MICROINFORMATIQUE

NOTE D'APPLICATION 2

PREMIERS PROGRAMMES EN C POUR LE MODULE EZ430

Institut d'
Automatisation
Industrielle

Table des matières

1. Documents de référence

Les documents de références pour cette note d'application sont :

1. ***MSP430x2xx Family User's Guide*** :
   http://php.iai.heig-vd.ch/~lzo/micro/down/MSP430x2xx Family.pdf
2. ***MSP430 IAR Embedded Workbench® IDE User Guide***:
   http://php.iai.heig-vd.ch/~lzo/micro/down/EW430_UserGuide.pdf
3. ***Slides Numération***.
4. ***Slides Outils de développement.***
5. ***Slides Introduction au TI MSP430***
6. ***Slides Unité de traitement, Unité Arithmétique et logique, Entrées-sorties, Interruptions***

2. Objectifs de cette note d'application

Les diverses activités décrites dans cette note constituent une initiation pratique à la programmation en C du MSP430. La séquence de travaux proposés est conçue pour développer par étapes à la fois une pratique de base de la programmation d'un microcontrôleur et une première utilisation des composants clé du MSP430.

Toutefois l'étudiant-e qui aurait déjà des connaissances et/ou expériences dans ses domaines peut aussi sauter certains programmes de taches plus simples et passer directement à la réalisation des programmes demandés à la fin de chaque chapitre et qui généralement combinent plusieurs aspects dans un même code.

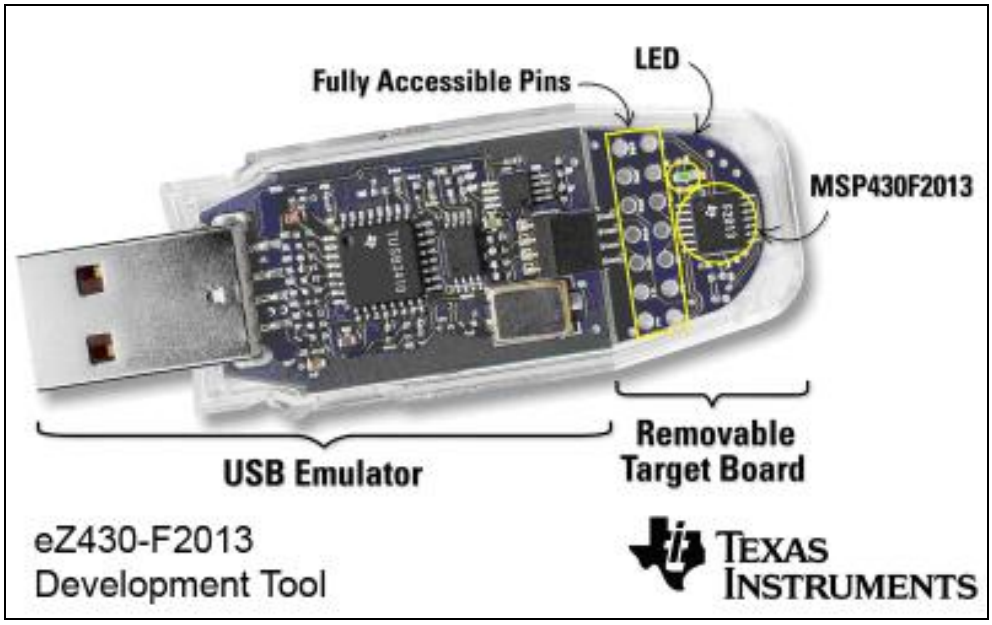
De toute manière il est **important de lire très attentivement** chaque chapitre et ligne de cette note et de comprendre chaque détail des exemples.

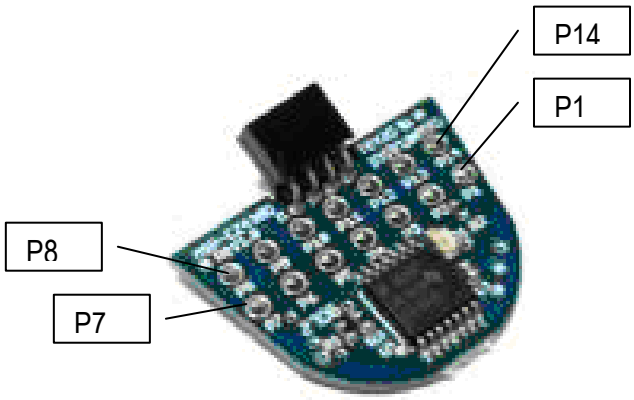
3. Paramétrages

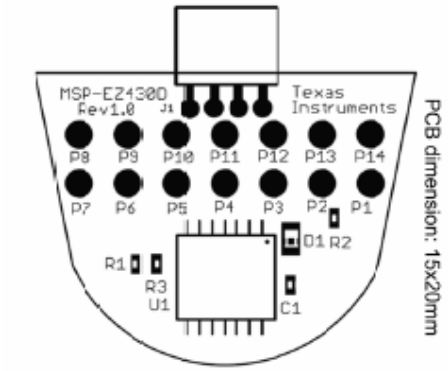
Les pilotes du FET Debugger et du stick MSP430-EZ430 doivent être installés sur le PC.
La création d'un projet en mode debugger se fait essentiellement de la même manière qu'en mode simulation.
Il faut veiller à que **dans chaque nouveau projet ouvert** les paramètres suivants soient validés (menu Projet -> Options) :

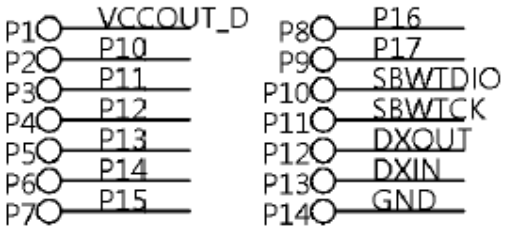
- Dans *General Options* :
  - A l'onglet *target* : MSP430F2012 (ou, si le cas MSP430F013, **vérifiez le type de composant utilisé !**)
  - A l'onglet *Library Confiiguration* : CLIB
- Dans *Debugger* :
  - Driver : FET Debugger
  - Dans *FET Debugger*, Connection : Texas Instrument USB-IF

4. Pinout des modules EZ430-F2012 et F2013





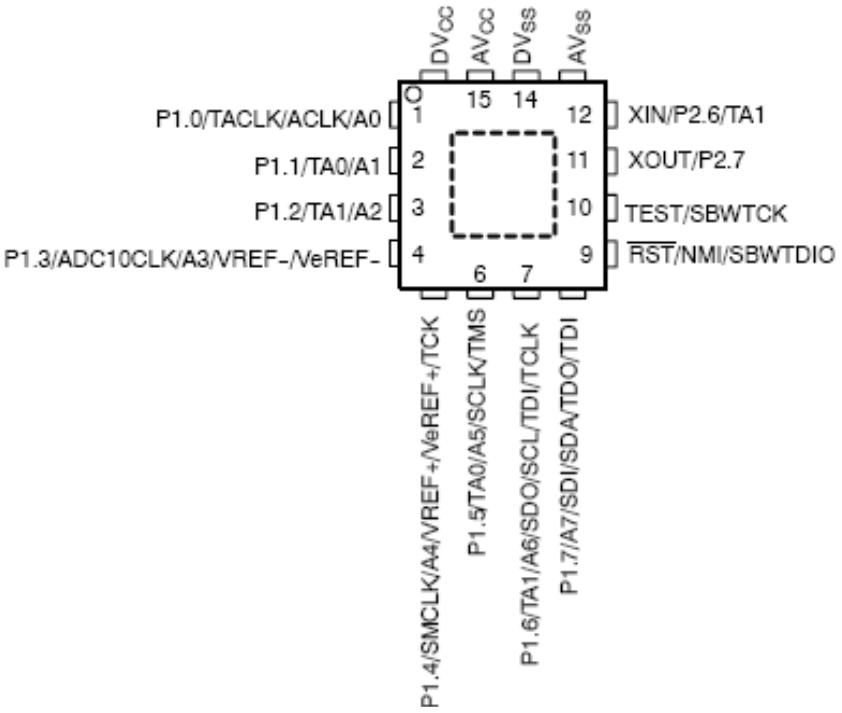




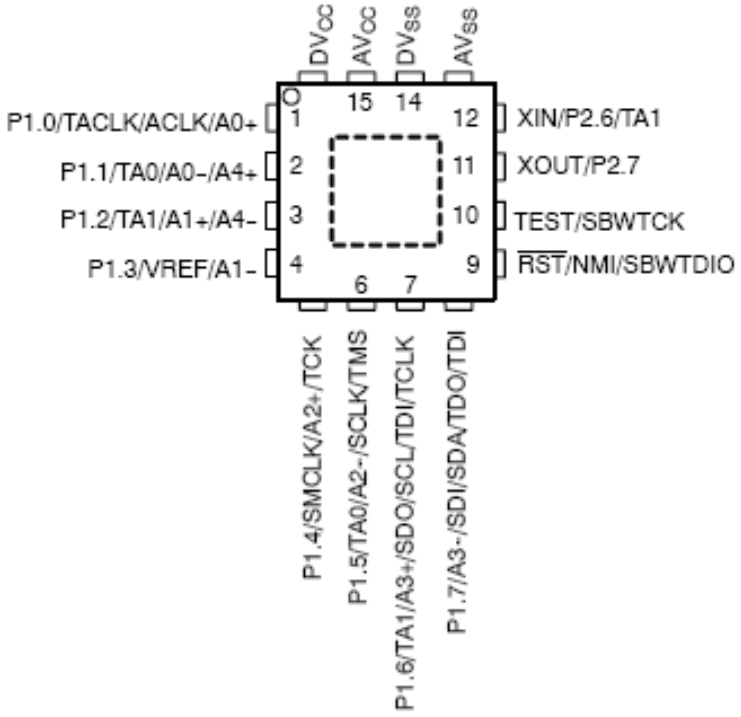
Le module à aussi une **LED**, associé au port P1.0.

Il y a deux modèles de microcontrôleurs légèrement différents associables à la clé USB EZ340 :

- **MSP430F2012**



- **MSP430F2013**

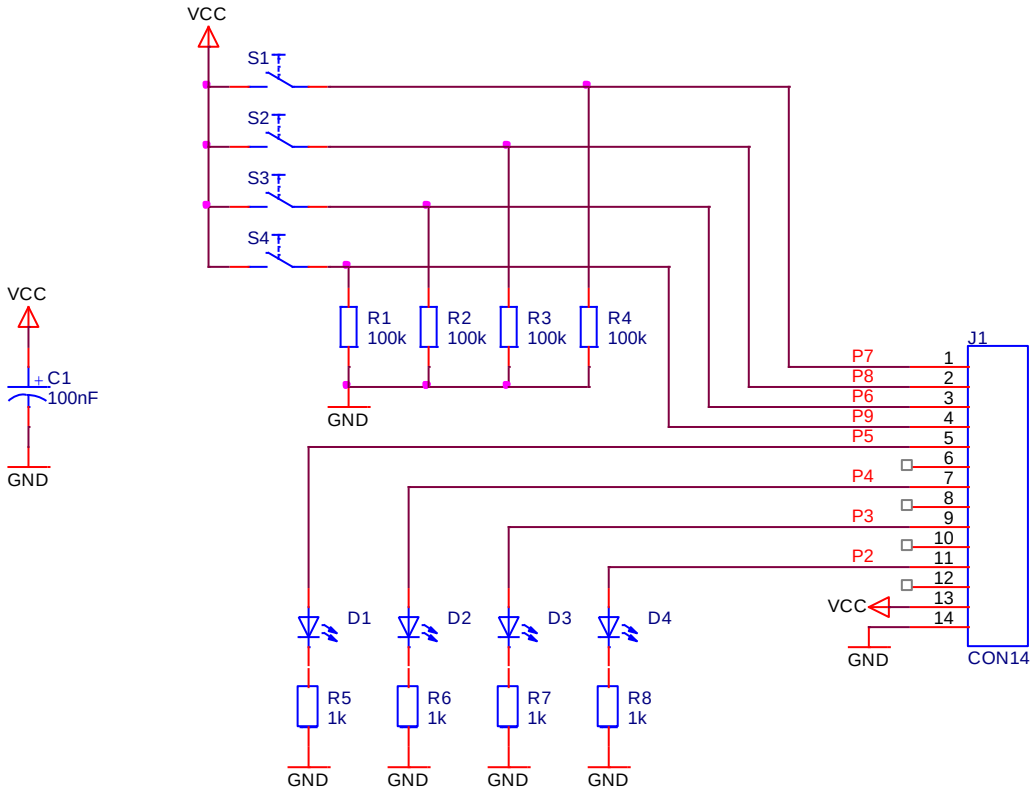


Dans les deux cas le port P1.0, qui se trouve sur la broche 2, est branché en sortie aussi sur une LED interne. On notera que ce type particulier de MSP430 a seulement le port P1 ainsi que deux broches du port P2 (P2.6 et P2.7) comme GPIO disponibles.

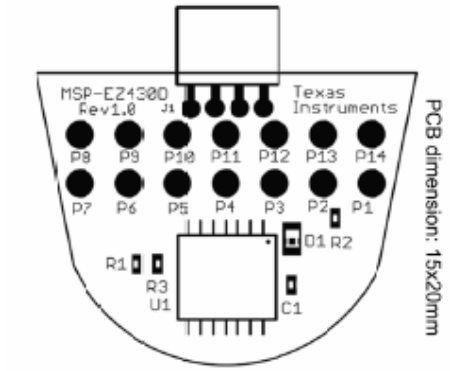
5. Circuit d'expérimentation

Pour ces premiers essais de programmation nous utilisons un petit circuit d'expérimentation relié au module eZ430 par un câble plat. Il comporte :

- 4 LEDs, associés aux ports P1.0 à P1.3.
- 4 boutons poussoirs, associés aux ports P1.4 à P1.7 : **attention**, les boutons ne sont pas câblés dans l'ordre avec les ports du eZ430, voir le tableau à la page suivante.



Le tableau suivant donne les correspondances entre les broches sur le eZ430, les registres GPIO et les éléments câblés sur le petit circuit d'expérimentation.



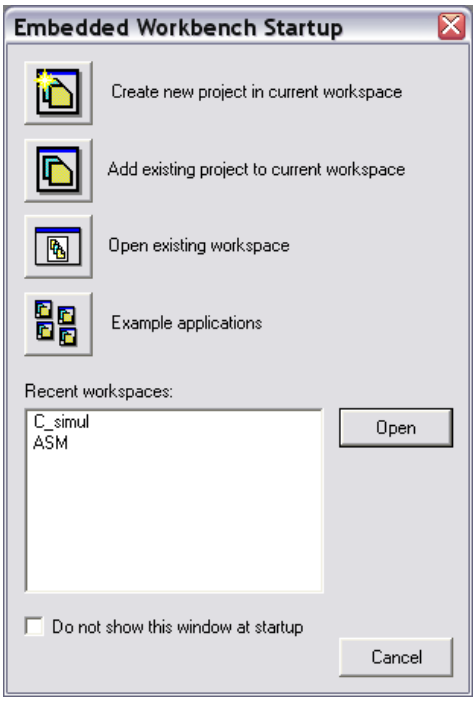
Broche sur le eZ430	Registre GPIO	LED	Bouton
P2	P1.0	D4	
P3	P1.1	D3	
P4	P1.2	D2	
P5	P1.3	D1	
P6	P1.4		S3
P7	P1.5		S1
P8	P1.6		S2
P9	P1.7		S4

6. *Création d'un workspace «EZ430» pour plusieurs projets*

1. Il est important **bien organiser l'espace de travail** pour la programmation du MSP430 : en principe vous devriez avoir déjà réservé un répertoire appelé **MSP430** dans lequel se trouvent déjà les applications en mode simulateur en assembleur (asm) et C.
2. Vous créerez ensuite **dans MSP430** un sous-répertoire **EZ430** et dans celui-ci un premier répertoire de projet (par exemple appelé **Projet_1**). On aura donc l'arborescence de répertoires suivante:

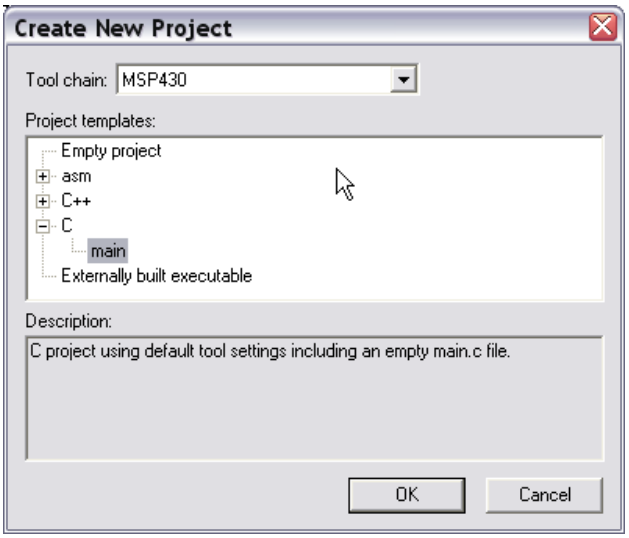
   MSP430 → EZ430 → Projet_1

3. Lancer IAR :



(probablement les workspaces ASM et C_simul sont préexistants et apparaissent dans votre fenêtre)
cliquez **Cancel**

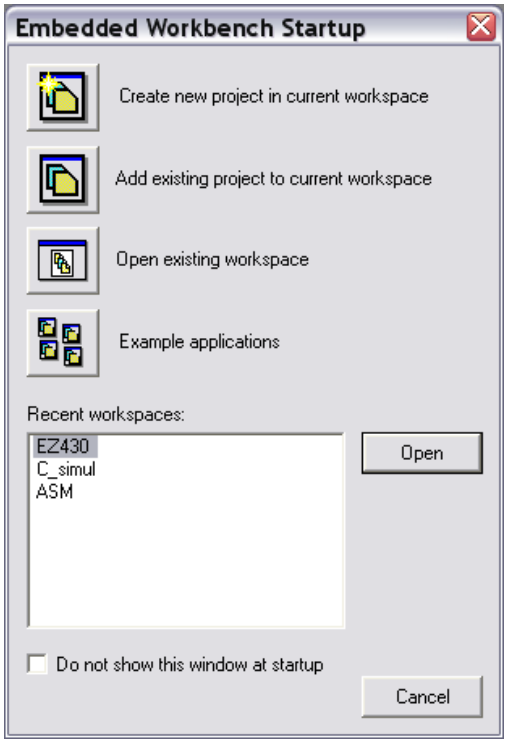
4. Créer un nouveau projet en C:



5. Et enregistrer le nouveau projet dans le répertoire **MSP430 / EZ430 / Projet_1**, sous le nom « LED_simple ».

6. Il faut maintenant encore sauver le nouveau Workspace:
sélectionnez **File → Save Workspace**. Donnez le nom EZ430 au Workspace (comme son répertoire) et sauvez-le dans le répertoire prédéfini (*MSP430 / EZ430*). Contrôlez ensuite qu'un fichier *EZ430.eww* a bien été créé dans ce répertoire.

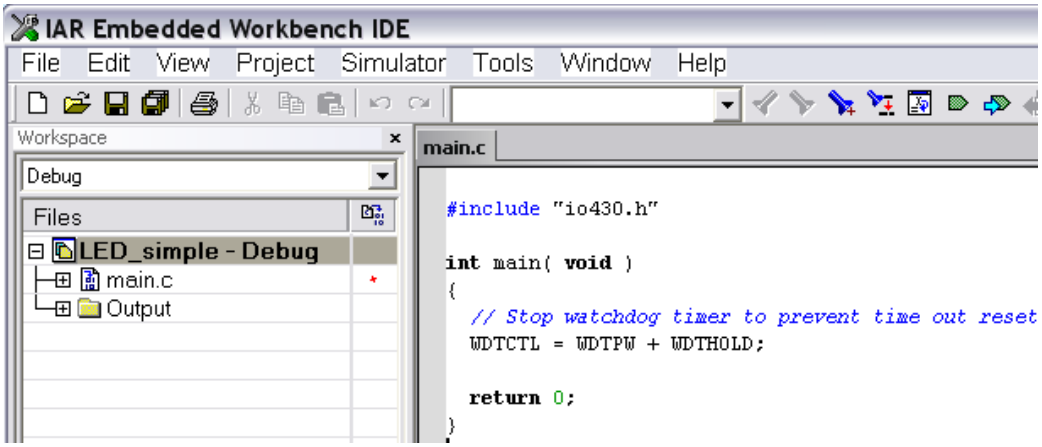
7. Ensuite, chaque fois qu'on relancera IAR, la fenêtre de démarrage (*Startup*) présentera le workspace EZ430.



Ouvrez-le et on accéderera directement au workspace avec tous les programmes réalisés entre temps.

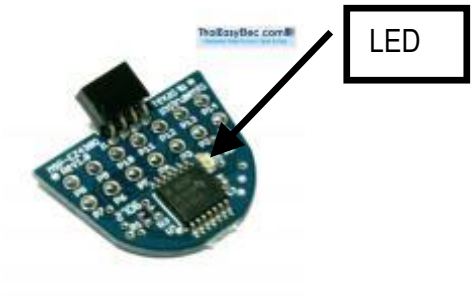
7. Le premier programme

Sélectionner le projet **LED_simple**:



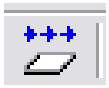
Dans l'éditeur, copier le programme suivant qui fait clignoter la LED sur le module eZ430 associé au port P1.0.

```
// Clignotement de la LED interne
//
*********************************************
#include "io430.h"

int main( void )
{
  unsigned int i;

  // Stop watchdog timer
  WDTCTL = WDTPW + WDTHOLD;

  P1DIR |= 0x01;                // on adresse P1.0 vers sortie (output)

  while (1) {
    P1OUT ^= 0x01;             // toggle P1.0 avec exclusive-OR
    for (i=0 ; i<10000 ; i++);
  }
}
```



Compiler (*make*) et lancer le mode *Debug*

et exécuter le programme.



Vérifiez que le programme est bien sauvé dans le fichier *main.c* du répertoire *MSP430 / EZ430 / Prog_1*.

Comprenez-vous **dans le détail** chaque instruction ?

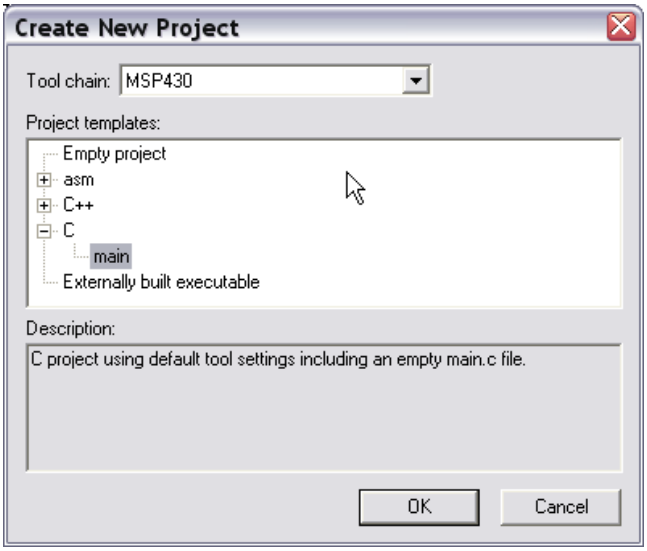
Modifier ensuite **la fréquence de clignotement**.

8. Création d'autres projets dans le même Workspace

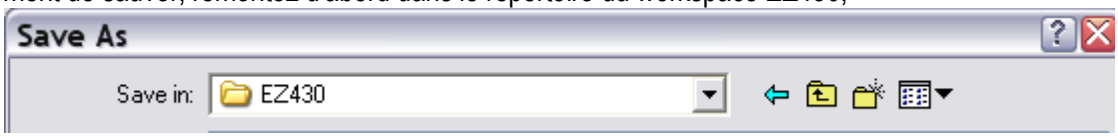
En règle générale, vous devrez **créer un nouveau projet** dans **un répertoire distinct** pour chaque nouvelle application et programme demandée.

Dans le **même workspace** EZ430, ajouter un nouveau projet « LED_switch » pour la première des applications décrite au prochain chapitre.

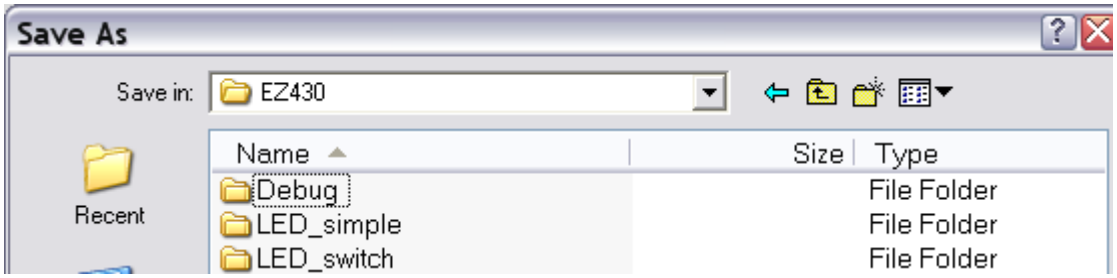
Pour cela sélectionnez le menu *Project -> Create New Project*. Sélectionnez *C -> main*.



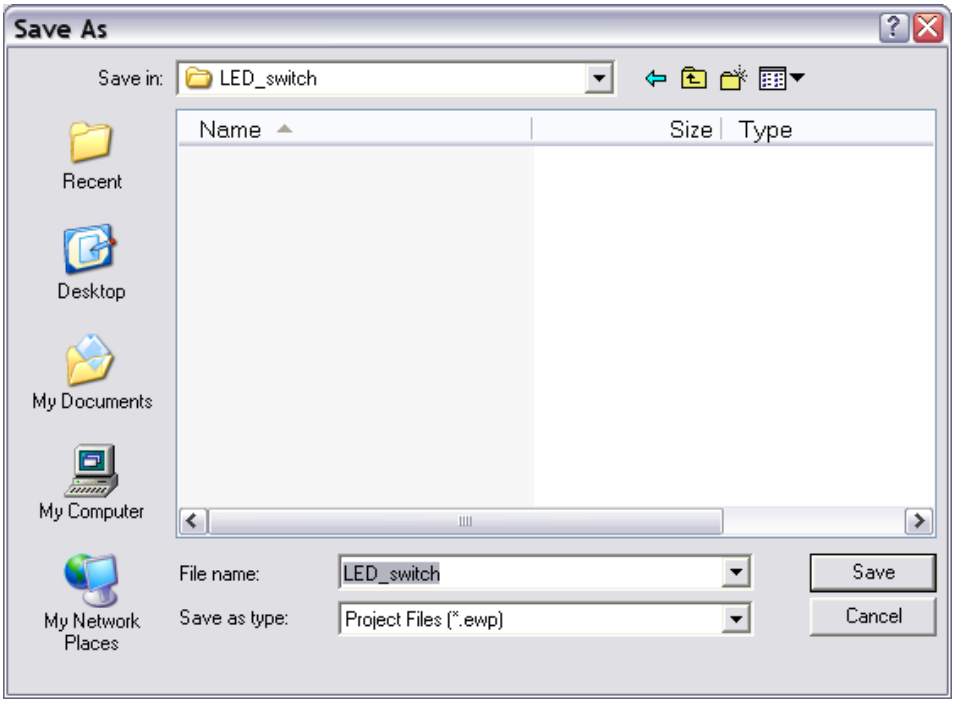
Au moment de sauver, remontez d'abord dans le répertoire du workspace EZ430,



Cliquez sur  et **créer un nouveau répertoire** « LED_switch ».



Allez dans ce nouveau répertoire et insérer le nom du projet.



On aura ainsi créé le fichier de projet ***LED_switch.ewp*** dans un répertoire de même nom:

Attention :
pour chaque nouveau projet il faut généralement que les paramètres suivants soient **validés à nouveau** (menu Projet -> Options) :

- Dans *General Options* :
  - A l'onglet *target* : MSP430F2012
  - A l'onglet *Library Confiiguration* : CLIB
- Dans *Debugger* :
  - Driver : FET Debugger
  - Dans *FET Debugger*, Connection : Texas Instrument USB-IF

On répéte qu'en règle générale, vous devrez **créer un nouveau projet** dans **un répertoire distinct** pour chaque **nouvelle application** et programme demandés.

Ceci également quand vous effectuez des variantes.
Sinon la version précédente sera **effacée** !

9. Programmation simple des GPIO

1. Copier la source du programme LED_simple dans le nouveau projet **LED_switch**, créé au chapitre précédent.

   Modifier ensuite le programme pour qu'on allume la LED sur le EZ430 associé au port P1.0, quand on presse un switch-poussoir (sur le circuit d'expérimentation) connecté au port P1.4 (broche 6).

   Par exemple par la séquence

```
if ((P1IN & 0x10) != 0)
  P1OUT = 0x01;
else
  P1OUT = 0x00;
```

   dans la boucle `while(1)`.

   Mais il est aussi possible de réaliser la même programmation par **une seule ligne de programme** en utilisant des opérateurs binaires (&, |, ^, etc.) . laquelle, par exemple ?

   Il est également possible d'utiliser les champs de bits définis dans le fichier inclus **io430x20x2.h** .

   Ici ces définitions

```
__no_init volatile union
{
  unsigned __READ char P1IN;  /* Port 1 Input */

  struct
  {
    unsigned __READ char P0IN_0  : 1;
    unsigned __READ char P0IN_1  : 1;
    unsigned __READ char P0IN_2  : 1;
    unsigned __READ char P0IN_3  : 1;
    unsigned __READ char P0IN_4  : 1;
    unsigned __READ char P0IN_5  : 1;
    unsigned __READ char P0IN_6  : 1;
    unsigned __READ char P0IN_7  : 1;
  } P1IN_bit;
} @ 0x0020;


__no_init volatile union
{
  unsigned char P1OUT;  /* Port 1 Output */

  struct
  {
    unsigned char P1OUT_0        : 1;
    unsigned char P1OUT_1        : 1;
    unsigned char P1OUT_2        : 1;
    unsigned char P1OUT_3        : 1;
    unsigned char P1OUT_4        : 1;
    unsigned char P1OUT_5        : 1;
    unsigned char P1OUT_6        : 1;
    unsigned char P1OUT_7        : 1;
  } P1OUT_bit;
} @ 0x0021;
```

2. Ouvrir un **nouveau projet** et réaliser un programme qui fait ainsi que quand on presse un bouton, la **LED reste allumée**. Si on presse ensuite le même bouton, **elle s'éteint**. On peut faire cela par des boucles `while()`, par exemple :

```
while(P1IN & 0x10);     // waits here while button isn't depressed
while(!(P1IN & 0x10));  // waits here while button is pushed in

P1OUT ^= 0x01;
```

   Pouvez-vous expliquer le déroulement de ce programme ?

3. Réaliser un programme qui lit l'état des **quatre switches** adressés par P1.4 à P1.7 et le répercute sur les **quatre LEDs** associés respectivement aux ports P1.0 à P1.3. Par exemple, quand l'utilisateur presse le premier bouton (à partir de la droite), la LED correspondante (la première à partir de la droite) s'allume. Quand ensuite on presse à nouveau le même bouton la LED s'éteint.

   a. Programmer la direction des divers P1.X (PxDIR)

   b. Répercuter chaque P1IN.n sur un correspondant P1OUT.m

   Ici aussi the opérateurs binaire &, |, ^, <<, >> peuvent être très utiles.

   Attention : prendre en compte le fait que les boutons ne sont pas câblés dans le même ordre que les ports P1.4 a P1.7 (voir tableau à la page 7).

4. Réaliser ensuite une variante qui lit l'état des switches adressés par P1.4 à P1.7 et les répercute sur les LEDs **en sens inverse**. Quand l'utilisateur presse le premier bouton (à partir de la gauche), la LED correspondante doit être la première à partir de la gauche, et ainsi de suite.

5. Les GPIO en sortie sont souvent utilisées pour commander des moteurs pas-à-pas ou brushless (en PWM). Dans ces cas les sorties correspondantes aux différentes bobine du moteur sont séquencées de manière à faire tourner le moteur dans le sens et avec la vitesse voulue.

   Réalisez un programme qui **allume et éteint les 4 LEDs une après l'autre en séquence de chenillard**, une après l'autre, après la dernière le cycle recommence avec la première.

   Pour l'instant on obtiendra un temps d'attente pour le défilement et le flashing des LEDs par des simples boucles d'attente de type :

```
for (i=0 ; i<30000 ; i++) ;
```

6. Finalement, **combinons tous ces aspects**: on veut un programme avec la **spécification suivante** :
   a. Quand on presse le **bouton 1**, les 4 LEDs s'allument, on le re-presse et elles s'éteignent.
   b. Quand on presse le **bouton 2**, les 4 LEDs s'allument et s'éteignent en séquence de chenillard, une après l'autre, après la dernière le cycle recommence avec la première
   c. Quand on presse le **bouton 3**, les 4 LEDs flashent ensemble régulièrement.
   d. Dans tous les cas pour passer d'un mode à l'autre, il faut d'abord l'éteindre en repressant le même bouton qui a commandé l'allumage voulu.

   Pour ces exemples on réalisera un temps d'attente pour le défilement ou le flashing des LEDs par des simples boucles de type :

```
for (i=0 ; i<30000 ; i++) ;
```

   Les trois modes d'allumage des LEDs peuvent, par exemple, être géré par un *switch* :
   (mais plusieurs autres structures (*if, while*) sont également possibles …) , par exemple :

```
#include "io430.h"
#define WAIT_UNTIL_SOME_P1_NOT_ZERO    while (!(P1IN & 0xF0))
#define WAIT_UNTIL_ALL_P1_ARE_ZERO     while (P1IN & 0xF0)

int main( void )
{
  int i,j;
  unsigned char p1_in;

  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;

  P1DIR = 0x0F;       // P1.0-3 output
  P1OUT = 0;

  while(1)
  {
    WAIT_UNTIL_SOME_P1_NOT_ZERO;
    p1_in = P1IN ;

    WAIT_UNTIL_ALL_P1_ARE_ZERO;

    switch (p1_in & 0xF0) {

    case 0x10:   // premier bouton
      do
      { P1OUT = 0x0F;
        for (j=0 ; j<30000 ; j++);
      } while (!(P1IN & 0x10));  // reboucle tant que P1.4 n'est pas activé
      P1OUT = 0;
      break;

    case 0x20:  // deuxième bouton, doit activer une séquence en chenillard
      ...

      break;

    case 0x40:   // troisième bouton, ...
      ...

      break;
    }
    P1OUT = 0;  // on éteint tout

    WAIT_UNTIL_ALL_P1_ARE_ZERO;
  }
}
```

10. Interruptions

A guise d'introduction aux interruptions on programmera des *interrupts* sur les bits 4-7 du port P1 qui sont associés aux boutons de la carte d'expérimentation.

La préparation dans main() sera du type :

```
#include "io430.h"
#include "intrinsics.h"

int main( void )
{
  // Stop watchdog timer to prevent time out reset
  WDTCTL = WDTPW + WDTHOLD;

  P1IES &= 0x00;   // interrupt P1.x au flancs montants
  P1IE  |= 0xF0;   // enable interrupt P1.x
  P1IFG &= ~0xF0;  // on s'assure que les flags sont bien à zéro
  __enable_interrupt();

  while(1)
  { . . .
    . . .
  }
```

La routine d'interruption est écrite comme :

```
// routine d'interruption qui survient lorsqu'un des P1.x a un flanc
#pragma vector=PORT1_VECTOR
__interrupt void Port1_ISR (void) // PORT1ISR (void)
{
  if (P1IFG & 0x10) // par exemple, test si l'interruption est causé par P1.4
   ...faites «toggler» une LED ...

  P1IFG = 0;        // il FAUT remettre le flag à zéro !
}
```

10.1 EXERCICES

1. Réaliser un programme qui a la même spécification que l'exercice 9.3 – il lit l'état des switches adressés par P1.4 à P1.7 et le répercute sur les LEDs associées à P1.0 – P1.3 - mais en déclenchant l'allumage des LEDs par une interruption programmée sur le port 1.

2. Ensuite réaliser une application qui **reprogramme par des interruptions** l'application 9.6 à la page précédente (LEDs actives en trois modes différents déclenchés par les switches P1.4 à P1.6), ce qui permet d'éviter les boucles d'attente entre pression et levée établies avec la méthode précédente.

   L'utilisation des interruptions permet aussi facilement de passer d'un mode à l'autre sans devoir l'éteindre préalablement par le même bouton.

   Qui aurait le temps et voudrait quelles points bonus à valoir sur le note pourrait aussi programmer une « danse des LEDs », déclenchée par le bouton S4 (P1.7). Bonus extra pour la séquence plus amusante !

11. Introduction aux timers

Le MSP430 contient un système d'horloges flexible conçu spécialement pour une utilisation avec une batterie. Le système d'horloges met à disposition de l'utilisateur plusieurs sources différentes selon le modèle du MSP430 et en fonction des besoins en consommation et en fréquence :

– ACLK (Auxillary clock) pour les applications qui doivent consommer très peu et qui n'ont pas besoin de très hautes fréquences (32'768 Hz max). Dans le cas du F2012/13 le ACLK peut utiliser soit un oscillateur interne à basse fréquence (VLO – *very low oscillatior*, fréquence 12 kHz), ou recevoir le signal d'un quartz externe.

– MCLK (Master clock) qui est utilisée par le CPU, utilisée pour les applications qui ont besoin de hautes fréquences

– SMCLK (Sub main clock) utilisable pour les modules périphériques, qui par défaut a une fréquence de 32 · 32'768 = 1'048'576 Hz.

Comme le MCLK, le SMCLK utilisera généralement l'oscillateur interne rapide DCO – *digitallly controlled oscillator* capable, en agissant sur des registres appropriés, jusqu'a 16 Mhz.

11.1 TIMERS INTERNES

Le MSP430 possède en général deux Timer internes: Timer A et Timer B qui ont presque les mêmes caractéristiques, le B ayant quelques possibilités en plus.

Les modèles F2012 et F2013 n'ont que le Timer A.

Nous allons décrire ici le principe très simplifié de fonctionnement du Timer A pour comprendre comment le temps est géré. Le Timer A contient deux registres TAR (compteur) et TACCR0 (capture/comparaison) et est relié à une source d'horloge configurable.

Plusieurs modes de comptage sont au choix:

- mode continu (CONTINUOUS) dans lequel le TAR est incrémenté jusqu'a sa valeur maximale 0xFFFF avant de générer une interruption,
- mode montant (UP) dans lequel le TAR est incrémenté jusqu'a la valeur contenue dans TACCR0 avant de générer une interruption,
- mode alterné (UP/DOWN) dans lequel TAR est incrémenté jusqu'à la valeur de TACCR0 puis décrémenté jusqu'a TACCR0 avant de générer l'interruption.

L'incrémentation ou la décrémentation est déclenchée par des fronts montants sur l'entrée du registre de comparaison en fonction des cycles d'horloge.

Le timer possède d'autres registres TACCR1, TACCR2 (le timer B en contient 6 au total) permettant de générer des interruptions à des intervalles différents. Plusieurs flags sont modifiables pour contrôler le timer, par exemple pour

- sélectionner l'horloge (ex. ACLK ou SMCLK) et la source (ex. VLO ou DCO)
- définir le mode de contage
- définir la fréquence
- mais aussi remettre le TAR à zéro, ou activer les interruptions, etc.

11.2 EXEMPLES

L'exemple suivant illustre un simple Timer dans un programme qui fait clignoter, en déclenchant des interruptions par le Timer_A, la LED interne de P1.0 à une fréquence de 1 Hz.

```
#include "io430.h"
#include "intrinsics.h"

void main(void)
{ float freq;

  WDTCTL = WDTPW + WDTHOLD;                 // Stop WDT

  P1DIR |= 0x01;                            // P1.0 output
  P1OUT = 0;
  CCTL0 = CCIE;                             // CCR0 interrupt enabled

  BCSCTL1 |= DIVA_0;                        // ACLK/1
  BCSCTL3 |= LFXT1S_2;                      // ACLK = VLO
  freq = 2; // Hz
  CCR0 = 12000 / freq - 1;
  TACTL = TASSEL_1 + MC_1;                  // ACLK, upmode

  __enable_interrupt();
  while (1);
}

// Timer A0 interrupt service routine
#pragma vector=TIMERA0_VECTOR
__interrupt void Timer_A (void)
{
  P1OUT ^= 0x01;                            // Toggle P1.0
}
```

Ici c'est le ACLK « sourcé » sur le VLO à environ 12 kHz qui est utilisé.
Exécutez le programme et vérifiez au moyen de la documentation (*MSP430x2xx Family User's Guide, chapitre 12*) et également avec les définitions dans le fichier *io430x20x2.h* **toutes les significations** des divers registres utilisés ici: par exemple avec un tableau explicatif dans votre rapport.

Il est aussi possible de réaliser un clignotement de 1 Hz par le SMCLK (dont la fréquence par défaut est ~1 MHz) en remplaçant la section centrale du programme par

```
  BCSCTL2 |= DIVS_3;                        // SMCLK source = DCO / 8
  CCR0 = 65535;
  TACTL = TASSEL_2 + MC_1;                  // SMCLK, upmode
```

Expliquez le sens des paramètres et aussi pourquoi on ne pourrait pas obtenir une fréquence plus basse que 1 Hz (à moins d'agir en amont sur la fréquence du DCO …).

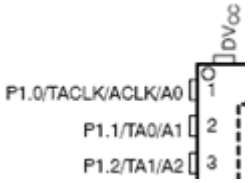
11.3 EXERCICE

Réaliser un programme qui permet de varier la fréquence de clignotement de la LED sur P1.0 au moyen des deux premiers boutons-poussoirs associés respectivement à P1.4 et P1.5. Chaque actionnement du premier bouton, programmé au moyen d'une interruption sur le port correspondant, augmentera la fréquence d'un facteur 2. Le deuxième bouton-poussoir permet de réduire la fréquence également d'un facteur 2 à chaque actionnement.

11.4 RÉALISATION DE PWM AU MOYEN D'UN TIMER

On veut maintenant utiliser un timer pour obtenir le rapport cyclique d'un PWM sur le port P1.2, qui sera ici matérialisé par la luminosité d'une LED.

Se référer au cours sur les GPIO et vérifier que la broche no.3 (P1.2) a comme alternative sur le F2012/F2013 la sortie Timer A1.



Insérer donc dans le préambule du programme le code pour générer un PWM sur la LED D2 :

```
void main(void)
{
  WDTCTL = WDTPW + WDTHOLD;                 // Stop WDT
  P1DIR |= 0x06;                            // P1.1 et P1.2 output
  P1SEL |= 0x04;                            // P1.2 TA1 option
  CCR0 = 512-1;                             // Période du PWM
  CCTL1 = OUTMOD_7;                         // CCR1 reset/set
  CCR1 = 51;                                // CCR1 PWM duty cycle
  TACTL = TASSEL_2 + MC_1;                  // SMCLK, up mode

  P1OUT |= 0x02                             // signal continue sur P1.1 pour
                                            // comparaison

  while (1);
}
```

Décrivez les significations des divers registres utilisés ici: par exemple avec un tableau explicatif dans votre rapport.
Faites ensuite varier le rapport cyclique afin de changer la luminosité de la LED.

11.5 EXERCICE

Réaliser un programme qui permet de varier le rapport cyclique sur la sortie TA1, matérialisée par la LED associée à cette sortie (broche no. 3) au moyen des deux premiers switches-poussoirs associés respectivement à P1.4 et P1.5.

Chaque actionnement du premier bouton, programmé au moyen d'interruptions sur le port correspondant, augmentera le rapport cyclique (donc la luminosité de la LED) entre le minimum de 0%, ensuite 1%, ensuite par incréments de 3%, jusqu'au maximum de 100%. L'actionnement du deuxième bouton-poussoir doit permettre de réduire e rapport cyclique dans le même ordre.

12. Rapport

Le rapport sera une mise au propre de vos propres notes prises durant ce travail: un simple log-book et enregistrement des divers programmes réalisés et testés.

Il inclura pour chaque tache effectuée

- Titre de la tache.
- Copier-coller des *parties significatives* des listings produits.
- **Les réponses à toutes les questions posées**.
- Remarques, résultats, conclusions et suggestions applicables pour chaque cas.