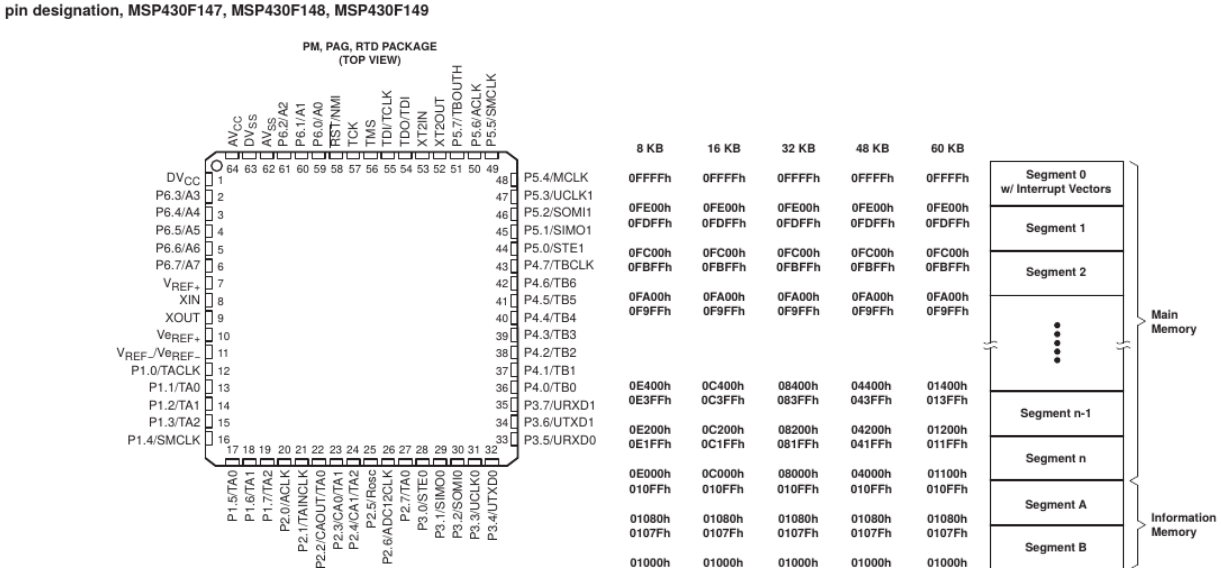
Sujet informatique embarquée – examen M1

J.-M Friedt, 27 mai 2015

Tous les documents sont autorisés, toutes les connexions à internet sont autorisées mais les communications par téléphone mobile sont **proscrites**. Chaque question sera notée sur 1 point. Les questions bonus sont considérées comme plus difficiles et ne méritent pas de s'y attarder en cas de manque de temps.

Afin d'illustrer la flexibilité et la généralité de `gcc`, on se propose d'apprivoiser un nouveau microcontrôleur 16 bits, le MSP430. Il s'agit d'une architecture Von Neuman très classique et simple d'accès, notamment grâce à l'excellente documentation de Texas Instruments. En particulier, parmi la gamme du MSP430, nous nous intéresserons à un composant en particulier, le MSP430F149. Il s'agit d'un petit microcontrôleur caractérisé par sa très faible consommation, approprié pour les applications autonomes fonctionnant sur batterie. Comme souvent, deux documentations sont utiles pour maîtriser un nouveau microcontrôleur : un guide d'utilisateur général de l'architecture [1] et la déclinaison spécifique du microcontrôleur qui nous intéresse [2]. Le premier document fournit des informations valables pour tout membre de la famille des MSP430, la seconde le brochage et les fonctionnalités fournies par une implémentation particulière. Notre objectif sera, *basé sur ces deux documentations* et la version pour MSP430 de `gcc`, de développer quelques codes pour cette nouvelle plate-forme.



1. **Quelle quantité de mémoire volatile est disponible ?**
2. **En supposant que nous réservons 256 octets pour la pile, combien de variables de type `unsigned short` pouvons nous stocker en mémoire ? Justifiez.**
3. **Quel est le rôle de la pile ? Que se passe-t-il si le nombre de variable créé devient trop important et empiète sur la zone allouée à la pile ?**
4. **À quelle adresse initialiser le pointeur de pile sur un MSP430F149 ?**

Bonus 1 : lors de la compilation, nous indiquerons à `gcc` que nous utilisons un MSP430F149 par l'option `-mmcu=msp430f149`. Un fichier définit l'organisation de la mémoire de ce composant : où se trouve-t-il ? son contenu est-il cohérent avec les réponses fournies ci-dessus ?

Un programme est composé de deux fichiers, un programme principal nommé `prog.c` comprenant la fonction `main()` – qui fait appel à une fonction `affiche()` qui n'y a pas été définie, et un second fichier nommé `fichier.c` implémentant la fonction `affiche()`.

5. **Quelle ligne de commande permet de compiler l'objet `affiche.o` ?**
6. **Comment lier le programme principal et la fonction ? Démontrer sur un exemple concret sur ordinateur.**

Bonus 2 : proposer un `Makefile` automatisant ce processus. En particulier, on indiquera clairement où se trouvent les espaces et les tabulations comme séparateur.

1. `http://www.ti.com.cn/cn/lit/ug/slau049f/slau049f.pdf`
2. `http://www.ti.com/lit/ds/symlink/msp430f149.pdf`

1 Faire clignoter une LED

La première étape pour s'approprier un microcontrôleur tient en la lecture de la *datasheet* pour comprendre à quelle adresse se trouve quel registre. En particulier, diverses configurations de registres sont nécessaires pour atteindre l'objectif visé sur chaque broche. On désire en particulier faire clignoter deux diodes électro-luminescentes (LED) connectées à deux broches du port 1.

Port	Register	Short Form	Address	Register Type	Initial State
P1	Input	P1IN	020h	Read only	–
	Output	P1OUT	021h	Read/write	Unchanged
	Direction	P1DIR	022h	Read/write	Reset with PUC
	Interrupt Flag	P1IFG	023h	Read/write	Reset with PUC
	Interrupt Edge Select	P1IES	024h	Read/write	Unchanged
	Interrupt Enable	P1IE	025h	Read/write	Reset with PUC
	Port Select	P1SEL	026h	Read/write	Reset with PUC

7. **Basé sur les informations ci-dessus, comment faire clignoter une LED. En particulier, quelles sont les étapes d'initialisation qui doivent précéder la boucle infinie ?**
8. **Supposons que nous ne voulions faire clignoter que les bits 3 et 4 (Px.3 et Px.4) sans toucher aux autres bits du port. Comment passer ces bits particuliers à 1 ? Même question pour placer ces bits à 0 sans toucher aux autres bits du registre.**
9. **En supposant que P3.4 soit connecté à une LED qui indique à l'utilisateur l'état du système, comment accéder à une communication asynchrone compatible RS232 ?**
10. **À quel emplacement mémoire se trouve le registre P1SEL ?**

Bonus 3 : un compilateur dans un langage de haut niveau tel que le C permet d'appeler un registre par son nom au lieu de l'appeler par son adresse. Dans quel fichier se trouve la définition de P1SEL pour le MSP430F149 ? La définition correspond-t-elle à la réponse fournie ci-dessus ?

2 Ressources occupées

Nous avons vu que des bibliothèques – telles que `libc` ou `libm` – facilitent la vie du développeur au détriment des ressources occupées. Dans de nombreux cas, un table d'allocation des polices de caractère est nécessaire : il s'agit d'un tableau contenant, pour chaque caractère affichable, la liste des pixels qui s'allument et qui s'éteignent sur l'écran. Chaque caractère est défini par les pixels qui s'allument sur un élément unitaire d'affichage à l'écran (Fig. 1) : on suppose que chaque élément affichable (caractère) est composé de 5 colonnes et 10 lignes, et que la police de caractères définit 256 symboles (table ASCII étendue par exemple).

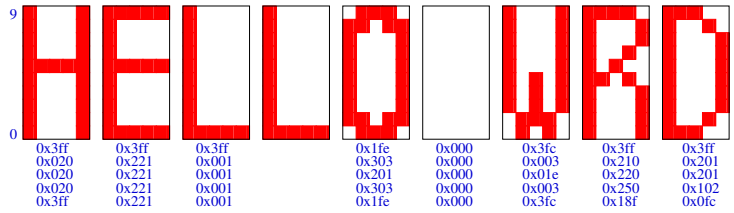


Figure 1 – Extrait d'une police de caractères – ici les lettres nécessaires à afficher le message HELLO WORLD.

11. **Un masque définit les LEDs allumées sur chaque colonne. En supposant que nous nous contentions de stocker le motif sur un mot de 16 bits (au détriment des 6 derniers bits perdus inutilement), est-il possible de placer le tableau de la police de caractères en RAM ? Justifiez.**
12. **Fournir un bout de code qui définit un caractère de la police de caractères.**

Préfixer une définition de variable ou de tableau par le mot clé `const` informe le compilateur que le contenu de cette variable ne peut être modifié par le programme. Par conséquent, le compilateur s'autorise à placer la définition en mémoire non-volatile.

13. **Quelle est la taille de la mémoire non-volatile sur MSP430F149 ?**
14. **Cette taille est-elle suffisante pour stocker la police de caractères ? Justifiez.**

Bonus 4 : la commande `size` (et sa déclinaison pour MSP430 `msp430-size` permet de connaître la quantité de mémoire occupée dans chaque segment (mémoire volatile nommée `bss` ou `data`, ou non-volatile nommée `text`). Vérifier par une analyse d'un code, implémentant un bout de police de caractères, compilé avec ou sans le préfixe `const` devant le tableau, que la zone mémoire occupée passe de RAM à ROM.

3 Initialisation du processeur et chien de garde

Cette étude illustre le code d'initialisation du microprocesseur, et les effets indésirables qu'il peut engendrer s'il n'est pas étudié avec soin.

Un programme avait pour habitude de compiler fort bien avec les "anciennes" versions de `gcc`, et en particulier sa version 3.3.2 (datée de 2003). Une version plus récente de `gcc` (4.6.3, de 2012) induit un programme qui "ne fonctionne plus comme avant".

Un programme compilé au moyen de `msp430-gcc` commence désormais par

```
00004000 <__watchdog_support>:
    4000:       55 42 20 01     mov.b   &0x0120,r5
    4004:       35 d0 08 5a     bis     #23048, r5      ;#0x5a08
    4008:       82 45 00 11     mov     r5,     &0x1100

0000400c <__init_stack>:
    400c:       31 40 00 39     mov     #14592, r1      ;#0x3900

00004010 <__do_copy_data>:
    4010:       3f 40 00 00     mov     #0,     r15     ;#0x0000
    4014:       0f 93           tst     r15
    4016:       08 24           jz      $+18            ;abs 0x4028
    4018:       92 42 00 11     mov     &0x1100,&0x0120
    401c:       20 01
    401e:       2f 83           decd    r15
    4020:       9f 4f 80 40     mov     16512(r15),4352(r15);0x4080(r15), 0x1100(r15)
    4024:       00 11
    4026:       f8 23           jnz     $-14            ;abs 0x4018

00004028 <__do_clear_bss>:
    4028:       3f 40 00 00     mov     #0,     r15     ;#0x0000
    402c:       0f 93           tst     r15
    402e:       07 24           jz      $+16            ;abs 0x403e
    4030:       92 42 00 11     mov     &0x1100,&0x0120
    4034:       20 01
    4036:       1f 83           dec     r15
    4038:       cf 43 00 11     mov.b   #0,     4352(r15);r3 As==00, 0x1100(r15)
    403c:       f9 23           jnz     $-12            ;abs 0x4030

0000403e <main>:
...
```

Avec une version plus ancienne (3.3.2) de `msp430-gcc`, la compilation du code donnait

```
00001100 <_reset_vector__>:
    1100:       b2 40 80 5a     mov     #23168, &0x0120 ;#0x5a80
    1104:       20 01

00001106 <_copy_data_init__>:
    1106:       3f 40 74 5e     mov     #24180, r15     ;#0x5e74
    110a:       3e 40 00 02     mov     #512,   r14     ;#0x0200
    110e:       3d 40 18 02     mov     #536,   r13     ;#0x0218
    1112:       0d 9e           cmp     r14,    r13     ;
    1114:       05 24           jz      $+12            ;abs 0x1120
    1116:       fe 4f 00 00     mov.b   @r15+,  0(r14)  ;
    111a:       1e 53           inc     r14             ;
    111c:       0e 9d           cmp     r13,    r14     ;
    111e:       fb 2b           jnc     $-8             ;abs 0x1116

00001120 <_clear_bss_init__>:
    1120:       3f 40 18 02     mov     #536,   r15     ;#0x0218
    1124:       3d 40 18 02     mov     #536,   r13     ;#0x0218
    1128:       0d 9f           cmp     r15,    r13     ;
    112a:       05 24           jz      $+12            ;abs 0x1136
    112c:       cf 43 00 00     mov.b   #0,     0(r15)  ;r3 As==00
    1130:       1f 53           inc     r15             ;
    1132:       0f 9d           cmp     r13,    r15     ;
    1134:       fb 2b           jnc     $-8             ;abs 0x112c
```

```
00001136 <_end_of_init__>:
    1136:       30 40 40 11     br      #0x1140         ;
...

00001140 <main>:
    119e:       b2 40 80 5a     mov     #23168, &0x0120 ;#0x5a80
...
```

Register	Short Form	Register Type	Address	Initial State
Watchdog timer control register	WDTCTL	Read/write	0120h	06900h with PUC
SFR interrupt enable register 1	IE1	Read/write	0000h	Reset with PUC
SFR interrupt flag register 1	IFG1	Read/write	0002h	Reset with PUC†

15. **Qu'est-ce qu'un chien de garde ? (*watchdog*). À quoi sert-il ?**
16. **Quel est son état dans le code issu de `msp430-gcc` 4.6.3 ? Dans le code issu de `msp430-gcc` 3.3.2 ? Justifiez.**
17. **Quelle est la conséquence si on n'y prend garde ? Pourquoi le code semble-t-il "ne plus fonctionner" ? S'il est actif, au bout de combien de temps le chien de garde se déclenche-t-il dans les configurations proposées ci-dessus ?**

WDTCTL, Watchdog Timer Register

15	14	13	12	11	10	9	8
Read as 069h WDTPW, must be written as 05Ah							

7	6	5	4	3	2	1	0
WDTHOLD	WDTNMIES	WDTNMI	WDTTMSEL	WDTCNTCL	WDTSSEL	WDTISx	
rw-0	rw-0	rw-0	rw-0	r0(w)	rw-0	rw-0	rw-0

Bonus 5 : quelle option de `msp430-gcc` version 4.6.3 (installé sur les PCs) permet de gérer le chien de garde implicitement à la compilation, sans action explicite du programmeur ?

4 TinyOS sur MSP430 : un environnement exécutif

Un niveau d'abstraction plus élevé est accessible en remplaçant la programmation en assembleur ou en C par un environnement exécutif nommé TinyOS [3]. Cet environnement exécutif est aussi fonctionnel, en plus du MSP430 représenté par la plateforme TelosB [4] sur microcontrôleurs Atmel Atmega (plateforme MicaZ) ou ST Microelectronics STM32.

18. **Quels sont les intérêts d'un environnement exécutif sur microcontrôleur ?**

L'exemple le plus trivial, qui se contente de faire clignoter 3 diodes, se trouve dans le répertoire d'exemples `apps/Blink` et se compile par `make telosb` pour donner en résultat le fichier `build/telosb/main.exe`. Ce fichier est fourni dans le répertoire `/home/jmfriedt/exam_M1` du PC.

```
-rw-r--r-- 1 jmfriedt jmfriedt 2650 May 10 07:40 $HOME/tinyos-main/apps/Blink/BlinkAppC.nc
-rw-r--r-- 1 jmfriedt jmfriedt 2924 May 10 07:40 $HOME/tinyos-main/apps/Blink/BlinkC.nc
-rwxr-xr-x 1 jmfriedt jmfriedt 12630 May 10 07:47 main.exe
-rw-r--r-- 1 jmfriedt jmfriedt  7362 May 10 07:47 main.hex
```

Un programme aux fonctionnalités identiques du point de vue utilisateur est écrit en assembleur et génère les fichiers suivants :

```
-rwxr-xr-x 1 jmfriedt jmfriedt 10943 May 10 07:54 led.exe
-rw-r--r-- 1 jmfriedt jmfriedt   468 May 10 07:54 led.hex
-rwxr-xr-x 1 jmfriedt jmfriedt   673 May 10 07:48 led.S
```

19. **Quels sont les fichiers qui peuvent être flashés dans le microcontrôleur et dont la taille est représentative de l'occupation de ROM ? Est-ce que ces deux programmes tiennent dans un MSP430F149 ? Justifiez.**
20. **Analyser les fonctions et bibliothèques requises par TinyOS et commenter sur la taille du code résultant.**

3. `http://sourceforge.net/p/tinyosonstm32/code/ci/master/tree/`
4. `http://www.willow.co.uk/html/telosb_mote_platform.php`

Réponses

1. Le MSP430F149 possède 2 KB de RAM
2. (2048-256)/2=896 variables de type short qui occupent 2 octets chacune
3. La pile permet au microcontrôleur de stocker des variables temporaires, passer les arguments aux fonctions appelées ainsi que le pointeur de programme au moment du saut à une fonction. Si la pile est corrompue par l'écriture dans une variable stockée dans une zone mémoire occupée par la pile, soit le passage de paramètres est corrompu, et dans le pire cas le programme plante à cause de la perte de la valeur du *program counter* puisque le microcontrôleur ne sait plus comment revenir au programme principal en quittant la fonction appelée.
4. Empiler une variable sur pile *décrémente* le pointeur de pile : il faut donc placer le pointeur de pile *à la fin de la RAM*, soit l'adresse 0x0A00-1=0x9FF (cf p.15 de la documentation du MSP430F149).

Bonus 1 : `locate msp430f149` indique l'existence de `/usr/msp430/lib/ldscripts/msp430f149/memory.x` qui contient la ligne `ram (wx)  : ORIGIN = 0x0200, LENGTH = 0x0800 /* END=0x0a00, size 2K */` qui est cohérent avec les affirmations ci-dessus.

5. `msp430-gcc -mmcu=msp430f149 -c affiche.c`
6. `msp430-gcc -mmcu=msp430f149 -o executable main.c affiche.o` qui fait appel à `msp430-ld`.

Bonus 2 :

```
all: executable
executable: main.o affiche.o
msp430-gcc -mmcu=msp430f149 -o executable main.o affiche.o
main.o: main.c
msp430-gcc -mmcu=msp430f149 -c main.c
affiche.o: affiche.c
msp430-gcc -mmcu=msp430f149 -c affiche.c
```

avec les occurrences de `msp430-gcc` précédées d'une tabulation.

7. PxSEL à 0 configure le port en GPIO, PxDIR à 1 configure le port en sortie, et PxOUT définit l'état de la broche (1 pour fournir la tension d'alimentation, 0 pour la masse). PxSEL et PxDIR se configurent en dehors de la boucle infinie, PxOUT varie dans la boucle.
8. Utilisation des masques : passer à 1 demande un OU logique avec le masque 0x18 (bit 3 est 0x08, bit 4 est 0x10) : `PxOUT |= 0x18;`. Pour placer à 0, on effectue un ET logique avec l'inverse de ce masque : `PxOUT &=~0x18;` La réponse passant par le XOR (`PxOUT ^= 0x18;`) est acceptée.
9. Le MSP430F149 possède deux ports de communication asynchrone, UTXD0 et UTXD1. UTXD0 est sur P3.4 : il nous reste donc le port 1 sur P3.6 donc nous devons passer les bits P3.6 et P3.7 de P3SEL à 1 pour utiliser les fonctions secondaires de ces broches.
10. P1SEL se trouve à l'emplacement 0x26 (page 9-7 de SLAU049F).

Bonus 3 : `locate msp430f149 | grep h$` indique l'existence de `/usr/msp430/include/msp430f149.h` qui définit `#define P1SEL_  0x0026  /* Port 1 Selection */`, en cohérence avec notre réponse précédente.

11. 2 octets/colonne×5 colonnes/caractère×256 caractères=2560 octets >2 KB. La table ne peut pas tenir en RAM.
12. `short tableau[256*5]={0x123,0x456,0x789,0xABC,0xDEF};`
13. 60 KB (cf p.15 de la datasheet du MSP430F149)
14. Cette taille sera suffisante pour stocker les 2560 octets.

Bonus 4 : en définissant un tableau comme ci-dessus de la forme nous obtenons une occupation des ressources de

```
$ msp430-size executable
   text    data     bss     dec     hex filename
    140    2560       2    2702     a8e executable
```

Si nous préfixons cette définition de `const` pour donner
`const short tableau[256*5]={0x123,0x456,0x789,0xABC,0xDEF};` alors l'occupation de ressources devient

```
$ msp430-size executable
   text    data     bss     dec     hex filename
   2700       0       2    2702     a8e executable
```

donc l'intégralité des ressources occupées est en mémoire non-volatile (flash) au lieu d'occuper plus de 2 KB en RAM. La seule subtilité dans cette démonstration est que le premier cas (dépassement de capacité de RAM) ne compile pas avec l'option `-mmcu=msp430f149` car `gcc` détecte le dépassement de capacité (`region 'ram' overflowed by 514 bytes`) et nous devons compiler avec une option de microcontrôleur possédant des ressources plus étendues, par exemple le MSP430F1611 par `-mmcu=msp430f1611`.

15. Le chien de garde est un compteur matériel qui, s'il n'est pas remis à 0 par le programme principal exécuté par le microcontrôleur, induit une réinitialisation (relance du programme) lorsque la valeur maximale du compteur est atteinte. Il s'agit d'un mécanisme qui garantit le redémarrage du système embarqué en cas de défaillance logicielle.
16. `msp430-gcc` 4.6.3 initialise le *watchdog* avec l'état 0x5a08. 0x5A sur l'octet de poids fort est le code permettant la configuration du compteur, et 0x08 sur l'octet de poids faible met à 1 le bit `WDTCNTCL` donc initialise le compteur. On note en particulier que `WDTHOLD=0` donc le compteur est actif. Au contraire, `msp430-gcc` 3.3.2 initialisait le *watchdog* par 0x5a80 donc place à 1 le bit `WDTHOLD` qui désactive donc le chien de garde (page 199 ou 10-8 de SLAU049F).
17. Si le chien de garde est actif et n'est pas réinitialisé périodiquement par le programme principal, le microcontrôleur se réinitialise (reset) continuellement. Le code semble donc ne plus fonctionner puisque le programme redémarre continuellement. En supposant que le chien de garde est cadencé par l'oscillateur DCO interne à 1 MHz, alors la réinitialisation se produit tous les $10^6/32768 = 31$ Hz=32 ms (tel que décrit p.199 de SLAU049F).

Bonus 5 : la commande `msp430-gcc -v --help |& grep wat` indique qu'une nouvelle option de msp430-gcc a été ajoutée sous la forme `-mdisable-watchdog` documenté comme `Link the crt0 modules that disable the watchdog on startup`.

18. Portabilité du code, utilisation de bibliothèques fournies par les autres développeurs du système exécutif et accessibles au travers d'une API documentée.
19. Les deux formats compréhensibles par un microcontrôleur et épurés des symboles de déverminage et autres structures de données issues de la compilation (image prête à être copiée en mémoire non volatile du microcontrôleur) sont les `.bin` (option `-O binary` de `objcopy`) et `.hex` (option `-O ihex` de `objcopy`). Le fichier hexadécimal contient environ deux caractères par octet qui sera flashé dans le microcontrôleur, donc on divise par 2 la taille des fichiers `.hex` pour estimer la quantité de mémoire non-volatile nécessaire. `led.hex` issu du programme assembleur tient en $468/2 \simeq 234$ octets, largement inférieur aux 60 KB disponibles. `main.hex` issu de TinyOS occupe $7362/2 \simeq 3681$ et tient aussi en mémoire flash du MSP430F149.
20. Le code de TinyOS est considérablement plus volumineux à cause des fonctions liées à l'ordonnanceur et la gestion des ressources par chaque tâche. Bien que ne puissions le voir dans la liste des symboles de `main.exe`, l'ordonnanceur de TinyOS fait appel à `memset()` et donc aux bibliothèques de `libc`.