MSP430 TP2 : Memory-Mapped Input/Output

Introduction

Rappelez-vous, dans le TP précédent, on a allumé et éteint une diode en écrivant en mémoire à des adresses particulières. Dans ce TP, on va aller plus loin, et se pencher sur les mécanismes mis en jeu dans la communication entre CPU et périphériques.

À savoir : Les entrées-sorties

Du point de vue du processeur, un périphérique se présente comme un ensemble de registres, qui permettent d'échanger de l'information entre le CPU et le périphérique. On peut distinguer informellement trois sortes de registres :

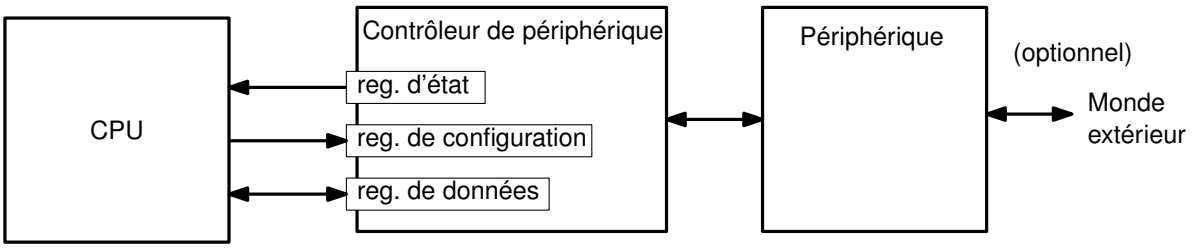
— les *registres d'état* fournissent de l'information sur l'état du périphérique. Ils sont typiquement accessibles en lecture seulement : le processeur peut lire leur contenu, mais pas le modifier.

— les *registres de contrôle* ou *de configuration* sont utilisés par le CPU pour configurer et contrôler le périphérique. Ils seront typiquement accessibles en lecture-écriture, ou parfois en écriture seulement.

— les *registres de données* permettent d'envoyer des données au périphérique (en écrivant dedans depuis le CPU) ou de recevoir des données de la part d'un périphérique (en lisant dedans).

Dans certains cas, un même registre peut appartenir à plusieurs de ces catégories, par exemple s'il contient à la fois des informations d'état (en lecture seule) et des information de configuration (en lecture/écriture).

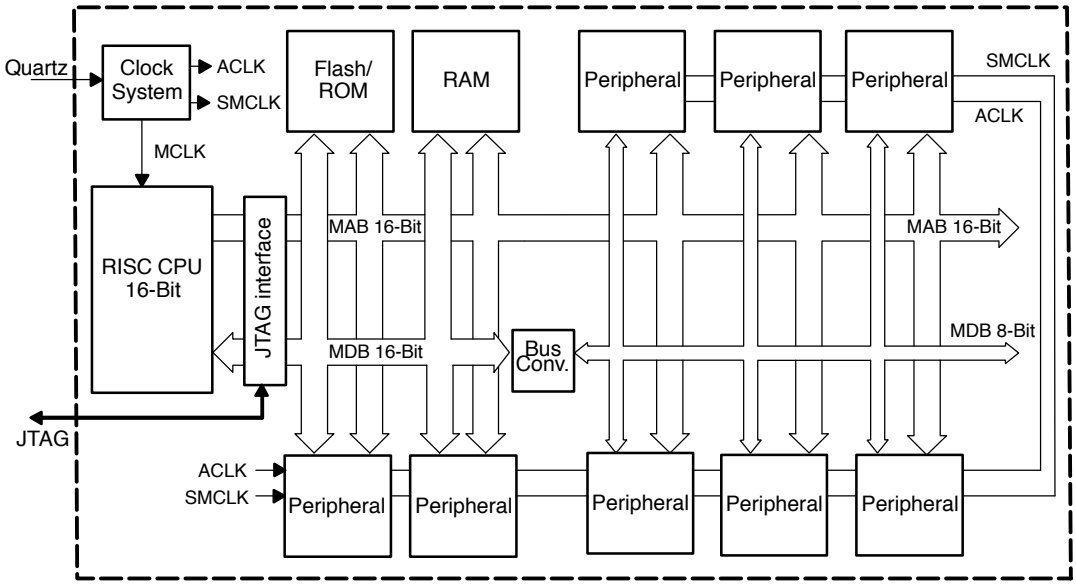
La circuiterie contenant ces registres est appelée le *contrôleur* du périphérique. Physiquement parlant, elle est parfois située sur le périphérique lui-même, par exemple un contrôleur de disque dur. Parfois au contraire elle est placée plus près du processeur, et reliée ensuite au périphérique proprement dit par un moyen quelconque. Par exemple, votre carte vidéo est reliée à votre écran par un câble VGA ou HDMI. L'architecture générique est illustrée ci-dessous :



Les registres matériels doivent pouvoir être accédés individuellement par le CPU. Comme pour les cases mémoire, on leur donne donc chacun une adresse distincte. Certains processeurs distinguent les adresses de mémoire et les adresses de registres matériels ; ils offrent alors des instructions distinctes pour accéder aux uns et aux autres. À l'inverse, la majorité des processeurs utilisent un unique *espace d'adressage* uniforme : certaines adresses correspondent à de la mémoire, et d'autres à des registres matériels. Les entrées-sorties se font alors avec les mêmes instructions que les accès mémoire classiques. On parle ainsi d'entrées/sorties «projetées en mémoire», ou *Memory-Mapped Input/Output*.

Utile pour le TP : le plan mémoire du msp430

Du point de vue du CPU, la mémoire principale et les périphériques se présentent tous comme des cases mémoire. Certains registres matériels font 16 bits, et occupent donc deux adresses consécutives (à gauche sur le schéma ci-dessous). Certains autres registres ne font que 8 bits, et occupent une seule adresse (à droite sur le schéma ci-dessous). Vous aurez aussi remarqué que la «mémoire» est elle-même composée d'une région de RAM (en lecture-écriture) et d'une région de mémoire flash (en lecture seule).



Pour s'y retrouver, la documentation technique nous indique le «plan d'adressage» (en VO, la *memory map*) c'est à dire une cartographie des différentes régions de l'espace d'adressage de la machine :

Address		Access
FFFFh	Interrupt Vector Table	Word/Byte
FFC0h		
FFBFh	Flash/ROM	Word/Byte
3100h		
30FFh	RAM	Word/Byte
1100h		
	Reserved	No access
01FFh	16-Bit Peripheral Modules	Word
0100h		
00FFh	8-Bit Peripheral Modules	Byte
0010h		
000Fh	Special Function Registers	Byte
0000h		

Pour les plus curieux : allez voir comment ces mêmes informations sont présentées dans la doc TI, en particulier [msp430.pdf page 22] et [datasheet.pdf page 16].

Exercice 1 Quelle est la taille totale de l'espace d'adressage, en kilo-octets ? Quelle est la taille de notre mémoire flash ? De notre RAM ?

Exercice 2 Combien d'adresses sont réservées aux périphériques accessibles par octet ? (Vous pouvez considérer que les *Special Function Registers* sont des registres matériels comme les autres.) Combien d'adresses sont réservées aux périphériques accessibles par mot ?

1 Accès aux registres matériels par nom symbolique

Dans cette partie, on va s'intéresser à la façon dont on peut accéder aux registres matériels depuis un programme écrit en C.

Exercice 3 Créez un nouveau répertoire TP1, et retapez [1] dans un fichier `tp2.c` le programe suivant :

```
#include <msp430fg4618.h>

volatile unsigned int i;

int main(void)
{
    // set P5.1 to output direction
    P5DIR = 2;

    for(;;)
    {
        // software delay
        for(i=0;i<20000;i++)
            {}//do nothing

        // turn red LED on
        P5OUT = 2;

        for(i=0;i<20000;i++)
            {}//do nothing

        // turn red LED off
        P5OUT = 0;

    }
}
```

Exercice 4 Compilez ce programme et transférez-le sur la carte. Constatez que la diode LED4 clignote. Désassemblez l'exécutable, et ouvrez le fichier `tp2.lst`.

Exercice 5 On s'intéresse seulement au code de la fonction `main()`, vous pouvez ignorer tout ce qui vient avant. Quelle est l'adresse de la première instruction de `main()` ? Quelle est la dernière adresse ? Dessinez la zone correspondante sur la memory map page précédente.

Exercice 6 Toujours en regardant le code de `main()`, identifiez les opérandes numériques qui représentent des adresses. Pour chacune de ces adresses, placez-la sur la memory map, et identifiez, dans le programme C, les lignes qui causent ces accès.

Commentaire Comparez ce programme C avec celui du TP précédent. Vous remarquerez que les adresses des registres matériels ne sont plus indiquées explicitement dans le code. Du point de vue du programmeur, pourtant, tout se passe comme s'il existait des variables nommées P5DIR, P5OUT, etc. Ces noms sont tirés de la documentation du matériel, ils identifient chacun un registre matériel appartenant à un périphérique.

1. Franchement, ça va plus vite de retaper un programme de cette longueur-là plutôt que d'essayer de le copier-coller (ce qui va foirer) et devoir réparer les dégâts ensuite. Et en plus, c'est formateur. Mais c'est vous qui voyez.

Explication : La correspondance entre ces noms d'usage et les adresses des registres est faite par `msp430fg4618.h`. Ce fichier contient une longue série de déclarations qui ressemblent[2] à ça :

```
...
#define P5OUT   (  *((volatile unsigned char*) 0x31)  )
#define P5DIR   (  *((volatile unsigned char*) 0x32)  )
...
```

Exercice 7 Dans votre programme, commentez la ligne `#include<msp430fg4618.h>`, et rajoutez les bons `#define` pour que le GCC compile quand même votre programme correctement. Faites valider par un enseignant.
Dans la suite, vous pourrez commenter ces définitions, et rétablir le `#include`.

2 Entrées-sorties logiques, aka General Purpose Input/Output

Maintenant qu'on a compris comment la chaîne de compilation s'y prend pour nous rendre confortable l'accès au matériel, on va s'intéresser d'un peu plus près au fonctionnement de celui-ci. Après tout, comment se fait-il qu'on puisse allumer une diode (à l'exérieur de la puce) en écrivant à une certaine adresse mémoire ? Sur le MSP430, le bus mémoire ne sort pourtant pas de la puce.
C'est un exemple de situation où le *périphérique* qui nous intéresse (la diode proprement dite) n'est accessible qu'indirectement, en passant un *contrôleur* qui possède des registres matériels branchés sur le bus mémoire. Cette distinction est un peu artificielle, et dans la pratique on confond souvent périphérique et contrôleur. Mais c'est important de garder à l'esprit que le logiciel n'a pas d'autre moyen de communication direct avec le monde que le bus mémoire de la machine de Von Neumann.

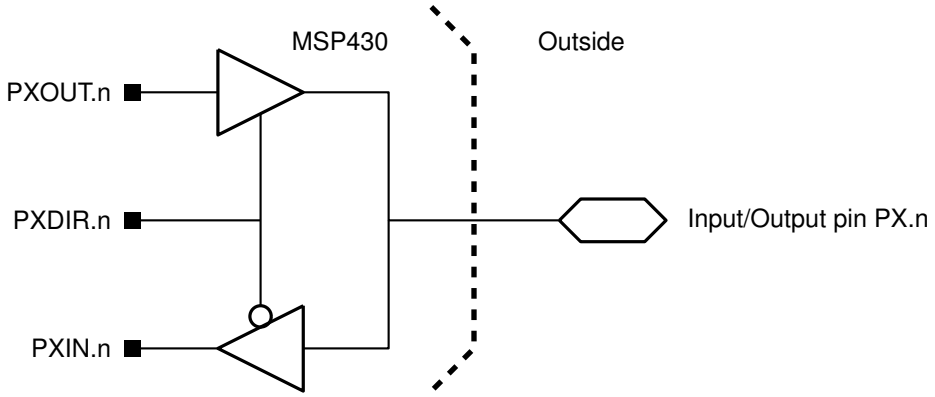
À savoir : General Purpose Input/Output

Une broche GPIO est une patte d'entrée-sortie logique destinée à un usage logiciel. Lorsqu'elle est configurée en sortie, on peut écrire un 0 ou un 1 dans un registre matériel pour positionner électriquement la broche à 0V ou à 3V. Lorsqu'elle est configurée en entrée, son état électrique externe est reflété dans un registre matériel : 0 pour «plutôt 0V», ou 1 pour «plutôt 3V». On peut ainsi connaître l'état logique de la patte en lisant la valeur du registre matériel.

2. Pour les plus curieux : ouvrez donc le fichier `/usr/msp430/include/msp430fg4618.h` et trouvez les déclarations exactes

Utile dans ce TP : les ports GPIO du MSP430

Sur le MSP430, les broches GPIO sont groupées par paquets de 8 que la documentation [a] appelle des «ports» : P1, P2, P3... Chaque port PX est ainsi associé à huit broches PX.0, PX.1 ... PX.7. La circuiterie attachée à une certaine broche PX.n est illustrée ci-dessous :



Dans le schéma ci-dessus, les carrés noirs à gauche représentent des booléens accessibles depuis le logiciel. Chaque port PX possède trois registres matériels PXDIR, PXIN, et PXOUT de 8 bits chacun. Chaque n-ième bit de ces registres est associé à la n-ième broche du port correspondant.

Par exemple, le bit P4DIR.3 permet de choisir la direction de la broche n^o 3 du port P4. Écrire un 1 dans ce bit configure la broche en sortie. On peut ensuite piloter le niveau électrique de la broche en modifiant la valeur du bit P4OUT.3. Inversement, écrire un 0 dans P4DIR.3 configure la broche P4.3 en entrée, et on peut alors consulter le niveau électrique de la broche en lisant la valeur du bit P4IN.3.

Chaque bit de chacun de ces registres est indépendant : on peut configurer certaines broches d'un même port en entrée et d'autres en sortie, etc. Dans chaque registre, le bit de poids le plus fort correspond à la broche n^o 7, et le bit de poids le plus faible correspond à la broche n^o 0.

a. Pour les plus curieux : tout ceci est documenté dans msp430x4xx.pdf à la page 409

2.1 GPIO en sortie : les diodes

Exercice 8 Sur le schéma bloc page 2, dessinez le port P5, la diode, la broche P5.1, et placez les trois registres P5DIR, P5IN, et P5OUT.

Exercice 9 Ressortez votre sujet du TP1 et retrouvez le schéma électrique de la carte mère. Constatez que la diode rouge LED4 y est connectée à la broche P5.1, c'est-à-dire la deuxième broche du port 5. Trouvez à quelles broches sont connectées les diodes LED1 et LED2.

Exercice 10 Modifiez votre programme pour faire clignoter toutes les diodes simultanément. Aidez-vous de mspdebug et de l'encadré page suivante pour faire la mise au point de votre programme : quelles valeurs faut-il écrire à quelles adresses, etc.

Extrait de la documentation : msp430x4xx.pdf page 413

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
P2	Input	P2IN	028h	Read only	–
	Output	P2OUT	029h	Read/write	Unchanged
	Direction	P2DIR	02Ah	Read/write	Reset with PUC
P3	Input	P3IN	018h	Read only	–
	Output	P3OUT	019h	Read/write	Unchanged
	Direction	P3DIR	01Ah	Read/write	Reset with PUC
P4	Input	P4IN	01Ch	Read only	–
	Output	P4OUT	01Dh	Read/write	Unchanged
	Direction	P4DIR	01Eh	Read/write	Reset with PUC
P5	Input	P5IN	030h	Read only	–
	Output	P5OUT	031h	Read/write	Unchanged
	Direction	P5DIR	032h	Read/write	Reset with PUC
P6	Input	P6IN	034h	Read only	–
	Output	P6OUT	035h	Read/write	Unchanged
	Direction	P6DIR	036h	Read/write	Reset with PUC

2.2 GPIO en entrée : les boutons

On va maintenant s'intéresser aux deux boutons poussoirs (dans le coin en bas à droite de la carte mère). Comme ils sont connectés à des broches GPIO du MSP430, on va pouvoir venir *scruter* leur état depuis le logiciel, ce qui va nous permettre d'écrire des programmes interactifs.

Exercice 11 Sur le schéma électrique, retrouvez les deux boutons SW1 et SW2. À quelles broches GPIO sont-ils connectés ?

Exercice 12 Dans votre programme, avant la boucle infinie, rajoutez un `while` qui attend un appui sur un bouton :

```
main()
{
    /* configuration */

    while ( /* button is not pressed */ )
    {
       /* do nothing */
    }

    for(;;)
    {
         /* blink */
    }
}
```

Exercice 13 Vous remarquerez que cette approche simpliste ne permet pas de distinguer deux appuis successifs. Pour ce faire, il faut non seulement détecter quand le bouton est appuyé, mais aussi quand il est relâché. Modifiez votre programme pour coller à la structure suivante :

```
main()
{
    /* configuration */

    for(;;)
```

```
    {
        /* turn all LEDS off */

        while ( /* button is not pressed */ )    { /* do nothing */ }
        while ( /* button is pressed */ )        { /* do nothing */ }

        /* turn on green LED  */

        while ( /* button is not pressed */ )    { /* do nothing */ }
        while ( /* button is pressed */ )        { /* do nothing */ }

        /* turn on yellow LED */

        while ( /* button is not pressed */ )    { /* do nothing */ }
        while ( /* button is pressed */ )        { /* do nothing */ }

        /* turn on red LED */

        while ( /* button is not pressed */ )    { /* do nothing */ }
        while ( /* button is pressed */ )        { /* do nothing */ }
    }
}
```

Faites valider par un enseignant.

3 L'écran à cristaux liquides

Afin de communiquer avec vos programmes de façon un peu plus conviviale qu'avec une diode, vous allez dans la suite écrire un *pilote de périphérique* (en anglais, *device driver*) pour l'écran LCD.

Comme expliqué dans l'introduction, il faut bien distinguer le *périphérique* qu'on veut piloter (ici, l'écran) et son *contrôleur*, avec lequel on pourra communiquer grâce à ses registres matériels. Notre MSP430 est justement équipé d'un *contrôleur LCD*, c'est à dire un composant dont le rôle est d'appliquer les bonnes tensions (analogiques) sur les bonnes broches de l'écran, de façon à noircir les zones voulues.

On vous donne, en annexe de cet énoncé (cf page 10, et sur Moodle) une fonction `lcd_init()` qui est un début de pilote. Cette fonction *initialise* le contrôleur LCD, en écrivant les bonnes valeurs dans ses registres de configuration, mais elle ne touche pas à ses registres de données. Dans les exercices qui suivent, c'est vous qui allez compléter ce pilote avec différentes fonctionnalités.

Exercice 14 Pour commencer, recopiez la fonction `lcd_init()` dans un fichier `lcd.c`, et ajoutez un fichier `lcd.h` pour y déclarer son prototype. Appelez cette fonction au début de votre fonction `main()`.

Utile dans ce TP Le contrôleur LCD possède une plage de 20 octets de *mémoire vidéo*[3] accessible depuis le processeur aux adresses 145 à 164 (incluses). Le contenu de cette mémoire est affiché sur l'écran LCD : chaque bit permet de noircir ou d'effacer un pixel.

Exercice 15 Écrivez deux fonctions `lcd_clear()` et `lcd_fill()`. La première efface complètement l'écran, et la seconde le noircit entièrement. Invoquez ces deux fonctions depuis votre `main()` pour vous assurer qu'elles marchent correctement.

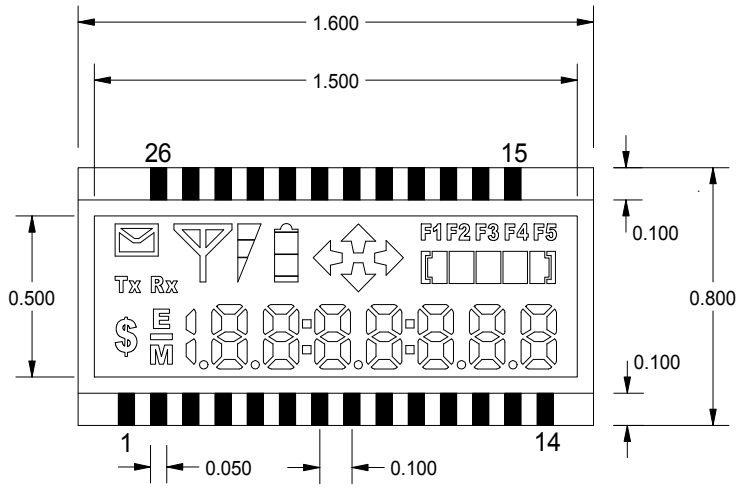
3. Vous remarquerez que la documentation utilise le terme équivalent de *registres de données* : cf msp430x4xx.pdf p 749

Exercice 16 Par essai et erreur, trouvez quel bit de la mémoire vidéo correspond au symbole «enveloppe». Ensuite, repérez l'emplacement correspondant et notez ENV dans la bonne case du tableau ci-dessous.

Address	7							0
0A4h								
0A3h								
0A2h								
0A1h								
0A0h								
09Fh								
09Eh								
09Dh								
09Ch								
09Bh								
09Ah								
099h								
098h								
097h								
096h								
095h								
094h								
093h								
092h								
091h								

Commentaire Le contrôle des autres pixels se passe de la même façon que pour l'enveloppe, en écrivant dans les bons bits de la mémoire vidéo. L'écran possède un certaine quantité d'autres symboles, et aussi une série de chiffres implémentés en *affichage 7-segments*. Tous ces éléments sont détaillés dans l'encadré page suivante. En particulier, chaque chiffre 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.

Exercice 17 Comme vous l'avez fait pour le symbole «enveloppe», trouvez quels bits de la mémoire vidéo il faut mettre à 1 pour afficher le nombre 42. Notez les positions correspondantes dans le tableau ci-dessus.

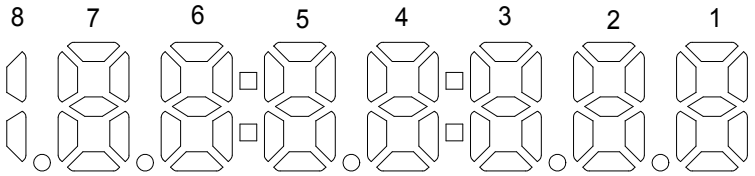


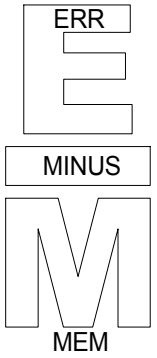
SBLCDA4

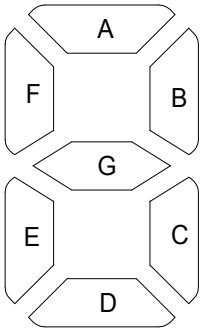
Rev 1.0.0 2 June 2003

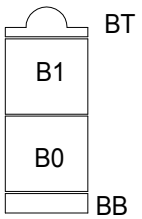
SoftBaugh, Inc.
4080 McGinnis Ferry Road
Suite 604
Alpharetta, GA 30005
www.softbaugh.com
(770) 772-8111
(770) 772-9030 (fax)

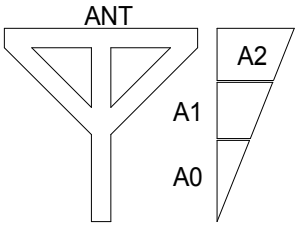
Pins both sides, 0.100" pitch, approx. 20mil wide at PCB, approx 0.250" overall
Glass is 0.8" wide, pins are 0.9" apart at PCB (0.050" knee on each side)

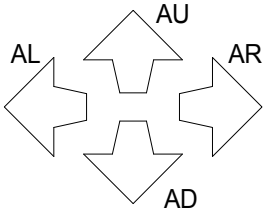


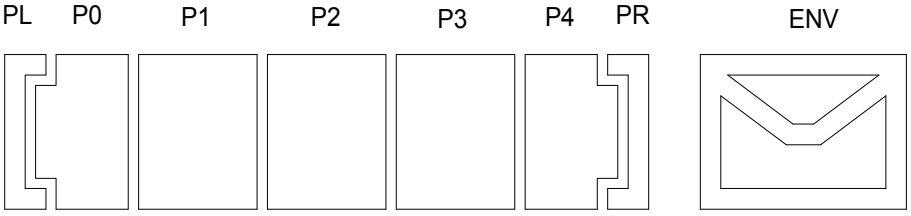












Exercice 18 Écrivez maintenant une fonction `void lcd_display_digit(int pos, int digit)` qui affiche un chiffre donné à une position donnée.

Exercice 19 Écrivez une fonction `void lcd_display_number(unsigned int number)` qui prend un nombre en paramètre, le décompose en chiffres, et affiche chacun de ces chiffres sur l'écran.

Exercice 20 Dans votre `main()`, ajoutez une boucle infinie qui incrémente un nombre à chaque itération, et l'affiche sur l'écran. Ajoutez une temporisation, ou une boucle d'attente sur les boutons, pour ralentir l'exécution et pouvoir observer votre driver en action !

Annexe : Configuration du contrôleur LCD

Note : ce fichier est également disponible sur Moodle.

```
#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 (cf motherboard.pdf p6)

  // 4-mux operation needs all  4 common lines (COM0-COM3). COM0 has
  // its dedicated pin  (pin 52, cf datasheet.pdf p3),  so 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 (cf msp430.pdf page 750)
  LCDACTL = 0b00011101;
  // bit 0 turns on the LCD_A module
  // bit 1 unused
  // bit 2 enables LCD segments
  // bits 3-4 set LCD mux rate: 4-mux
  // bits 5-7 set LCD frequency

  // Configure port pins
  //
  // mappings  are  detailed  on motherboard.pdf p19: our  screen  has  22
  // segments, wired to MCU pins S4 to S25 (shared with GPIO P8, P9,
  // and P10.0 to P10.5)
  LCDAPCTL0 = 0b01111110;
  // bit 0: MCU S0-S3   => not connected to the screen
  // bit 1: MCU S4-S7   => screen pins S0-S3   (P$14-P$11)
  // bit 2: MCU S8-S11  => screen pins S4-S7   (P$10-P$7)
  // bit 3: MCU S12-S15 => screen pins S8-S11  (P$6 -P$3)
  // bit 4: MCU S16-S19 => screen pins S12-S15 (P$2, P$1, P$19, P$20)
  // bit 5: MCU S20-S23 => screen pins S16-S19 (P$21-P$24)
  // bit 6: MCU S24-S25 => screen pins S20-21  (P$25, P$26)
  // bit 7: MCU S28-S31 => not connected to the screen

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