# TP2 : Développement Croisé

Après le premier TP qui vous a permis de vous familiariser avec la plate-forme, l'objectif de ce TP est d'étudier un peu plus en détail certains concepts centraux en Architecture : communication avec le matériel au travers de registres projetés en mémoire (*memory-mapped IO*), scrutation de signaux (*polling*), taille et représentation des types de données, compilation séparée. On manipulera également plusieurs outils techniques de bas niveau : exécution pas-à-pas, examen de la mémoire, émulation, programmation en assembleur, etc.

La finalité de ce TP sera de programmer un jeu de dés électronique. On s'intéressera donc successivement aux différents ingrédients nécessaires : affichage sur l'écran LCD, interaction au travers des boutons, et tirage de nombres aléatoires. Attention : une bonne partie du code que vous allez écrire vous sera certainement utile dans la suite du TP, et également dans les TP suivants. En conséquence, gardez bien sous la main une copie de ce vous écrivez, de façon à pouvoir le réutiliser par la suite.

**Important** Pour chacune de ces questions, on vous demande de toujours **justifier** votre réponse en donnant une référence précise indiquant *où* vous avez trouvé les informations que vous avancez : **dans quel document, à quelle page**, etc. Une réponse non justifiée sera considérée comme fausse.
Si vous tapez votre compte-rendu à l'ordinateur (ce qui est vivement conseillé puisqu'il comportera une certaine quantité de code), vous pouvez même ajouter un *copier-coller* ou une capture d'écran des pages en question. De cette facon-là, il nous sera nettement plus facile de suivre votre raisonnement.
Rappel : Ces documents sont tous disponibles sur la page moodle des TP d'Architecture.

## Documentations techniques

[Motherboard.pdf] *MSP430FG4618/F2013 Experimenter's Board User's Guide*.

[CPU.pdf] Chapter 3 of *MSP430x4xx Family User's Guide*.

[MSP430.pdf] *MSP430x4xx Family User's Guide*.

[datasheet.pdf] *MSP430xG461x Device Datasheet*.

[LCD.pdf] SoftBaugh *SBLCDA4 Specification*.

[workbench.pdf] *MSP430 IAR Embedded Workbench IDE User Guide*.

[compiler.pdf] *MSP430 IAR C/C++ Compiler Reference Guide*.

[assembler.pdf] *MSP430 IAR Assembler Reference Guide*.

[linker.pdf] *IAR Linker and Library Tools Reference Guide*.

[libc.pdf] *IAR C LIBRARY FUNCTIONS Reference Guide*.

## 1 Pilote de l'écran LCD

L'affichage sur un écran à cristaux liquides se fait au travers d'un *contrôleur LCD*, c'est à dire un composant dont le rôle est d'appliquer les bonnes tensions électriques aux bons endroits de l'écran, de façon à noircir les zones voules. Suivant les fabricants, le contrôleur est intégré avec l'écran, auquel cas on communique avec lui par un certain protocole de commandes d'affichage, ou bien l'écran ne comporte aucune intelligence, et il faut lui adjoindre un contrôleur externe.

C'est dans cette deuxième situation que nous nous trouvons ici : le modèle d'écran monté sur la carte est le *SoftBaugh SBLCDA4* (vous trouverez la *datasheet* correspondante [LCD.pdf] sur Moodle), qui ne comporte pas de contrôleur, mais il se trouve que notre MSP430 en possède un parmi ses périphériques internes [MSP430.pdf, chapitre 26]. Notez au passage que le chapitre 25 décrit également un contrôleur LCD similaire, mais qui n'est pas présent sur le MSP430FG4618. Le nôtre est le *LCD_A*, décrit au chapitre 26.

## 1.1 Prise en main

L'objectif de cette partie du TP est d'écrire un *pilote de périphérique*, qui permettra plus tard à vos programmes d'afficher des informations sur l'écran.
On vous donne, en annexe de cet énnoncé (cf p. 10), un début de pilote pour le contrôleur *LCD_A*. La configuration du contrôleur se fait comme pour les autres périphériques internes du MSP430, au travers de registres projetés en mémoire.

Créer un nouveau projet «TP2» dans IAR *Embedded Workbench*, et y créer des fichiers nommés `main.c`, `lcd.h` et `lcd.c`. Copier le code de `lcd_init()` dans `lcd.c`, indiquer le prototype de la fonction dans `lcd.h`, et écrire un programme `main()` qui appelle cette fonction :

```
#include "msp430fg4618.h"
#include "lcd.h"

int main(void)
{
    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer

    lcd_init();

    for(;;); // endless loop
}
```

**Important** Au cours de ce TP, on va s'intéresser d'assez près à certains aspects du jeu d'instructions et des périphériques du MSP430. Afin ne pas être gêné par les particularités du processeur *MSP430X* présent dans notre cible, nous allons forcer à la main certaines des options de la chaîne de compilation.

- Ouvrir la fenêtre d'*options du projet* : dans le cadre *Workspace*, sélectionner votre *projet* (le cube bleu clair) puis cliquer sur le menu *Project > Options*.
- Dans la catégorie *General Options*, onglet *Target*, choisir comme *Device* le MSP430FG4618 (comme pour le TP1).
- Ensuite, dans la catégorie *Linker*, onglet *Config*, cocher la case *Override default* dans *Linker command file*. Le chemin du fichier, qui était grisé, devient modifiable, mais on va le laisser à cette valeur (c'est à dire `$TOOLKIT_DIR$\CONFIG\lnk430FG4618.xcl`)
- Enfin, dans la catégorie *General Options*, onglet *Target*, choisir comme *Device* le «modèle» *Generic MSP430 device*.

*Note :* à chaque transfert du programme sur la carte, IAR *Embedded Workbench* va maintenant se plaindre que la cible choisie (*Generic MSP430 device*) ne correspond pas à la puce rencontrée en réalité (MSP430FG4618). Les deux sont cependant compatibles et on peut ignorer tranquillement ce message.

Au passage, assurez-vous que votre projet est bien configuré pour exécuter le programe sur la carte, et non pas dans l'émulateur. Pour cela, dans les *options de projet*, catégorie *Debugger*, onglet *Setup*, il faut bien choisir *FET Debugger* dans la liste déroulante, et non pas *Simulator*.

## 1.2 Initialisation du contrôleur

**Question 1** Dans quels registres matériels la fonction `lcd_init()` écrit-elle ?

**Question 2** À quels périphériques ces registres appartiennent-ils ? Vous trouverez ces informations en parcourant le guide des MSP430x4xx [MSP430.pdf] (Rappel : justifiez votre réponse).

**Question 3** En vous aidant de la datasheet du MSP430FG4618 [datasheet.pdf], indiquer la taille de ces différents registres, et à quel emplacement ils sont projetés dans l'espace d'adressage.

La fonction `lcd_init()` écrit dans ces registres matériels en utilisant leur nom d'usage (souvent le même que dans la documentation du matériel), exactement comme s'il s'agissait d'autant de variables globales du programme. Mais ces «variables» ne sont pas situées dans la RAM ! Pour chacune d'entre elles, son adresse aura été fixée «de force» lors de la compilation.

**Question 4** Toujours selon la datasheet [datasheet.pdf], dans quelle plage d'adresses est projetée la mémoire vive du du MSP430FG4618 ? (Rappelez-vous de justifier chacune de vos réponses en indiquant d'où proviennent les informations que vous donnez).

On va maintenant s'intéresser de plus près à la façon dont sont déclarées les noms d'usage que le programme utilise pour écrire dans les registres des périphériques. En cliquand avec le bouton droit sur la ligne `#include`, ouvrir le fichier d'en-tête `"msp430fg4618.h"` puis trouver les déclarations de ces «variables».

**Question 5** En vous aidant du manuel du compilateur [compiler.pdf, chapitre *Compiler extensions*], expliquer en détail les lignes 42 et 43 du fichier `"msp430fg4618.h"`.

Exécuter le programme : tous les «pixels» [1] de l'écran deviennent noirs . Cependant l'écran scintille, car la vitesse de rafraîchissement est trop lente.

Pour remédier à ce «problème», on va maintenant s'intéresser à comprendre comment fonctionne le contrôleur LCD, et comment il est configuré. Vous trouverez les explications à ce sujet au chapitre 26 du guide [MSP430.pdf]. En particulier, lisez les pages 26-1 à 26-8.

**Question 6** Le contrôleur est cadencé par l'horloge ACLK, qui dans notre cas oscille à 32Khz. Avec les valeurs de configuration présentes par défaut dans `lcd_init()`, quelle est la fréquence de rafraîchissement $f_{Frame}$ qu'on est en train d'observer sur l'écran ?

**Question 7** Modifier le code de `lcd_init()` pour accélérer le rafraîchissement jusqu'à supprimer le scintillement. Expliquer quel(s) paramètre(s) vous avez modifié dans le programme, et quelle fréquence de rafraîchissement vous avez choisie (toujours en justifiant vos réponses).

## 1.3 Découverte de l'écran

Une fois configuré, le composant *LCD_A* de notre MSP430 permet au logiciel d'afficher des informations sur l'écran via un banc de mémoire appelé LCDMEM. Le rôle du contrôleur est de transcrire le contenu de cette «mémoire vidéo» sur l'écran, comme expliqué dans le guide [MSP430.pdf, paragraphe 26.2.1].
Grâce à des déclarations appropriées dans `"msp430fg4618.h"`, LCDMEM est vu depuis le programme comme s'il s'agissait d'un tableau d'octets.

**Question 8** Modifier la fonction `lcd_init()` pour qu'elle efface l'écran plutôt que le noircir entièrement.

Le principe de l'affichage LCD est bien illustré par la figure 26–4 du guide [MSP430.pdf, page 26-9] : chaque segment est un dipôle, et c'est la tension électrique appliquée entre ses deux bornes qui détermine si le segment se noircit ou pas. Pour éviter d'avoir à câbler $2n$ broches pour piloter $n$ segments, l'écran de la figure 26–4 utilise une seule broche *commune*, reliée à l'une des extrémités de tous les segments. Ainsi, il suffit de $n+1$ broches pour commander $n$ segments : les *n segment pins* sont notées $SP_1 \cdots SP_n$ dans le guide, et la *common pin* est notée COM.

1. Ces pixels sont en général appelés «segments» dans un écran LCD, car ils sont typiquement organisés sept par sept pour permettre d'afficher des chiffres. On parle d'*affichage sept-segments*.

Mais quand le nombre de segments est important, avoir $n+1$ broches sur le boîtier reste trop encombrant. Pour économiser encore de l'espace, une autre technique consiste à utiliser *deux* broches de référence, qu'on note alors COM0 et COM1. Chaque segment est relié d'un côté à l'une de ces broches communes, et de l'autre côté à l'une des *segment pins*. De cette façon, une même broche $SP_i$ permet de piloter *deux* segments : celui situé entre $SP_i$ et COM0, et celui situé entre $SP_i$ et COM1. Cette technique est illustrée par la figure 26–6, on parle alors de fonctionnement en *2-mux*.

Cette même idée peut être poussée plus loin, en augmentant le nombre de ces broches de référence : 2-mux, 3-mux, 4-mux, etc. Le contrôleur LCD de notre MSP430 sait piloter des écrans utilisant jusqu'à 4 lignes communes (4-mux), ce qui est le cas de l'écran présent sur notre carte mère [LCD.pdf].

**Question 9** Combien de broches le boîtier de l'écran possède-t-il ? Combien sont des broches communes, combien sont des *segment pins* ?

**Question 10** En conséquence, combien de segments l'écran lui-même comporte-t-il ?
Comme indiqué sur la figure 26–2 du guide [MSP430.pdf], chaque bit de la mémoire vidéo correspond à un segment, c'est à dire une combinaison *segment pin* / *common pin*. En mode direct (appelé *statique* dans le guide), les lignes COM1, COM2 et COM3 ne sont pas utilisées, et donc seuls certains bits sont significatifs, comme illustré par la figure 26–5. En mode *4-mux*, les quatre lignes COM0···COM3 sont utilisées, et donc chaque bit de la mémoire vidéo est significatif. Cette situation est illustrée par la figure 26–11. Cependant, comme l'indique le texte : *This is only an example. Segment mapping in a user's application depends on the LCD pinout and on the MSP430-to-LCD connections.* Dans notre cas, les correspondances seront différentes, et il va nous falloir trouver comment sont faits les branchements.

**Question 11** Selon la documentation de l'écran [LCD.pdf], quelles broches faut-il stimuler pour afficher sur l'écran le signe «dollar» ?

**Question 12** En vous aidant du schéma électrique de la carte mère [Motherboard.pdf, page 15], déterminer quelle est la combinaison de broches correspondante sur le MSP430.

**Question 13** Ajouter à votre programme une fonction `display_dollar()` qui affiche le symbole dollar.

## 1.4 Affichage de nombres

Le pilotage des autres segments de l'écran se passe exactement de la même manière que pour le signe dollar. En particulier, comme l'indique la datasheet de l'écran [LCD.pdf, page 4], chacun des chiffres est constitué de sept segments notés A à G. Par exemple, pour afficher le nombre 42, il faudra allumer les segments B2, C2, F2, G2, et A1, B1, D1, E1, G1.

Pour vous faciliter la tâche, on vous donne ci-dessous une fonction qui prend en entrée un nombre, le décompose en chiffres, et affiche chacun de ces chiffres.

```
void lcd_display_number(unsigned int number)
{
    lcd_clear();
    if(number == 0 )
        lcd_display_digit(0, 0);

    int i=0;
    while(number)
    {
        lcd_display_digit(i,number%10);
        number /= 10;
        i++;
    }
}
```

**Question 14** Écrire les deux fonctions manquantes pour que le code ci-dessus puisse s'exécuter :

— `void lcd_clear()` effacera l'écran, et

— `void lcd_display_digit(int pos, int digit)` affichera un chiffre donné à une position donnée.

## 2 Passage de paramètres, exécution pas-à-pas, et examen de la pile

Dans cette partie, nous allons observer d'un peu plus près le code assembleur que génère le compilateur, en particulier pour les appels de fonctions. Afin que ce code ne soit pas inutilement complexe et contre-intuitif, nous allons tout d'abord désactiver les optimisations de compilation. Pour cela, ouvrir votre fenêtre d'*options du projet*, puis dans la catégorie *C/C++ compiler*, onglet *Optimizations*, fixer le niveau d'optimisation à *None*.

On considère maintenant le programme suivant :

```
#include "msp430fg4618.h"
#include "lcd.h"

void lcd_display_seven_digits(int a, int b, int c, int d, int e, int f, int g)
{
  lcd_display_digit(1,a);
  lcd_display_digit(2,b);
  lcd_display_digit(3,c);
  lcd_display_digit(4,d);
  lcd_display_digit(5,e);
  lcd_display_digit(6,f);
  lcd_display_digit(7,g);
}

int main(void)
{
    WDTCTL = WDTPW + WDTHOLD; // Stop watchdog timer

    lcd_init();
    lcd_display_seven_digits(1,2,3,4,5,6,7);
}
```

**Question 15** En utilisant la «vue» *Disassembly* du *debugger* (menu *View*), donner le code assembleur qui implémente l'appel à `lcd_display_seven_digits()`, y compris le passage des arguments.

**Question 16** En vous aidant de la documentation du compilateur [compiler.pdf], en particulier la partie sur la convention d'appel, commenter chaque ligne du code assembleur en question. Rappel : comme pour toutes les questions de ce TP, justifiez votre réponse en indiquant précisément *d'où* vous tirez chaque information.

**Question 17** Ajouter dans votre code un point d'arrêt à la ligne `lcd_display_digit(6,f)` (bouton droit de la souris, puis *Toggle Breakpoint*), puis exécuter le programme jusque là. À l'aide des différentes «vues» du *debugger* (*Memory*, *Registers*, et/ou *Stack*) dessiner (en entier) l'état de la pile d'exécution, et commenter chaque mot mémoire.

**Remarques** L'un des objectifs de cette question est de vous faire découvrir et manipuler différentes fonctionnalités d'un *debugger*. Vous serez largement plus efficace dans n'importe quel travail de programmation si vous savez tirer parti de ce genre d'outils, alors n'hésitez pas à vous en servir le plus souvent possible, en particulier dans ces TP d'architecture !

— Pour observer finement l'exécution de votre programme, vous pouvez l'exécuter en mode pas-à-pas, par exemple grâce à la commande *Step Into* (menu *Debug*).

— Les diverses vue mémoire (*Stack*, *Memory*...) sont paramétrables. Par exemple, on peut afficher l'information mot-par-mot plutôt que octet-par-octet en cliquant avec le bouton droit de la souris puis en choisissant *2x units*.
— N'hésitez pas à parcourir la documentation du *debugger* [workbench.pdf, chapitre 4] pour plus de détails. En particulier, le tutoriel des pages 37 à 47 est une lecture incontournable.

# 3 Générateur pseudo-aléatoire

Dans le TP précédent, vous avez écrit du code pour scruter les ports GPIO et réagir à chaque appui d'un bouton (ou plus exactement, à chaque relâchement du bouton). On va aujourd'hui réutiliser cette fonctionnalité pour non plus faire cligonter une diode, mais pour afficher des nombres.
Ajouter à votre programme la fonction suivante, et dans votre boucle principale, afficher le résultat du tirage à chaque appui de bouton.

```
unsigned int alea()
{
    static unsigned int n = 1;

    n = n * 3 + 5 ;
    return n;
}
```

On voit évidemment que ce générateur est loin d'être «aléatoire», cependant, pour de nombreux usages il sera parfaitement suffisant. En réalité, de nombreux générateurs pseudo-aléatoires suivent exactement ce schéma, avec juste des valeurs plus compliquées à la place de 1, 3 et 5 (c'est le cas par exemple de la fonction `rand()` dans la GNU Libc pour MSP430).

## 3.1 Représentation des entiers

On va s'intéresser dans cette partie du TP à la façon dont les données sont représentées lors de l'exécution, et à l'impact de cette représentation sur les calculs effectués par le programme.

Dans le langage C, lorsque le programmeur déclare une variable de type *entier* (`char`, `short`, `int`, `long`...), le compilateur est relativement libre de choisir comment il va implémenter cette variable. Sur nos machines de bureau, les entiers classiques (`int`) sont typiquement codés sur quatre octets, pour exploiter naturellement les caractéristiques du processeur (registres 32 bits, UAL 32 bits, etc). Mais cette façon de faire est loin d'être universelle !

La façon dont ICC, le compilateur d'IAR, implémente les variables entières est documentée dans le chapitre *Data Representation* du manuel [compiler.pdf, p. 169].

**Question 18** Selon la documentation, sur combien de bits la variable *n* sera-t-elle stockée ?

La plupart du temps, cette problématique ne nous préoccupera pas beaucoup : pour un indice de boucle allant de 1 à 10, stocker la variable sur 16, 32 ou 64 bits ne fera pas de différence. Cependant, il y a des cas où c'est important :

**Question 19** Écrire un petit programme avec trois variables entières non-signées : $a = 30000$, $b = 40000$, et $c = a * b$. Afficher la valeur de $c$ sur l'écran, ou examiner la variable à l'aide du *debugger*, et expliquer ce qu'on observe.

**Question 20** Dans le cas général, il n'est ainsi pas possible de multiplier deux valeurs d'un certain type, et d'obtenir une valeur du même type. De quel type aurait-on dû déclarer *c* pour éviter le débordement ?
Modifier le programme en conséquence, et observer le résultat à l'aide du *debugger*. On remarquera que la variable *c* est toujours calculée de façon «incorrecte». Encore une fois, il faut revenir aux spécifications [2] pour comprendre ce qui se passe. Notre programme calcule le résultat de la multiplication de *a* par *b*, *puis* stocke cette valeur dans la variable *c*. Or, le langage C considère que lors de l'évaluation d'une expression arithmétique en nombre entiers, chaque résultat intermédiaire a pour type *le type le plus large* parmi les opérandes. Ici, notre résultat intermédiaire $a * b$ est donc encore un `unsigned int`, quelque soit le type de *c*. Il est donc tronqué de force à 16 bits, avant d'être rangé dans *c*.

**Question 21** Modifier encore le programme pour que la multiplication $c = a * b$ soit calculée «correctement», et valider le résultat à l'aide du *debugger*.
Notre générateur aléatoire, lui, n'est pas gêné par ce mode de calcul, mais au contraire il en tire parti : après quelques itérations, la multiplication déborde et donne des résultats fantaisistes, qu'on utilise comme source d'aléa.

Lorsque dans un programme, la largeur sur laquelle sont stockés les entiers est une question importante, il devient assez inconfortable de compter uniquement sur la documentation du compilateur pour nous garantir telle ou telle taille. De plus, un changement d'architecture, ou de compilateur, risque d'entraîner sans prévenir des changements intempestifs de comportement.
Pour favoriser la lisibilité et la réutilisabilité du code, la bibliothèque standard du C offre donc des types entiers supplémentaires, plus explicites : `int16_t` permet de déclarer une variable entière signée de 16 bits, `uint8_t` déclare une variable non signée de 8 bits, etc. Lorsqu'un programmeur utilise ces types «à largeur explicite» il a la garantie d'obtenir le même comportement quelque soit l'environnement dans lequel il est exécuté.

Pour avoir accès à ces nouveaux types dans IAR, il faut inclure l'en-tête `<stdint.h>`, et modifier les *options du projet* : menu *Project > Options*, puis dans la fenêtre d'options, catégorie *General Options*, onglet *Library Configuration*, choisir dans la liste déroulante «Normal DLIB» au lieu de «CLIB» [3].

**Question 22** Modifier le générateur pseudo-aléatoire pour utiliser ces types à largeur explicite.

## 3.2 Différentes manières de multiplier

Le processeur du MSP430 ne dispose pas d'une instruction de multiplication (du type `mul.w RS, RD`) comme on en trouverait sur un processeur plus puissant (x86, ARM). Lorsque le programme C contient des multiplications, le compilateur doit donc faire preuve de créativité.

**Question 23** Dans notre fonction `alea()`, comment le compilateur a-t-il implémenté la multiplication, et pourquoi ? Expliquer en détail chacune des instructions assembleur de la fonction. Vous pouvez par exemple exécuter le programme en pas-à-pas et observer l'évolution du contenu des registres.
Changer les paramètres 3 et 5 pour de plus grands nombres, jusqu'à forcer le compilateur à implémenter une «vraie» multiplication.
Observer les instructions ainsi générées. Elles font appel à un périphérique de notre microcontrôleur judicieusement appelé le *multiplieur*, qui est décrit au chapitre 9 du guide MSP430 [Motherboard.pdf].

**Question 24** Le code de la fonction `alea()` comporte maintenant plusieurs instructions `mov.w`, c'est à dire des lectures ou des écritures directes en mémoire. Vers/depuis quelles adresses transfère-t-on des données ? Déterminer de quels registres matériels il s'agit en cherchant ces adresses dans la documentation du multiplieur [Motherboard.pdf, chapitre 9].

2. Ces spécifications sont définies au sein de l'ISO (International Organization for Standardisation), et publiées dans un document appelé le *standard* du langage C, disponible par exemple à cette adresse : `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853`.

3. Pour les curieux, les différences entre ces deux libc sont expliquées dans le manuel du compilateur [compiler.pdf, pages 47 et 75].

**Question 25** Expliquer en détail chacune des instructions assembleur de la fonction. Vous pouvez par exemple exécuter le programme en pas-à-pas et observer l'évolution du contenu des registres.

**Question 26** Dans les *options de projet*, rubrique *General Options*, il y a trois manières de se servir du multiplieur : *direct access*, *library calls*, ou bien «pas du tout». Compiler successivement le programme avec les trois options, et commenter le résultat : dans chaque cas, comment la multiplication est-elle réalisée ?

```
uint16_t mul(uint16_t x, uint16_t y)
{
  return x*y;
}
```

**Question 27** Écrire la fonction `mul()` ci-dessus puis comparer quantitativement les tailles de code généré et les temps d'exécution de `mul(42,170)` pour les trois manières de compiler la multiplication.

Pour mesurer les temps d'exécution, vous devrez faire tourner le programme sur le simulateur (inclus dans IAR Embedded Workbench), et non pas sur la carte, de façon à pouvoir profiter du *compteur de cycles* inclus dans le simulateur (*Cycle Counter*, dans la vue *registers*).[4] Pour cela, ouvrir les *options de projet*, catégorie *Debugger*. Dans l'onglet *Setup*, choisir *Simulator* pour exécuter votre programme sur l'émulateur, et choisir *FET Debugger* pour exécuter votre programme sur la carte. Attention, n'oubliez pas de revenir à *FET Debugger* pour la suite du TP !

**Question 28** Commenter les résultats obtenus, et comparer les avantages et inconvénients respectifs de ces trois manières de compiler la multiplication.

### 3.3 Programmation en Assembleur

À titre de comparaison, on va maintenant écrire une nouvelle version de notre fonction `mul`, cette fois-ci en assembleur. Créer et ajouter dans le projet un nouveau fichier `mul.asm` avec le contenu suivant :

```
PUBLIC mul      ; export 'mul' to the outside world
RSEG   CODE     ; this is a relocatable segment containing code

mul:            ; entry point to the function
    ret

END             ; end of file
```

Dans le fichier `.c`, retirer le corps de la fonction `mul()` et le remplacer par un point-virgule pour ne laisser qu'une *déclaration* de fonction.

Cliquer ensuite sur *Download and Debug*. Notre programme est maintenant constitué de plusieurs *modules*, écrits dans des langages différents. IAR va donc faire successivement :

- la *compilation* de chaque fichier C (traduction depuis le C vers le langage machine) ;
- l'*assemblage* du fichier assembleur (traduction depuis le langage ASM vers le langage machine) ;
- *l'édition de liens* entre les deux, pour générer un seul programme exécutable.

Un compilateur C travaille module par module, sans avoir de vue globale du programme (on parle de *compilation séparée*). Lors de la compilation de `main()`, le compilateur ICC ne sait donc pas encore à quelle adresse sera placée la fonction `mul()`. Il sait seulement, grâce à la *déclaration* de la fonction, comment l'appeler, c'est à dire comment lui passer des arguments et quelle forme aura la valeur de retour. Le compilateur peut donc d'ores et déjà générer le code correspondant aux appels, même s'il est obligé de laisser l'adresse en blanc pour l'instant.

4. Le comptage de cycles est également possible lorsqu'on exécute le programme sur la carte, mais il faut alors forcer l'exécution pas-à-pas stricte au niveau assembleur (menu *Emulator > Force Single Stepping*) ce qui rend la manipulation assez fastidieuse !

L'édition de liens consiste, au contraire, à choisir des emplacements pour tous les «morceaux» du programme : où placer le code de chaque fonction, où placer les données, etc. On a indiqué dans le fichier `.asm` que notre morceau `mul` était un *segment relogeable* (RSEG) contenant du code, l'éditeur de liens est donc libre de le placer là où ça l'arrange. Il pourra ensuite *résoudre* les appels à la fonction `mul` en insérant l'adresse dans toutes les instructions `CALL`

On peut donc déjà exécuter le programme obtenu, même si la fonction `mul` ne fait rien pour l'instant.

**Question 29** Exécuter le programme en l'état. Que retourne `mul()` ? Pourquoi ? (Rappel : justifiez votre réponse !)

**Question 30** Compléter la fonction pour qu'elle calcule la multiplication par une simple boucle itérative, c'est à dire en additionnant $x$ sur lui-même $y$ fois (ou l'inverse). Commenter chaque ligne de votre programme assembleur.

Vous aurez besoin pour cette question de la documentation du jeu d'instructions MSP430 [MSP430.pdf, p. 3-17 et suivantes], du manuel de l'assembleur IAR [assembler.pdf], et dans une certaine mesure du manuel du compilateur ICC [compiler.pdf, p. 83 et suivantes].

**Question 31** Comparer en termes de temps d'exécution et de taille de code cette nouvelle implémentation de la multiplication avec les trois précédentes, et conclure.

## 4 Facultatif : jouons un peu avec le MSP430

Au cours de ce TP, on a successivement programmé un pilote pour l'écran LCD, pour les boutons, un générateur aléatoire : en bref, on a maintenant tous les mécanismes de base nécessaires pour programmer un jeu de dés, comme par exemple le 421, le yams, ou le *black-jack*.

Il est temps de faire preuve de créativité. À vos claviers !

## Annexe : Configuration du contrôleur LCD

```
#include "msp430fg4618.h"

/*  Initialize the LCD_A controller

    claims P5.2-P5.4, P8, P9, and P10.0-P10.5
    assumes ACLK to be default 32khz (LFXT1)
*/
void lcd_init()
{
    // our LCD screen is a SBLCDA4 => 4-mux operation (SLAU213 p4)

    // 4-mux operation needs all  4 common lines (COM0-COM3). COM0 has
    // a dedicated pin  (pin 52, cf SLAS508H p3),  but let's claim the
    // other three.
    P5DIR |= (BIT4 | BIT3 | BIT2); // pins are output direction
    P5SEL |= (BIT4 | BIT3 | BIT2); // select 'peripheral' function (VS GPIO)

    // Configure LCD controller
    LCDACTL =
        (1<<0) | // turn on the LCD_A module
        (0<<1) | // unused bit
        (1<<2) | // enable LCD segments
        (3<<3) | // LCD mux rate: 4-mux
        (7<<5) ; // frequency select

    // Configure port pins
    //
    // mappings  are  detailed  on  SLAU213  p15: our  screen  has  22
    // segments, wired to MCU pins S4 to S25 (shared with GPIO P8, P9,
    // and P10.0 to P10.5)
    LCDAPCTL0 =
        (0 << 0) | // MCU S0-S3   => not connected to the screen
        (1 << 1) | // MCU S4-S7   => screen pins S0-S3   (P$14-P$11)
        (1 << 2) | // MCU S8-S11  => screen pins S4-S7   (P$10-P$7)
        (1 << 3) | // MCU S12-S15 => screen pins S8-S11  (P$6 -P$3)
        (1 << 4) | // MCU S16-S19 => screen pins S12-S15 (P$2, P$1, P$19, P$20)
        (1 << 5) | // MCU S20-S23 => screen pins S16-S19 (P$21-P$24)
        (1 << 6) | // MCU S24-S25 => screen pins S20-21  (P$25, P$26)
        (0 << 7) ; // MCU S28-S31 => not connected to the screen

    LCDAPCTL1 = 0 ; // MCU S32-S39 => not connected to the screen

    // Note: as we do not intend to support battery-powered operation,
    // we don't need to mess with charge pumps and such.

    // clear all LCD  memory (cf SLAU056J p. 26-4)
    int j;
    for( j=0 ; j<20 ; j++)
    {
        LCDMEM[j] = 0xFF;
    }
}
```