heig-vd
Haute Ecole d'Ingénierie et de Gestion
Du Canton du Vaud

Hes·so
Haute Ecole Spécialisée
de Suisse Occidentale

MICROINFORMATIQUE

NOTE D'APPLICATION 4 (V10)

COMMUNICATION SERIE

MESURE DE TEMPERATURE

MODES A BASSE CONSOMMATION

Institut d'
Automatisation
Industrielle

1. Introduction

1.1 PARAMÈTRAGES

IMPORTANT :
Créez chaque nouveau programme sous forme d'**un projet séparé dans son directoire propre**.

Pour chaque nouveau projet il faut veiller à que les paramètres suivants soient validés (menu Projet -> Options) :

- Dans *General Options* :
  - A l'onglet *target* : MSP430FG4617
  - A l'onglet *Library Confiiguration* : CLIB
- Dans C/C++ Compiler :
  - A l'onglet Optimizations : None (Best debug support)
- Dans *Debugger* :
  - Driver : FET Debugger

1.2 DOCUMENTS DE REFERENCE

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

1. ***MSP430x4xx Family.pdf*** : chapitre 18 – USART Peripheral Interface, SPI Mode.
2. ***EW430_CompilerReference.pdf :*** MSP430 IAR C/C++ Compiler, Reference Guide
3. ***Slides:*** UART, USART – SPI, Low-power modes
4. ***Data sheet*** capteur de température TC77, en annexe à cette note

2. Mesure de température

2.1 BUT DE CET EXERCICE

Le but de cet exercice est de réaliser avec la carte IAI MSP430 une mesure de température à partir d'un capteur lié au microcontrôleur par un bus sériel SPI.

Le port utilisé est le port 4 par conséquent il s'agit du périphérique USART1.

Le capteur est en mode Slave et le microcontrôleur en mode Master.

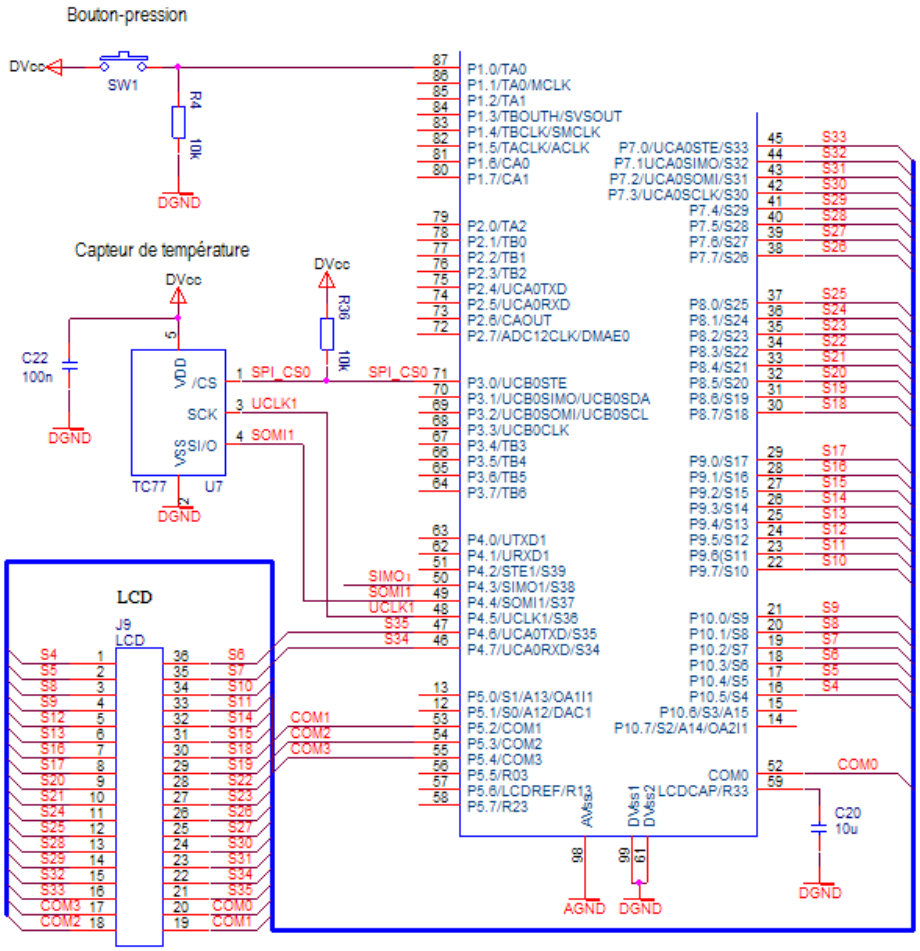


Figure 2-1 : Hardware utile pour l'application

Après traitement, la température doit être affichée sur l'écran LCD avec une résolution de 0.1°C ou 0.1°F. La sélection de l'unité est contrôlée par le commutateur à pression SW1.
Certaines routines sont données d'autres à compléter, une vue d'ensemble de l'application est nécessaire.

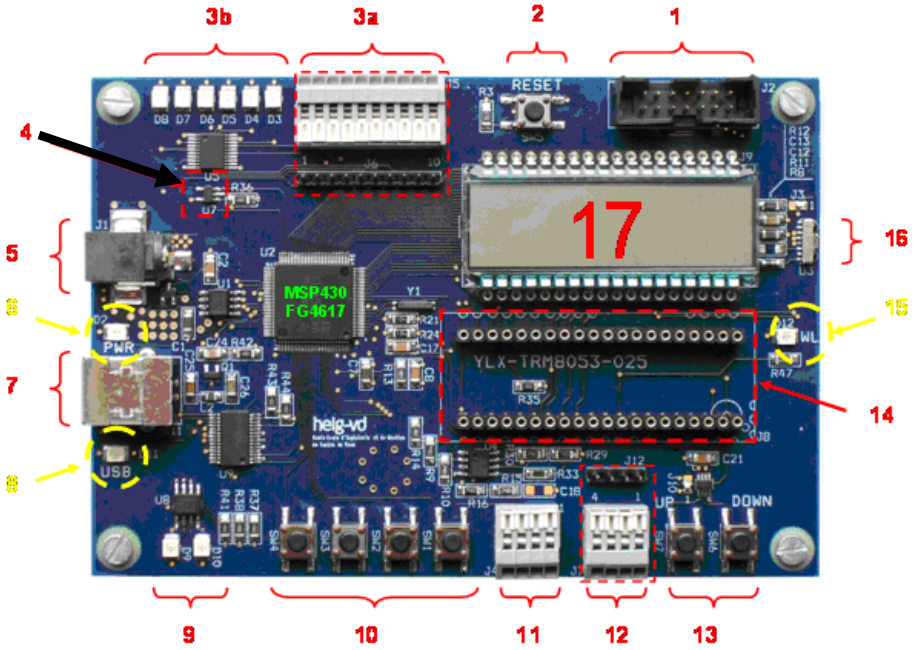


Figure 2-2 : Le capteur de température est indiqué au no.4 .

2.2 DESCRIPTION DU HARDWARE

2.2.1 Caractéristiques du capteur de température

Le capteur de température peut être décrit à l'aide de son schéma bloc. Il s'agit d'une mesure de température par l'intermédiaire de la variation des caractéristiques d'une diode. Un convertisseur Sigma-Delta dont la fréquence de suréchantillonnage est de 30kHz permet d'obtenir une résolution de 13 bits.
Une interface MICROWIRE compatible SPI permet d'atteindre deux registres. L'un pour la configuration du système (mode de travail, mode basse consommation) ainsi que le contrôle du circuit en phase de test et de calibration. Le second registre contient la valeur de la mesure.

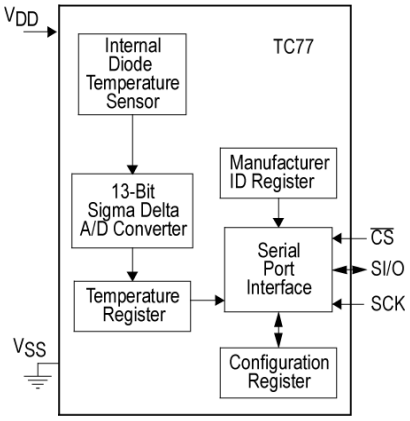


Figure 2-3 : Schéma bloc du capteur de température

La fonction de conversion est donnée par la Figure 2-4.

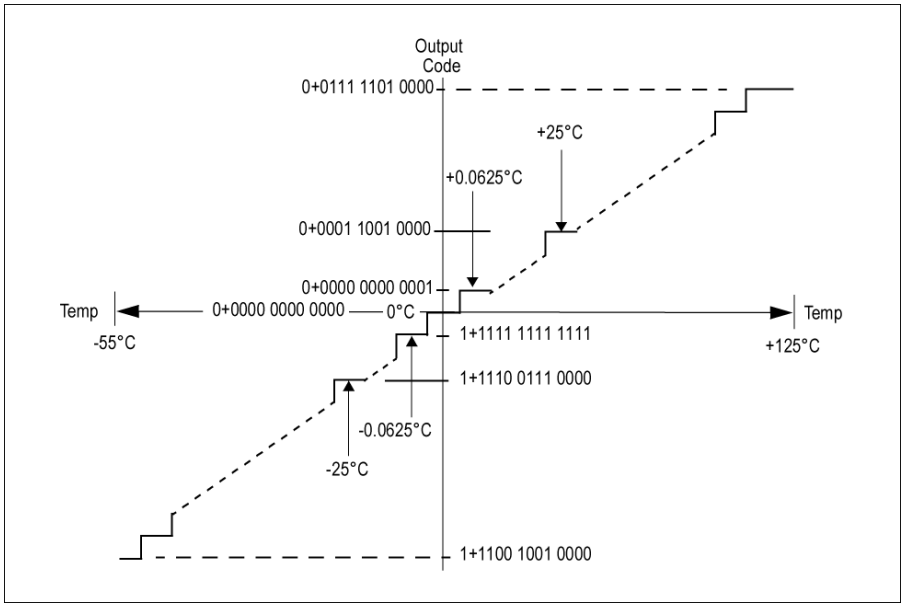


Figure 2-4 : Fonction de conversion

Il s'agit d'un code en complément à deux. La résolution vaut **0.0625°C**.

Les deux bits de poids faibles ne doivent pas être pris en compte (bus en haute impédance). Le bit 2 est toujours à 1, excepté lors de la première conversion suivant une tension d'alimentation dépassant 1.6V (superviseur de tension intégré).

La valeur numérique utile est donc donnée sur les 13 bits de poids forts.

Le tableau ci-dessous montre le codage de la température.

Température	Valeur numérique de sortie [LSB] Binaire				Hexadécimal
+125°C	0011	1110	1000	01zz	07D0
+25°C	0000	1100	1000	01zz	0190
+0.0625°C	0000	0000	0000	11zz	0001
0°C	0000	0000	0000	01zz	0000
-0.0625°C	1111	1111	1111	11zz	1FFF
-25°C	1111	0011	1000	01zz	1E70
-55°C	1110	0100	1000	01zz	1C90

Tableau 2-1 : Table de conversion

Le transfert des données par le bus SPI peut être effectué lorsque le signal $\overline{CS}$ est forcé au niveau bas, ainsi que le signal de l'horloge CLK. Le bit de poids fort (MSB) est placé sur la ligne SI/0 au flanc descendant de $\overline{CS}$, puis chaque bit est placé sur le bus au flanc descendant du CLK. Il faut donc 13 périodes d'horloge pour avec une transmission complète d'une donnée. Les données doivent être capturées sur le flanc montant de l'horloge. Au flanc montant du signal la communication est interrompue. Elle recommencera lors du prochain flanc descendant.

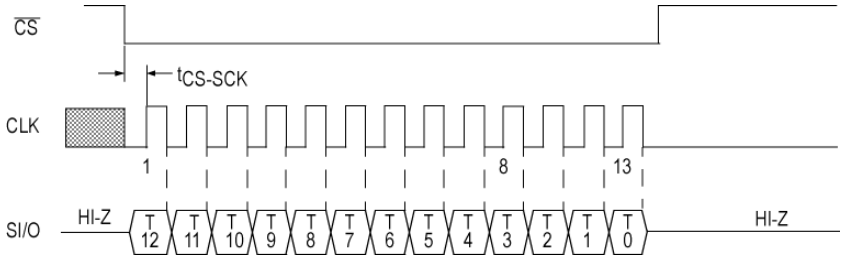


Figure 2-5 : Mode lecture de la température

A l'enclenchement (power up) le capteur de température se trouve en mode de conversion continue de la température.
Il est possible de changer de mode en plaçant le capteur en basse consommation (shutdown). Pour ce faire, il faut forcer le signal $\overline{CS}$ à 0, lire les 13 bits correspondant à une donnée (mesure de température), lire le bit suivant qui sera un 1 si la donnée est valide puis pour les deux derniers périodes de l'horloge, la ligne SI/0 se met en haute impédance. Les trois derniers bits ne font donc pas partie de la donnée.

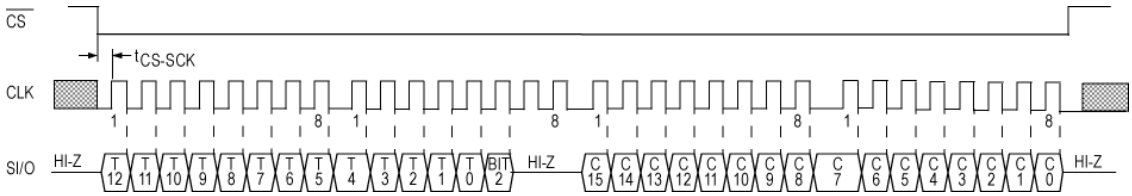


Figure 2-6 : Mode lecture température puis écriture du registre de configuration

Sans changer le signal $\overline{CS}$ il est alors possible d'écrire dans le registre de contrôle.
En écrivant 0x000 le capteur travaillera en mode conversion continue, en écrivant 0xFFFF, le capteur se mettra en mode basse consommation. Une lecture en basse consommation permet de connaître l'ID du fabricant. Pour plus de détail voir le datasheet de Microchip

2.2.2 Caractéristiques du bus SPI du MSP430

Comme le montre la Figure 2-1, seule la ligne SOMI est utilisée pour les données. La ligne correspondant au signal $\overline{CS}$ est issue de la broche P3.0. L'horloge correspond au signal UCLK. LE signal STE n'est pas utilisé puisqu'il y a qu'un seul Master (le capteur de température est considéré comme un Slave)

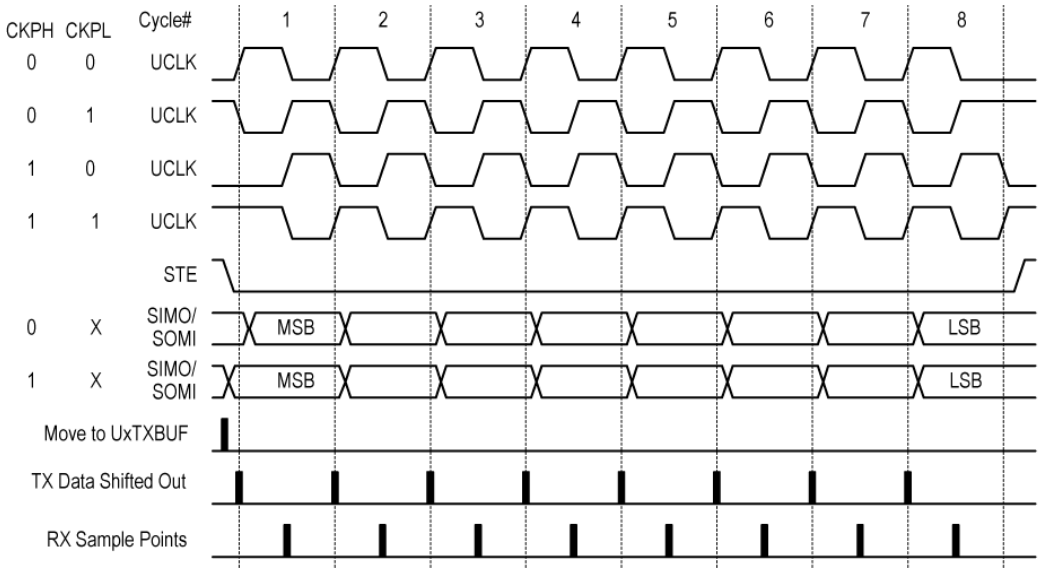


Figure 2-7 : Chronogramme de l'interface USART en mode SPI

2.2.3 Registres de configuration du SPI du MSP430

Les paragraphes suivants donnent des indications sur les registres à configurer au niveau du microcontrôleur pour réaliser l'application : voir en même temps le programme de base à la section 2.3 suivante.

2.2.3.1 Configuration de l'USART1

L'USART1 est utilisé en mode SPI. Le port P4 doit être configuré en tenant compte des spécifications du bus imposées par le capteur de température (TC77). Attention toute la configuration doit se faire avec le flag SWRST à 1 (U1CTL[0]).

2.2.3.1.1 Registre de contrôle de l'USART : U1CTL (0x078)

Les flags **CHAR** (U1CTL[4]), **SYNC** (U1CTL[2]), **MM** (U1CTL[1]) et **SWRST** (U1CTL[0]) doivent être configurés. Les autres bits de U1CTL sont à 0 par défaut et ne doivent pas être modifiés

2.2.3.1.2 Registre de contrôle de transmission de l'USART : U1TCTL (0x079)

Les flags **CKPH** (U1TCTL[7]), **CKPL** (U1TCTL[6]), **SSEL1** (U1TCLT[5])]), **SSEL0** (U1TCLT[5]) et **STC** (U1TCLT[1]). Les autres bits de U1CTL sont à 0 par défaut et ne doivent pas être modifiés.

2.2.3.1.3 Registre de contrôle de réception de l'USART : U1RCTL (0x7A)

Ce registre contient des flags d'erreur, il n'est pas utilisé dans cette application.

2.2.3.1.4 Registre de contrôle de modulation de l'USART : U1MCTL (0x7B)

Le registre U1MCTL doit être forcé à 0x00 lorsque l'USART est en mode SPI.

2.2.3.1.5 Registre de contrôle du Baud rate de l'USART : UxBR0 et UxBR1 (0x07C, 0x07D)

Ces deux registres sont utilisés pour fixé le Baud rate en divisant la fréquence de l'horloge de référence (ACLK = 32'768 Hz dans notre cas). Une division par 2 au minimum est nécessaire (U1BR1 = 0x00, U1BR0=0x02).

2.2.3.1.6 Registre de contrôle de l'activation de l'UART1 : ME2 (0x005)

Le flag **USPIE1** (ME2[4]) permet l'activation du SPI. Les autres bits de ME2 ne doivent en aucun cas être modifiés

2.2.3.1.7 Registre de contrôle d'activation des interruptions l'UART1 : IE2 (0x001)

Le flag **UTXIE1** IE2[5] et le flag **URXIE1** IE2[4] sont des flag de masquage d'interruption. Ils doivent les deux être forcés à 0 afin de s'assurer que les interruptions de l'USART en mode SPI sont masquées.

2.2.3.1.8 Registre des flags d'interruptions l'UART1 : IFG2 (0x003)

Le flag **URXIFG1** IFG2[4] indique qu'un caractère est disponible dans le buffer U1RXBUF.

2.2.3.2 Configuration du port P4

Le port P4 est utilisé pour le bus SPI. Les broches sont assignées de la manière suivante :

- P4.3 $\rightarrow$ SIMO (sortie – pas utilisée pour le capteur de température)
- P4.4 $\rightarrow$ SOMI (entrée)
- P4.5 $\rightarrow$ UCLK (sortie)

Les registres suivants doivent donc être configurés P4DIR (0x20), P4SEL (0x30). Attention seuls les bits correspondant aux broches P4.4, P4.5 doivent être modifié.

2.2.3.3 Configuration du port P3

Le port P3 est utilisé pour générer le signal $\overline{CS}$ (Chip Select) du bus SPI. Les broches sont assignées de la manière suivante :

- P3.0 $\rightarrow$ $\overline{CS}$ (sortie)

Les registres suivants doivent donc être configurés P3DIR (0x01A), P3OUT (0x019), P3SEL (0x01B). Attention seul le bit correspondant à la broche P3.0 doit être modifié.

2.3 EXERCICE DE BASE

Ouvrir un nouveau projet (toujours dans un directoire séparé …)

Copier et exécuter le programme suivant qui affiche la valeur (en bits) du capteur.

Comprendre et commenter **tous les paramètres** qui configurent l'UART-SPI..

Réaliser un projet basé sur la programmation en C comprenant tous les fichiers suivants :

- `main.c` : fichiers contenant le programme principal de l'application
- `LCD.c` et `LCD.h` : fichiers de contrôle de l'affichage
- `CapteurTemp.c` et `CapteurTemp.h` : fichiers de contrôle du capteur de température

Les fichiers LCD.c, LCD.h sont donnés et ne doivent pas être modifiés.

Le fichier CapteurTemp.c comprend les fonctions suivantes :

```
#include "msp430xG46x.h"
// Initialisation de l'USART1 en mode SPI ************************
void SPI_1_init(void)
{
  ME2 |= USPIE1;                    // Enable USART1 SPI mode
  U1CTL =  à configurer voir section 2.2.3.1.1.plus haut ...
  U1TCTL=  à configurer voir section 2.2.3.1.2.plus haut ...
  U1BR0 = 0x02 ;
  U1BR1 = 0x00 ;                     // UCLK/2
  U1MCTL = 0x00 ;                    // no modulation

// Configuration de la ligne CS (Chip Select) P3.0
  P3OUT |= 0x01;
  P3SEL  = 0x00;
  P3DIR |= 0x01;

// Configuration du port P4 pour le périphérique USART1 en mode SPI
  P4DIR &= ~BIT4 ;                   // SOMI1
  P4SEL |= BIT4 ;
  P4DIR |= BIT5 ;                    // UCLK1
  P4SEL |= BIT5 ;
}

// Lecture de la donnée brute du capteur TC77
// 16 bits signés (3 LSBs à supprimer) unité = 0.0625 °C ***************
signed short capteurTemp_read(void)
{ signed short data;

  P3OUT &= ~BIT0;                 // Activation du CS

  U1TXBUF = 0x00 ;                // Chargement du buffer U1TXBUF
  while ( !(U1TCTL & 0x01)) ;      // Attente de la réception complet (8 bits)
                   // TXEPT=1 indique que le buffer de transmission est vide
  data = U1RXBUF << 8 ;            // lecture des MSBs

  U1TXBUF = 0x00 ;              // Chargement du buffer de transmission U1TXBUF
  while ( !(U1TCTL & 0x01)) ;      // Attente de la réception complet (8 bits)
                   // TXEPT=1 indique que le buffer de transmission est vide
  data = data | U1RXBUF ;          // lecture des LSBs

  P3OUT |= BIT0 ;                  // relève enable : désactivation du CS
  return(data) ;             // Retour de la valeur brute fournie par le capteur
}
```

Questions :

- Décrivez la signification de U1CTL et U1TCTL ainsi que de tous les paramètres les définissant.
- Quelle est ici la fréquence du clock de la transmission ?
- Quelle est la définition de P4SEL ? Pourquoi ?
- Quelle est la fonction de l'instruction `U1TXBUF = 0x00`

Ceci est un exemple du fichier `main.c`.

```
#include "msp430xG46x.h"
#include "intrinsics.h"     //  unction intrinsèque : __no_operation()
#include « LCD.h »           // déclaration des fonctions LCD
#include « capteurTemp.h »    // déclaration des fonctions capteurTemp.c
#include <stdio.h>  // déclaration de la fonction « sprintf »

void main( void )
{
  int i ;                          // variable signée de 16 bits
  unsigned int n ;                 // variables non signées de 16 bits
  char str[9] ;                    // Tableau de 9 caractères (8 bits)

  WDTCTL = WDTPW + WDTHOLD;      // Stop watchdog timer

  LCD_init() ;                     // initialisation de l'affichage LCD
  SPI_1_init() ;// initialisation de l'interface avec le capteur de température

  while(1)       // Boucle infinie
  {
// Suppression des 3 bits de poids faibles du mot de 16 bits issu du capteur
    n = capteurTemp_read()>>3 ;

    // Création d'une chaîne de caractère ASCII à afficher
    sprintf(str, « T=%4.1f C »,n*0.0625) ;
    LCD_print(str) ;

    // Attente avant une nouvelle acquisition de température
    for (i=0 ; i<1000 ; i++)
      __no_operation() ;
  }
}
```

Vous devez aussi écrire le fichier `capteurTemp.h` avec les définitions des fonctions contenues dans `capteurTemp.c` et qui doit se trouver dans le **même directoire**.

2.3.1 Description du software

Le capteur de température TC77 fournit, par une interface MICROWIRE compatible SPI une mesure de température codée sur 16 bits.
Les trois bits de poids faibles doivent être éliminés car ils ne contiennent pas d'information utile.
Après conversion, la température doit être affichée sur un affichage LCD en degrés Celsius T=xx.x C

2.3.1.1 Utilisation alternative des structures et union

Grâce à l'utilisation du fichier d'entête io430.h → io430xG46x.h, il est aussi possible d'atteindre explicitement n'importe quel flag (bit d'un registre spécifique). Par conséquent, l'écriture d'un bit d'un registre – par exemple U1CTL – peut être aussi réalisée selon l'instruction *Nom_du_registre*_bit.*Nom_du_flag*.
Par exemple :
U1CTL_bit.SYNC = 1 est équivalent à U1CTL |= 0x04
U1CTL_bit.SYNC = 0 est équivalent à U1CTL &=0xFB

Pour plus de détails, il est vivement recommandé de consulter le fichier io430xG46x.h.

2.3.1.2 Affichage LCD

L'affichage des températures est réalisé par une transformation de la valeur binaire de 16 bits issue du capteur en une valeur entière ne tenant compte que des 13 bits utiles (décalage arithmétique à droite de 3 bits). Cette valeur, multipliée par le gain du capteur (0.0625V/LSB) donne une valeur fractionnaire qu'il faut ensuite transformer en caractères ASCII avant de l'afficher sur l'affichage LCD.

La solution la plus facile est d'utiliser la routine `sprintf` de la librairie **clib**. Les instructions sont les suivantes :
Pour une température exprimée en degrés Celsius :

```
sprintf(str, "T=%4.1f C",n*0.0625);
```

ou str est de type char str[9] et est un tableau (chaine de caractères) constitué de 9 caractères (8 bits) sous forme ASCII. La Figure 2-8 donne un exemple pour une température T=22.6°C. Le neuvième caractère n'est pas affiché. Il indique simplement la fin de la chaine de caractères.

⊟ str	"T=22.6 C"
├ [0]	'T' (0x54)
├ [1]	'=' (0x3D)
├ [2]	'2' (0x32)
├ [3]	'2' (0x32)
├ [4]	'.' (0x2E)
├ [5]	'6' (0x36)
├ [6]	' ' (0x20)
├ [7]	'C' (0x43)
└ [8]	'' (0x00)

Figure 2-8 : Tableau de 9 caractères ASCII pour affichage

L'affichage LCD est contrôlé par plusieurs routines : voir la **note d'application No. 3**.

```
void LCD_clear(void);  // éteint tous les segments de l'affichage
void LCD_init(void);   // initialisation de l'ensemble des ports dédiés à l'affichage LCD
void LCD_print(char* str);  // affiche les caractères ASCII du tableau str .
```

2.4 EXERCICES ULTERIEURS

Dans l'exercice de base toutes les opérations se font dans la boucle principale while(1).
Afin d'aller dans la direction d'une application à basse consommation ***il est nécessaire de vider*** cette boucle de toutes ses opérations qui seront programmées dans des routines d'interruption.

Important: avant de programmer ces variantes copiez votre programme de base et créez un nouveau projet.

2.4.1 Affichage par un timer

Programmer un simple timer qui générera des interruptions toutes les secondes (freq = 1 Hz) pour afficher le résultat.
Le préambule du main.c inclura par exemple:

```
// Définition d'un timer_A avec fréquence = 1 Hz
  CCR0 = 32768 - 1;
  TACTL = TASSEL_1 + MC_1;  // ACLK, upmode
  CCTL0 = CCIE;             // CCR0 interrupt enabled

  __enable_interrupt();
```

Justifiez pourquoi la fréquence dans ce cas est bien de 1 Hz.

Ensuite programmez sa fonction d'interruption où passera le code pour l'affichage.

```
#pragma vector= TIMERA0_VECTOR       // Timer_A TACCR0 interrupt handler
__interrupt void int_timerA(void)
{
  unsigned int n;
  char str[9];

  P2OUT ^= 0xC0;   // clignotement pour vérifier la fréquence du timer
// Suppression des 3 bits de poids faibles du mot de 16 bits issu du capteur
  n = data >> 3;
  sprintf(str, "T=%4.1fC",n*0.0625);
  LCD_print(str);     // Affichage de la chaîne de caractères
}
```

Vous devrez définir comme globales les variables en commun avec le main.c . lesquelles ?

L'activation de P2OUT (LEDs clignotantes) permettra de vérifier que le timer *travaille* bien à 1 Hz. P2DIR sera donc aussi initialisé dans le main.c .

Pensez évidemment à supprimer le code d'affichage précédent dans le main.c !

2.4.2 Lecture des mesures de température dans une routine d'interruption

L'étape suivante est de déplacer aussi le code de la fonction CapteurTemp.c dans la routine d'interruption que l'UART peut déclencher.
Pour cela il suffira d'ajouter dans le main.c

```
    IE2 = URXIE1;
```

dans la fonction SPI_1_init(void).

Question : quelle est la signification de ces paramètres ?

Ensuite on doit programmer la routine d'interruption

```
#pragma vector=USART1RX_VECTOR
__interrupt void USART1RX_ISR (void)
{
  if (IFG2 & UTXIFG1)           // USART1 TX buffer ready?
  { IE2 = 0;
    P2OUT ^= 0x03;  // clignotement pour vérifier la fréquence d'échantillonnage

    data = U1RXBUF << 8;          // lecture des MSBs

    U1TXBUF = 0x00;        // Chargement du buffer de transmission U1TXBUF
    while (!(U1TCTL & 0x01));     // Attente de la réception du 2ème octet
                // TXEPT=1 indique que le buffer de transmission est vide
    data = data | U1RXBUF;        // lecture des LSBs

    P3OUT |= BIT0;                // relève enable: désactivation du CS
    P3OUT &= ~BIT0;               // Activation du CS
    U1TXBUF = 0;           // Chargement du buffer de transmission U1TXBUF
    IE2 = URXIE1;
  }
}
```

Question : pourquoi redéfinit-on plusieurs fois IE2 et P3OUT ?

Initialisez les sorties P2 afin de visualiser les fréquences du timer d'affichage (sur P2.6-7) et de l'échantillonnage.
Ensuite diminuez la fréquence d'échantillonnage à environ **3 Hz**, ce qui peut être vérifié en première approximation par le clignotement des LEDs P2.0-1.

Normalement à ce point vous devriez aussi avoir entièrement *vidé* la boucle `while(1)` du `main.c`.
Notez que l'exemple montré ici n'est que très partiellement optimal puisque l'exécution attend dans la routine d'interruption que le deuxième byte soit disponible.

On pourrait améliorer cela en reprogrammant cette routine d'interruption pour lire un seul byte et ensuite sortir. Une instruction avec une variable globale d'état du type

```
quel_byte ^= 1 ;
```

exécutée à chaque appel peut être utilisée pour discriminer entre les deux bytes de la lecture et ainsi permettrait de ne rester que le minimum de temps dans la routine d'interruption.
Si vous tentez cela, faites-le dans un nouveau projet, afin de ne pas risquer de perdre l'acquis précédent …

2.4.3 Affichage de la température aussi en Kelvin

Ajouter une interruption sur le port 1 d'entrée afin que quand on presse le bouton P1.0 on passe d'un affichage en degrés Celsius vers un en degrés Kelvin en forme « **`T=YYY.YK`** » (attention, pas plus que 8 caractères pour le LCD), et vice-versa.
Pour cela la routine d'interruption peut être utilisée comme définition de mode (état) : selon le mode la boucle principale `while(1)` affichera ainsi la température.

```
#pragma vector=PORT1_VECTOR
__interrupt void Port1_ISR (void) // PORT1ISR (void)
{   if (P1IFG == 0x01)
    mode ^= 1;

  P1IFG = 0;
}
```

3. *Modes à basse consommation*

A côté du mode de fonction normal le MSP430 peut être programmé pour fonctionner en plusieurs modes à basse consommation.

Les modes low-power 0-4 sont configurés avec les bits CPUOFF, OSCOFF, SCG0, et SCG1 dans le registre de statut (SR).
Ainsi les modes low-power 0-4 sont activés et gérés par les fonctions qui adressent le SR, que vous trouverez dans le fichier **intrinsics.h** où elles sont définies comme suit.

```
  __intrinsic void           __bic_SR_register(unsigned short);
  __intrinsic void           __bis_SR_register(unsigned short);
  __intrinsic unsigned short __get_SR_register(void);
  __intrinsic void           __bic_SR_register_on_exit(unsigned short);
  __intrinsic void           __bis_SR_register_on_exit(unsigned short);
  __intrinsic unsigned short __get_SR_register_on_exit(void);

// Control bits in the processor status register, SR.
#define __SR_GIE     (1<<3)
#define __SR_CPU_OFF (1<<4)
#define __SR_OSC_OFF (1<<5)
#define __SR_SCG0    (1<<6)
#define __SR_SCG1    (1<<7)

/* Functions for controlling the processor operation modes*/
#define __low_power_mode_0() (__bis_SR_register(  __SR_GIE      \
                                                | __SR_CPU_OFF))

#define __low_power_mode_1() (__bis_SR_register(  __SR_GIE      \
                                                | __SR_CPU_OFF  \
                                                | __SR_SCG0))

#define __low_power_mode_2() (__bis_SR_register(  __SR_GIE      \
                                                | __SR_CPU_OFF  \
                                                | __SR_SCG1))

#define __low_power_mode_3()                    \
  (__bis_SR_register(  __SR_GIE                 \
                     | __SR_CPU_OFF             \
                     | __SR_SCG0                \
                     | __SR_SCG1))

#define __low_power_mode_4()                    \
  (__bis_SR_register(  __SR_GIE                 \
                     | __SR_CPU_OFF             \
                     | __SR_SCG0                \
                     | __SR_SCG1                \
                     | __SR_OSC_OFF))

#define __low_power_mode_off_on_exit()          \
  (__bic_SR_register_on_exit(  __SR_CPU_OFF     \
                             | __SR_SCG0        \
                             | __SR_SCG1        \
                             | __SR_OSC_OFF))
```

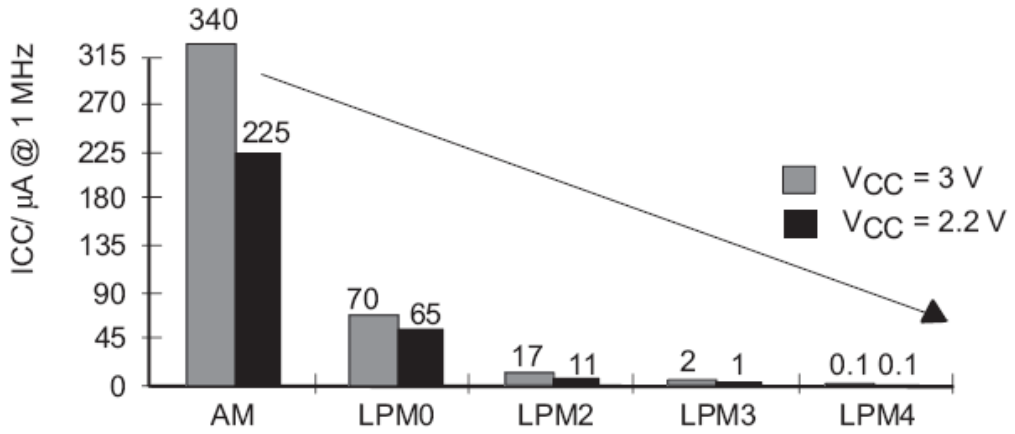
L'avantage d'inclure les bits de contrôle de mode CPUOFF, OSCOFF, SCG0, and SCG1 dans le SR est que le mode de fonctionnement actuel est sauvé sur le stack lorsqu'une routine de gestion d'interrupt est exécutée.

Le fichier **msp430xG46x.h** défini en plus plusieurs macros utiles pour activer et désactiver les différents modes :

```
#define CPUOFF              (0x0010)
#define OSCOFF              (0x0020)
#define SCG0                (0x0040)
#define SCG1                (0x0080)

/* Low Power Modes coded with Bits 4-7 in SR */
#define LPM0                (CPUOFF)
#define LPM1                (SCG0+CPUOFF)
#define LPM2                (SCG1+CPUOFF)
#define LPM3                (SCG1+SCG0+CPUOFF)
#define LPM4                (SCG1+SCG0+OSCOFF+CPUOFF)
#define LPM0_bits           (CPUOFF)
#define LPM1_bits           (SCG0+CPUOFF)
#define LPM2_bits           (SCG1+CPUOFF)
#define LPM3_bits           (SCG1+SCG0+CPUOFF)
#define LPM4_bits           (SCG1+SCG0+OSCOFF+CPUOFF)

#define LPM0      _BIS_SR(LPM0_bits)     /* Enter Low Power Mode 0 */
#define LPM0_EXIT _BIC_SR_IRQ(LPM0_bits) /* Exit Low Power Mode 0 */
#define LPM1      _BIS_SR(LPM1_bits)     /* Enter Low Power Mode 1 */
#define LPM1_EXIT _BIC_SR_IRQ(LPM1_bits) /* Exit Low Power Mode 1 */
#define LPM2      _BIS_SR(LPM2_bits)     /* Enter Low Power Mode 2 */
#define LPM2_EXIT _BIC_SR_IRQ(LPM2_bits) /* Exit Low Power Mode 2 */
#define LPM3      _BIS_SR(LPM3_bits)     /* Enter Low Power Mode 3 */
#define LPM3_EXIT _BIC_SR_IRQ(LPM3_bits) /* Exit Low Power Mode 3 */
#define LPM4      _BIS_SR(LPM4_bits)     /* Enter Low Power Mode 4 */
#define LPM4_EXIT _BIC_SR_IRQ(LPM4_bits) /* Exit Low Power Mode 4 */
```



Modes de fonctionnement et consommation associée

3.1 EXERCICE DE BASE

Ouvrir un nouveau projet (toujours dans un directoire séparé …)

On veut réaliser une simple application de démonstration avec deux timers, basés respectivement sur le ACLK et le SMCLK, et qui font chacun clignoter une LED.

Définir donc dans le main.c un timer_B basé sur le SMCLK avec une fréquence de 4 Hz :

```
TBCTL = TBSSEL_2 + MC_1 + ID_3;     // SMCLK/8, upmode
TBCCR0 = 32768 - 1;          // Définition d'un timer_B avec fréquence = 4 Hz
TBCCTL0 = CCIE;              // CCR0 interrupt enabled
```

Question : Identifiez les paramètres ici définis et justifiez pourquoi la fréquence est bien de 4 Hz.

Définir ensuite un autre timer_A basé sur le ACLK avec une fréquence de 1 Hz.

Ecrire pour **chaque timer** sa fonction d'interruption CCR0, qui fera clignoter une LED, par exemple pour le timer_B :

```
#pragma vector= TIMERB0_VECTOR    // Timer_B TBCCR0 interrupt vector handler
__interrupt void int_timerB(void)
{
  P2OUT ^= BIT7;                    // Toggle P2.7
}
```

Le timer_A fera clignoter la LED P2.6 et sera configuré de manière similaire.

Réaliser l'application d'abord en mode normale : donc avec une boucle `while(1)` - vide, évidemment – et vérifier que les LED clignotent bien avec les fréquences désirées (respectivement 1 et 4 Hz).

Ensuite remplacer le `while(1)` par **`__low_power_mode_X()`**, X définissant le mode choisi entre 0 et 4.

Observez et décrivez les effets sur le programme dans les divers modes entre 0 et 4.

3.2 EXERCICES ULTERIEURS

3.2.1 Passage d'un mode à l'autre

Ajoutez une fonction d'interruption sur les switches P1 afin de passer d'un mode basse consommation à l'autre. Une pression sur le bouton poussoir SW1 enclenchera le mode 1, sur SW2 le mode 2 et ainsi de suite. Comme indicateur du mode en cours, programmez l'allumage d'une LED parmi les quatre les plus à droite (P2.0 à P2.3).

Pour enclencher un mode dans une fonction d'interruption il est nécessaire d'utiliser la fonction `__bis_SR_register_on_exit (LPMX_bits);` X définissant le mode entre 0 et 4.

Pour réactiver un timer qui avait été arrêté il faut aussi d'abord sortit du low-power par une instruction `LPM4_EXIT;`

Vérifiez que `LPM4_EXIT` permet de sortir en fait de n'importe quel mode low-power (aussi < 4).

3.2.2 Lecture du capteur de température SPI en mode low-power

Reprenez la dernière version du programme de lecture du capteur de température SPI, et remplacez la boucle `while(1)` par par **`__low_power_mode_X()`**, X définissant le mode entre 0 et 4.

Observez les effets sur le programme dans les différents modes low-power et finalement implémentez dans la version finale le mode le plus économique qui laisse fonctionner le programme.

4. Communication par bus I2C

4.1 BUT DE CET EXERCICE

Le but de cet exercice est de réaliser la communication entre le MSP430 et un circuit de type PCA9530 qui est un gradateur de lumières (LED dimmer) pilotant 2 LEDs.
Il communique avec le microcontrôleur via un bus I²C.

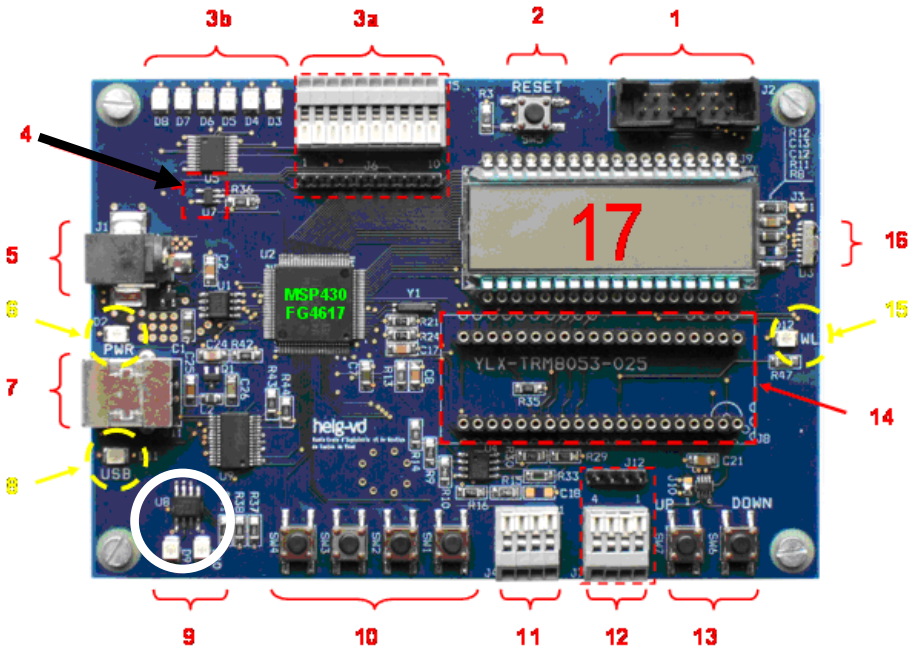


Figure 4-1 : Le LED dimmer PCA9530 est indiqué au no.9 .

4.2 BUS UTILISE

Le bus I²C (Inter-Integrated Circuit), développé à l'origine par Philips, est un bus fréquemment utilisé pour la communication basse-vitesse avec des périphériques. Il était typiquement utilisé à la base pour relier tous les périphériques (récepteur télécommande, affichage, …) au microprocesseur dans des appareils grands public.

Figure 4-2 : logo I²C

Ce bus a une configuration maître-esclave.

Le maître initie la communication et va écrire ou lire des données dans un esclave donné. Chaque esclave doit donc avoir une adresse distincte. Il peut naturellement y avoir plusieurs esclaves, mais aussi plusieurs maîtres (multi-master mode) ; ce cas ne sera toutefois pas abordé ici.

La communication est sérielle synchrone sur 2 lignes : le maître génère un clock série (SCL, serial clock) et les données transitent sur la ligne bidirectionnelle SDA (serial data). Chaque acteur du bus a des sorties en drain-ouvert ; les lignes ayant chacune une résistance pull-up.

4.3 CIRCUIT

Le circuit génère sa propre horloge pour piloter les LEDs en PWM.

Pour chaque LED, la fréquence de l'horloge peut être divisée, et on peut également régler le rapport cyclique désiré.

Ainsi, suivant la fréquence et le rapport cyclique choisis, chaque LED peut soit clignoter, soit être allumée avec une intensité réglable.

CONTROL REGISTER DEFINITION

B2	B1	B0	REGISTER NAME	TYPE	REGISTER FUNCTION
0	0	0	INPUT	READ	INPUT REGISTER
0	0	1	PSC0	READ/ WRITE	FREQUENCY PRESCALER 0
0	1	0	PWM0	READ/ WRITE	PWM REGISTER 0
0	1	1	PSC1	READ/ WRITE	FREQUENCY PRESCALER 1
1	0	0	PWM1	READ/ WRITE	PWM REGISTER 1
1	0	1	LS0	READ/ WRITE	LED SELECTOR

Figure 4-3 : registres internes du PCA9530

Il s'agit donc d'écrire dans les registres PSC0, PWM0, PCS1 et PWM1 du PCA9530 pour contrôler les sorties LED0 (D9), respectivement LED1 (D10).

La période du PWM est donnée par :

$$T_{PWMn} = \frac{PSCn + 1}{152} \qquad 4.1$$

et le rapport cyclique :

$$\eta_n = \frac{PWMn}{256} \qquad 4.2$$

Selon la documentation et le schéma, l'adresse du circuit et fixée à la valeur 0x60 (1100000b).

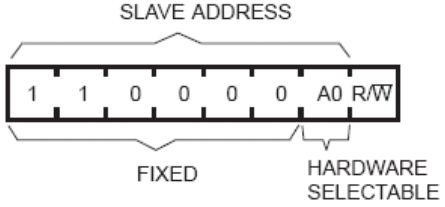


Figure 4-4 : Composition de l'adresse du PCA9530

4.4 BRANCHEMENTS

Dans notre cas, il n'y y sur le bus qu'un seul maître (le MSP430), et un seul esclave (le gradateur) présents sur le bus :

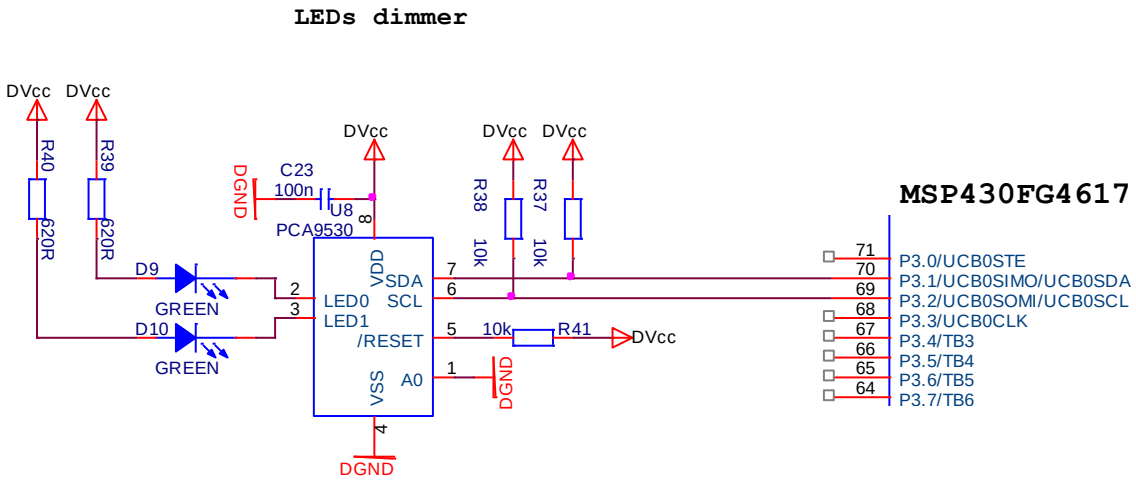


Figure 4-5 : branchements

Le module du microcontrôleur utilisé pour la communication est le « USCI_B0 ».

4.5 COMMUNICATION

4.5.1 Etapes générales

Il faut toujours garder à l'esprit que le bus I²C est un bus multi-master, et que les esclaves peuvent mettre plus ou moins de temps à réagir. Il faut donc :

- Attendre que le bus soit libre avant de communiquer
- Que l'esclave quittance les communications.

Voici les étapes d'une communication :

1. Dans le préambule du programme : configuration des registres du MSP430 régissant la communication I2C par le module USCI.
2. Le maître attend que le bus soit libre, puis initie la communication par une « start condition » (flanc descendant de SDA durant SCL haut).
3. Le maître envoie l'adresse du périphérique esclave avec qui il veut communiquer (7 bits), ainsi qu'un bit R/W indiquant le sens des données à suivre.
   L'esclave doit quittancer la bonne réception de ceci en tirant la ligne de donnée en bas.
4. Les données sont transmises. Il peut y avoir plusieurs octets.
5. Le maître termine la communication par une « stop condition » (flanc montant de SDA durant SCL haut).

4.5.2 Dans notre cas

1. Générer « start condition », attendre qu'elle ait été générée.
2. Envoyer adresse et le sens des données (transmission), attendre quittance de la part de l'esclave, ce qui va générer un interrupt dans le MSP430.
3. Envoi des données (plusieurs octets)
   - Envoi du byte de « Control ».

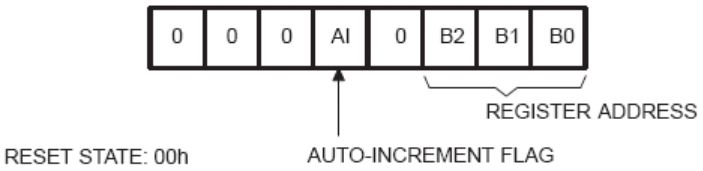


Figure 4-6 : Composition du byte de contrôle

L' « auto-increment » sert à indiquer qu'on veut écrire dans plusieurs registres successivement. L'adresse (« B2, B1, B0 ») indique dans quel registre du PCA9530 sera chargée la valeur qui suit.

   - Envoi des valeurs de fréquence et rapport cycliques des deux LEDs :
     - « Frequency prescaler » du PWM0

$$PSC0 = \frac{T_{PWM0} \cdot 152}{1000} - 1$$

       avec T_{PWM0} [msec] = période = 1 / fréquence du PWM0
     - Rapport cyclique du PWM0 rapportée à 255.
     - « Frequency prescaler » du PWM1 :

$$PSC1 = \frac{T_{PWM1} \cdot 152}{1000} - 1$$

     - Rapport cyclique du PWM1 rapportée à 255.
     - LED selector (dans notre cas : LED0 sur PWM0, LED1 sur PWM1)
4. Générer « stop condition ».

4.6 PROGRAMMATION

4.6.1 Configuration

Les registres à configurer sont les suivants :

No	Registre	Nom court	Bits à configurer
1	Port 3 select register	P3SEL	Bits 1 et 2
2	USCI_B1 control register 1	UCB0CTL1	UCSWRST, UCSSELx
3	USCI_B1 control register 0	UCB0CTL0	UCMST + UCMODE_3 + UCSYNC
4	USCI_B0 bit rate control register 1 & 0	UCB0BR1, UCB0BR0	tous
5	USCI_B0 I2C slave address register	UCB0I2CSA	tous

Tableau 4-1 : Registres à initialiser

No	Registre	Nom court	Bits à utiliser
6	USCI_B1 transmit buffer register	UCB0TXBUF	tous
2	USCI_B1 control register 1	UCB0CTL1	UCTR, UCTXSTT, UCTXSTP
7	SFR interrupt enable register 2	IE2	UCB0TXIE

Tableau 4-2 : Registres à utiliser lors de communication

1. P3SEL : Configurer correctement le port 3 pour utiliser les pins P3.1 et P3.2 en I²C.
2. UCB0CTL1
   - Mettre le module USCI_B0 en reset pendant sa configuration à l'aide de « UCSWRST ». Enlever le reset à la fin de la configuration
   - Sélectionner le signal d'horloge qui sera utilisé pour la communication I2C. Une valeur de « UCSSEL_2 » ou « UCSSEL_3 » (SMCLK) convient.
   - Le bit « UCTR » détermine le sens de transit des données dans la phase suivant celle d'adressage (0=lecture, 1=écriture).
   - Mettre « UCTXSTT » à 1 génère une condition de start.
   - Mettre « UCTXSTP » à 1 génère une condition de stop.
3. UCB0CTL0. Configurer comme suit :
   - UCMST configure le module en I2C master
   - UCMODE_3 configure le module en I2C
   - UCSYNC configure le module en synchrone
4. UCB0BR1, UCB0BR0 : Configurer le diviseur de clk de manière à obtenir 100kHz au maximum comme fréquence du signal SCL (f_{SMCLK} 1MHz).
5. Charger l'adresse du slave qui sera utilisée pour les communications à suivre.
6. Simplement charger la valeur qu'on veut transmettre.

7. UCB0TXIE sert à activer les interruptions pour ce module. L'interruption aura lieu dès que l'évenement commandé a eu lieu, ce peut être une « start condition », la transmission d'une adresse, d'une donnée, ou une « stop condition ».

Au besoin, aidez-vous du datasheet du MSP430 que vous avez reçu, chapitre 21 (« Universal Serial Communication Interface, I2C Mode »).

4.6.2 Machine d'état

Le cheminement expliqué au §4.5.2 peut très avantageusement se faire par interruption en gérant le système comme une machine d'état très simple :

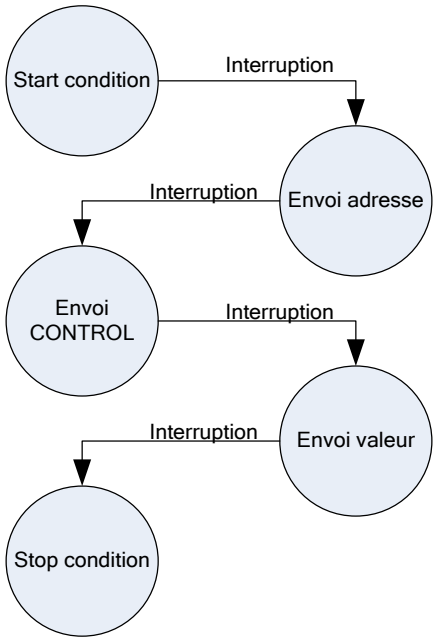


Figure 4-7 : Machine d'état pour transmission:

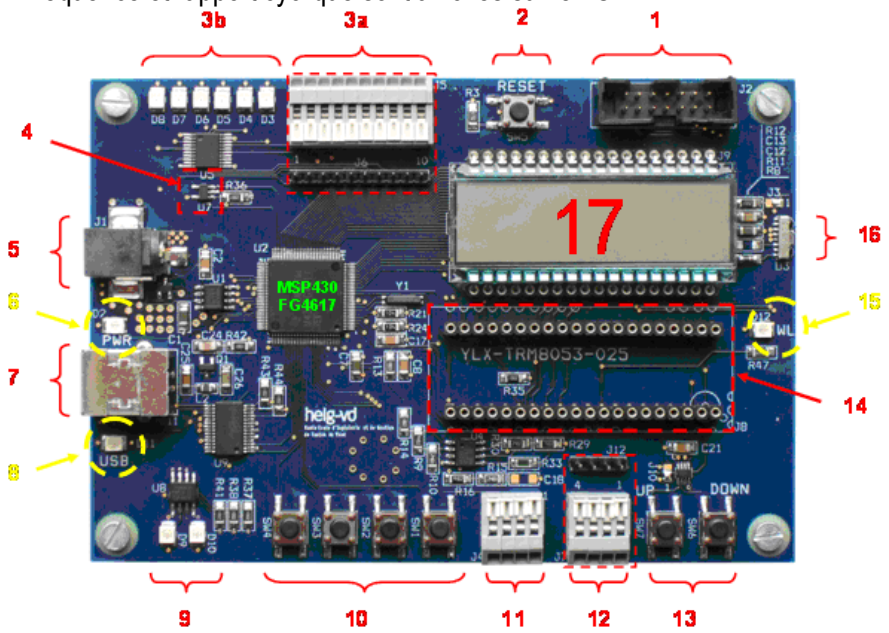
Cette machine d'état est, dans notre exemple du chapitre suivant, implémenté dans la fonction d'interruption du `USCIAB0TX_VECTOR` (page suivante). Le flag d'autoincrément étant ici positif, plusieurs (5) bytes sont envoyés consécutivement entre le *CONTROL* et la *Stop condition* . L'ordre d'envoi correspond à un incrément du champ B2-B1-B0 du byte de *CONTROL* comme indiqué à la *Figure 4-3*, page 16.

4.7 EXERCICE DE BASE

Ouvrir un nouveau projet (toujours dans un directoire séparé …)

Copier et exécuter le programme suivant qui :

1. Initialise le PCA9350.
2. Fait clignoter les LEDs (No. 9 sur la figure suivante) avec une fréquence de 2 Hz et un rapport cyclique de 50% .
3. Fréquence et rapport cyclique sont affichés sur le LCD.



Comprendre et commenter **tous les paramètres** qui configurent l'USCI utilisée en I2C.

Réaliser un projet basé sur la programmation en C comprenant les fichiers suivants :

- main.c : fichiers contenant le programme principal de l'application
- LCD.c et LCD.h : fichiers de contrôle de l'affichag
- ledDimmer.c et ledDimmer.h : fichiers de contrôle du PCA9350

Les fichiers LCD.c, LCD.h sont donnés et ne doivent pas être modifiés.

Le fichier ledDimmer.c comprend les fonctions suivantes :

```
//*******************************************************************************
#include "msp430xG46x.h"
#include "intrinsics.h"

#define max( a, b ) ( ((a) > (b)) ? (a) : (b) )
#define min( a, b ) ( ((a) < (b)) ? (a) : (b) )
#define E_IDLE  0

unsigned short state;
unsigned short controlReg, freqPrescReg_1, pwmReg_1, freqPrescReg_2, pwmReg_2;

//*******************************************************************************
//Initialisation de l'I2C
void ledDimmer_init(void)
{
  P3SEL |= 0x06;                            // Assign I2C pins to USCI_B0
  UCB0CTL1 |= UCSWRST;                      // Enable SW reset
  UCB0CTL0 = à configurer, voir section 4.6.1 plus haut  // I2C Master, synchronous mode
  UCB0CTL1 = à configurer ...               // Use SMCLK, keep SW reset
```

```
  UCB0BR0 = 11;                             // fSCL
  UCB0BR1 = 0;
  UCB0I2CSA = 0x60;                         // Set slave address (fixe)
  UCB0CTL1 &= ~UCSWRST;                     // Clear SW reset, resume operation

  state = E_IDLE;

  UCB0CTL1 |= UCTR + UCTXSTT; // I2C TX, start condition
                              // (va envoyer le start + slave address)
  IE2 |= UCB0TXIE;            // Enable TX ready interrupt
}

/*******************************************************************************
Interruption TX - La transmission est faite en machine d'états:
On fait une start condition.  Lorsque c'est fait, cette routine d'interrupt
survient, on transmet le 1er byte, elle re-survient, on transmet le 2e byte,
etc... jusqu'au dernier byte, suivi d'une stop condition.  */

#pragma vector = USCIAB0TX_VECTOR
__interrupt void USCIAB0TX_ISR(void)
{
  switch(state) {
  case E_IDLE:
    UCB0TXBUF = 0x11; //auto-increment + register address=1 (PCS0)
    state++;
    break;
  case 1:
    UCB0TXBUF = freqPrescReg_1; //f = fmax (152Hz)
    state++;
    break;
  case 2:
    UCB0TXBUF = pwmReg_1; // rapport cyclique = 0 (éteint)
    state++;
    break;
  case 3:
    UCB0TXBUF = freqPrescReg_2; // f = fmax (152Hz)
    state++;
    break;
  case 4:
    UCB0TXBUF = pwmReg_2; // rapport cyclique = 0 (éteint)
    state++;
    break;
  case 5:
    UCB0TXBUF = 0x0E; // LED0 sur PWM0, LED1 sur PWM1
    state++;
    break;
  case 6:             // STOP condition
    UCB0CTL1 |= UCTXSTP;  // I2C stop condition
    IFG2 &= ~UCB0TXIFG;   // Clear USCI_B0 TX int flag
    IE2 &= ~UCB0TXIE;     // disable TX ready interrupt
    state = E_IDLE;
    break;
  default:  // erreur...
    UCB0CTL1 |= UCTXSTP;  // I2C stop condition
    IFG2 &= ~UCB0TXIFG;   // Clear USCI_B0 TX int flag
    IE2 &= ~UCB0TXIE;     // disable TX ready interrupt
    state = E_IDLE;
    break;
  }
}

/*******************************************************************************
Ecriture des 2 LEDs
period : période du pwm en [ms]
dutycycle : rapport cyclique en [%]      */

void ledDimmer_write (unsigned int period_1, unsigned char dutyCycle_1,
                      unsigned int period_2, unsigned char dutyCycle_2)
{
  // détermine control register
  controlReg = 0x11;  // auto-increment + register address=(LedNr)
```

```
  // détermine frequency prescaler - équation: T=(prescaler+1)/152
  freqPrescReg_1 =
       (((unsigned long)max(period_1,7)*(unsigned long)152)/(unsigned long)1000)-1;
  freqPrescReg_2 =
       (((unsigned long)max(period_2,7)*(unsigned long)152)/(unsigned long)1000)-1;

  // détermine PWM register (0=min, 255=max)
  pwmReg_1 = ((unsigned int)min(dutyCycle_1,100)*(unsigned int)255)/(unsigned int)100;
  pwmReg_2 = ((unsigned int)min(dutyCycle_2,100)*(unsigned int)255)/(unsigned int)100;

  state = E_IDLE;             // lance l'envoi des datas par I2C
  UCB0CTL1 |= UCTR + UCTXSTT; // I2C TX, start condition
                              // (va envoyer le start + slave address)
  IE2 |= UCB0TXIE;            // Enable TX ready interrupt
}
```

Questions :

- Décrivez la signification de `UCB0CTL0` et `UCB0CTL1` ainsi que de tous les paramètres les définissant.
- Quelle est ici la fréquence du clock de la transmission ? (observez `UCB0BR0` et `UCB0BR1`)
- Quel est le rôle de la fonction `USCIAB0TX_ISR()` ?
  Vérifiez qu'elle joue bien le rôle de machine d'état.
- Quels sont les paramètres de la fonction `ledDimmer_write(…)` et que fait-elle ?
- Observez les macros `min(…)` et `max(..)` : savez-vous les interpréter ?

Et maintenant le main.c :

```
//*******************************************************************************
#include "msp430xG46x.h"
#include "intrinsics.h"
#include "ledDimmer.h"
#include "LCD.h"
#include "stdio.h"


unsigned int duty = 50, freq = 2;  // valeurs initiales de clignotement des LEDs

void main( void )
{
  int i;
  char str[9];

  WDTCTL = WDTPW + WDTHOLD;      // Stop watchdog timer

  LCD_init();
  ledDimmer_init();

  __enable_interrupt();

  ledDimmer_write (1024/freq, duty, 1024/freq, duty);

  while(1)
  {
    for (i=0 ; i<30000 ; i++);
    sprintf (str,"%4d %3d", freq, duty);
    LCD_print (str);
  }
}
```

4.8 EXERCICE ULTERIEUR

4.8.1 Changement de fréquence et rapport cyclique dans le programme

On veut maintenant qu'en cliquant les boutons SW1-4 on puisse varier ces deux paramètres. Ajoutez donc une fonction d'interruption sur P1 , par exemple :

```
// *******************************************************
#pragma vector=PORT1_VECTOR
__interrupt void Port1_ISR (void)
{
  P2OUT ^= 0x01;
  if (P1IFG == 0x01)
    duty += 10;
  if (P1IFG == 0x02)
    duty -= 10;
  if (P1IFG == 0x04)
    freq *= 2;
  if (P1IFG == 0x08)
    freq /= 2;

  duty = min (duty, 100);
  duty = max (duty, 0);
  freq = min (freq, 1024);
  freq = max (freq, 1);

  ledDimmer_write(1024/freq, duty, 1024/freq, duty);

  P1IFG = 0;
}
```

Evidemment vous devez aussi initialiser cette interruption dans le main.c .

Questions :

- Il est utile de définir ici aussi les macros `min(…)` et `max(..)` ?
  Que se passerait-il si on ne les utilisait pas ?
- Décrivez ce que font ici les quatre switches SW1-4 (quel switch fait quoi …).

5. *Application de synthèse*

5.1 SPECIFICATION

En guise de conclusion on veut réaliser une application avec la spécification suivante :

- Lecture et affichage avec une fréquence de 1 Hz du capteur de température SPI.
- Dans le préambule du programme la température initiale T_O est enregistrée comme référence.
- Le circuit PCA9350 (bus I2C) est utilisé pour convertir la valeur de l'incrément de température ($\Delta T = T - T_O$) en deux signaux :
  a. Un rapport cyclique de PWM à haut fréquence (~1,5 KHz), qui se manifeste comme intensité lumineuse sur la LED0 avec l'échelle suivante :

     $$\text{rapport cyclique (\%)} = 5 + \Delta T*100 / 5$$

  b. Une fréquence proportionnelle à l'incrément de température, qui se manifeste comme fréquence de clignotement (rapport cyclique de 50%) sur la LED1 avec l'échelle suivante :

     $$\text{freq (Hz)} = 1 + 2 * \Delta T$$

- Le switch SW1 permet par pressions successives du bouton de changer l'affichage LCD entre :
  1. Température absolue (en Celsius)
  2. Incrément de température ($\Delta T = T - T_O$)
  3. Fréquence de la LED1
  4. Intensité (rapport cyclique) de la LED0
- Enfin toute l'application tourne en **mode basse consommation**.

5.2 PROGRAMMATION

On peut partir du programme réalisé au point 2.4.2 .
Recopiez tous ses fichiers dans un nouveau directoire et déclarez un nouveau projet.

Il faut y inclure les modules `capteurTemp.c, ledDimmer.c, LCD.c,` ainsi que les fichiers de déclaration correspondants (`.h`) .
Pour le module `capteurTemp.c`, vous prendrez celui qui contient la fonction `capteurTemp_read()`.

Le main.c débutera par exemple comme dans le listing suivant :

```
#include "msp430xG46x.h"
#include "intrinsics.h"     // fonctions intrinsèques
#include "LCD.h"            // déclaration des fonctions LCD
#include "capteurTemp.h"    // déclaration des fonctions Capteur Température
#include "ledDimmer.h"
#include <stdio.h>          // déclaration de la fonction sprintf

unsigned int mode = 0;
unsigned int quel_byte;
float T_init;
int data;

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

// initialisation de P1.0 (bouton pression SW1)
  P1DIR  = 0x00;            // configuration de P1.0 en entrée
  P1IE  |= 0x01;            // enable interrupt P1.0
  P1IFG  = 0;

  P2DIR = 0xC3;
  P2OUT = 0x80;

  LCD_init();        // initialisation de l'affichage LCD
  SPI_1_init();      // initialisation de l'interface SPI - capteur de température
  ledDimmer_init();  // initialisation du PCA9530

  T_init = 0.0625*(capteurTemp_read()>>3);  // lecture de la temperature initiale
... (suite du programme)
```

On ajoutera ensuite la définition d'un timer pour obtenir des interruptions toutes les secondes, par exemple :

```
  // définition d'un timer_A avec fréquence = 1 Hz
  CCR0 = 32768/1 - 1;
  TACTL = TASSEL_1 + MC_1;  // ACLK, upmode
  CCTL0 = CCIE;             // CCR0 interrupt enabled
```

Terminez le main.c avec l'activation des interruptions et la boucle.

Le concept de base est que ce timer doit déclencher la lecture du capteur en activant (une fois toutes les secondes) l'interrupt du buffer SPI (vector=USART1RX_VECTOR) :

```
#pragma vector= TIMERA0_VECTOR    // Timer_A TACCR0 interrupt vector handler
__interrupt void int_timerA(void)
{
  P2OUT ^= 0xC0;    P2OUT |= 0x02;

  P3OUT &= ~BIT0;               // Activation du CS
  U1TXBUF = 0x00;               // Chargement du buffer de transmission U1TXBUF
  quel_byte = 0;
  IE2 |= URXIE1;
}
```

La lecture de la température et l'affichage LCD se font ainsi dans la routine d'interruption suivante:

```
#pragma vector=USART1RX_VECTOR   // Interrupt du buffer SPI
__interrupt void USART1RX_ISR (void)
{
  unsigned int n, period;
  char str[9];                   // chaine de 9 caractères
  float DT, freq;
  unsigned char duty;

  if (IFG2 & UTXIFG1) {          // USART1 TX buffer rempli ?
    IE2 = 0;                     // disable interrupt

    if (quel_byte == 0) {        // lecture du premier byte (MSBs)
      data = U1RXBUF << 8;

      quel_byte = 1;             // lance lecture 2ème byte
      IE2 |= URXIE1;
      U1TXBUF = 0x00;            // chargement du buffer de transmission U1TXBUF
    }
    else {                       // lecture du 2ème byte (LSBs)
      data = data | U1RXBUF;
      P3OUT |= BIT0;             // relève enable: désactivation du CS
      // Supression des 3 bits de poids faibles du mots de 16 bits issu du capteur
      n = data >> 3;
      DT = n * 0.0625 - T_init;
      freq = 1 + 2*DT;
      period = (unsigned int)(1000.0 / freq);
      duty = 5 + (char)(DT * 100 / 5);

      switch (mode) {   // Affichage sur le LCD
      case 0:                    // Température
        sprintf(str, "TE=%4.1fC", n * 0.0625);
        break;
      case 1:                    // Incrément de température
        sprintf(str, "DT=%4.1fC", DT);
        break;
      case 2:                    // Sortie en fréquence
        sprintf(str, "FR=%3.1fHZ", freq);
        break;
      case 3:                    // Sortie en amplitude
        sprintf(str, "AM=%3d   ", duty);
        break;
      }
      LCD_print(str);            // Affichage de la chaîne de caractères

      ledDimmer_write(period, 50, 7, duty);
    }
  }
}
```

Remarquez comment cette routine sera appelée deux fois pour lire les deux bytes du capteur SPI. Ce qui demande que les variables ***quel_byte*** et ***data*** soient déclarées globales.

C'est seulement lors de la deuxième interruption que la température et ses élaborations en termes de PWM et fréquence des LEDs sont calculées et affichées.

Ensuite l'envoi sur l'I2C est commandé par la fonction `ledDimmer_write(...)`, ce qui enclenche les interruptions du `USCIAB0TX_VECTOR`, dont la programmation se trouve dans le fichier `ledDimmer.c`.

Une fois transmise la « Stop condition » toutes ces interruptions sont desactivées (*disabled*) , sauf celle sur Timer_A, et le MSP430 sera effectivement en veille.

La fonction d'interruption sur P1 permet de changer de mode d'affichage :

```
#pragma vector=PORT1_VECTOR   // routine d'interruption sur P1
__interrupt void Port1_ISR (void)
{
  if (P1IFG == 0x01) {
    P1IE = 0;
    P2OUT ^= 0x01;
    mode += 1;
    if (mode == 4)
      mode = 0;
  }
  P1IFG = 0;
  P1IE  = 0x01;
}
```

Enfin exécutez le programme en mode basse consommation.

Afin de visualiser l'intervalle de temps durant lequel le MSP430 est actif, remarquez qu'on allume la LED P2.1 au début de l'interruption du Timer_A.

Il convient donc d'éteindre celle même LED lors de la « Stop condition » dans la fonction `__interrupt void USCIAB0TX_ISR(void) :`

```
...
case 6:            // STOP condition
    UCB0CTL1 |= UCTXSTP;  // I2C stop condition
    IFG2 &= ~UCB0TXIFG;   // Clear USCI_B0 TX int flag
    IE2 &= ~UCB0TXIE;     // disable TX ready interrupt
    state = E_IDLE;
    P2OUT &= ~0x02;
    break;
...
```

Le court flash de cette LED visualise ainsi le temps durant lequel le MSP430 est effectivement actif.

Questions :

Décrivez **avec vos mots** le fonctionnement du programme et en particulier l'enclenchement des divers interrupts ainsi que le rôle de la fonction `ledDimmer_write(...)`.
Suggestion : suivez le fil des divers registres *interrupt enable* xxIEx ainsi que de la variable *quel_byte*, et tentez de dessiner un organigramme (*flow chart*) du programme.

Quel est le mode basse consommation le plus économique possible pour ce programme ?

Dans ce mode, quels composants sont actifs et quels inactifs ?

Comment se fait-il que le programme tourne dans ce mode même s'il utilise le SMCLK pour la communication I2C ?

6. *Rapport*

- **Répondez à toutes les questions posées.**
- Présentez vos listing and commentez d'éventuels changements et améliorations.