TP2, 5TC-BED, ez430-RF2500
Memory-Mapped I/O, Bouton

October 1, 2019

But du TP

En cours, on vous a présenté l'architecture du processeur MSP430 et l'ensemble des périphériques associés à ce processeur sur les cartes "eZ430-R2500". Ce TP a pour but de comprendre le principe de l'interaction du microcontroleur avec le monde extérieur (General Pupose Input/Output, GPIO) et le principe du contrôle de périphérique mappé en mémoire (Memory mapped I/O)

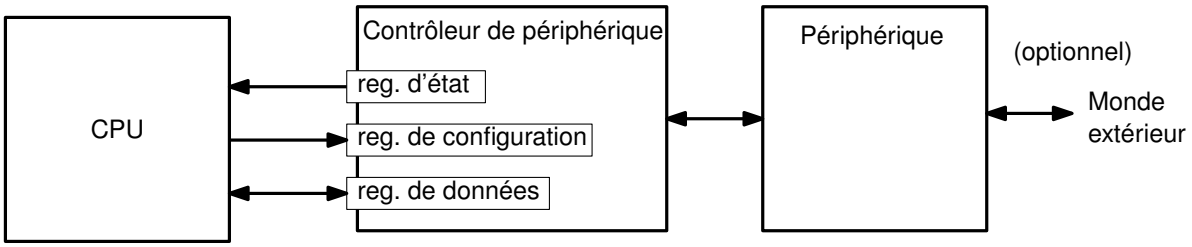
À savoir: Les entrées-sorties

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 informations de configuration (en lecture/écriture).

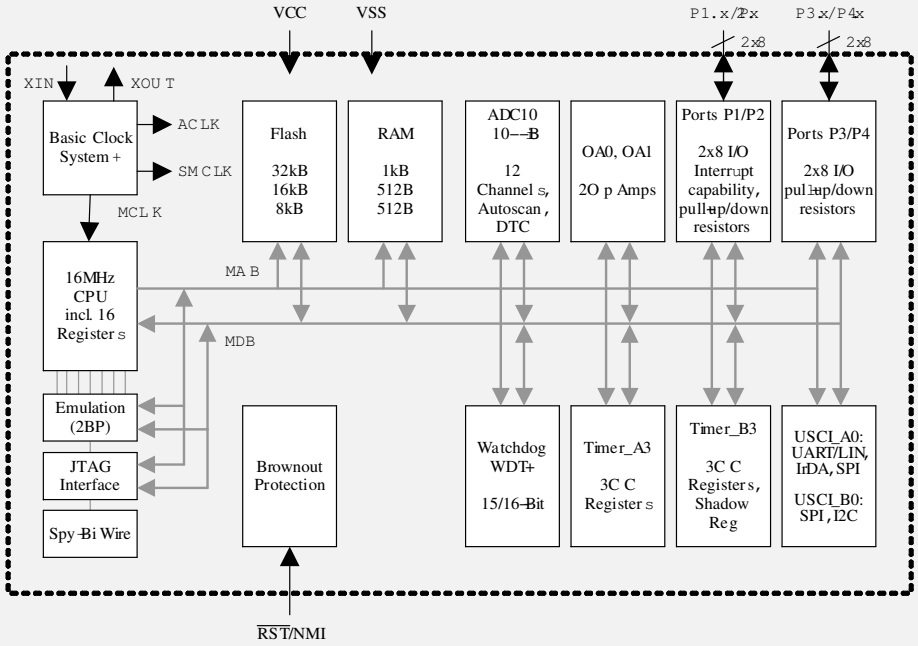
La circuiterie contenant ces registres est appelée le *contrôleur* du périphérique. Physiquement parlant, elle est parfois située sur le périphérique lui-même, par exemple un contrôleur de disque dur. Parfois au contraire elle est placée plus près du processeur, et reliée ensuite au périphérique proprement 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:



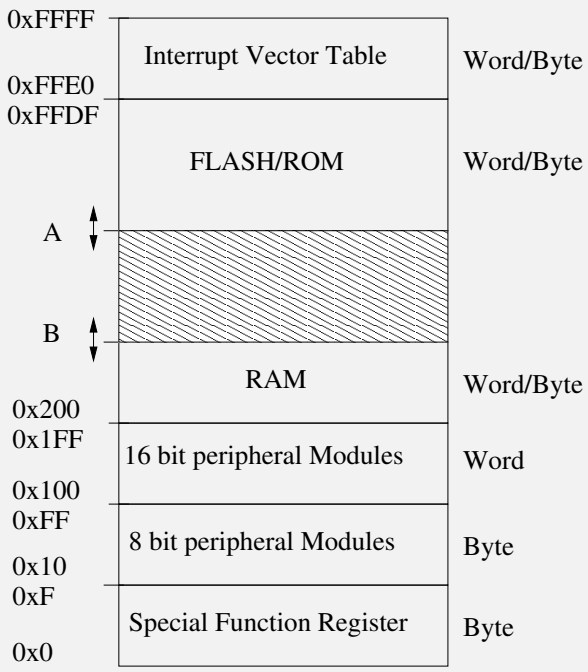
Les registres matériels doivent pouvoir être accédés individuellement par le CPU. Comme pour les cases mémoire, on leur donne donc chacun une adresse distincte. Certains processeurs distinguent les adresses de mémoire et les adresses de registres matériels ; ils offrent alors des instructions distinctes 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:



Sur le MSP430F2274 présent sur la carte ez430, A=`0x8000` et B=`0x05FF`. Ces informations sont bien sur extraites des documentation [datasheet.pdf page 18/1-4] pour le schéma ci-dessus et [docMSP430f2274.pdf p16] pour les valeurs de A et B.

Exercice 1 Quelle est la taille de la RAM et de la Flash sur le MSP430F2274 ?

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 dans un fichier `tp2.c` le programe suivant:

```
#include <msp430f2274.h>

volatile unsigned int i;

int main(void)
{
    // set P1.1 to output direction
    P1DIR = 1;

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

        // turn red LED on
        P1OUT = 1;

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

        // turn red LED off
        P1OUT = 0;

    }
}
```

Exercice 4 Compilez ce programme et transférez-le sur la carte. Constatez que la diode LED4 clignote.

Exercice 5 Désassemblez l'exécutable avec la commande:
`msp430-objdump -d tp2.elf > tp2.lst`
et ouvrez le fichier `tp2.lst` et repérez le code de la fonction `main`. Repérez respectivement l'initialisation des diodes, les deux boucles de temporisation, la boucle infinie, et les instructions qui allument et éteignent la diode.

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

Explication : La correspondance entre ces noms d'usage et les adresses des registres est faite par: `/usr/msp430/include/msp430f2274.h`. Ce fichier contient une longue série de déclarations qui ressemblent à ça:

```
...
#define P1OUT   (  *((volatile unsigned char*) 0x21)  )
#define P1DIR   (  *((volatile unsigned char*) 0x22)  )
...
```

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.

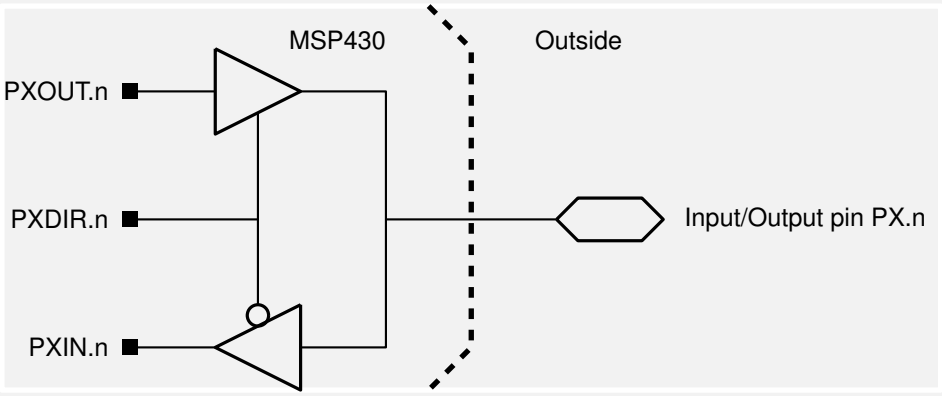
C'est un exemple de situation où le *périphérique* qui nous intéresse (la diode proprement dite) n'est accessible qu'indirectement, en passant un *contrôleur* qui possède des registres matériels branchés sur le bus mémoire. Cette distinction est un peu artificielle, et dans la pratique on confond souvent 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.

Utile dans ce TP: les ports GPIO du MSP430

Sur le MSP430, les broches GPIO sont groupées par paquets de 8 que la documentation appelle des «ports»: P1, P2, P3... Chaque port PX est ainsi associé à huit broches PX.0, PX.1 ... PX.7. La circuiterie (simplifiée) 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.

2.1 GPIO en sortie: les diodes

Exercice 7 Sur le schéma le schéma électrique de la carte mère ez430 (page 8). Constatez que la diode rouge LED0 y est connectée à la broche P1.0, c'est-à-dire la première broche du port 1. Trouvez à quelles broches sont connectées les diodes LED1 et le bouton.

Exercice 8 Modifiez votre programme pour faire clignoter toutes les diodes simultanément. En plus de P1DIR on mettra P1SEL à 0.

GPIO ou périphérique interne

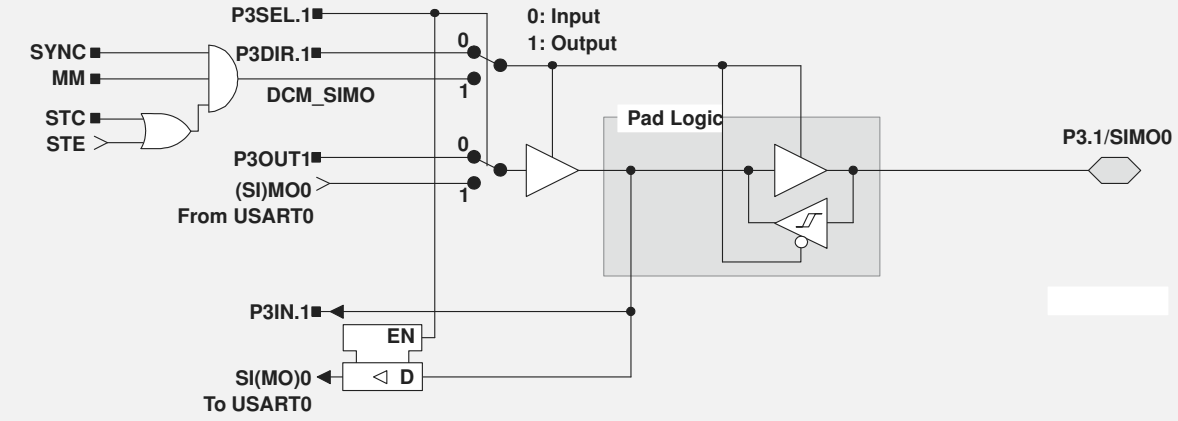
Sur tous les MSP430 les ports sont multiplexés, soient ils sont utilisés comme port d'entrée/sortie généraux (GPIO) soient ils sont utilisés pour adresser un périphérique, soit de l'interieur (du MSP430) soit de l'extérieur (par l'intemédiaire du pad externe). La macro utilisée pour cela est P1SEL (pour le port 1, voir ci dessous), certains ports (comme le port 1) peuvent aussi être utilisé pour générer une interuption lorsqu'un signal arrive sur le port. Les macros utilisées pour cela sont P1IIE, P1IES et P1IFG, nous les verrons plus tard.

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

La figure ci-dessous, extraite de la doc du MSP430F149 détaille le mécanisme exact du port 3.1 qui peut être configuré soit en GPIO soit pour recevoir le signal SIMO d'un protocole UART (que nous étudierons plus tard).

input/output schematic (continued)

port P3, P3.1, input/output with Schmitt–trigger



Exercice 9 Sur quelle documentation et à quelle page trouvez vous ce à quoi va être utile le P1.0 et P1.1 sur le MSP430 de l'EZ430?

2.2 GPIO en entrée: les boutons

On va maintenant s'intéresser au bouton poussoir (un peu caché sur le cache en plastique, en général on peut l'accéder avec un stylo bille). Comme il est connecté à une broches GPIO du MSP430, on va pouvoir venir *scruter* son état depuis le logiciel, ce qui va nous permettre d'écrire des programmes interactifs.

Exercice 10 Sur le schéma électrique, retrouvez le bouton S1. À quelle broche GPIO est-il connecté? Quelle est la valeur envoyée sur le port quand le bouton est pressé.

La documentation du MSP430f2274 (p 58) explique comment connaître l'état par défaut du GPIO correspondant au bouton lorsqu'il est ouvert: il y a une resistance de rappel (pull-up/down resistance enable P1REN) qui indique qu'il y a une valeur par défaut lorsque le circuit est ouvert et que cette valeur par défaut est la valeur positionnée sur P1OUT. Pour configurer correctement le bouton de l'EZ430, il faut donc positionner le

bit 2 de P1REN à 1, et le bit 2 de P1OUT à 1. Danscette configuration la valeur lue sur le port du bouton (P1.2) lorsque le bouton n'est pas appuyé est '1'.

comment positionner un bit sur un registre d'un octet

Pour positionner un bit dans un registre d'un octet, si on utilise une affectation simple, on écrase les autre bits avec une valeur arbitraire. Pour ne modifier qu'un bit, on doit utiliser un masque et utiliser un '*et*' bit à bit (pour positionner a 1), ou un '*ou*' bit à bit (pour positionner à 0).
Par exemple: positionner le bit 2 de P1REN à 1:
`P1REN = P1REN | 0b100` ou simplement: `P1REN |= 0b100`
Positionner le bit 3 de P1OUT à 0:
`P1OUT = P1OUT & ~0b1000` ou de manière équivalent `P1OUT &= ~0x4`

Exercice 11 Dans votre programme, avant la boucle infinie, rajoutez un `while` qui attend un appui sur un bouton pour allumer une diode:

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

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

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

Exercice 12 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 all LEDS on  */

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

         /* turn green LED off and red LED on  */

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

         /* turn RED LED off and green led on  */

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

    }
}
```

Faites valider par un enseignant. on remarque que l'utilisation du bouton en detectant les phases avec une boucle while est délicate, on l'utilisera plutôt par l'intermédiaire des interuptions.

A Schematic de la carte EZ430

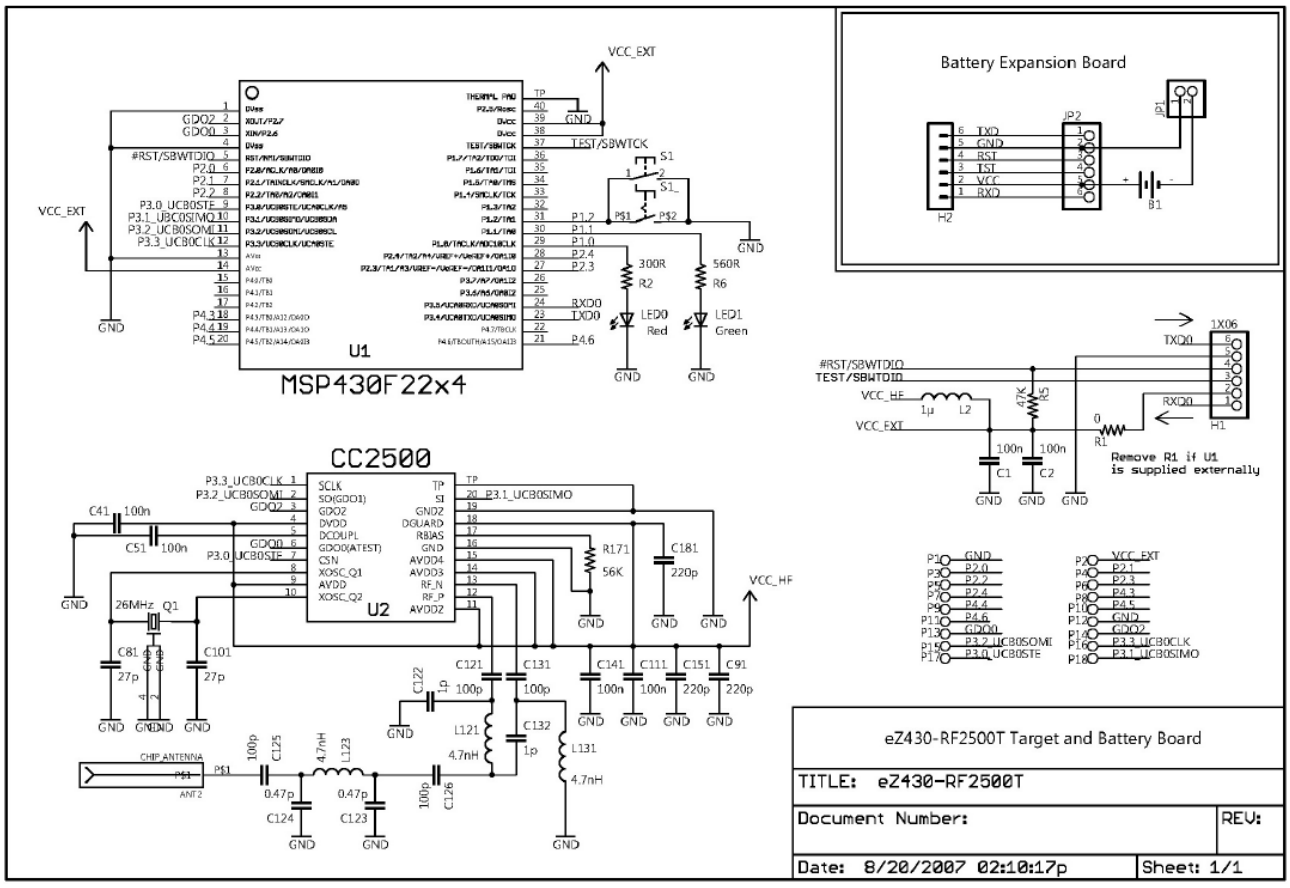


Figure 1: Schema de la carte EZ430 (docEZ430.pdf, p 16)