Programmation de la carte Explorer16

1 Introduction

Lors des différents laboratoires en rapport avec les microcontrôleurs (µC), vous serez amenés à programmer des contrôleurs fabriqués par *Microchip*, et plus particulièrement ceux de la famille des dsPIC.

Ces composants, comme de nombreux autres contrôleurs de moyenne puissance, se programment en langage C. Contrairement au C/C++ utilisé pour réaliser des programmes pour des processeurs « classiques » tournant sur des PC ou des smartphones modernes, il n'y a généralement pas de système d'exploitation pour servir d'intermédiaire entre le programme utilisateur et le matériel : le programme écrit constitué l'entièreté des instructions exécutées par le processeur (notons toutefois qu'il est possible de réaliser un système d'exploitation dans ce but).

Cet accès direct au matériel implique qu'il est nécessaire de fournir à l'utilisateur des méthodes de bas niveau permettant d'interagir avec les différents périphériques. Ces méthodes seront décrites tout au long de ce document.

Ce document est décomposé en trois parties :

- Description physique de la carte électronique utilisée lors des laboratoires
- Introduction à la programmation du µC
- Description des différents périphériques disponibles, et guide de programmation individuel

2 Carte à µC

2.1 Introduction

La carte à µC utilisée au laboratoire est construite autour d'un dsPIC33 de Microchip. Ce µC à architecture 16bits est un des plus puissants de sa famille et peut fonctionner jusque 40MIPS. La Figure 1 présente un schéma bloc simplifié. Ce dsPIC intègre plusieurs blocs fonctionnels :

- un **microprocesseur**, cerveau du système qui exécute les opérations et contrôle les différents périphériques à l'aide de signaux de commande ;
- une **mémoire ROM** de type Flash de 256Ko, accueillant le programme devant être exécuté par le processeur ainsi que certaines constantes. Elle ne peut être réécrite en fonctionnement, cette opération doit se faire avant l'exécution du programme via un circuit de programmation présent sur la carte ;
- une **mémoire RAM** de type SRAM de 30ko, contenant toutes les variables temporaires ;
- des **entrées et sorties numériques** regroupées en port (PORTA…PORTG), sur lesquels sont notamment connectées les LED, le clavier et des boutons poussoirs ;
- des **timers** (horloges) fournissant une base de temps permettant entres autres d'exécuter des actions de manière périodique ;
- des **interfaces séries** (UART, I2C, SPI, CAN) ;
- des **convertisseurs A/N** 10 et 12 bits.

Pour rappel : le microcontrôleur intègre tous ces éléments dans une seule puce électronique.

Sur la Figure 2 se trouve le dessin du boitier de la puce électronique. On y retrouve l'ensemble des pattes (=bornes) du µC, ainsi que leur nom. Ces noms seront utiles par la suite.

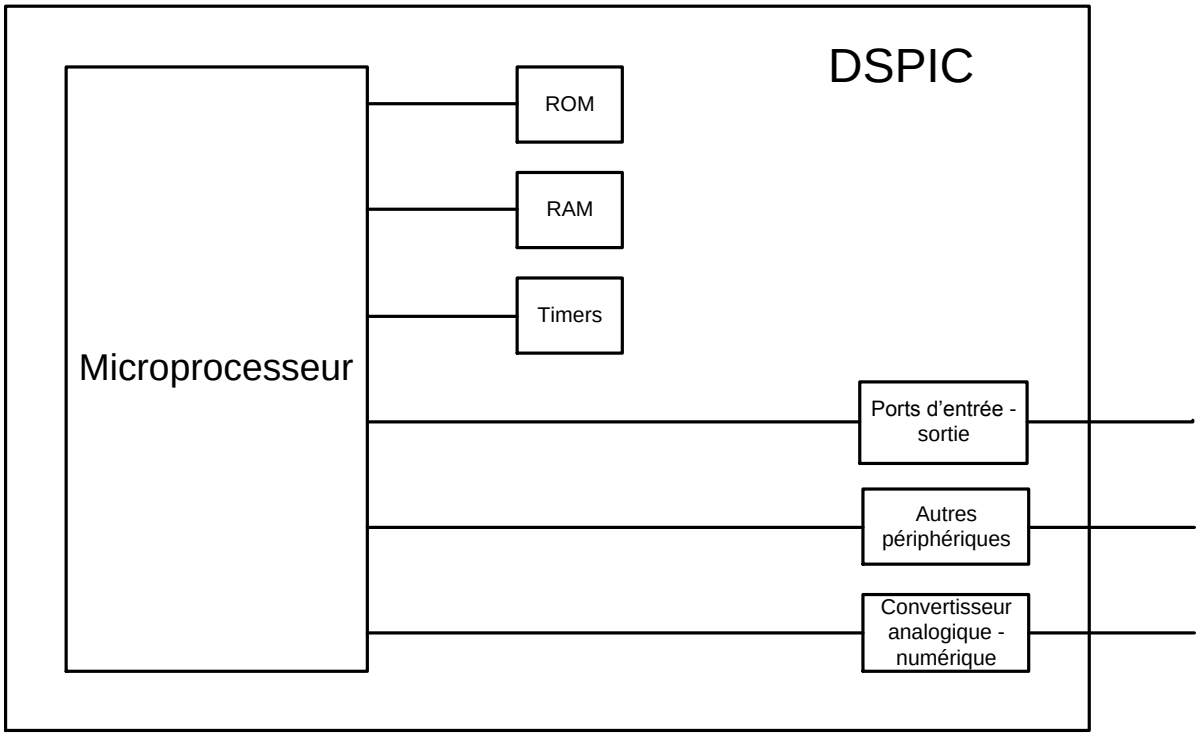


Figure 1 : Schéma bloc simplifié du dsPIC33F

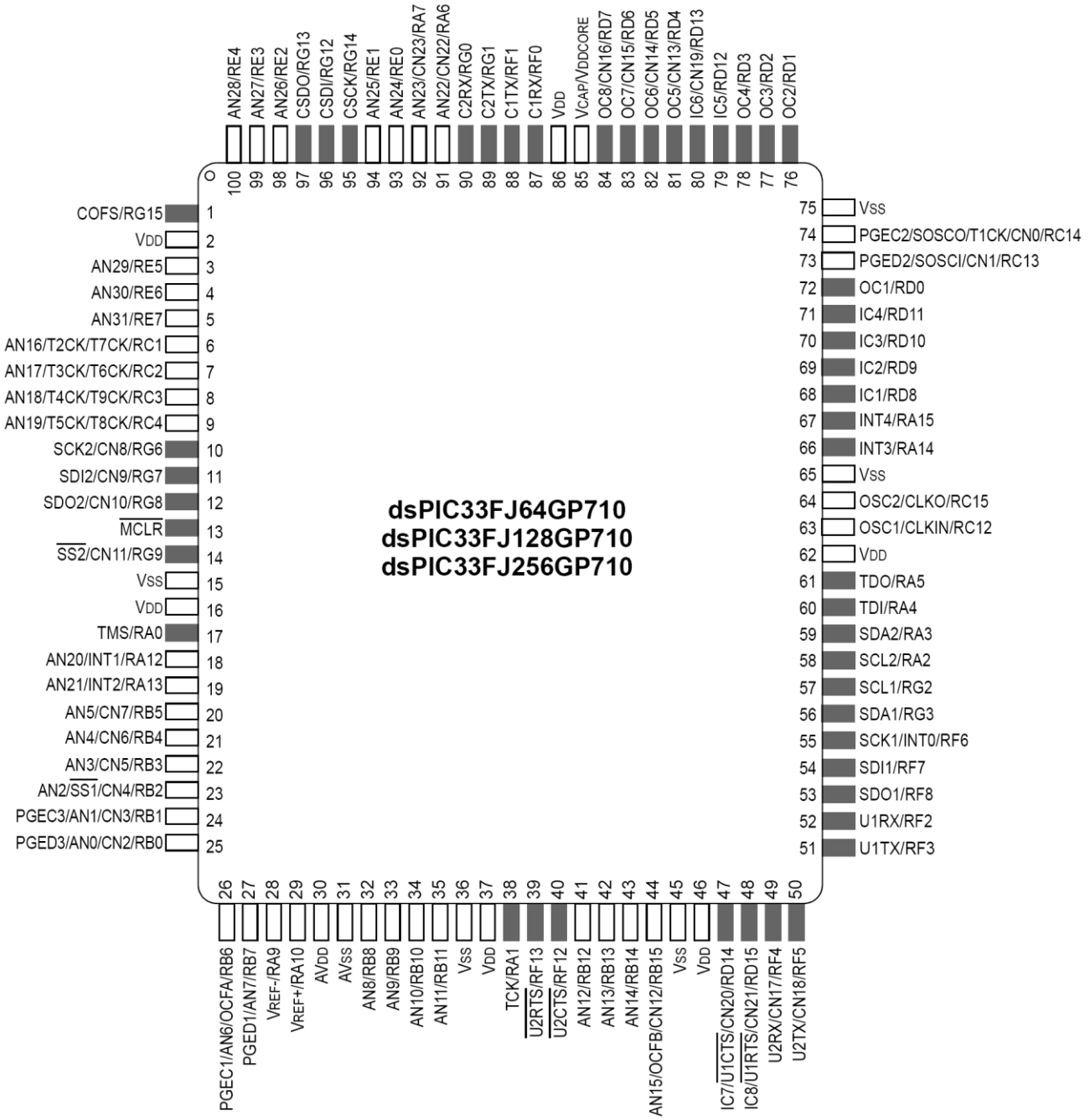


Figure 2 : Schéma des pattes du dsPIC33FJ256GP710

Lors des laboratoires, le système que nous utiliserons est constitué de deux cartes :

- une carte principale Explorer 16 contenant le dsPIC conçu par Microchip
- une carte d'extension rajoutant des circuits permettant d'interagir avec les processus

L'ensemble est alimenté en 3,3V.

Un schéma de principe du système complet est fourni à la Figure 3. Le circuit de programmation permet d'écrire un programme dans la ROM du contrôleur.

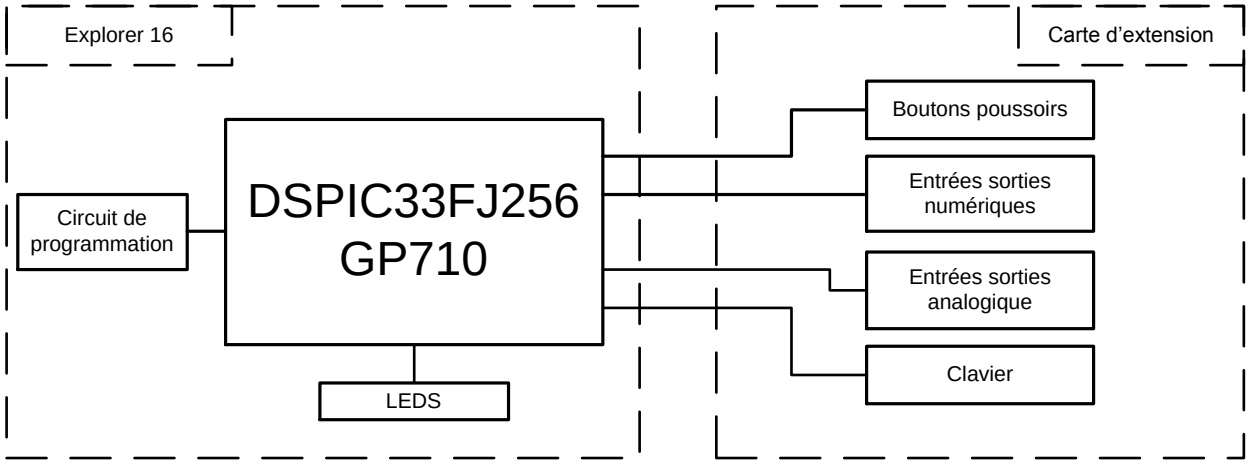


Figure 3 : Schéma de principe du système à µC

2.2 Carte Explorer 16

Cette carte est celle qui contient le dsPIC et le programmateur/débogueur. Un schéma bloc est donné à la Figure 4. On peut y voir :

1. Emplacement pour le µC
2. Connecteur d'alimentation externe. Il alimente des régulateurs qui fournissent des tensions de 3,3V et 5V à toute la carte.
3. LED d'indication de mise sous tension
4. Connecteur DB9 et circuit d'interface RS232. Ce circuit transforme les données provenant de l'UART 2 du dsPIC (pattes 39, 40, 49 et 50) en signaux respectant le standard RS232.
5. Capteur de température. Ce n'est pas celui-ci qui sera utilisé pour les laboratoires.
6. Connecteur USB et µC servant à la programmation et au débogage du dsPIC.
7. Connecteur ICD (non utilisé)
8. Commutateur (non utilisé)
9. Ecran LCD de 2 lignes de 16 caractères. Il est connecté aux ports RD4, RD5, RB15 et RE0 à RE7. Ces I/O ne pourront donc pas être utilisées pour une application.
10. Connecteur pour un LCD supplémentaire. (non utilisé)
11. Boutons connectés sur MCLR#, RD6, RD7, RD13 et RA7. Sur chacune de ces pattes, un pull-up à 3,3V est connecté. (non utilisé)
12. Potentiomètre (non utilisé)
13. 8 LEDs connectées sur RA0 à RA7.
14. Multiplexeurs (non utilisé).
15. Oscillateur à quartz à 8MHz. Cet oscillateur est utilisé pour fournir une horloge à 32MHz via la PLL interne.
16. EEPROM série (non utilisé)
17. Emplacement de prototypage (non utilisé)
18. Connecteur d'extension
19. Connecteur PICkit 2 (non utilisé)
20. Connecteur JTAG (non utilisé)

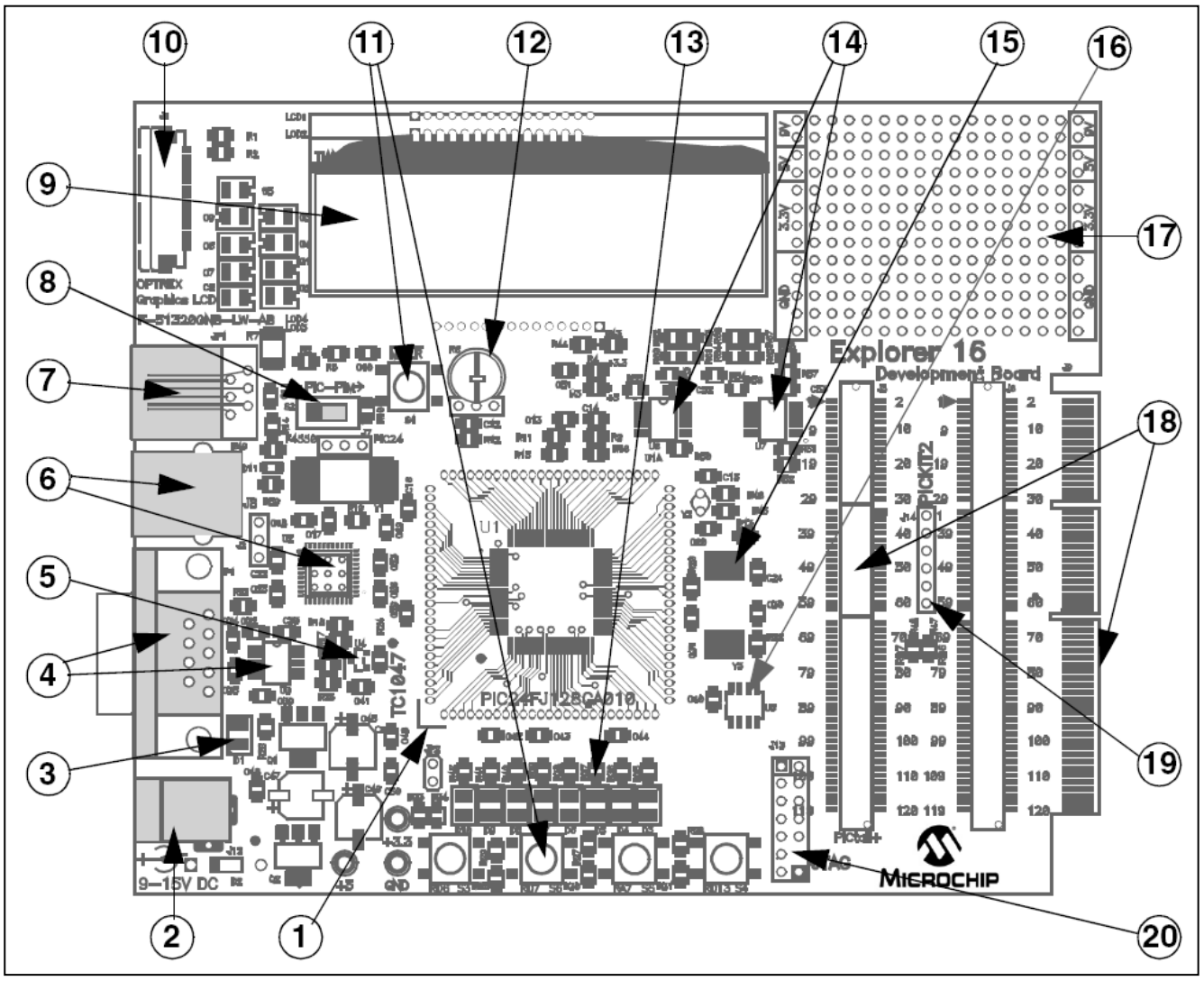


Figure 4 : Schéma bloc de l'Explorer 16

2.2.1 LED

Huit LED sont présentes sur la carte Explorer 16. Elles sont connectées aux pattes RA0 à RA7 et se comportent comme de simples I/O. Pour allumer/éteindre une LED, il suffit d'écrire un '1' ou un '0' sur la patte correspondante.

2.3 Carte d'extension

La carte d'extension est constituée de circuits utilisés dans les différents laboratoires d'électronique. Elle nous permet d'interagir aisément avec l'Explorer16 et contient notamment :

- des boutons,
- un clavier,
- des connecteurs pour des I/O numériques,
- un connecteur pour des entrées analogiques,
- un conditionneur pour une sortie analogique permettant de sortir du son,
- un connecteur pour les sorties PWM.

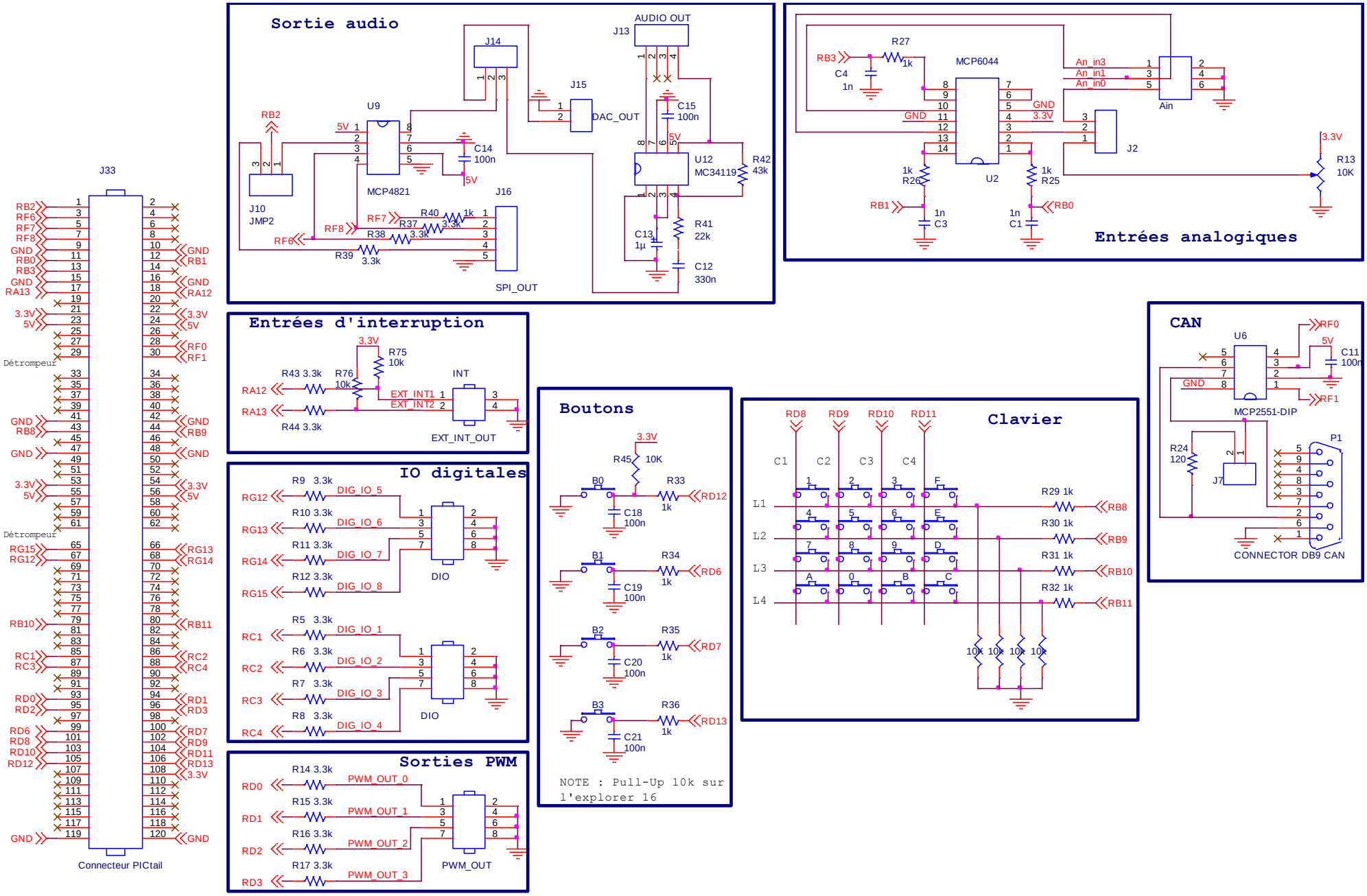


Figure 5 : Schéma électrique de la carte d'extension

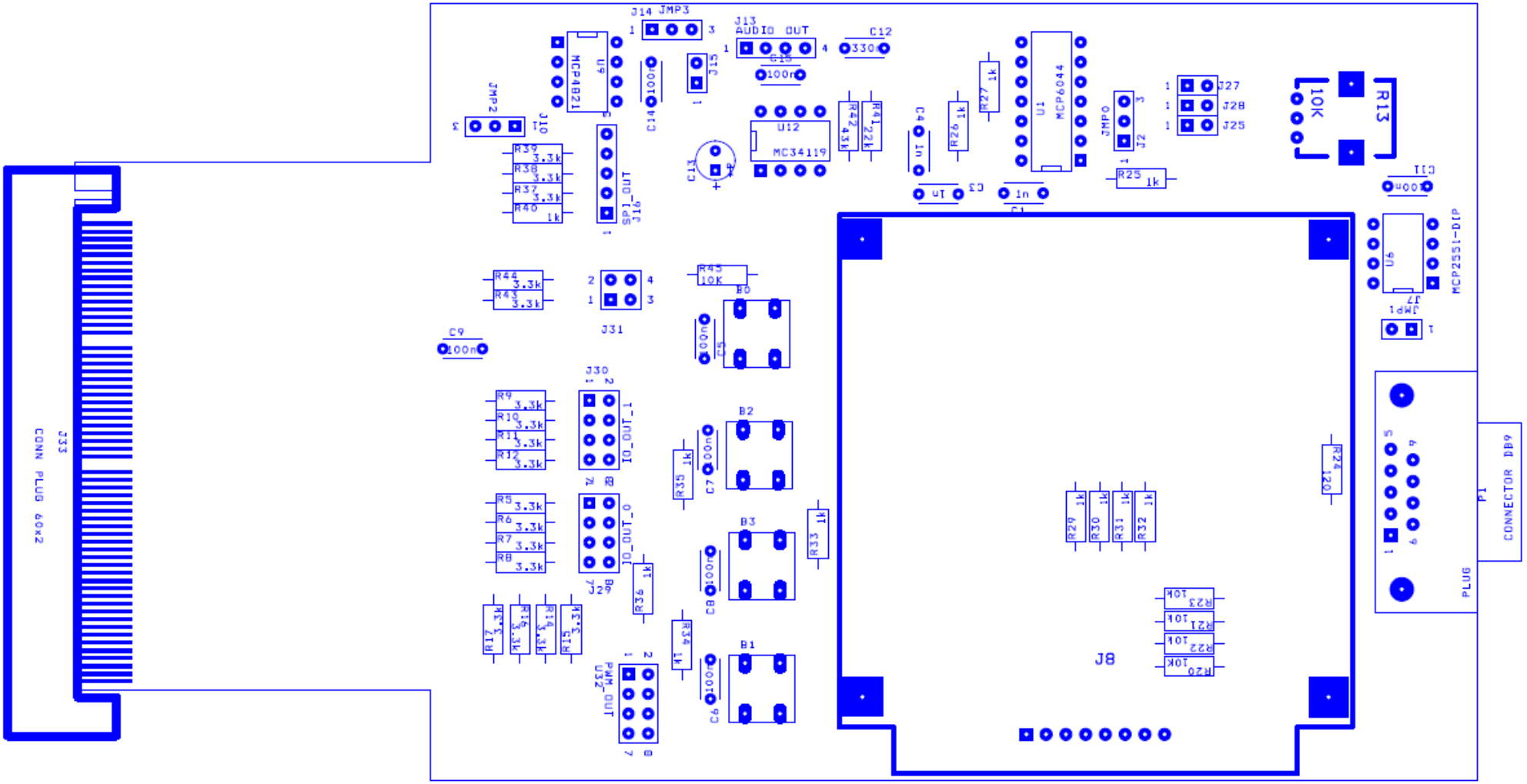


Figure 6 : Emplacement de chaque composant sur la carte d'extension

2.3.1 Boutons-poussoirs

Quatre boutons sont connectés sur la carte d'extension. En regardant la carte d'extension et en partant du haut, ceux-ci sont connectés aux pattes RD12, RD7, RD13 et RD6 du µC.

Le schéma de principe est donné sur la Figure 7. Lorsque l'on n'appuie pas sur un bouton, la patte correspondante mise à 3.3V via la résistance (nommée *pull-up* car elle « tire » la tension vers le haut lorsque l'on n'impose pas volontairement la tension). Lorsque l'on appuie sur le bouton, la patte est reliée à la masse, c'est-à-dire 0V.

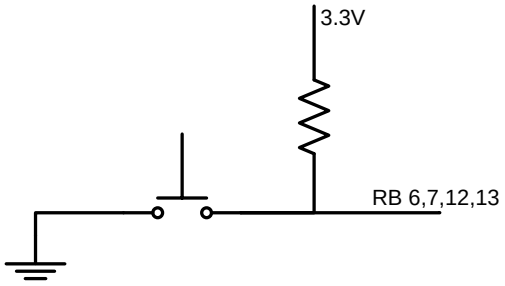


Figure 7 : Boutons-poussoirs

2.3.2 Entrées/sorties digitales

Grâce aux connecteurs DIO, vous disposez de 8 entrées-sorties digitales (RC1 à RC4 et RG12 à RG15). Une de ces sorties vous permettra de vérifier à l'aide d'un oscilloscope la période de votre Timer.

Note : le nom des pattes n'est pas marqué sur la carte d'extension. Pour trouver à quelle borne du connecteur est reliée la patte, se baser sur le nom des résistances (R5 à R17) du schéma de la Figure 5 : ce nom figure également sur la carte d'extension.

2.3.3 Entrées analogiques

Trois entrées analogiques sont disponibles. Elles sont connectées à des amplis-op montés en suiveur pour protéger les entrées du dsPIC. Ces trois entrées correspondent aux pattes AN0, AN1 et AN3 de la Figure 2 (pattes 22, 24 et 25). L'entrée AN0 peut être connectée soit au connecteur, soit à un potentiomètre, grâce au *jumper* J2. Ce potentiomètre offre la possibilité de tester facilement votre code de configuration de l'ADC en fournissant une tension réglable entre 0 et 3,3V.

2.3.4 Sortie audio

Le DSPIC ne contient pas de convertisseur numérique-analogique, un convertisseur externe a donc été ajouté. Le principe est de convertir un nombre codé sur 12 bits en une tension comprise entre 0V et 3.3V.

A la sortie du convertisseur se trouve l'étage de sortie utilisé lors du montage de la radio au laboratoire d'ELEC-H-301 : un filtre passe-haut suivi de l'ampli de puissance NJM2113. Un connecteur permet de brancher un haut-parleur.

2.3.5 Sorties PWM

PWM est l'acronyme pour *Pulse Width Modulation*. Comme son nom l'indique, ce périphérique permet de générer une onde périodique avec des créneaux de largeur variable. Un exemple d'application est la commande de moteurs à courant continu par un hacheur.

3 Programmation

Microchip fournit plusieurs compilateurs pour ses µCs. Le choix du compilateur dépend non seulement du langage de programmation mais également de la famille du µC (dsPIC, PIC18, PIC32, …). Lors des laboratoires, vous programmerez en langage C.

Même si le dsPIC utilisé a une puissance de calcul impressionnante pour un µC, ces ressources sont plus limitées que le processeur de votre ordinateur. Il faut donc en tenir compte lorsque vous programmez pour un µC.

Le programme d'exemple qui vous est fourni a un canevas type simple. En général, la procédure « *main()* » commence par une initialisation des différents périphériques du microcontrôleur. Pendant cette phase, toutes les interruptions doivent être désactivées étant donné que le fonctionnement du périphérique initialisé peut être incertain (les interruptions sont automatiquement désactivés à la mise sous tension du µC, il n'y a donc rien à faire avant l'initialisation des périphériques).

La suite de cette procédure est constituée d'une boucle infinie puisque sur un système embarqué, le programme ne peut pas se terminer. C'est dans cette boucle infinie que certaines taches cycliques ou de « polling » sont effectuées.

Le fichier LCD.c présent dans votre projet contient les procédures d'initialisation et d'affichage du LCD. Ces procédures sont décrites dans le fichier LCD.h.

Il est important d'arriver à écrire un code propre et facilement réutilisable dans un autre projet. Pour cela, en général, on regroupe toutes les fonctions écrites pour commander un même périphérique dans un seul fichier source « .c ». A chaque fichier source contenant des fonctions réutilisables, on associe un fichier « header » de même nom contenant, en plus d'éventuelles autres définitions de constantes et variables, l'entête de ces fonctions (exemple : LCD.c et LCD.h présents dans le sous-répertoire de votre projet)

Si vous voulez utiliser les fonctions définies dans le fichier *nom_fichier.c*, vous devez :

- inclure le fichier en question à votre projet ;
- ajouter une ligne d' « include » au début du fichier utilisant les fonctions :

  #include "*nom_fichier.h*"

4 Syntaxe

Le langage C utilisé pour coder sur un dsPIC est fort semblable à l'ANSI C permettant de coder sur un ordinateur. Les différences viennent du fait que certaines fonctionnalités ne sont pas présentes sur le µC ou sur l'ordinateur.

4.1 Types de variable

Le Tableau 1 montre les types de nombres entiers supportés par le compilateur et le Tableau 2, les types à virgule flottante.

Les pointeurs peuvent également être utilisés.

Il est très important de tenir compte du type de chaque variable. Par exemple, tous les registres sont des mots de 16 bits ; en conséquence, l'instruction « *myVar = PORTB ;* » ne fonctionnera correctement que si « *myVar »* est une variable d'au moins 16 bits (pour plus d'information sur ce que fait cette commande, consulter la section dédiée aux entrées/sorties.

Tableau 1 : Types d'entiers

Type	Bits	Min	Max
char, signed char	8	-128	127
unsigned char	8	0	255
short, signed short	16	-32768	32767
unsigned short	16	0	65535
int, signed int	16	-32768	32767
unsigned int	16	0	65535
long, signed long	32	-2^{31}	$2^{31} - 1$
unsigned long	32	0	$2^{32} - 1$
long long, signed long long	64	-2^{63}	$2^{63} - 1$
unsigned long long	64	0	$2^{64} - 1$

Tableau 2 : Types à virgule flottante (E = exposant, N= valeur normalisée approximative)

Type	Bits	E Min	E Max	N Min	N Max
float	32	-126	127	2^{-126}	2^{128}
double	32	-126	127	2^{-126}	2^{128}
long double	64	-1022	1023	2^{-1022}	2^{1024}

Le produit de deux nombres de 16 bits donne un résultat sur 32 bits. L'instruction « *c = a*b ;* » où *a* et *b* sont des variables de type *int* (16 bits), ne fournira un résultat correct que si la variable c est de type *long* (32 bits).

4.2 Valeur binaire et hexadécimale

Pour assigner une valeur en hexadécimal à une variable ou à un registre, la valeur hexadécimale doit être précédée de « 0x » (ex : int i = 0x01AF;). Pour assigner une valeur en binaire, le nombre binaire doit être précédé de « 0b » (ex : char a = 0b00001010;).

Remarque : la manière dont vous assignez votre variable n'influence pas sa valeur. Toutes les données sont stockées en binaire en mémoire. Si vous écrivez « unsigned char a = 0x0A; », cela équivaut à « unsigned char a = 0b00001010; » ou à « unsigned char a = 10; ».

4.3 Les registres

Un registre est une case mémoire interne au microcontrôleur pouvant être accédée par le programme. Il peut contenir :

- des bits de configuration du µC ou d'un périphérique (ex : choisir si une patte donnée est une entrée ou une sortie) ;
- des bits de statut indiquant l'état du µC ou d'un périphérique (ex : fin d'une conversion analogique-numérique) ;
- des bits de données d'un périphérique (ex : état d'une entrée numérique, résultat d'une conversion, période d'un timer)

Bien que les registres soient vus comme n'importe quelle variable dans le programme, ils présentent toutefois une différence fondamentale avec cette dernière : modifier l'état d'un registre a pour effet de modifier le comportement du hardware !

4.3.1 Accès aux registres

Pour que le compilateur puisse identifier les différents registres dans le code, un fichier « .h » correspondant au type de micro-contrôleur dsPIC est inclus dans le projet. Il définit les noms à utiliser pour chacun des registres. Ces noms sont les mêmes que ceux indiqués dans les notices. Ce fichier définit également des structures à chacun des registres permettant d'y accéder bit-à-bit. Lorsqu'une configuration est associée à plusieurs bits d'un registre, vous n'avez accès qu'à l'ensemble des bits.

L'accès à un registre entier, c'est-à-dire aux 16 bits, est effectué grâce à son nom qui s'utilise comme une variable. Par exemple, pour placer la valeur 5000 dans le registre PR1 du timer 1, il suffit d'écrire « PR1 = 5000 ; ».

Pour accéder à un bit ou un ensemble de bits particulier d'un registre, utiliser le nom du registre suivi du suffixe « bits », puis d'un point suivi du nom du bit (ou de l'ensemble de bits) à modifier (à trouver dans la notice ou dans le présent document).

Par exemple :

- Pour modifier la sortie RB1, vous pouvez écrire « LATBbits.LATB1 = 1; »
- Par contre, pour modifier la priorité de l'interruption du Timer1 codée sur 3 bits, vous ne pouvez pas accéder aux trois bits séparément. Vous devez écrire « IPC0bits.T1IP = 0b001; ».

Exemple de modification de registre

Sur la Figure 8 se trouve un exemple de registre que vous serez amenés à utiliser, permettant de contrôler le fonctionnement du Timer 1, et portant le nom de T1CON. Il s'agit d'un mot (=variable) de 16 bits, et la valeur de chacun des bits va modifier le comportement du Timer 1.

La variable T1CON est vue comme un nombre non-signé par le µC, il est donc possible d'écrire un nombre dans la variable (ex T1CON = 8230 ;) et de modifier tous les bits en une fois, mais ce n'est la méthode la plus intuitive, ni la plus claire lorsque l'on désire relire son code, c'est pourquoi nous allons décrire une méthode plus évidente.

En **gras** sur la figure se trouve le nom des différents champs contenus dans cette variable. Il est possible d'accéder à ces champs de manière individuelle. Par exemple nous voyons que le champ TON sert à activer ou désactiver le Timer. Pour y accéder, nous écrivons par exemple

T1CONbits.TON = 1 ;

Cette ligne pour effet de ne modifier que le bit correspondant à TON, et dans le cas présent cela lancera le Timer. Cette méthode d'accès est identique pour tous les registres.

Note : dans la suite du texte, nous utiliserons souvent des phrases du type « le bit T1IF du registre IFS0 ». Cela signifie que ce bit est un champ du registre IFS0, et est accessible via la commande IFS0bits.T1IF.

R/W-0	U-0	R/W-0	U-0	U-0	U-0	U-0	U-0
TON	—	TSIDL	—	—	—	—	—
bit 15							bit 8

U-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	U-0
—	TGATE	TCKPS<1:0>		—	TSYNC	TCS	—
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15 **TON:** Timer On bit
1 = Starts the timer
0 = Stops the timer

bit 14 **Unimplemented:** Read as '0'

bit 13 **TSIDL:** Stop in Idle Mode bit
1 = Discontinue timer operation when device enters Idle mode
0 = Continue timer operation in Idle mode

bit 12-7 **Unimplemented:** Read as '0'

bit 6 **TGATE:** Timer Gated Time Accumulation Enable bit
When TCS = 1:
This bit is ignored
When TCS = 0:
1 = Gated time accumulation enabled
0 = Gated time accumulation disabled

bit 5-4 **TCKPS<1:0>:** Timer Input Clock Prescale Select bits
11 = 1:256 prescale value
10 = 1:64 prescale value
01 = 1:8 prescale value
00 = 1:1 prescale value

bit 3 **Unimplemented:** Read as '0'

bit 2 **TSYNC:** Timer External Clock Input Synchronization Select bit
When TCS = 1:
1 = Synchronize external clock input
0 = Do not synchronize external clock input
When TCS = 0:
This bit is ignored. Read as '0'. Timerx uses the internal clock when TCS = 0

bit 1 **TCS:** Timer Clock Source Select bit
1 = External clock from TxCK pin
0 = Internal clock (Fosc/2)

bit 0 **Unimplemented:** Read as '0'

Figure 8 : Registre T1CON

4.3.2 Modification d'un registre par masque

Il est souvent utile de modifier certains bits d'un registre ou d'une variable sans modifier les autres bits. Pour cela, il faut travailler à l'aide de masques. Par exemple, supposons que variable « a » de type « char » soit initialisé avec la valeur 0x48. Si vous voulez mettre à 0 le bit 6 de « a », effectuez l'opération suivante :

a = a & 0xBF ;

Ce qui se traduit par :

01001000
AND 10111111
a 00001000

En utilisant un masque dont tous les bits étaient à 1 excepté le bit 6, le bit 6 de « a » est bien mis à 0.

Le Tableau 3 rappelle les différents opérateurs logiques en C.

Tableau 3 : Opérateurs logiques en C

Opérateur	Signification	Syntaxe
&	ET bit à bit	9 & 12 (1001 & 1100)
|	OU bit à bit	9 | 12 (1001 | 1100)
^	OU exclusif bit à bit	9 ^ 12 (1001 ^ 1100)

4.4 Variables globales

Pour des raisons de simplicité et d'efficacité, les variables globales sont couramment utilisées dans les programmes pour µC.

La déclaration de ces variables est très simple lorsqu'elles sont utilisées que dans une seule unité de compilation[1] ou module. Dans ce cas, cette déclaration se fait en dehors de toutes fonctions.

Par contre, lorsqu'une variable globale est partagée par plusieurs modules, cette déclaration produira une erreur de compilation. En effet, lors de la génération du code objet[2] d'un programme constitué de plusieurs modules, chaque unité est compilée séparément donnant un fichier objet pour chacune d'elle. Ensuite, ces fichiers objet sont liés à l'aide d'un éditeur de liens fournissant ainsi le code pouvant être exécuté par le µC.

Il est donc nécessaire d'utiliser une méthode pour indiquer au compilateur et à l'éditeur de lien que des variables globales sont utilisées par plusieurs modules. La méthode utilise des déclarations dissymétriques : certaines déclarations sont des définitions et d'autres sont des références. Un seul module peut définir un nom, et plusieurs unités peuvent le référencer. Pour cela, il faut fournir la référence à l'aide d'un fichier d'entête (« .h ») dans lequel la variable est déclarée avec le mot-clé « extern ». Ce mot-clé indique que la variable est définie ailleurs.

Le code suivant montre les déclarations de variables globales dans plusieurs modules.

```
/***************main.c***************/        //module 1
int variable_globale1 ;              //Définition des variables globales
int variable_globale2 ;

int main(void)
{
        variable_globale1=0 ;              //Initialisation des variables globales
        variable_globale2=0 ;
        while(1){                          //boucle infinie
                if(variable_globale1)      variable_globale2==1 ;  //utilisation des variables
        }
}
/***************common.h***************/
extern int variable_globale1 ;             //Déclaration des références
extern int variable_globale2 ;

/***************utilisation.c***************/  //Module 2
#include « common.h » ;                  //Importation des références
void utilisation(void)
{
        if(variable_globale2){             //Utilisation des références
                variable_globale1==0 ;
                variable_globale2==0 ;
        }else
                variable_globale2==1 ;
}
```

[1] Unité de compilation : fichier « .c » contenant des définitions de fonctions.

[2] Code objet : programme en langage machine produit par le compilateur.

5 Première prise en main

Ce chapitre fournit des informations simplifiées qui vous serviront pour les deux premiers laboratoires. Des explications plus complètes sont données dans la suite de ce document.

Vous serez amenés à utiliser les versions plus complètes dans les labos suivants. Lors des laboratoires d'introduction, des simplifications sont réalisées par une pré-configuration des registres.

5.1 Ports d'entrée-sortie

Les pattes d'entrée-sortie (I/O pour input-output) constituent le moyen le plus basique de communiquer avec l'extérieur.

Le DSPIC contient différentes séries d'I/O, nommées PORTA à PORTG et contenant chacune de 8 à 16 pattes distinctes. Les I/O du port A sont nommées RA0 à RA15, le principe est le même pour les autres ports (RB0 à RB15, RE0 à RE7, …).

Chaque patte peut être configurée en entrée ou en sortie. Avant de lire ou d'écrire sur une patte, il faut donc indiquer au processeur s'il s'agit d'une entrée ou d'une sortie. Ceci se fait par l'intermédiaire des registres TRIS. Pour indiquer que la patte RA3 du port A est une entrée, il faut mettre à 1 le quatrième bit du registre TRISA (la numérotation des pattes d'un port commence à 0). L'écriture d'un 0 sur ce même bit correspond à la configuration de la patte en sortie. Pour cela, il est possible d'accéder au bit à l'aide des structures :

```
TRISAbits.TRISA3=1 ;
```

ou de configurer tout un port en une fois :

```
TRISB = 0x1E5C ; // Configure tout le port B en une seule fois.
```

Ensuite, pour modifier l'état d'une sortie ou connaître l'état d'une entrée, il faut également accéder à des registres. Une écriture dans le registre LATx modifiera l'état des pattes du port x. Par exemple pour imposer une tension de 3,3V sur la patte RA4, il faut mettre à 1 le cinquième bit du registre LATA :

```
LATAbits.LATA4 = 1 ; //écrit un '1' sur la patte RA4
```

Pour connaître l'état d'une patte d'un port, il faut lire dans le registre PORTx le bit correspondant à cette patte.

Exemple :

```
Int myVar = PORTB ; //lit la valeur de toutes les pattes du PORTB comme un
                    mot de 16 bits et l'écrit dans la variable myVar.
If(PORTBbits.RB10){…}// vérifie si la patte RB10 est à l'état 1.
```

En résumé, trois registres sont utilisés :

- TRISx permet de choisir le sens de la borne (entrée ou sortie)
- LATx permet d'imposer la tension sur une borne de sortie
- PORTx permet de lire la tension sur une borne d'entrée

Note : par la suite, vous devrez faire attention à ce qu'une patte donnée ne soit pas également utilisée par le convertisseur analogique-numérique. Pour ce faire, lisez la section consacrée aux bornes d'IO plus loin dans ce guide

5.2 Timer

Les timers permettent d'exécuter une fonction à intervalle fixe, de mesurer une durée ou d'obtenir une base de temps.

Le principe est le suivant : on commence par entrer une période dans le registre PRx (ou x est le numéro du timer, de 1 à 9). Cette période est exprimée en nombre de cycles d'exécution du processeur. Le processeur a été configuré pour exécuter les instructions à une fréquence de 40 MHz, nommée Fcy par la suite.

```
PR2 = 10000 ; //indique au timer 2 que sa période est de
```
$\frac{10000}{40\,10^6} = 250\mu s$

Le processeur va alors incrémenter un compteur à chaque coup d'horloge jusqu'à arriver à la valeur de PR2. A ce moment, le bit T2IF, qui au départ vaut '0', est mis à '1' par le processeur, ce qui permet de détecter le changement via un test dans le code du µC. Dans le même temps, le compteur est automatiquement remis à zéro et le comptage reprend. Pour les timers 1 à 3, ce bit se trouve dans le registre IFS0.

Attention : ce bit doit être remis à '0' par une instruction dans le programme. Dans le cas contraire un test du type

```
if(IFS0bits.T1IF == 1)
```

serait toujours vérifié.

Ce bit est en réalité le flag de l'interruption associée au timer (voir la section 0 sur les interruptions).

Pour lancer le timer, il suffit de mettre à '1' le bit TON du registre TxCON. Mettre ce même bit à '0' gèle le compteur à sa valeur actuelle.

```
T1CONbits.TON = 1 ; //lance le timer1
```

Le registre des timers étant sur 16 bits, la période que l'on peut représenter est limitée à maximum 1638µs, bien insuffisante dans de nombreux cas. Une manière de contourner ce problème est d'utiliser un *prescaler*. Il s'agit d'un diviseur de l'horloge alimentant le timer par un facteur 8, 64 ou 256. Le prescaler se règle en modifiant le champ `TxCONbits.TCKPS` codé sur 2 bits. Il peut prendre les valeurs :

TCKPS=	Division
0b00 (défaut)	1
0b01	8
0b10	64
0b11	256

En résumé, les étapes pour **configurer** le Timer sont

- Calcul de la période
- Ecriture de la période dans PRx
- Eventuellement, ajouter un préscaling
- Lancer le timer

Les étapes à réaliser **une fois le timer lancé** sont

- Vérification du bit TxIF afin de voir si la période est écoulée
- Si le test est vérifié, ne pas oublier de remettre le bit à '0'

5.3 Le convertisseur analogique-numérique

Le microcontrôleur intègre un convertisseur analogique-numérique (analog to digital converter , ou ADC). Par défaut, la grandeur numérique comprise entre 0V et 3.3V obtenue est codée sur 10 bits. Une tension de 0V donnera un résultat de 0 et une tension de 3.3V donnera un résultat de 1023.

Le convertisseur possède 32 entrées analogiques nommées AN0 à AN31, dont seules 3 sont accessibles sur la carte d'extension (AN0, AN1 et AN3).

La mise à 0 du bit SAMP du registre AD1CON1 permet de démarrer une conversion. Cette conversion n'est pas instantanée, elle prend quelques µs.

```
AD1CON1bits.SAMP=0 ; //lance une conversion
```

Une fois la conversion terminée, son résultat est écrit dans le registre ADC1BUF0 (lu comme une variable de 16 bits) et le bit AD1IF du registre IFS0 est mis à '1'. La fin de la conversion peut donc être détectée en en vérifiant la valeur. Ce bit doit être remis à zéro manuellement. AD1IF est également le flag d'interruption du convertisseur (voir plus loin)

La carte est configurée pour connecter un potentiomètre à la patte AN0. Ce potentiomètre est monté en diviseur résistif, permettant ainsi de faire la tension sur la patte AN0 de 0V à 3.3V.

Il est également possible de coupler l'ADC au Timer 3, de sorte que l'échantillonnage et la conversion soient démarrés automatiquement à chaque débordement du Timer. Pour ce faire, le champ `AD1CON1bits.SSRC` doit être modifié (voir Figure 9). Dans ce cas, aucun traitement software n'est

nécessaire pour démarrer le convertisseur lorsque le timer a terminé, tout est réalisé en hardware. Le bit T3IF ne doit donc ni être testé, ni être remis à '0'. Il est en revanche nécessaire de lancer le Timer 3 via le bit T3CONbits.TON

5.4 Les interruptions

Dans un système à µC, de nombreux événements peuvent survenir à tout moment. Ces événements sont généralement dit « asynchrones », ce qui signifie que leur moment d'arrivée n'est pas prévisible dans le code.

L'apparition d'un événement peut être vérifiée de deux manières :

- Par scrutation (ou polling) systématique et continuelle d'un bit nommé flag. La plupart des événements (timers, fin de conversion …) préviennent en effet de leur occurrence en modifiant un bit spécifique. La vérification régulière de ce bit permet de traiter l'évènement en exécutant les tâches appropriées. A titre d'exemple, le bit TxIF permettant de vérifier le débordement d'un timer est également son bit de flag.
- Par déclenchement d'une interruption. Lorsque le flag lié à un événement est activé, le processeur interrompt automatiquement le programme et exécute une fonction dédiée à cette source d'interruption. Cette fonction est nommée routine de service d'interruption ou ISR (Interrupt Service Routine). L'ISR est démarrée automatiquement par le processeur et ne doit pas appelée par une instruction dans le programme.
  La procédure pour qu'une interruption soit déclenchée à l'apparition d'un événement est la suivante :
  - Dans l'étape de configuration, l'interruption est activée en mettant à '1' le bit *Interrupt Enable* qui lui correspond. Dans le cas des timers, ces bits sont `IEC0bits.TxIE` (où x est le numéro du timer (1 à 3)) et le bit correspondant à la fin d'une conversion A/N est `IEC0bits.AD1IE`.
  - Ecriture de l'ISR, correspondant au traitement de l'événement.
    Cette fonction **doit** avoir la syntaxe suivante :
    `void _ISR _nom_de_l'interruption (void) {…}`, où le nom de l'interruption est `_TxInterrupt` ou `_ADC1Interrupt` pour les périphériques utilisés dans les premiers labos. Attention à la présence d'un espace entre « void » et « _ISR » ainsi qu'entre « _ISR » et « _T1Interrupt »

  Le flag lié à l'interruption doit être remis à zéro dans la routine. Pour les timers, ce bit est `IFS0bits.T1IF` et pour l'ADC, il se nomme `IFS0bits.AD1IF`

R/W-0	U-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0
ADON	—	ADSIDL	ADDMABM	—	AD12B	FORM<1:0>	
bit 15							bit 8

R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0 HC,HS	R/C-0 HC, HS
	SSRC<2:0>		—	SIMSAM	ASAM	SAMP	DONE
bit 7							bit 0

Legend:	HC = Cleared by hardware	HS = Set by hardware	
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15 **ADON:** ADC Operating Mode bit
1 = ADC module is operating
0 = ADC is off

bit 14 **Unimplemented:** Read as '0'

bit 13 **ADSIDL:** Stop in Idle Mode bit
1 = Discontinue module operation when device enters Idle mode
0 = Continue module operation in Idle mode

bit 12 **ADDMABM:** DMA Buffer Build Mode bit
1 = DMA buffers are written in the order of conversion. The module provides an address to the DMA channel that is the same as the address used for the non-DMA stand-alone buffer.
0 = DMA buffers are written in Scatter/Gather mode. The module provides a Scatter/Gather address to the DMA channel, based on the index of the analog input and the size of the DMA buffer.

bit 11 **Unimplemented:** Read as '0'

bit 10 **AD12B:** 10-bit or 12-bit Operation Mode bit
1 = 12-bit, 1-channel ADC operation
0 = 10-bit, 4-channel ADC operation

bit 9-8 **FORM<1:0>:** Data Output Format bits
For 10-bit operation:
11 = Signed fractional (DOUT = sddd dddd dd00 0000, where s = .NOT.d<9>)
10 = Fractional (DOUT = dddd dddd dd00 0000)
01 = Signed integer (DOUT = ssss sssd dddd dddd, where s = .NOT.d<9>)
00 = Integer (DOUT = 0000 00dd dddd dddd)
For 12-bit operation:
11 = Signed fractional (DOUT = sddd dddd dddd 0000, where s = .NOT.d<11>)
10 = Fractional (DOUT = dddd dddd dddd 0000)
01 = Signed Integer (DOUT = ssss sddd dddd dddd, where s = .NOT.d<11>)
00 = Integer (DOUT = 0000 dddd dddd dddd)

bit 7-5 **SSRC<2:0>:** Sample Clock Source Select bits
111 = Internal counter ends sampling and starts conversion (auto-convert)
110 = Reserved
101 = Reserved
100 = Reserved
011 = MPWM interval ends sampling and starts conversion
010 = GP timer (Timer3 for ADC1, Timer5 for ADC2) compare ends sampling and starts conversion
001 = Active transition on INTx pin ends sampling and starts conversion
000 = Clearing sample bit ends sampling and starts conversion

bit 4 **Unimplemented:** Read as '0'

bit 3 **SIMSAM:** Simultaneous Sample Select bit (only applicable when CHPS<1:0> = 01 or 1x)
When AD12B = 1, SIMSAM is: U-0, Unimplemented, Read as '0'
1 = Samples CH0, CH1, CH2, CH3 simultaneously (when CHPS<1:0> = 1x); or
Samples CH0 and CH1 simultaneously (when CHPS<1:0> = 01)
0 = Samples multiple channels individually in sequence

bit 2 **ASAM:** ADC Sample Auto-Start bit
1 = Sampling begins immediately after last conversion. SAMP bit is auto-set.
0 = Sampling begins when SAMP bit is set

bit 1 **SAMP:** ADC Sample Enable bit
1 = ADC Sample/Hold amplifiers are sampling
0 = ADC Sample/Hold amplifiers are holding
If ASAM = 0, software can write '1' to begin sampling. Automatically set by hardware if ASAM = 1.
If SSRC = 000, software can write '0' to end sampling and start conversion. If SSRC ≠ 000, automatically cleared by hardware to end sampling and start conversion.

bit 0 **DONE:** ADC Conversion Status bit
1 = ADC conversion cycle is completed.
0 = ADC conversion not started or in progress
Automatically set by hardware when A/D conversion is complete. Software can write '0' to clear DONE status (software not allowed to write '1'). Clearing this bit does NOT affect any operation in progress. Automatically cleared by hardware at start of a new conversion.

Note 1: The 'x' in ADxCON1 and ADCx refers to ADC 1 or ADC 2.

Figure 9 : Registre AD1CON1

6 Interaction avec les périphériques

Dans les sections qui suivent, nous allons décrire les principaux périphériques des dsPIC. Après une introduction à son fonctionnement, nous décrirons le mode opératoire recommandé pour les laboratoires. Dans de nombreux cas, ces périphériques possèdent des modes de fonctionnement plus complexes, mais inutiles dans notre cas. Si cela vous intéresse, leur fonctionnement complet se trouve dans la notice de la famille dsPIC (plus de 1000 pages !)

7 Ports I/O

7.1 Introduction

Le dsPIC33FJ256GP710 possède 100 pattes dont 85 peuvent être utilisé comme des entrées/sorties digitales. Ces pattes sont le moyen le plus simple pour le processeur d'interagir avec le monde extérieur. En tant qu'entrée, permettent de lire l'état logique de la patte. En tant que sorties, elles permettent au processeur d'imposer une tension de 0V ou de 3.3V sur la borne. Les autres pattes servent à l'alimentation, au reset et à l'oscillateur.

Toutes les IO sont partagées avec d'autres périphériques (UART, SPI, ADC,…). Généralement, une patte ne peut utiliser qu'une seule de ces fonctions à un instant donné, mais cette fonction peut être changée à tout moment dans le programme.

Une I/O comporte donc des multiplexeurs permettant de sélectionner sa fonction. Lorsqu'un périphérique est en fonction, les pattes correspondant ne peuvent pas être utilisées en I/O banale.

La structure d'une entrée/sortie est schématisée à la Figure 10.

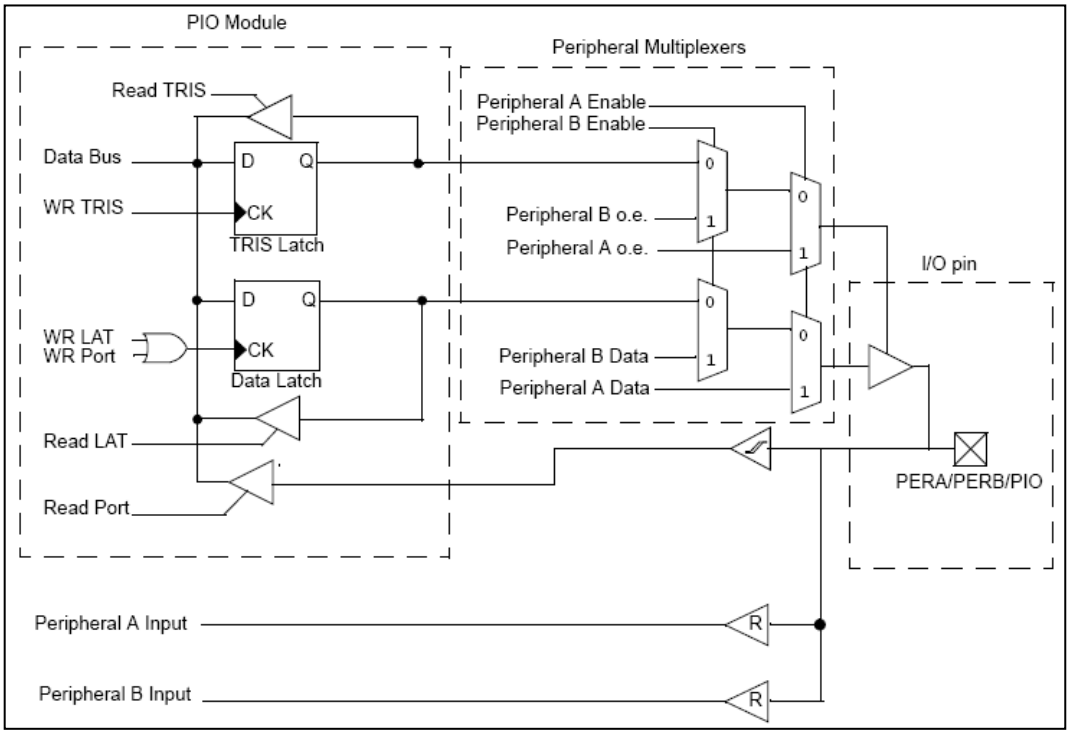


Figure 10 : Schéma bloc d'une I/O

Les I/O sont rassemblées en ports identifié par un « R » suivi d'une lettre (A, B, C, D, E, F et G) sur la Figure 2. Un port est composé de maximum 16 I/O.

7.2 Registres de contrôle des I/O

Trois registres sont associés à chaque port, permettant de les configurer :

- TRISx : registre contrôlant la direction (entrée ou sortie)
- PORTx : registre d'I/O, utilisé en lecture
- LATx : registre du « Data Latch », utilisé en écriture

Le « x » représente une lettre qui identifie un port particulier (A, B,…).

Chaque I/O d'un port est associée à un bit dans chacun de ces trois registres.

7.2.1 Registre TRIS

Ce registre permet de définir si une patte est une entrée ou une sortie. Si un bit de ce registre est à 1, la patte associée est une entrée, tandis que si le bit est à 0, la patte est une sortie (pour le retenir, « 0 » ressemble au « O » de « Output »). Après un reset, toutes les pattes sont configurées en entrée.

Exemple de code :

- Accès au registre TRIS du port B : TRISB = 0x00F0;
- Accès au bit 2 du registre TRIS du port B : TRISBbits.TRISB2 = 0;

7.2.2 Registre PORT

Le registre PORT permet d'accéder directement à la tension des bornes. En théorie, ce registre permet non seulement de lire l'état logique ('0' pour 0V, '1' pour 3.3V) de la patte, mais également d'en imposer l'état via le programme (ceci est tributaire de la valeur écrite dans le registre TRIS correspondant : on ne peut imposer l'état d'une borne depuis le programme du µC si celle-ci est configurée en entrée).

Toutefois, en raison de problèmes apparaissant lors des opérations de « read-modify-write », il est recommandé de n'utiliser le registre PORT que lors des lectures (ie. patte en entrée), et jamais lors des écritures. Ces problèmes sont décrits dans le chapitre « Entrée-sorties numériques » du syllabus de cours

Exemple de code :

- Accès au bit 2 du registre PORT du port B (borne RB2) , et écriture de la valeur (0 ou 1) dans la variable a :
  a = PORTBbits.RB2;
- Lecture simultanée de toutes les bornes du port B, et écriture des 16 bits correspondant dans la variable a : a = PORTB;

7.2.3 Registre LAT

Ce registre élimine le problème qui peut apparaître avec les instructions « read-modify-write ». Lors de l'écriture de la valeur à imposer sur une borne de sortie, la valeur est écrite dans un buffer au lieu de directement imposer la tension de la borne.

Exemple de code :

- Accès au registre LAT du port B : LATB = 0x00F0;
- Accès au bit 2 du registre LAT du port B : LATBbits.LATB2 = 0;

7.2.4 Multiplexage avec les périphériques internes

Quand un périphérique est activé, le buffer de sortie de la patte associée est en général contrôlé par ce dernier, cependant certaines sorties doivent être configurées dans le code à l'aide des registres cités ci-dessus.

L'UART est un exemple de contrôle du buffer de sortie par le périphérique. Quand l'UART est activé, les registres PORT et TRIS n'ont aucun effet et ne peuvent pas être utilisés pour écrire sur les pattes RX et TX (pattes de réception et de transmission). La plupart des périphériques de communication contrôlent le buffer de sortie de leurs pattes associées.

7.2.5 Particularité des entrées analogiques

Certaines bornes d'IO du µC peuvent également servir d'entrée au convertisseur analogique-numérique. Ces deux fonctions ne sont pas disponibles simultanément, et il est nécessaire de spécifier au µC dans quel mode la borne est utilisée via des registres spécifiques.

Par défaut, ces bornes mixtes sont mises en mode analogique, et doivent donc être reconfigurées avant usage si vous voulez les utiliser en entrée/sortie logique (cf 7.3 pour le mode opératoire)!

Note : cette étape ne nous dispense pas de spécifier la direction de la borne via le registre TRIS

Les registres AD1PCFGL et AD1PCFGH sont décrits aux Figure 27 et Figure 28 de la section 13.

7.3 Mode opératoire

Voici, en résumé, les étapes nécessaires à la configuration d'une borne d'entrée/sortie

- Repérer la borne sur le schéma de la Figure 2. La borne aura généralement plusieurs « noms », par exemple AN20/INT1/RA12
- Si cette liste de noms en contient un du type ANx (x allant de 0 à 31), cela signifie que l'entrée x du convertisseur analogique numérique est également présente sur la patte. Si c'est le cas, il est nécessaire de configurer la patte en mode numérique (elle est en mode analogique par défaut) en écrivant un '1' dans le bit PCFGx du registre AD1PCFGL (de AN0 à AN15) ou AD1PCFGH (de AN16 à AN31)

  Exemple : on souhaite utiliser la borne RB1 en tant que sortie digitale. La Figure 2 nous indique que le nom complet de cette borne est PGEC3/**AN1**/INT3/**RB1**, elle sert donc d'entrée au convertisseur analogique-numérique.
  Avant de l'utiliser en tant que sortie, nous devons donc spécifier au processeur que la borne est utilisée en mode binaire, via la ligne AD1PCFGLbits.PCFG**1** = 1 .

- Configurer la direction (entrée ou sortie) à l'aide du registre TRIS correspondant. Par exemple, TRIS**A**bits.TRIS**A6** = 1 met la borne RA6 en entrée. (= 0 pour la mettre en sortie)

Et les étapes pour interagir avec cette borne lors du programme sont:

- Pour écrire sur la patte, et ainsi fixer la tension à sa sortie, utiliser les registres LAT. Par exemple LAT**B**bits.LAT**B3** = 0 impose 0V sur la borne RB3
- Pour lire la tension présente sur la patte et écrire la valeur correspondante (0 ou 1) dans une variable, utiliser les registres PORT. Par exemple z= PORT**A**bits.R**A2** affecte l'état logique de la borne RA2 à la variable z.

7.4 Remarque liée au clavier

Dans le cas du clavier, le laps de temps entre l'instruction d'écriture pour mettre une colonne à l'état haut et le passage effectif de la colonne à HI est de plusieurs cycles machine. Ceci est dû à la sortance assez faible du dsPIC, provoquant une montée lente de la tension.

Pour détecter correction la pression sur une touche, vous devrez appeler la fonction « __delay_us() » entre l'instruction d'écriture sur une colonne et de lecture sur les lignes.

8 Timer

8.1 Introduction

Un Timer/Counter est un périphérique permettant de compter un nombre d'événements, et de prévenir l'utilisateur lorsque ce nombre atteint un nombre prédéfini.

De manière générale, on peut compter :

- Un nombre de périodes de l'horloge du processeur, on parle alors de *Timer* car, connaissant la durée d'une période, on peut en déduire une durée arbitraire
- Un nombre d'événements externes, et plus précisément le nombre de flancs montant apparaissant sur une borne d'entrée spécifique. On parle alors de *Counter*

Dans la suite de ce document, nous nous concentrerons sur l'utilisation en tant que Timer.

Pour rappel (cf. cours), le principe général d'un timer est le suivant :

- à chaque coup d'horloge (40 MHz dans le cadre des labos, mais cette fréquence est réglable), un registre de 16bits, nommé TMRx (x étant le numéro du timer) est incrémenté de 1
- la valeur de ce registre est ensuite comparée à celle contenue dans un second registre, dit « de période », nommé PRx
- En cas d'égalité, le timer repart de 0 (ie. TMRx = 0) et prévient de son débordement en mettant à '1' un bit spécifique nommé TxIF (que l'on trouve dans le registre IFS0 à IFS4 selon le numéro du timer)

Le dsPIC33F possède 9 timers, numérotés de 1 à 9, qui présentent des légères différences.

Individuellement, un timer possède un registre de période de 16bits. Ceci implique qu'il ne compter que de 0 à 65535. Pour pallier à ce problème, il est possible de combiner deux timers pour obtenir un registre de période de 32bits, ou bien on peut demander à un timer de ne s'incrémenter qu' « une fois sur n » , on parle alors de prescaling. Ces modes spécifiques sont également décrits ci-dessous

8.2 Utilisation en mode 16 bits

Avant d'utiliser un timer, il est nécessaire de préciser sa période exprimée en nombre de cycles machine via le registre PRx qui lui est associé :

```
PR2 = 10000 ; //indique au timer 2 que sa période est de 10000/(40 10^6) = 250µs
```

Lors de cette configuration, il est également possible de spécifier au processeur que le débordement du timer doit provoquer l'envoi d'une interruption (cf section consacrée aux interruptions pour plus de détails). Pour ce faire, il faut activer l'interruption en écrivant '1' dans le bit TxIE du registre IEC0 (timers 1 à 3) ou IEC1 (timers 4 et 5) :

```
IEC0bits.T1IE ; //active l'interruption du timer 1
```

Il suffit enfin de le lancer, en écrivant '1' dans le bit TON de son registre de contrôle TxCON (où x est le numéro du timer)

```
T3CONbits.TON = 1 ; //lance le timer 3
```

Lorsque le timer déborde, son bit de flag est mis à 1. Les mesures à prendre changent selon que l'interruption associée est activée ou non.

- Si l'interruption est désactivée, le débordement se détecte en testant la valeur du bit de flag (ex `if(IFS0bits.T3IF == 1)`)
- Si l'interruption est activée, ce test est inutile et la routine d'interruption est automatiquement appelée par le processeur

Dans tous les cas, le bit TxIF doit être remis à zéro de manière software (ie. par une instruction de votre programme.

8.3 Prescaling

Le registre des timers étant sur 16 bits, la période que l'on peut représenter est limitée à maximum 1638µs dans le cas d'une fréquence d'horloge de 40MHz. Cette période est bien insuffisante dans de nombreux cas. Une manière de contourner ce problème est d'utiliser un *prescaler*. Il s'agit d'un diviseur de l'horloge incrémentant le timer par un facteur 8, 64 ou 256 : le timer ne s'incrémente donc qu'une fois sur N. Le prescaler se règle en modifiant le champ `TxCONbits.TCKPS` codé sur 2 bits. Il peut prendre les valeurs :

TCKPS=	Division
0b00 (défaut)	1
0b01	8
0b10	64
0b11	256

8.4 Mode opératoire en mode 16bits

En résumé, les étapes à suivre lors de la configuration du timer x sont les suivantes :

- Ecriture de la période dans **PRx**
- Configuration éventuelle du prescaler , via le champ **TxCONbits.SSRC**
- Activation éventuelle de l'interruption liée au timer via le bit **IFS0bits.TxIE** (timer 1 à 3) ou **IFS1bits.TxIE** (timer 4 et 5)
- Lancement du timer via le bit **TxCONbits.TON**

8.5 Utilisation en mode 32bits

Il est possible de combiner les timers 2 et 3 ainsi que les timers 4 et 5 afin de former un timer possédant une période codée sur 32bits, permettant donc de compter jusqu'à $2^{32} - 1$ flancs d'horloge.

Dans la suite, nous décrivons le fonctionnement pour la paire timer 2 / timer 3. La paire 4/5 a un fonctionnement identique.

Lorsque la paire est configurée en mode 32bits, le timer 2 s'incrémente à chaque coup d'horloge comme normalement. En revanche, il ne retourne pas à zéro lorsque TMR2 = PR2, mais uniquement lorsqu'il arrive à sa période maximale de 65535. Lorsqu'il retourne à zéro le registre de comptage du timer 3 s'incrémente de 1.

Lorsque TMR2 = PR2 ET TMR3 = PR3, alors seulement l'ensemble retourne à zéro et le flag est mis à '1'.

Le timer 2 contient donc les 16 bits de poids faible, et le timer 3 les 16 bits de poids fort

Note : le timer de 32 bits est contrôlé par le registre du timer 2 (notamment pour le lancement du timer ou la configuration de prescaler). En revanche, c'est le flag du timer 3 (IFS0bits.T3IF) qui est mis à '1' lors du débordement !

8.6 Mode opératoire en mode 32bits

En résumé, les étapes à suivre lors de la configuration du timer 2/3 (ou 4/5, 6/7 et 8/9) en mode 32 bits sont les suivantes :

- Faire basculer la paire 2/3 en mode 32 bits via le bit **T2CONbits.T32**
- Configuration éventuelle du prescaler, via le champ **T2CONbits.SSRC**
- Ecriture des 16 bits de poids faible de la période dans **PR2**
- Ecriture des 16 bits de poids fort de la période dans **PR3**
- Activation éventuelle de l'interruption liée au timer via le bit **IFS0bits.T3IE**
- Lancement du timer via le bit **T2CONbits.TON**

8.7 Registres de contrôle

R/W-0	U-0	R/W-0	U-0	U-0	U-0	U-0	U-0
TON	—	TSIDL	—	—	—	—	—
bit 15							bit 8

U-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	U-0
—	TGATE	TCKPS<1:0>		—	TSYNC	TCS	—
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15 **TON:** Timer On bit
1 = Starts the timer
0 = Stops the timer

bit 14 **Unimplemented:** Read as '0'

bit 13 **TSIDL:** Stop in Idle Mode bit
1 = Discontinue timer operation when device enters Idle mode
0 = Continue timer operation in Idle mode

bit 12-7 **Unimplemented:** Read as '0'

bit 6 **TGATE:** Timer Gated Time Accumulation Enable bit
When TCS = 1:
This bit is ignored
When TCS = 0:
1 = Gated time accumulation enabled
0 = Gated time accumulation disabled

bit 5-4 **TCKPS<1:0>:** Timer Input Clock Prescale Select bits
11 = 1:256 prescale value
10 = 1:64 prescale value
01 = 1:8 prescale value
00 = 1:1 prescale value

bit 3 **Unimplemented:** Read as '0'

bit 2 **TSYNC:** Timer External Clock Input Synchronization Select bit
When TCS = 1:
1 = Synchronize external clock input
0 = Do not synchronize external clock input
When TCS = 0:
This bit is ignored. Read as '0'. Timerx uses the internal clock when TCS = 0

bit 1 **TCS:** Timer Clock Source Select bit
1 = External clock from TxCK pin
0 = Internal clock (Fosc/2)

bit 0 **Unimplemented:** Read as '0'

Figure 11 : Registre T1CON pour type A (Timer Control)

R/W-0	U-0	R/W-0	U-0	U-0	U-0	U-0	U-0
TON	—	TSIDL	—	—	—	—	—
bit 15							bit 8

U-0	R/W-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	U-0
—	TGATE	TCKPS<1:0>		T32	—	TCS	—
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15 **TON:** Timerx On bit
When T32 = 1 (in 32-bit Timer mode):
1 = Starts 32-bit TMRy(1):TMRx timer pair
0 = Stops 32-bit TMRy(1):TMRx timer pair
When T32 = 0 (in 16-bit Timer mode):
1 = Starts 16-bit timer
0 = Stops 16-bit timer

bit 14 **Unimplemented:** Read as '0'

bit 13 **TSIDL:** Stop in Idle Mode bit
1 = Discontinue timer operation when device enters Idle mode
0 = Continue timer operation in Idle mode

bit 12-7 **Unimplemented:** Read as '0'

bit 6 **TGATE:** Timerx Gated Time Accumulation Enable bit
When TCS = 1:
This bit is ignored
When TCS = 0:
1 = Gated time accumulation enabled
0 = Gated time accumulation disabled

bit 5-4 **TCKPS<1:0>:** Timerx Input Clock Prescale Select bits
11 = 1:256 prescale value
10 = 1:64 prescale value
01 = 1:8 prescale value
00 = 1:1 prescale value

bit 3 **T32:** 32-Bit Timerx Mode Select bit
1 = TMRx and TMRy(1) form a 32-bit timer
0 = TMRx and TMRy(1) form separate 16-bit timer

bit 2 **Unimplemented:** Read as '0'

bit 1 **TCS:** Timerx Clock Source Select bit
1 = External clock from TxCK pin
0 = Internal clock (Fosc/2)

bit 0 **Unimplemented:** Read as '0'

Note 1: TMRy is a Type C timer (y = 3, 5, 7 and 9).

Figure 12 : Registre TxCON pour timers 2,4,6,8

R/W-0	U-0	R/W-0	U-0	U-0	U-0	U-0	U-0
TON(2)	—	TSIDL(1)	—	—	—	—	—
bit 15							bit 8

U-0	R/W-0	R/W-0	R/W-0	U-0	U-0	R/W-0	U-0
—	TGATE(2)	TCKPS<1:0>(2)		—	—	TCS(2)	—
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15 **TON:** Timerx On bit(2)
1 = Starts 16-bit Timerx
0 = Stops 16-bit Timerx

bit 14 **Unimplemented:** Read as '0'

bit 13 **TSIDL:** Stop in Idle Mode bit (1)
1 = Discontinue timer operation when device enters Idle mode
0 = Continue timer operation in Idle mode

bit 12-7 **Unimplemented:** Read as '0'

bit 6 **TGATE:** Timerx Gated Time Accumulation Enable bit(2)
When TCS = 1:
This bit is ignored
When TCS = 0:
1 = Gated time accumulation enabled
0 = Gated time accumulation disabled

bit 5-4 **TCKPS<1:0>:** Timerx Input Clock Prescale Select bits(2)
11 = 1:256 prescale value
10 = 1:64 prescale value
01 = 1:8 prescale value
00 = 1:1 prescale value

bit 3-2 **Unimplemented:** Read as '0'

bit 1 **TCS:** Timerx Clock Source Select bit(2)
1 = External clock from TxCK pin
0 = Internal clock (Fosc/2)

bit 0 **Unimplemented:** Read as '0'

Note 1: When 32-bit timer operation is enabled (T32 = 1) in the Type B Timer Control (TxCON<3>) register, TSIDL bit must be cleared to operate the 32-bit timer in Idle mode.

2: These bits have no effect when the 32-bit timer operation is enabled (T32 = 1) in the Type B Timer Control (TxCON<3>) register.

Figure 13 : Registre TxCON pour timers 3,5,7,9

9 Interruptions

9.1 Introduction

Une interruption est une suspension de l'exécution du code courant pour traiter un événement d'origine interne ou externe, périodique ou apériodique.

Ces évènements sont par exemple : un flanc sur une entrée spécifique, une comparaison satisfaite (Timer), une fin de conversion analogique/numérique, la réception d'un caractère par l'UART, ... Le dsPIC33F permet 118 sources d'interruption.

Un évènement peut être détecté de deux manières qui différent par la fréquence à laquelle est vérifié de l'apparition d'un évènement et par la modalité du démarrage de son traitement :

- La scrutation récurrente dans le code (généralement, le test d'un flag dans la boucle infinie). Cette méthode est appelée *polling*. Cet évènement est alors traité juste après ce test soit dans le code principal, soit par l'appel d'une fonction. Dans ce cas, le µC continue l'exécution du programme et n'interrompt aucune autre tâche.
  Cette opération est typiquement réalisée par un test du type :
  `if(IFS0bits.T1IF==1)`
- La prise en compte automatique des flags dès la fin de l'instruction en cours. Dans ce cas, l'évènement provoque l'arret automatique de la tâche en cours afin d'exécuter le code lié à l'interruption, nommé *routine d'interruption ou ISR (Interrupt Service Routine)*.

Il est important de souligner que dans le cas du polling, l'appel à une fonction de traitement est une ligne de code. Tandis que si l'interruption est autorisée, aucune ligne de code ne se trouve dans le programme pour l'appel de l'ISR.

Configurer une interruption revient à écrire du code à deux endroits différents :

- Dans la fonction *main*, il est nécessaire de signaler au processeur qu'il doit tenir compte de l'événement en tant que source d'interruption
- Hors de toute fonction, écrire la routine d'interruption. Il s'agit d'une fonction qui sera appelée par le processeur dès que l'événement en question survient

9.2 Déroulement d'une interruption

Lorsque, pour quelque raison que ce soit, un flag d'interruption est mis à '1' ET que l'interruption correspondante est active, le processeur procède aux étapes suivantes :

1. Le µC termine l'exécution de l'instruction en cours ;
2. Le contexte actuel (variables locales) est sauvegardé
3. La routine d'interruption est automatiquement appelée par le processeur
4. Au terme de l'exécution de la routine, restauration du contexte
5. Le programme reprend son exécution normale, à partir de l'endroit où il avait été interrompu

Toutes ces étapes sont réalisées automatiquement et indépendamment de la tâche en cours.

Pour éviter qu'une interruption ne soit appelée en boucle, il est nécessaire de remettre à '0' le flag dpar une instruction d'écriture au sein de l'ISR

9.3 Syntaxe pour la routine d'interruption

De façon à ce que le compilateur puisse lier chaque source d'interruption à sa routine, cette dernière doit être définie de la manière suivante :

`void _ISR nom_interruption(void){ }` Note : attention aux espaces !

Le nom « nom_interruption » doit correspondre à une des entrées de la colonne « Primary Name » du

Tableau 4. Ce nom comment toujours un underscore '_'

Tableau 4 : Vecteur d'interruption

Primary Name	Interrupt Source
_INT0Interrupt	INT0 – External Interrupt 0
_IC1Interrupt	IC1 – Input Compare 1
_OC1Interrupt	OC1 – Output Compare 1
_T1Interrupt	T1 – Timer1
_DMA0Interrupt	DMA0 – DMA Channel 0
_IC2Interrupt	IC2 – Input Capture 2
_OC2Interrupt	OC2 – Output Compare 2
_T2Interrupt	T2 – Timer2
_T3Interrupt	T3 – Timer3
_ADC1Interrupt	ADC1 – ADC 1
_INT1Interrupt	INT1 – External Interrupt 1
_ADC2Interrupt	ADC2 – ADC 2
_T4Interrupt	T4 – Timer4
_T5Interrupt	T5 – Timer5
_INT2Interrupt	INT2 – External Interrupt 2
_U2RXInterrupt	U2RX – UART2 Receiver
_U2TXInterrupt	U2TX – UART2 Transmitter

9.4 Registres de contrôles et de statuts des interruptions

Le registre INTCON1 () contient le bit NSTDIS permettant ou non "l'interrupt nesting". Si ce bit est activé, l'interruption en cours ne sera jamais interrompue. Les autres sont des bits de statuts pour les sources de problèmes du µC.

Les registres IFSx (x étant un chiffre de 0 à 7) comportent tous les flags des requêtes d'interruption. Ce sont ces flags qui doivent être remis à zéro à la fin de chaque ISR.

Les registres IECx (x étant un chiffre de 0 à 7) permettent d'autoriser ou d'inhiber chaque source d'interruption.

Note : attention à ne pas confondre le bit d'autorisation (ex IEC0bits.T1IE) avec le bit de flag (IFS0bits.T1IF) : le premier indique au processeur que si l'événement se produit, il entraine l'envoi d'une interruption alors que le second indique que l'événement s'est produit (ce bit est mis à '1' dès que l'événement se produit, et ce même si l'interruption correspondant n'est pas activée)

9.5 Procédure d'initialisation des interruptions

Activer une interruption liée à un périphérique est très simple :

1. configurer les registres propres au périphérique,
2. autoriser l'interruption (bit xxxIE des registres IECy , ex IEC0bits.AD1IE = 1 pour l'ADC). Ce nom est précisé dans la section dédiée à chaque périphérique
3. Ne pas oublier d'écrire la routine d'interruption en dehors du main()

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	DMA1IF	AD1IF	U1TXIF	U1RXIF	SPI1IF	SPI1EIF	T3IF
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
T2IF	OC2IF	IC2IF	DMA0IF	T1IF	OC1IF	IC1IF	INT0IF
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15 **Unimplemented:** Read as '0'

bit 14 **DMA1IF:** DMA Channel 1 Data Transfer Complete Interrupt Flag Status bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 13 **AD1IF:** ADC1 Conversion Complete Interrupt Flag Status bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 12 **U1TXIF:** UART1 Transmitter Interrupt Flag Status bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 11 **U1RXIF:** UART1 Receiver Interrupt Flag Status bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 10 **SPI1IF:** SPI1 Event Interrupt Flag Status bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 9 **SPI1EIF:** SPI1 Fault Interrupt Flag Status bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 8 **T3IF:** Timer3 Interrupt Flag Status bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 7 **T2IF:** Timer2 Interrupt Flag Status bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 6 **OC2IF:** Output Compare Channel 2 Interrupt Flag Status bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 5 **IC2IF:** Input Capture Channel 2 Interrupt Flag Status bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 4 **DMA0IF:** DMA Channel 0 Data Transfer Complete Interrupt Flag Status bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 3 **T1IF:** Timer1 Interrupt Flag Status bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 2 **OC1IF:** Output Compare Channel 1 Interrupt Flag Status bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 1 **IC1IF:** Input Capture Channel 1 Interrupt Flag Status bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

bit 0 **INT0IF:** External Interrupt 0 Flag Status bit
1 = Interrupt request has occurred
0 = Interrupt request has not occurred

Figure 14 : Registre IFS0 (Interrupt Flag Status)

U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	DMA1IE	AD1IE	U1TXIE	U1RXIE	SPI1IE	SPI1EIE	T3IE
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
T2IE	OC2IE	IC2IE	DMA0IE	T1IE	OC1IE	IC1IE	INT0IE
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15 **Unimplemented:** Read as '0'

bit 14 **DMA1IE:** DMA Channel 1 Data Transfer Complete Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 13 **AD1IE:** ADC1 Conversion Complete Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 12 **U1TXIE:** UART1 Transmitter Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 11 **U1RXIE:** UART1 Receiver Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 10 **SPI1IE:** SPI1 Event Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 9 **SPI1EIE:** SPI1 Error Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 8 **T3IE:** Timer3 Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 7 **T2IE:** Timer2 Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 6 **OC2IE:** Output Compare Channel 2 Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 5 **IC2IE:** Input Capture Channel 2 Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 4 **DMA0IE:** DMA Channel 0 Data Transfer Complete Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 3 **T1IE:** Timer1 Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 2 **OC1IE:** Output Compare Channel 1 Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 1 **IC1IE:** Input Capture Channel 1 Interrupt Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

bit 0 **INT0IE:** External Interrupt 0 Enable bit
1 = Interrupt request enabled
0 = Interrupt request not enabled

Figure 15 : Registre IEC0 (Interrupt Enable Control)

File Name	ADR	Bit 15	Bit 14	Bit 13	Bit 12	Bit 11	Bit 10	Bit 9	Bit 8	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	All Resets
INTCON1	0080	NSTDIS	OVAERR	OVBERR	COVAERR	COVBERR	OVATE	OVBTE	COVTE	SFTACERR	DIV0ERR	DMACERR	MATHERR	ADDRERR	STKERR	OSCFAIL	—	0000
INTCON2	0082	ALTIVT	DISI	—	—	—	—	—	—	—	—	—	INT4EP	INT3EP	INT2EP	INT1EP	INT0EP	0000
IFS0	0084	—	DMA1IF	AD1IF	U1TXIF	U1RXIF	SPI1IF	SPI1EIF	T3IF	T2IF	OC2IF	IC2IF	DMA0IF	T1IF	OC1IF	IC1IF	INT0IF	0000
IFS1	0086	U2TXIF	U2RXIF	INT2IF	T5IF	T4IF	OC4IF	OC3IF	DMA2IF	IC8IF	IC7IF	AD2IF	INT1IF	CNIF	—	MI2C1IF	SI2C1IF	0000
IFS2	0088	T6IF	DMA4IF	—	OC8IF	OC7IF	OC6IF	OC5IF	IC6IF	IC5IF	IC4IF	IC3IF	DMA3IF	C1IF	C1RXIF	SPI2IF	SPI2EIF	0000
IFS3	008A	FLTAIF	—	DMA5IF	DCIIF	DCIEIF	QEIIF	PWMIF	C2IF	C2RXIF	INT4IF	INT3IF	T9IF	T8IF	MI2C2IF	SI2C2IF	T7IF	0000
IFS4	008C	—	—	—	—	—	—	—	—	C2TXIF	C1TXIF	DMA7IF	DMA6IF	—	U2EIF	U1EIF	FLTBIF	0000
IFS5	008E	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
IFS6	0090	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
IFS7	0092	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
IEC0	0094	—	DMA1IE	AD1IE	U1TXIE	U1RXIE	SPI1IE	SPI1EIE	T3IE	T2IE	OC2IE	IC2IE	DMA0IE	T1IE	OC1IE	IC1IE	INT0IE	0000
IEC1	0096	U2TXIE	U2RXIE	INT2IE	T5IE	T4IE	OC4IE	OC3IE	DMA2IE	IC8IE	IC7IE	AD2IE	INT1IE	CNIE	—	MI2C1IE	SI2C1IE	0000
IEC2	0098	T6IE	DMA4IE	—	OC8IE	OC7IE	OC6IE	OC5IE	IC6IE	IC5IE	IC4IE	IC3IE	DMA3IE	C1IE	C1RXIE	SPI2IE	SPI2EIE	0000
IEC3	009A	FLTAIE	—	DMA5IE	DCIIE	DCIEIE	QEIIE	PWMIE	C2IE	C2RXIE	INT4IE	INT3IE	T9IE	T8IE	MI2C2IE	SI2C2IE	T7IE	0000
IEC4	009C	—	—	—	—	—	—	—	—	C2TXIE	C1TXIE	DMA7IE	DMA6IE	—	U2EIE	U1EIE	FLTBIE	0000
IEC5	009E	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
IEC6	00A0	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
IEC7	00A2	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	—	0000
IPC0	00A4	—	T1IP<2:0>			—	OC1IP<2:0>			—	IC1IP<2:0>			—	INT0IP<2:0>			4444
IPC1	00A6	—	T2IP<2:0>			—	OC2IP<2:0>			—	IC2IP<2:0>			—	DMA0IP<2:0>			4444
IPC2	00A8	—	U1RXIP<2:0>			—	SPI1IP<2:0>			—	SPI1EIP<2:0>			—	T3IP<2:0>			4444
IPC3	00AA	—	—	—	—	—	DMA1IP<2:0>			—	AD1IP<2:0>			—	U1TXIP<2:0>			4444
IPC4	00AC	—	CNIP<2:0>			—	—	—	—	—	MI2C1IP<2:0>			—	SI2C1IP<2:0>			4444
IPC5	00AE	—	IC8IP<2:0>			—	IC7IP<2:0>			—	AD2IP<2:0>			—	INT1IP<2:0>			4444
IPC6	00B0	—	T4IP<2:0>			—	OC4IP<2:0>			—	OC3IP<2:0>			—	DMA2IP<2:0>			4444
IPC7	00B2	—	U2TXIP<2:0>			—	U2RXIP<2:0>			—	INT2IP<2:0>			—	T5IP<2:0>			4444
IPC8	00B4	—	C1IP<2:0>			—	C1RXIP<2:0>			—	SPI2IP<2:0>			—	SPI2EIP<2:0>			4444
IPC9	00B6	—	IC5IP<2:0>			—	IC4IP<2:0>			—	IC3IP<2:0>			—	DMA3IP<2:0>			4444
IPC10	00B8	—	OC7IP<2:0>			—	OC6IP<2:0>			—	OC5IP<2:0>			—	IC6IP<2:0>			4444
IPC11	00BA	—	T6IP<2:0>			—	DMA4IP<2:0>			—	—	—	—	—	OC8IP<2:0>			4444
IPC12	00BC	—	T8IP<2:0>			—	MI2C2IP<2:0>			—	SI2C2IP<2:0>			—	T7IP<2:0>			4444
IPC13	00BE	—	C2RXIP<2:0>			—	INT4IP<2:0>			—	INT3IP<2:0>			—	T9IP<2:0>			4444

Legend: — = unimplemented, read as '0'. Reset values are shown in hexadecimal.

Figure 16 : Plan des registres de contrôle des interruptions

10 Interruption externe

Le dsPIC33F possède 5 entrées d'interruption externe. Chacune d'elles peut être configurée pour déclencher une interruption sur un flanc montant ou descendant à l'aide du registre INTCON2 ().

Pour qu'un flanc déclenche une interruption, il ne faut pas oublier de configurer la patte en entrée, en plus de l'initialisation générale des sources d'interruptions.

R/W-0	R-0	U-0	U-0	U-0	U-0	U-0	U-0
ALTIVT	DISI	—	—	—	—	—	—
bit 15							bit 8

U-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	INT4EP	INT3EP	INT2EP	INT1EP	INT0EP
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15 **ALTIVT:** Enable Alternate Interrupt Vector Table bit
1 = Use alternate vector table
0 = Use standard (default) vector table

bit 14 **DISI:** DISI Instruction Status bit
1 = DISI instruction is active
0 = DISI instruction is not active

bit 13-5 **Unimplemented:** Read as '0'

bit 4 **INT4EP:** External Interrupt 4 Edge Detect Polarity Select bit
1 = Interrupt on negative edge
0 = Interrupt on positive edge

bit 3 **INT3EP:** External Interrupt 3 Edge Detect Polarity Select bit
1 = Interrupt on negative edge
0 = Interrupt on positive edge

bit 2 **INT2EP:** External Interrupt 2 Edge Detect Polarity Select bit
1 = Interrupt on negative edge
0 = Interrupt on positive edge

bit 1 **INT1EP:** External Interrupt 1 Edge Detect Polarity Select bit
1 = Interrupt on negative edge
0 = Interrupt on positive edge

bit 0 **INT0EP:** External Interrupt 0 Edge Detect Polarity Select bit
1 = Interrupt on negative edge
0 = Interrupt on positive edge

Figure 17 : Registre INTCON2 (Interrupt Control 2)

11 PWM / Générateur d'ondes rectangulaires

11.1 Introduction

Le module PWM (Pulse Width Modulation) permet, comme son nom l'indique, de générer sur une borne d'E/S un signal en onde carrée dont la fréquence ainsi que la durée de l'état haut peuvent être modifiée.

Le dsPIC33F possède 8 périphériques « Output Compare » connectés au port RD (RD0 à RD7, renommées à cette occasion OC1 à OC8). Dans la suite, un 'x' remplacera le numéro du périphérique Output Compare (1 à 8) dans les différents noms de registres et de sorties (x = 1 à 8). La lettre 'y' sera quant à elle, utilisée pour désigner le Timer 2 ou 3.

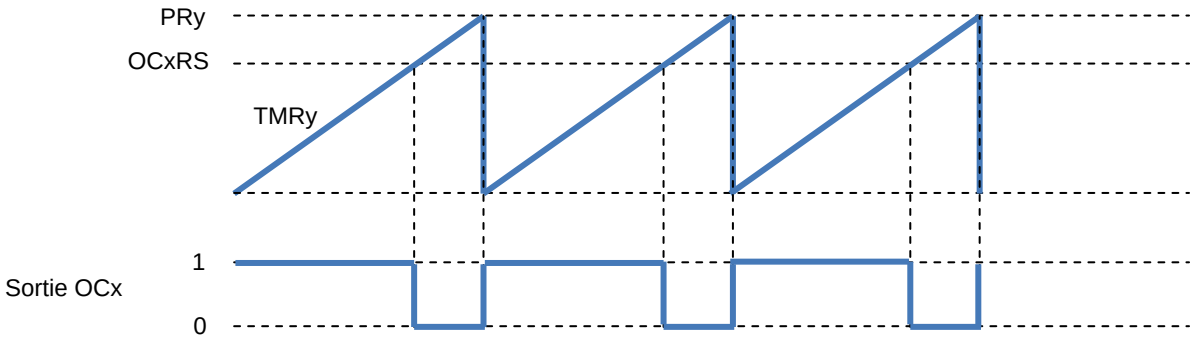
En effet, le principe de fonctionnement du module PWM peut être expliquée comme celui d'un Timer possédant deux registres de comparaison au lieu d'un :

- Le registre PRy permet, comme d'habitude, de fixer la période du Timer. Dans notre cas, cela a pour effet de fixer la période de l'onde rectangulaire
- Le registre OCxRS contient la durée de l'état haut de l'onde rectangulaire. Cette durée est exprimée dans les mêmes unités que PRy

Le fonctionnement du module est le suivant :

- Au lancement du timer 2 ou 3, la sortie OCx est mise à '1' et le registre TMRy commence à s'incrémenter
- Lorsque TMRy = OCxRS , la sortie OCx bascule à '0'
- Lorsque TMRy = PRy , le timer déborde et retourne à zéro. Une nouvelle période commence et la sortie OCy est remise à '1'

La figure suivante illustre ce principe :



La période de la PWM vaut donc $(PRy + 1) \cdot T_{CY}$

La largeur de l'impulsion vaut $(OCxR + 1) \cdot T_{CY}$

Note : on préfère souvent parler de rapport cyclique, représentant la fraction du temps passée à l'état haut lors d'une période de l'onde carrée. Dans notre cas, ce rapport cylique vaut donc $D = \frac{OCxR+1}{PRy+1}$

11.2 Mode opératoire

La procédure à suivre pour configurer le module PWM est :

- Ecrire la durée de l'état haut dans OCxRS ;
- Configurer la patte correspondante à OCx en sortie (TRISDbits.TRISD… = 0), utiliser la Figure 3 pour repérer la patte correspondante ;
- Choisir si c'est le Timer 2 ou 3 qui servira de base de temps (OCxCONbits.OCTSEL) ;
- Configurer le Timer choisi (TyCON, PRy) , cf. section dédiée au Timers ;
- Configurer le périphérique en mode PWM (OCxCONbits.OCM = 0b110 dans notre cas) ;
- Initialiser éventuellement une interruption sur le Timer y ;
- Démarrer le Timer y via le bit TyCONbits.TON.

Le registre OCxRS peut ensuite être modifié à tout moment pour changer le rapport cyclique. On ne joue généralement pas sur la période du Timer après son activation.

11.3 Registre

U-0	U-0	R/W-0	U-0	U-0	U-0	U-0	R/W-0
—	—	OCSIDL	—	—	—	—	—
bit 15							bit 8

U-0	U-0	U-0	R-0, HC	R/W-0	R/W-0	R/W-0	R/W-0
—	—	—	OCFLT	OCTSEL		<OCM2:0>	
bit 7							bit 0

Legend:	HC = Cleared in Hardware	
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'
-n = Value at POR	'0' = Bit is cleared	

bit 15-14 **Unimplemented:** Read as '0'

bit 13 **OCSIDL:** Stop Output Compare x in Idle Mode Control bit
1 = Output compare x halts in CPU Idle mode
0 = Output compare x continues to operate in CPU Idle mode

bit 12-5 **Unimplemented:** Read as '0'

bit 4 **OCFLT:** PWM Fault Condition Status bit
1 = PWM Fault condition has occurred (cleared in hardware only)
0 = No PWM Fault condition has occurred (this bit is used only when OCM<2:0> = 111)

bit 3 **OCTSEL:** Output Compare x Timer Select bit
1 = Timer3 is the clock source for Output Compare x
0 = Timer2 is the clock source for Output Compare x

bit 2-0 **OCM<2:0>:** Output Compare x Mode Select bits
111 = PWM mode with fault protection. PWM mode on OCx, Fault pin is enabled
110 = PWM mode without fault protection. PWM mode on OCx, Fault pin is disabled
101 = Continuous Pulse mode. Initialize OCx pin low, generate continuous output pulses on OCx pin
100 = Delayed One-Shot mode. Initialize OCx pin low, generate single output pulse on OCx pin
011 = Toggle mode. Compare event toggles OCx pin
010 = Active High One-Shot mode. Initialize OCx pin high, compare event forces OCx pin low
001 = Active Low One-Shot mode. Initialize OCx pin low, compare event forces OCx pin high
000 = Module Disabled. Output Compare module is disabled

Note: The user software must disable the associated Output Compare module when writing to the output compare control registers to avoid malfunctions.

Figure 18 : Registre OCxCON (Output Compare x Control)

12 UART/RS-232 : transmission série

12.1 Introduction

Le RS-232 est un standard de communication série (les bits sont envoyés un à un) servant à établir une communication entre différents circuits selon un protocole défini.

La transmission série est asynchrone, ce signifie qu'il n'y a pas de synchronisation au niveau du bit (on ne transmet pas d'horloge pour synchroniser). Les mots (ensemble des bits envoyés lors d'une seule transmission) ont tous la même longueur, mais l'intervalle entre mots ne doit pas être régulier.

Le *baud-rate* (fréquence d'émission des bits) et le format de transmission (la composition du mot) doivent être connus à l'avance et les différents appareils doivent être configurés de manière identique. Ce format de transmission est illustré à la Figure 19. Il suit toujours le même schéma avec quelques variantes :

- le start bit, servant à signaler le début d'un envoi
- le mot à transmettre, composé de 5 à 9 bits selon les variantes
- un bit de parité permettant de détecter des erreurs de transmission ; les différentes variantes sont :
  - pas de bit de parité émis
  - parité paire ("even parity") : le bit est calculé pour que l'ensemble des bits de données et de la parité contienne un nombre pair de 1
  - parité impaire ("odd parity") : le bit est calculé pour que l'ensemble des bits de données et de la parité contienne un nombre impair de 1
  - bit de parité toujours à 1
  - bit de parité toujours à 0

  La parité permet la détection d'une erreur simple : si un seul bit a été altéré lors de la transmission, le récepteur, qui refait le même calcul de parité, s'en apercevra. Il ne sera pas capable de déterminer quel bit est erroné. Une erreur double est indétectable.
- le stop bit; les variantes portent sur la longueur du stop bit : la valeur la plus courante est 1 temps de bit, mais dans certains cas la longueur peut être de 1,5 ou 2 temps de bit.

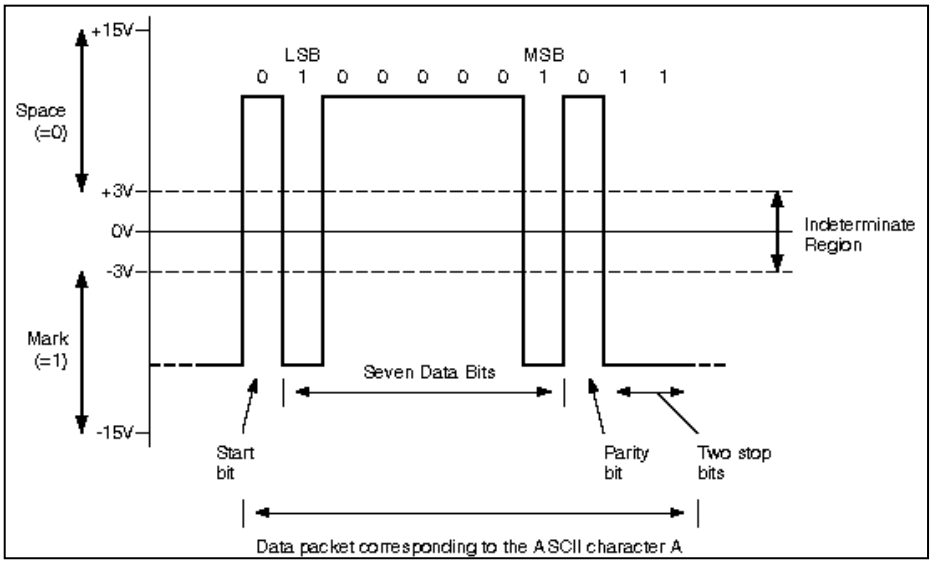


Figure 19 : Format de la transmission RS-232

Au niveau du contrôleur, la transmission et la réception des mots sont assurées par un périphérique nommé UART (*Universal Asynchronous Receiver and Transmitter).*

Les PC munis d'un port série sont également compatibles avec le protocole RS-232.

Note : ne pas confondre l'UART (périphérique gérant l'envoi et la réception des données) et RS232 (le protocole de communication).

Vu du programmeur, l'utilisation de la transmission série assistée par un UART consiste à :

- programmer le *baud rate* et le format de transmission une fois pour toutes lors de l'initialisation de l'UART

en transmission

- vérifier (par polling d'un bit de statut, ou par interruption issue de l'UART) que le registre à décalage de transmission est vide. L'UART du dsPIC possède une file d'attente de trois mots.
- écrire le mot à transmettre dans ce registre

en réception

- savoir (par polling d'un bit de statut, ou par interruption issue de l'UART) que le registre à décalage de réception est plein
- lire le mot reçu qui est présent dans ce registre
- consulter le statut de l'UART pour voir s'il n'y a pas eu d'erreur comme
  - format incorrect (« Framing Error »)
  - mauvaise parité
  - écrasement ou « overrun » : la donnée actuelle a été reçue avant que la précédente n'ait été lue ; au moins un mot a été perdu.

Dans ce projet, vous utiliserez une liaison à trois fils entre le système embarqué et le port série de l'ordinateur. Un schéma de câblage est donné à la Figure 20. La masse des deux parties est évidement commune et connectée à la patte 5 des connecteurs DB9. Les signaux de réceptions RX et de transmission TX (respectivement, pattes 2 et 3 sur le connecteur DB9) sont croisés de manière à ce que le Tx de l'un soit connecté au Rx de l'autre.

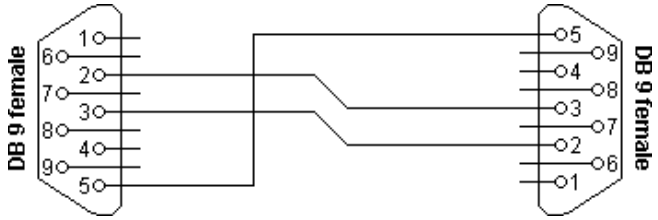


Figure 20 : Schéma de câblage d'une connexion RS-232 à trois fils.

12.2 Registres de contrôle

Cinq registres permettent de contrôler l'UART dont trois servent à configurer celui-ci :

- UxMODE : Registre configurant les modes de l'UART ;
- UxSTA : Registre de contrôles et de statuts ;
- UxBRG : Registre de 16 bits configurant le baud rate (voir sous-section 0)
- UxRXREG : Registre de réception
- UxTXREG : Registre de transmission (accessible seulement en écriture)

UxRXREG est le registre dans lequel le caractère reçu peut être lu, et UxTXREG est celui dans lequel le caractère à envoyer sur le port série doit être écrit.

Les buffers FIFO de transmission et de réception ont une profondeur de 4 caractères.

Les Figure 21 et Figure 22 montrent en détail les deux premiers registres.

12.3 Générateur de baud rate

L'UART du dsPIC inclut un générateur de baud rate (BRG). Celui peut fonctionner en basse ou haute vitesse selon la configuration du bit BRGH de UxMODE. Ce bit modifie la fréquence de l'horloge locale permettant le sur-échantillonnage des données reçues. Lorsqu'il vaut 0, le sur-échantillonnage est d'un facteur 16 par rapport au baud rate. Le facteur n'est plus que de 4 lorsque BRGH est initialisé à 1.

Ce bit a donc une influence sur la manière de calculer le contenu du registre UxBRG.

Pour BRGH égale à 0, le contenu du registre UxBRG est donné par :

$$UxBRG = \frac{F_{CY}}{16 * baud\ rate} - 1$$

Où F_{CY} est la fréquence du cycle-machine.

Lorsque BRGH vaut 1, le contenu du registre UxBRG est donné par :

$$UxBRG = \frac{F_{CY}}{4 * baud\ rate} - 1$$

12.4 Procédure d'initialisation de l'UART

La procédure à suivre pour initialiser l'UART est la suivante :

- Initialiser UxBRG pour le baud rate désiré (vérifier que l'arrondi ne crée pas une erreur de baud rate généré trop importante). Eventuellement, choisir le mode haute ou basse vitesse via le bit UxMODE.BRGH ;
- Configurer le format des données (UxMODEbits.PDSEL et UxMODEbits.STSEL) ;
- Choisir le mode d'interruption pour la réception (UxSTAbits.URXISEL , cf plus loin)
- Configuration éventuelle de l'interruption de réception(IEC0bits.U1RXIE pour UART1 , IEC1bits.U2RXIE pour UART2)
- Activer l'UART (UxMODEbits.UARTEN) ;
- Activer la transmission (UxSTA.UTXEN). (uniquement nécessaire en cas de transmission)

Les deux derniers points doivent être impérativement réalisés dans cet ordre.

12.5 Emission d'un caractère

Une fois la procédure d'initialisation terminée, il est possible à tout instant d'envoyer un caractère en suivant la procédure suivante

- Vérifier que la file d'attente d'envoi n'est pas pleine, via le bit UxSTAbits.UTXBF
- Ecrire la donnée à envoyer (sur 8 bits) dans UxTXREG

12.6 Réception de caractères

La donnée arrivant sur la patte UxRX du dsPIC est directement transférée dans le registre à décalage UxRSR. Ce dernier fait une conversion série-parallèle pour reconstituer le mot envoyé. La donnée est ensuite placée dans une file pouvant contenir jusque 4 données. Une lecture dans le registre UxRXREG permet alors de récupérer les données dans cette mémoire. Attention : cette donnée est supprimée de la file lorsqu'elle est lue.

Comme tout événement asynchrone, il est possible de vérifier l'intervention de la réception d'une donnée par interruption ou par polling du flag.

Le flag d'interruption est IFS0bits.U1RXIF pour l'UART1 , IFS1bits.U2RXIF pour UART2.

Lors de la phase d'initialisation, il est possible de choisir quel événement amènera à la modification du flag (et donc à l'envoi éventuel de l'interruption associée à la réception) en jouant sur le champ UxSTAbits.URXISEL : dès la réception d'un caractère, ou à différents niveaux de remplissage de la FIFO de réception UxRXREG

R/W-0	U-0	R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
UARTEN	—	USIDL	IREN(1)	RTSMD	—	UEN<1:0>	
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
WAKE	LPBACK	ABAUD	URXINV	BRGH	PDSEL<1:0>		STSEL
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15 **UARTEN:** UARTx Enable bit
1 = UARTx is enabled; UARTx pins are controlled by UARTx as defined by the UEN<1:0> and UTXEN control bits
0 = UARTx is disabled; UARTx pins are controlled by the corresponding PORT, LAT and TRIS bits

bit 14 **Reserved**

bit 13 **USIDL:** Stop in Idle Mode bit
1 = Discontinue operation when the device enters Idle mode
0 = Continue operation in Idle mode

bit 12 **IREN:** IrDA Encoder and Decoder Enable bit(1)
1 = IrDA encoder and decoder are enabled
0 = IrDA encoder and decoder are disabled

bit 11 **RTSMD:** Mode Selection for $\overline{\text{UxRTS}}$ Pin bit
1 = $\overline{\text{UxRTS}}$ is in Simplex mode
0 = $\overline{\text{UxRTS}}$ is in Flow Control mode

bit 10 **Reserved**

bit 9-8 **UEN<1:0>:** UARTx Enable bits
11 = UxTX, UxRX and BCLKx pins are enabled and used; $\overline{\text{UxCTS}}$ pin is controlled by port latches
10 = UxTX, UxRX, $\overline{\text{UxCTS}}$ and $\overline{\text{UxRTS}}$ pins are enabled and used
01 = UxTX, UxRX and $\overline{\text{UxRTS}}$ pins are enabled and used; $\overline{\text{UxCTS}}$ pin is controlled by port latches
00 = UxTX and UxRX pins are enabled and used; UxCTS, UxRTS and BCLKx pins are controlled by port latches

bit 7 **WAKE:** Enable Wake-up on Start bit Detect During Sleep Mode bit
1 = Wake-up is enabled
0 = Wake-up is disabled

bit 6 **LPBACK:** UARTx Loopback Mode Select bit
1 = Enable Loopback mode
0 = Loopback mode is disabled

bit 5 **ABAUD:** Auto-Baud Enable bit
1 = Enable baud rate measurement on the next character. Requires reception of a Sync field (55h); cleared in hardware upon completion.
0 = Baud rate measurement disabled or completed

bit 4 **URXINV:** Receive Polarity Inversion bit
1 = UxRX Idle state is '0'
0 = UxRX Idle state is '1'

bit 3 **BRGH:** High Baud Rate Select bit
1 = High speed
0 = Low speed

bit 2-1 **PDSEL<1:0>:** Parity and Data Selection bits
11 = 9-bit data, no parity
10 = 8-bit data, odd parity
01 = 8-bit data, even parity
00 = 8-bit data, no parity

bit 0 **STSEL:** Stop Selection bit
1 = 2 Stop bits
0 = 1 Stop bit

Note 1: This feature is only available for Low-Speed mode (BRGH = 0). See the specific device data sheet for details.

Figure 21 : Registre UxMODE (UART Mode)

R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	R-0	R-1
UTXISEL1	UTXINV	UTXISEL0	—	UTXBRK	UTXEN	UTXBF	TRMT
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R-1	R-0	R-0	R/C-0	R-0
URXISEL<1:0>		ADDEN	RIDLE	PERR	FERR	OERR	URXDA
bit 7							bit 0

Legend:	C = Clearable bit		
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15,13 **UTXISEL<1:0>** Transmission Interrupt Mode Selection bits
11 = Reserved
10 = Interrupt generated when a character is transferred to the Transmit Shift register and the transmit buffer becomes empty
01 = Interrupt generated when the last transmission is over (i.e., the last character has been shifted out of the Transmit Shift register) and all the transmit operations are completed
00 = Interrupt generated when any character is transferred to the Transmit Shift Register (which implies at least one location is empty in the transmit buffer)

bit 14 **UTXINV:** Transmit Polarity Inversion bit
1 = UxTX Idle state is '1'
0 = UxTX Idle state is '0'

bit 12 **Unimplemented:** Read as '0'

bit 11 **UTXBRK:** Transmit Break bit
1 = UxTX pin is driven low regardless of the transmitter state (Sync Break transmission – Start bit followed by twelve '0's and a Stop bit)
0 = Sync Break transmission is disabled or completed

bit 10 **UTXEN:** Transmit Enable bit
1 = UARTx transmitter enabled; UxTX pin is controlled by UARTx (if UARTEN = 1)
0 = UARTx transmitter disabled; any pending transmission is aborted and the buffer is reset; UxTX pin is controlled by PORT

bit 9 **UTXBF:** Transmit Buffer Full Status bit (read-only)
1 = Transmit buffer is full
0 = Transmit buffer is not full; at least one more data word can be written

bit 8 **TRMT:** Transmit Shift Register is Empty bit (read-only)
1 = Transmit Shift register is empty and the transmit buffer is empty (i.e., the last transmission has completed)
0 = Transmit Shift register is not empty; a transmission is in progress or queued in the transmit buffer

bit 7-6 **URXISEL<1:0>:** Receive Interrupt Mode Selection bits
11 = Interrupt flag bit is set when the receive buffer is full (i.e., has 4 data characters)
10 = Interrupt flag bit is set when the receive buffer is 3/4 full (i.e., has 3 data characters)
0x = Interrupt flag bit is set when a character is received

bit 5 **ADDEN:** Address Character Detect bit (bit 8 of received data = 1)
1 = Address Detect mode enabled. If 9-bit mode is not selected, this control bit has no effect.
0 = Address Detect mode disabled

bit 4 **RIDLE:** Receiver Idle bit (read-only)
1 = Receiver is Idle
0 = Data is being received

bit 3 **PERR:** Parity Error Status bit (read-only)
1 = Parity error has been detected for the current character
0 = Parity error has not been detected

bit 2 **FERR:** Framing Error Status bit (read-only)
1 = Framing error has been detected for the current character
0 = Framing error has not been detected

bit 1 **OERR:** Receive Buffer Overrun Error Status bit (clear/read-only)
1 = Receive buffer has overflowed
0 = Receive buffer has not overflowed. (Clearing a previously set OERR bit will reset the receive buffer and RSR to an empty state.)

bit 0 **URXDA:** Receive Buffer Data Available bit (read-only)
1 = Receive buffer has data; at least one more character can be read
0 = Receive buffer is empty

Figure 22 : Registre UxSTA (UART Status & Control)

13 Convertisseur analogique/numérique

13.1 Introduction

Le dsPIC33F dispose de 32 entrées analogiques multiplexées sur deux convertisseurs (ADC). Chaque ADC peut réaliser une conversion sur 10 ou 12 bits. Ce périphérique, aux modes de fonctionnement éventuellement très complexes, sera ici utilisé dans ses versions les plus simples.

Une conversion se déroule en deux étapes :

- l'échantillonnage : durant cette étape, la grandeur analogique est échantillonnée (on « prend une photo de la tension ») et est maintenue stable pour la numérisation. Cette étape est réalisée par un échantillonneur bloqueur (Sample and hold ou S/H). Celui-ci peut être vu comme un amplificateur connecté à une capacité via un interrupteur. Pendant l'échantillonnage, la capacité est connectée à l'amplificateur. A la fin de l'échantillonnage, l'amplificateur est déconnecté et la capacité garde la tension.
- la numérisation : c'est pendant cette étape que la grandeur analogique est convertie en une donnée numérique. Elle est lancée par la demande de début de numérisation

Note : par défaut, la phase d'échantillonnage ne démarre pas automatiquement et nécessite de jouer sur un bit particulier. Il est toutefois possible de rendre cette étape automatique en mettant le bit AD1CON1bits.ASAM à '1'

A chacune de ces étapes est associée une durée. La Figure 23 schématise le déroulement d'une conversion analogique/numérique.

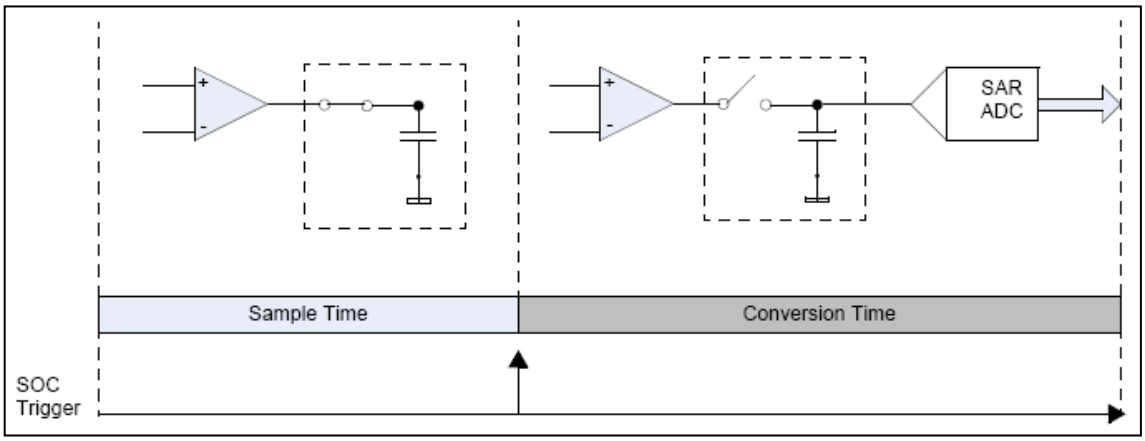


Figure 23 : Etapes d'une conversion analogique/numérique

Même dans son mode de fonctionnement le plus simple, il vous sera nécessaire de faire quelques choix :

- Le convertisseur peut numériser la tension sur 10bits ou 12bits, selon la valeur du bit AD1CON1bits.AD12B
- On peut choisir la borne sur laquelle le convertisseur va mesurer la tension (AN0, AN1 et AN3 sont accessibles sur la carte d'extension), via le registre AD1CHS0.CHS0A
- Il est possible de choisir l'événement qui va provoquer la demande de numérisation : cette demande peut être faite par software, à chaque débordement du timer 3,… Ceci se configure via le champ AD1CON1bits.SSRC

  Note 1 : dans le cas où le timer 3 lance la conversion, il n'est pas nécessaire de tester le flag d'interruption ou d'écrire la routine d'interruption de ce timer : à chaque débordement, il envoie automatiquement le signal de début de numérisation. Seule la fin de la conversion doit être gérée en software.

  Note 2 : si vous avez choisi de lancer la conversion par software, vous pouvez le faire à tout moment en écrivant un '0' dans le bit AD1CON1bits.SAMP

Une fois la conversion terminée, le résultat est écrit dans le registre ADC1BUF0. Dans le même temps, le flag d'interruption IFS0bits.AD1IF est mis à '1', ce qui peut éventuellement provoquer l'appel de la routine d'interruption du convertisseur si cette dernière a été activée au préalable via le bit IEC0bits.AD1IE.

La conversion est un processus relativement long pouvant prendre plusieurs dizaines de cycle machine. Lorsque l'on envoie une demande de conversion, on ne peut donc pas supposer que la donnée sera prête à la ligne de code qui suit : il est nécessaire de vérifier l'état du bit de fin de conversion

Comme vu dans la partie dédiée aux bornes d'entrée/sortie, les pattes d'entrée du convertisseur (nommées AN0 à AN31 sur la Figure 2) sont partagées avec des bornes d'i/o numériques. Il convient donc de vérifier que la borne voulue est bien mise en mode analogique :

- Repérer la borne voulue sur le schéma de la Figure 2. La borne aura généralement plusieurs « noms », par exemple AN20/INT1/RA12
- Configurer la patte en mode analogique (elle est en mode analogique par défaut) en écrivant un '0' dans le bit PCFGx du registre AD1PCFGL (de AN0 à AN15) ou AD1PCFGH (de AN16 à AN31). Pour l'exemple donné, ce sera le bit AD1PCFGH.PCFG20
- Configurer la borne en entrée grâce au registre TRIS correspondant. Pour l'exemple donné, ce sera TRISAbits.TRISA12

13.2 Horloge de l'ADC

Le module ADC utilise une horloge générée à partir soit d'un RC interne soit de l'horloge du cycles machine. Cette horloge (T_{AD}) permet de contrôler le temps de conversion. La numérisation nécessite 12 périodes (12 T_{AD}) en mode 10 bits et 14 périodes (14 T_{AD}) en mode 12 bits.

Lorsqu'elle est basée sur l'horloge du cycle machine (F_{CY}), l'horloge de l'ADC est configurable à l'aide des 8 bits ADCS de ADxCON3. La période T_{AD} est alors donnée par :

$$T_{AD} = T_{CY}(ADCS + 1)$$

Pour qu'une numérisation se déroule correctement, cette période doit être supérieure à 75ns.

13.3 Configuration de l'ADC

Les différentes étapes pour la configuration de l'ADC sont :

- Sélection du mode 10 ou 12 bits (AD1CON1bits.AD12B) ;
- Configuration de l'horloge du convertisseur (AD1CON3bits.ADCS) ;
- Repérer la borne voulue sur le schéma de la Figure 2. La borne aura généralement plusieurs « noms », par exemple AN20/INT1/RA12
- Configurer la patte en mode analogique (elle est en mode analogique par défaut) en écrivant un '0' dans le bit PCFGx du registre AD1PCFGL (de AN0 à AN15) ou AD1PCFGH (de AN16 à AN31). Pour l'exemple donné, ce sera le bit AD1PCFGH.PCFG20
- Configurer la borne en entrée grâce au registre TRIS correspondant. Pour l'exemple donné, ce sera TRISAbits.TRISA12
- Choix de la borne d'entrée du convertisseur (AD1CHS0bits.CHS0A)
- Passage en mode d'échantillonnage automatique (AD1CON1bits.ASAM=1)
- Choix du déclenchement de la conversion (manuel ou sur le Timer 3) via AD1CON1bits.SSRC
- Configuration éventuelle de l'interruption de l'ADC via le bit IEC0bits.AD1IE ;
- Si le Timer 3 a été choisi comme source du déclenchement, configurer sa période et le lancer. Il est possible d'utiliser le prescaler et/ou le mode 32bits de la paire Timer 2/3.
- Activer l'ADC (AD1CON1bits.ADON).

R/W-0	U-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0
ADON	—	ADSIDL	ADDMABM	—	AD12B	FORM<1:0>	
bit 15							bit 8

R/W-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0	R/W-0 HC,HS	R/C-0 HC, HS
SSRC<2:0>			—	SIMSAM	ASAM	SAMP	DONE
bit 7							bit 0

Legend:	HC = Cleared by hardware	HS = Set by hardware	
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15 **ADON:** ADC Operating Mode bit
1 = ADC module is operating
0 = ADC is off

bit 14 **Unimplemented:** Read as '0'

bit 13 **ADSIDL:** Stop in Idle Mode bit
1 = Discontinue module operation when device enters Idle mode
0 = Continue module operation in Idle mode

bit 12 **ADDMABM:** DMA Buffer Build Mode bit
1 = DMA buffers are written in the order of conversion. The module provides an address to the DMA channel that is the same as the address used for the non-DMA stand-alone buffer.
0 = DMA buffers are written in Scatter/Gather mode. The module provides a Scatter/Gather address to the DMA channel, based on the index of the analog input and the size of the DMA buffer.

bit 11 **Unimplemented:** Read as '0'

bit 10 **AD12B:** 10-bit or 12-bit Operation Mode bit
1 = 12-bit, 1-channel ADC operation
0 = 10-bit, 4-channel ADC operation

bit 9-8 **FORM<1:0>:** Data Output Format bits
For 10-bit operation:
11 = Signed fractional (DOUT = sddd dddd dd00 0000, where s = .NOT.d<9>)
10 = Fractional (DOUT = dddd dddd dd00 0000)
01 = Signed integer (DOUT = ssss sssd dddd dddd, where s = .NOT.d<9>)
00 = Integer (DOUT = 0000 00dd dddd dddd)
For 12-bit operation:
11 = Signed fractional (DOUT = sddd dddd dddd 0000, where s = .NOT.d<11>)
10 = Fractional (DOUT = dddd dddd dddd 0000)
01 = Signed Integer (DOUT = ssss sddd dddd dddd, where s = .NOT.d<11>)
00 = Integer (DOUT = 0000 dddd dddd dddd)

bit 7-5 **SSRC<2:0>:** Sample Clock Source Select bits
111 = Internal counter ends sampling and starts conversion (auto-convert)
110 = Reserved
101 = Reserved
100 = Reserved
011 = MPWM interval ends sampling and starts conversion
010 = GP timer (Timer3 for ADC1, Timer5 for ADC2) compare ends sampling and starts conversion
001 = Active transition on INTx pin ends sampling and starts conversion
000 = Clearing sample bit ends sampling and starts conversion

bit 4 **Unimplemented:** Read as '0'

bit 3 **SIMSAM:** Simultaneous Sample Select bit (only applicable when CHPS<1:0> = 01 or 1x)
When AD12B = 1, SIMSAM is: U-0, Unimplemented, Read as '0'
1 = Samples CH0, CH1, CH2, CH3 simultaneously (when CHPS<1:0> = 1x); or
Samples CH0 and CH1 simultaneously (when CHPS<1:0> = 01)
0 = Samples multiple channels individually in sequence

bit 2 **ASAM:** ADC Sample Auto-Start bit
1 = Sampling begins immediately after last conversion. SAMP bit is auto-set.
0 = Sampling begins when SAMP bit is set

bit 1 **SAMP:** ADC Sample Enable bit
1 = ADC Sample/Hold amplifiers are sampling
0 = ADC Sample/Hold amplifiers are holding
If ASAM = 0, software can write '1' to begin sampling. Automatically set by hardware if ASAM = 1.
If SSRC = 000, software can write '0' to end sampling and start conversion. If SSRC ≠ 000, automatically cleared by hardware to end sampling and start conversion.

bit 0 **DONE:** ADC Conversion Status bit
1 = ADC conversion cycle is completed.
0 = ADC conversion not started or in progress
Automatically set by hardware when A/D conversion is complete. Software can write '0' to clear DONE status (software not allowed to write '1'). Clearing this bit does NOT affect any operation in progress. Automatically cleared by hardware at start of a new conversion.

Note 1: The 'x' in ADxCON1 and ADCx refers to ADC 1 or ADC 2.

Figure 24 : Registre ADxCON1 (ADx Control 1)

R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADRC	—	—	SAMC<4:0>				
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
ADCS<7:0>							
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15 **ADRC:** ADC Conversion Clock Source bit
1 = ADC Internal RC Clock
0 = Clock Derived From System Clock

bit 14-13 **Unimplemented:** Read as '0'

bit 12-8 **SAMC<4:0>:** Auto Sample Time bits
11111 = 31 TAD
• • •
00001 = 1 TAD
00000 = 0 TAD

bit 7-0 **ADCS<7:0>:** ADC Conversion Clock Select bits
11111111 = TCY · (ADCS<7:0> + 1) = 256 · TCY = TAD
• • •
00000010 = TCY · (ADCS<7:0> + 1) = 3 · TCY = TAD
00000001 = TCY · (ADCS<7:0> + 1) = 2 · TCY = TAD
00000000 = TCY · (ADCS<7:0> + 1) = 1 · TCY = TAD

Figure 25 : Registre ADxCON3 (ADx Control 3)

R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
CH0NB	—	—	CH0SB<4:0>				
bit 15							bit 8

R/W-0	U-0	U-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
CH0NA	—	—	CH0SA<4:0>				
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15 **CH0NB:** Channel 0 Negative Input Select for Sample B bit
Same definition as bit 7.

bit 14-13 **Unimplemented:** Read as '0'

bit 12-8 **CH0SB<4:0>:** Channel 0 Positive Input Select for Sample B bits(1, 2)
Same definition as bit<4:0>.

bit 7 **CH0NA:** Channel 0 Negative Input Select for Sample A bit
1 = Channel 0 negative input is AN1
0 = Channel 0 negative input is VREFL

bit 6-5 **Unimplemented:** Read as '0'

bit 4-0 **CH0SA<4:0>:** Channel 0 Positive Input Select for Sample A bits(1, 2)
11111 = Channel 0 positive input is AN31
11110 = Channel 0 positive input is AN30
• • •
00010 = Channel 0 positive input is AN2
00001 = Channel 0 positive input is AN1
00000 = Channel 0 positive input is AN0

Note 1: The AN16 – AN31 pins are not available for ADC 2.
2: The 'x' in ADxCHS0 and ADCx refers to ADC 1 or ADC 2

Figure 26 : Registre ADxCHS0 (ADx Input Channel 0 Select)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PCFG31	PCFG30	PCFG29	PCFG28	PCFG27	PCFG26	PCFG25	PCFG24
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PCFG23	PCFG22	PCFG21	PCFG20	PCFG19	PCFG18	PCFG17	PCFG16
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15-0 **PCFG<31:16>:** ADC Port Configuration Control bits(1, 2)
1 = Port pin in Digital mode, port read input enabled, ADC input multiplexor connected to AVss
0 = Port pin in Analog mode, port read input disabled, ADC samples pin voltage

Note 1: On devices with less than 32 analog inputs, all PCFG bits are R/W by user. However, PCFG bits are ignored on ports without a corresponding input on device.
2: ADC2 only supports analog inputs AN0-AN15; therefore, no ADC2 Port Configuration register exists.

Figure 27 : Registre AD1PCFGH (ADC1 Port Configuration Register High)

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PCFG15	PCFG14	PCFG13	PCFG12	PCFG11	PCFG10	PCFG9	PCFG8
bit 15							bit 8

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0
PCFG7	PCFG6	PCFG5	PCFG4	PCFG3	PCFG2	PCFG1	PCFG0
bit 7							bit 0

Legend:			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	'1' = Bit is set	'0' = Bit is cleared	x = Bit is unknown

bit 15-0 **PCFG<15:0>:** ADC Port Configuration Control bits(1, 2, 3)
1 = Port pin in Digital mode, port read input enabled, ADC input multiplexor connected to AVss
0 = Port pin in Analog mode, port read input disabled, ADC samples pin voltage

Note 1: On devices with less than 16 analog inputs, all PCFG bits are R/W by user. However, PCFG bits are ignored on ports without a corresponding input on device.
2: On devices with two analog-to-digital modules, both AD1PCFGL and AD2PCFGL affect the configuration of port pins multiplexed with AN0-AN15.
3: The 'x' in ADxPCFGL and ADx refers to ADC 1 or ADC 2

Figure 28 : Registre ADxPCFGL (ADCx Port Configuration Register Low)