vers le MSP430 nous permet d'appréhender la complexité associée à l'utilisation de `libm` : le code nécessite la programmation de plus de 27 kB de mémoire flash. L'exécution du programme réserve une autre surprise : alors que nous constatons que le

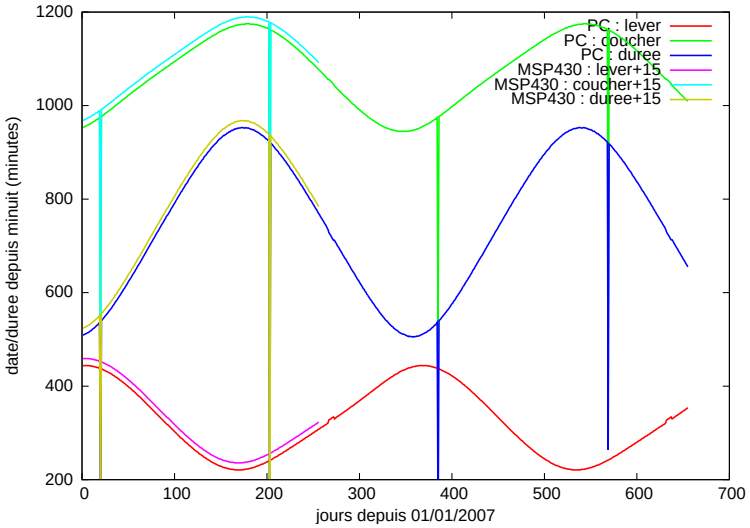


FIG. 9 – Calcul des horaires de lever et de coucher du soleil puis, par différence, de la durée du jour, en fonction du numéro du jour depuis le 1er janvier 2007. Le calcul sur PC a été effectué sur une durée de 655 afin de fournir une courbe sur plus d'un an, tandis que le calcul sur MSP430 a été fait sur 255 jours. Nous constatons que les deux séries de calcul – sur PC et MSP430 – fournissent les mêmes résultats. Les coubes issues du MSP430 ont été décalées de 15 minutes vers le haut par souci de clareté de la représentation graphique. Nous constatons que quelques cas particuliers induisent des résultats erronés pour des raisons que nous n'avons pas cherché à identifier. Pour les autres cas, le résultat est en accord à la minute près avec les données issues de `http ://aa.usno.navy.mil/data/docs/RS_OneYear.html`.

résultat de ce calcul est correct (Fig. 9), c'est surtout la durée de l'exécution qui nous rappelle la puissance démesurée des PC actuellement disponibles. Le calcul des horaires de lever et de coucher du soleil sur 255 jours consécutifs prend 5 minutes 24 secondes sur MSP430 (pour moins de 30 ms sur un Pentium MMX cadencé à 166 MHz). Au delà de la différence de fréquence d'horloge des processeurs, c'est avant tout l'implémentation logicielle du calcul flottant qui réduit les performances. Malgré tout, ce calcul ne prendrait donc que 1,27 seconde chaque jour dans une implémentation aux côtés d'une horloge temps réel. Pour un processeur consommant 300 μA sous 3,3V, ce calcul consomme donc 1,3 mJ, soit l'énergie générée et emmagasinée par la chute d'une bille de 100 g d'une hauteur de 1 cm environ avec un rendement de 10 % : l'opérateur d'une telle horloge devra donc souvent revenir remonter sa montre pour qu'elle continue a fonctionner ... [9] Une telle valeur n'est cependant pas en contradiction grossière avec les antiques pendules : dans notre cas, le poids placé à 1,8 m du sol devrait être remonté tous les 6 mois.

4 Programmation de la mémoire flash

Un dernier point qui vaut la peine d'être mentionné est la capacité du MSP430 à conserver des paramètres en mémoire non-volatile (flash) inscrits par un programme en cours d'exécution. Une telle fonctionnalité permet par exemple de conserver des coefficients de calibrage (par exemple de la sonde de température interne après calibrage de chaque composant individuel) ou les horaires d'alarme dans le cas de l'horloge temps réel.

[9] La variation d'énergie potentielle ΔE issue de la chute d'une bille de masse $m = 0,1$ kg sur une hauteur Δh dans un champ de gravitation $g = 9,8$ m.s^{-2} est $\Delta E = m \times g \times \Delta h$ soit avec un rendement de 10 %, $\Delta h = 10 \times \frac{\Delta E}{mg}$, qu'on résoud avec notre objectif d'obtenir $\Delta E = 1,3 \times 10^{-3}$ J.

La mémoire flash est segmentée en bloc : tandis qu'un mot peut être écrit individuellement, la réinitialisation de la mémoire nécessaire au préalable de chaque nouvelle écriture se fait nécessairement par bloc.

Les méthodes d'effacement et d'écriture en flash depuis un programme sont décrites dans la note d'application SLAA103 de Texas Instruments (http://www.gaw.ru/pdf/TI/app/msp430/slaa103.pdf). Dans un exemple qui nous intéressait, nous désirions conserver en mémoire non volatile quelque paramètres définis par l'utilisateur suite à une transmission par RS232. Nous commençons par effacer le bloc mémoire qui nous intéresse :

```
flash_rmw:
  mov.w   #0x0A500,&FCTL3
  mov.w   #0x0A502,&FCTL1
  mov.w   #00,&0x1000      ; erase flash segment B 1000-107F
  mov.w   #0x0A500,&FCTL1
  mov.w   #0x0A510,&FCTL3
  ret
```

Cette étape préalable est nécessaire car l'écriture en flash ne permet *que* de convertir des 1 en 0 : il nous faut donc partir d'une mémoire totalement initialisée à 0xFF pour être certain d'y stocker la valeur voulue [6, chap.10]. L'écriture de la valeur contenue dans le registre `tmp` à l'emplacement `tmp2` contenu dans l'espace mémoire qui vient d'être intialisé (0x1000-0x107F pour le segment B par exemple) se fait par

```
flash_cpw:
  mov.w   #0x0A500,&FCTL3
  mov.w   #0x0A540,&FCTL1
  mov.w   tmp,0(tmp2)            ; write_word at flash @
  mov.w   #0x0A500,&FCTL1
  mov.w   #0x0A510,&FCTL3
  ret
```

Mieux vaut éviter d'utiliser ces espaces mémoires pour des écritures répétées automatiques car le support flash est annoncé avec une durée de vie limitée de quelque dizaines de milliers d'écritures.

5 Conclusion

Nous avons présenté le microcontrôleur MSP430 de Texas Instruments, un composant spécialement dédié aux applications faible consommation – de l'ordre de quelques centaines de microampères – fonctionnant sur piles. Nous avons présenté les outils de développement en assembleur et C sous GNU/Linux. Les applications proposées se sont focalisées sur l'implémentation d'une horloge temps réel, avec une consommation au moins aussi faible que les composant dédiés disponibles commercialement, mais ajoutant la souplesse de l'implémentation logicielle et donc la capacité à ajuster le code aux besoins de l'application sans ajouter de composants additionels (eux même gourmands en énergie). Nous avons finalement estimé l'écart entre notre implémentation de l'horloge et la seconde "exacte" en utilisant pour référence le signal à 1 Hz issu d'un récepteur GPS. Nous avons complété cette étude par une présentation de l'utilisation du calcul flottant qui occupe une bonne partie des ressources du microcontrôleur, tant en occupation mémoire qu'en puissance de calcul.

Ce projet ne prétend pas répondre à toutes les questions sur l'électronique embarquée ou la distribution d'une source de temps précise, mais au contraire fournit quelques pistes qui mériteraient d'être approfondies pour que ces systèmes soient utilisables en pratique. Les perspectives de ce travail visent désormais à compléter la gamme des outils utilisables sur le MSP430 par un système d'exploitation. Nous avons vu que l'utilisation du C facilite certaines tâches au détriment de l'occupation de la mémoire et des performances (`libm` par exemple) : il est probable que les mêmes conclusions puissent être tirées avec l'ajout d'une couche additionnelle telle que TinyOS (http://www.tinyos.net/).

6 Remerciements

Texas Instruments a gracieusement fourni quelques MSP430 sous forme d'échantillons pour nos développements.

JMF remercie F. Lardet-Vieudrin pour les discussions sur les méthodes d'ajustement de la fréquence d'un oscillateur à quartz et l'accès aux composants associés. Ce travail n'aurait pu aboutir sans les discussions avec les collègues des équipes Temps-Fréquence du Laboratoire de Physique et Métrologie des Oscillateurs (FEMTO-ST/CNRS, Besançon) et de l'Observatoire de Besançon. AM et FB sont étudiants en master ELO à l'Université de Franche-Comté à Besançon.

L'ensemble des codes sources et données présentées dans ce document sont disponibles à http://jmfriedt.free.fr.

Références

[1] D. Stapes & T. Brandes, *The MSP430x1xx Basic Clock System – Texas Instruments Application Report SLAA081*, Juin 2000, disponible à http://www.gaw.ru/pdf/TI/app/msp430/slaa081.pdf

[2] B. Lane, *MSP430 Development with Linux*, Linux Journal (2006), disponible à http://www.linuxjournal.com/article/8682

[3] *MSP430x1xx User's Guide – SLAU049F*, disponible à http://focus.ti.com/lit/ug/slau049f/slau049f.pdf

[4] M. Mitchell, *Implementing a Real-Time Clock on the MSP430 – Application Report SLAA076A*, Janvier 2001, disponible à http://focus.ti.com/lit/an/slaa076a/slaa076a.pdf.

[5] S. Underwood, *mspgcc – A port of the GNU tools to the Texas Instruments MSP430 microcontrollers* (2003), disponible à http://sourceforge.net/project/showfiles.php?group_id=42303

[6] C. Nagy, *Embedded systems design using the TI MSP430 series*, Newnes (2003) est en grande partie une copie des datasheets, mais a le bon goût de fournir des exemples de codes fonctionnels

[7] M. Mitchell, *Using PWM Timer_B as a DAC – Application Reponse SLAA116*, Décembre 2000, disponible à http://focus.ti.com/lit/an/slaa116/slaa116.pdf

[8] P. Horowitz & W. Hill, *The Art of Electronics, 2nd. Ed*, Cambridge University Press (1989), p.274

[9] J. Karki, *Active Low-Pass Filter Design*, Texas Instruments Application Report SLOA049 (Octobre 2000), disponible à www.science.unitn.it/~bassi/Signal/TInotes/sloa049.pdf

[10] W. Lewandowski, P. Moussay, P. Guerin, F. Meyer & M. Vincent, *Testing the Oncore GPS receiver for time metrology*, 11th European and Time Forum (EFTF) (1997), pp.493-497

[11] O.E. Rudnev, Y.S. Shmaliy, E.G. Sokolinskiy, A.Y. Shmaliy & O.I. Kharchenko, *Kalman filtering of a frequency instability based on Motorola Oncore UT GPS timing signals*, 1999 Joint Meeting EFTF-IEEE IFCS (1999), pp. 251-258

[12] http://www.rt66.com/~shera/index_fs.htm

[13] http://www.tapr.org/pdf/GT_Eng_Notes.pdf et ftp://ftp.tapr.org/gps/motorola//oncore.eng.notes.pdf

[14] http://www.temex.com/var_prod/gallery/documents/SYNC_Applications_Notes/TV_Broadcast_needs_Precision_Frequency_Offset_TF12.pdf

[15] http://www.temex.com/var_prod/gallery/documents/SYNC_Applications_Notes/TETRA%20Synchronization_TF15.pdf

[16] M.A. Lombardi, C. Norman & W.J. Walsh, *The role of LORAN Timing in Telecommunications*, disponible à www.loran.org/news/Loran%20NIST%20RTCM%20Paper%202006.pdf

[17] Y. Granjon, *Automatique*, Dunod (2001) chap. 7 et 8, ou http://www.embedded.com/2000/0010/0010feat3.htm

[18] P. Duffet-Smith, *Practical Astronomy with your Calculator, 3rd Ed.*, Cambridge University Press (1988)

```
#include <signal.h>
#include <io.h>

#define asservit
#define  nbbalayage 4
#define  tabdebut 0x600 //start du tableau stockant
#define  tabfin 0x61c //stop du tableau -> taille 0x1c=14
                                        // vals (on sauve des short).
#define         tmp     R15 // <- 1er param de PID
#define  addr R14 // RAM=0x200-0x9FF <- 2eme param de PID
#define         DAC     R13 //  polar varicap <- 3eme param de PID
#define         CPT     R10

.global main            ;-------------------------------------------------------
main: MOV.W   #0x280,R1 ; stack = 0x200-0x280
        CALL    #Setup

mov.w #0,CPT
mov.w #0,DAC
mov #0x600,addr ; place le tableau dans la RAM A l'adresse 0x600
mov.w #0,&0x300
mov.w #0,&0x2F8

zero:  cmp #tabfin,addr ; init tableau a 0
jeq finzero
mov.w #0,@addr
inc.w addr ; inc 2 fois car on est en short (2 octets)
inc.w addr
jmp zero
finzero:mov #0x600,addr

Mainloop:
mov.b #0,tmp
at:     CMP.b   #1,tmp ; NE PAS se mettre en sleep qui coupe osc
jne at ; interruption nous sort de la boucle
// xor.b #4,&P1OUT  ; A VIRER POUR ETRE STABLE (pas de LED) !

// Creation du tableau rotatif qui contient les valeurs du timer A.
inc.b CPT
cmp #tabfin,addr ; si on est en fin de tablo
jne finadr ; alors on deplace l'element courant en
mov @addr,tmp ; ... debut de tablo et on y va
mov #tabdebut,addr
mov tmp,@addr
finadr: inc.w addr ; inc 2 fois car on est en short (2 octets)
inc.w addr

mov addr,tmp ; envoi adresses de stockage de timer A
call  #rs_TX
swpb tmp
call  #rs_TX

// Balayage en frequence en fonction de la valeur du DAC.
lcpt:   CMP.b   #nbbalayage,CPT ; fait nbbalayage acqusitions par valeur de DAC
jne suitecpt
mov.b #0,CPT

#ifndef asservit
add.w #0x01,DAC // incremente de 1 la valeur du DAC
cmp.w #1400,DAC // valeur limite du DAC (sinon pas d'osc)
jne dacok
mov.w #0x00,DAC // remise a zero de la valeur du DAC
dacok: mov.w DAC,&TBCCR3 ; inc DAC
#endif                          // fin de la boucle de calibration

suitecpt:mov.w &CCR0,tmp ; read timer A input capture current value
boucle: call  #rs_TX  // envoi sur port com la val du timer A
swpb tmp
call  #rs_TX
swpb tmp

#ifdef asservit
push R15
push R14 ; on veut se rappeler de la position du tablo
; push R13 ; DAC va etre remplace par PID.c
push R12 ; R11 et R12 seront modifies pas PID.c
push R11
CALL    #pid
mov.w tmp,DAC ; return() dans R15
call  #rs_TX ; envoi de la commande sur port com.
swpb tmp
call  #rs_TX

mov.w DAC,&TBCCR3  ; commande -> PWM ie DAC
pop R11
pop R12
; pop R13 ; pas de push donc pas de pop
pop R14
pop R15

//Envoi de somme de pid sur port com :
mov.w &0x300,tmp
call  #rs_TX
swpb tmp
call  #rs_TX
#else
mov.w DAC,tmp
call  #rs_TX
swpb tmp
call  #rs_TX
swpb tmp
call  #rs_TX
swpb tmp
call  #rs_TX
swpb tmp
#endif

; a chaque PPS, on renvoie la temperature du MSP430 => courbe f(T)
call #SetupADC12
call  #rs_TX
swpb tmp
call  #rs_TX

        JMP     Mainloop

//////// send tmp on UART0 ////////
rs_TX: bit.b #UTXIFG0,&IFG1 ; p.13-29: UTXIFG0 set when U0TXBUF empty
jz rs_TX
mov.b tmp,&TXBUF0
ret

;-------------------------------------------------------------------------------
;  Setup: Configure Modules and Control Registers
;-------------------------------------------------------------------------------
Setup:          MOV.w   #WDTPW+WDTHOLD,&WDTCTL  ; Stop Watchdog Timer

// Timer A (32 kHz) capture a chaque seconde le contenu de l'horloge
//setupTA:      MOV     #TASSEL0+TACLR, &TACTL  ; ACLK for Timer_A.
setupTA:        MOV.w   #TASSEL1+TACLR, &TACTL  ; SMCLK for Timer_A.
                BIS #MC1,&TACTL         ; start TA in "up to FFFF" mode
mov.w #CM0+CAP+CCIE+SCS,&TACCTL0 ; capture front montant
// jmfriedt: /opt/cdk4msp/msp430/include/msp430/timera.h
MOV #0x14,&P1OUT
MOV #0x12,&P1SEL ; p.9-4: P1_1 is TA, NOT GPIO
MOV #0x14,&P1DIR
                EINT                            ; Enable interrupts

  call #SetupUART0

setupTB: bis.b #0x08,&P4SEL ; TB3 for PWM (pin 39)
bis.b #0x08,&P4DIR ; TB3 for output
MOV.w   #TBSSEL1+TBCLR, &TBCTL  ; SMCLK for Timer_B.
mov.w #CCIE,&TBCCTL0 ; compare mode
mov.w #0xFFF,&TBCCR0 ; period of PWM: 8 bits DAC
mov.w #0x02E0,&TBCCTL3 // Capture/Compare Control Reg
mov.w #0x30,&TBCCR3 ; Vout initial
bis #MC0,&TBCTL

// a faire en dernier, apres que tous les ports soient OK (TBCCR3 initialized)
call #SetupOsc
                RET ; Done with setup.

////////////////////////////////////////////////////////////////////////////
SetupADC12: ...
SetupUART0: ...
////////////////////////////////////////////////////////////////////////////
SetupOsc:  bic.b  #XT2OFF,&BCSCTL1  ; XT2on
bic.b  #OFIFG,&IFG1  ; Clear OSC fault flag
mov  #0xFF,R15  ; R15 = Delay
SetupOsc1:  dec  R15  ; Additional delay ensure start
jnz  SetupOsc1  ;
bit.b  #OFIFG,&IFG1  ; OSC fault flag set?
jnz  SetupOsc  ; OSC Fault, clear flag again
bis.b  #SELM0+SELS,&BCSCTL2  ; MCLK = SMCLK = XT2      ?SELS?
ret

; l'exemple TI qui definit une table fixe et se compile sans entete est mauvais
interrupt(TIMERA0_VECTOR)               ; register interrupt vector
.global CCR0INT
CCR0INT:  mov.b #1,tmp ; set flag
                RETI

interrupt(TIMERB0_VECTOR)
.global CCIFG0
CCIFG0:  RETI ; leave CCR3 unchanged for DAC
```

TAB. 4 – Exemple issu de l'exemple d'horloge temps réel précédent, modifié afin de déclencher le compteur sur le quartz haute fréquence, capturer les dates de déclenchement du 1 PPS du GPS et générer un signal de contrôle issu d'une PID pour générer une PWM utilisée comme DAC pour asservir le quartz haute fréquence sur le 1 PPS. La fonction d'asservissement `pid.c` est décrite ailleurs (Tab. 5). Nous en avons éliminé les fonctions `SetupUART0` et `SetupADC12` présentées plus tôt dans ce texte.

```
#include "hardware.h"
#include <stdio.h>

#define consigne 4000
#define P       30
#define I       5
#define addre   0x300

unsigned short pid(unsigned short vc,unsigned short pos,unsigned short DAC)
// R15=vc, R14=pos, R13=DAC...
{
    signed short vp;
    signed short interval;
    signed short erreur;
    signed short somme;

    vp=*(unsigned short *)(0x2F8);      // old val compteur
    *(unsigned short *)(0x2F8)=vc;

    erreur=vc-vp-consigne;

    somme=*(unsigned short*)(addre);    // tableau de l'integrale
    somme-=*(unsigned short *)(pos);
    *(unsigned short*)(pos)=erreur;
    somme+=erreur;
    *(unsigned short*)(addre)=somme;

    DAC-=erreur/P;
    DAC-=((somme)/(14*I));              // 14 elements dans tableau

    if (DAC > 0x8000) DAC = 0x00;       // DAC<0
    if (DAC > 1400) DAC = 1400;

    return(DAC);
}
```

Tab. 5 – La fonction d'asservissement PID qui génère la commande de PWM à partir des valeurs successives des dates de détection du signal 1 PPS du GPS. Le choix des types des variables est fondamental pour à la fois permettre la gestion de valeurs négatives du contrôleur tout en passant des arguments compatibles avec la PWM (valeur entre 0 et 1400 afin de garantir l'oscillation de XT2).

```
#include <signal.h>
#include <io.h>

#define         an      R15
#define  mois    R14
#define         jour    R13
#define         tmp     R10

;-------------------------------------------------------------------------
;               Program main
;-------------------------------------------------------------------------
.global main
main:
RESET:
MOV.W   #0x880,R1 ; stack = 0x200-0x280 // NE MARCHE PAS AVEC 280
        CALL    #Setup

mov.w #2007,an
mov.w #0,mois
mov.w #0,jour

Mainloop:
mov an,tmp ; return() dans R15:R14 (unsigned long)
call  #rs_TX ; envoi sur port com.
mov mois,tmp ; return() dans R15:R14 (unsigned long)
call  #rs_TX
mov jour,tmp
call  #rs_TX

push R15
push R14 ; on veut se rappeler de la position du tablo
push R13 ; DAC va etre remplace par PID.c
CALL    #sunriseset
mov an,tmp ; return() dans R15:R14 (unsigned long)
call  #rs_TX ; envoi sur port com.
swpb tmp
call  #rs_TX
mov mois,tmp
call  #rs_TX
swpb tmp
call  #rs_TX

pop R13
pop R14
pop R15
inc.b jour
jne Mainloop
fin:    JMP     fin

//////// send tmp on UART0 ////////
rs_TX: bit.b #UTXIFG1,&IFG2 ; p.13-29: UTXIFG0 set when U0TXBUF empty
jz rs_TX
mov.b tmp,&TXBUF1
ret

;-------------------------------------------------------------------------
;  Setup: Configure Modules and Control Registers
;-------------------------------------------------------------------------
Setup:          MOV.w   #WDTPW+WDTHOLD,&WDTCTL  ; Stop Watchdog Timer

MOV #0x14,&P1OUT
MOV #0x12,&P1SEL ;
MOV #0x14,&P1DIR
                EINT                            ; Enable interrupts

  call #SetupUART0
call  #SetupUART1
call #SetupP3

                RET ; Done with setup.


; ***************** UART 0 Initialisation *************************
; nb: SMCLK has been divided by 4, must compensate for divider register

SetupUART0:  mov.b  #SWRST,&UCTL0 ; cf p.13-4
bis.b  #CHAR,&UCTL0  ; 8-bit characters: 13-22
mov.b  #SSEL0,&UTCTL0  ; UCLK = ACLK @ 32768 Hz
mov.b  #0x03,&UBR00  ;
mov.b  #0x00,&UBR10  ; cf p.13-16: 9600 bauds +/- 15%
mov.b  #0x4A,&UMCTL0  ;
bic.b  #SWRST,&UCTL0
bis.b  #UTXE0+URXE0,&ME1  ; Enable USART0 TXD/RXD
nop
ret

; ***************** Port 3 Initialisation *************************
SetupP3:  bis.b  #0xF0,&P3SEL  ; P3.4,5,6,7 = USART select
bis.b  #0x50,&P3DIR  ; P3.4,6 = output direction
ret ;

;-------------------------------------------------------------------------
; ***************** UART 1 Initialisation
; nb: SMCLK has been divided by 4, must
SetupUART1:  mov.b #SWRST,&UCTL1 ; cf p.13-4
  bis.b #CHAR,&UCTL1
mov.b #SSEL0,&UTCTL1
mov.b #0x03,&UBR01
mov.b #0x00,&UBR11
mov.b #0x4A,&UMCTL1
bic.b #SWRST,&UCTL1
bis.b #UTXE1+URXE1,&ME2
ret
```

Tab. 6 – Programme assembleur pour MSP430 chargé de générer les paramètres d'appel à la fonction C et d'initialiser les périphériques de communication pour fournir les résultats par RS232. Ce programme peut ainsi compléter l'horloge temps réel vue auparavant afin de déclencher un évènement à élévation du soleil connue plutôt qu'à une heure prédéterminée.

```
// passer de double a double ne change pas le resultat
// Practical Astronomy with your Calculator
// 3rd Ed, Peter Duffet-Smith, Cambridge Univ Press (1988)

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define DEBUG {}
//#define printf {}

#define epsilon_g (279.403303) // degs
#define omega_g (282.768422)   // degs
#define ecc (0.016713)
#define epsilon (23.43929)     // epoch 2000

#define mylat (+47.20)    // Besancon, France
#define mylon (+06.00)
//#define mylat (+42.37)  // Boston, MA
//#define mylon (-71.05)

// ATTENTION : int = 16 bits sur MSP430
double jours(short an,short mois,short jours)
{                                       // mois=0 si on a deja calcule les jours
 short jmois[12]={0,31,59,90,120,151,181,212,243,273,304,334};
 short i=0,cpt=0;
 for (i=1990;i<an;i++) {cpt+=365;if ((i%4==0)&&(i!=2000)) cpt+=1;}
 if (mois!=0) {
    cpt+=jmois[mois-1];
    if ((i%4==0)&&(i!=2000)&&(mois>1)) cpt++;
    }
 cpt+=jours;
 return((double)(cpt));
 }

void borne24(double *machin)
{while (*machin>24.) *machin-=24.;
 while (*machin<0.)  *machin+=24.;
}

void borne(double *machin)
{while (*machin>360.) *machin-=360.;
 while (*machin<0.)   *machin+=360.;
}

void ecliptic_to_equatorial(double* lambda,double *delta)
{double sin_delta,x,y;
 *lambda*=(M_PI/180.);
 sin_delta=sin(epsilon*M_PI/180.)*sin(*lambda);
 *delta=asin(sin_delta)*180./M_PI;
 y=sin(*lambda)*cos(epsilon*M_PI/180.);
 x=cos(*lambda);
 *lambda=atan(y/x)*180./M_PI;  // alpha deg
 if (x<0.) *lambda+=180; borne(lambda);  // chap. 27 p.41
 DEBUG("x=%f y=%f => alpha=%f\n",x,y,*lambda);
}

unsigned long sunriseset(unsigned short year,unsigned short month,unsigned short day)
// fournit UTCsunset et UTSsunrise pour D=date (jours depuis 010190)
{double S,T,T0,D,N,Msol,Ec,lsol,lsolp,beta,betap,GSTr1,GSTs1,GSTr2,GSTs2,H,GSTs,GSTr,T0p;
 unsigned long res;
 D=jours(year,month,day); // 17 aout 2007 = 6437
 DEBUG("jours=%5.0f since 01/01/1990\n",D);
 //D=-1461.+69.;// exemple p.95: alpha=23.332801 delta=-4.309187  10 March 1986
 //D=-522.;     // exemple p.91:       8h26'4"         19deg12'42"
 //D=-3444.;    // exemple p.88:       8h25'46"        19deg13'48"
 N=360./365.242191*D;
 Msol=N+epsilon_g-omega_g;borne(&Msol);
 Ec=360./M_PI*ecc*sin(Msol/180.*M_PI);
 lsol=N+Ec+epsilon_g;     borne(&lsol);
 lsolp=lsol+0.985647;
 DEBUG("lsol=%f\n",lsol);
 ecliptic_to_equatorial(&lsol,&beta);
 lsol/=15.;
 DEBUG("alpha=%f delta=%f\n",lsol,beta);
 DEBUG("lsolp=%f \n",lsolp);
 ecliptic_to_equatorial(&lsolp,&betap);
 lsolp/=15.;
 DEBUG("alphap=%f deltap=%f\n",lsolp,betap);
 // chap 33, p.52
 H=acos(-tan(beta *M_PI/180.)*tan(mylat*M_PI/180.))*180./M_PI/15.;
 GSTr1=24.+lsol -H-mylon/15.;borne24(&GSTr1);
 GSTs1=lsol+H-mylon/15.;     borne24(&GSTs1);
 H=acos(-tan(betap*M_PI/180.)*tan(mylat*M_PI/180.))*180./M_PI/15.;
 GSTr2=24.+lsolp-H-mylon/15.;borne24(&GSTr2);
 GSTs2=lsolp+H-mylon/15.;    borne24(&GSTs2);
 if (GSTr1>GSTr2) {GSTr2+=24.;GSTs2+=24.;}
// calculer T00 chap 12 p.17
  S=D-3653.5;
  T=S/36525.;
  T0=6.697374558+(2400.051336*T)+(0.000025862*T*T);borne24(&T0);
  T0p=T0-((mylon/15.)*1.002738);while (T0p<0.) T0p+=24.;
  if (GSTr1<T0p) {GSTr2+=24.;GSTr1+=24.;}
  if (GSTs1<T0p) {GSTs2+=24.;GSTs1+=24.;}
 DEBUG("GSTr1=%f GSTs1=%f\n",GSTr1,GSTs1);
 DEBUG("GSTr2=%f GSTs2=%f\n",GSTr2,GSTs2);
 DEBUG("T00=%lf T00p=%lf \n",T0,T0p);
 GSTs=(24.07*GSTs1-T0*(GSTs2-GSTs1))/(24.07+GSTs1-GSTs2);
 GSTr=(24.07*GSTr1-T0*(GSTr2-GSTr1))/(24.07+GSTr1-GSTr2);
 DEBUG("GSTr=%f GSTs=%f\n",(GSTr),(GSTs));
 GSTr-=T0;borne24(&GSTr);GSTr*=0.9972695663;
 GSTs-=T0;borne24(&GSTs);GSTs*=0.9972695663;
 GSTr-=0.0753; // LATITUDE DEPENDENT
 GSTs+=0.0753; // LATITUDE DEPENDENT
 // {hrise minrise hset minset}
 res=(unsigned long)(rintf(floorf(GSTr)))*256*256*256+(unsigned long)(rintf((GSTr-floor(GSTr))*60.))*256*256+
 return(res);
}

/*
int main(int argc,char **argv)
{unsigned long res;
 unsigned short k,hrise,minrise,hset,minset;
 if (argc>1)
   res=sunriseset(2007,0,atoi(argv[1]));
 else
   for (k=0;k<256;k++) {
   res=sunriseset(2007,0,k);
 hrise=  (res&0xFF000000)/(256*256*256);
 minrise=(res&0x00FF0000)/(256*256);
 hset=   (res&0x0000FF00)/256;
 minset= (res&0x000000FF);
 printf("day=%d UTr=%dh%dmin, UTs=%dh%dmin\n",k,hrise,minrise,hset,minset);
 }
}
*/
```

TAB. 7 – Le programme en C de calcul des horaires de lever et de coucher du soleil : en décommentant la partie `main()`, le programme fonctionne sur PC. Pour un fonctionnement sur MSP430, la fonction `sunriset()` est appelée depuis le programme assembleur présenté dans le tableau 6.