Microcontrôleurs

Valentin Gies

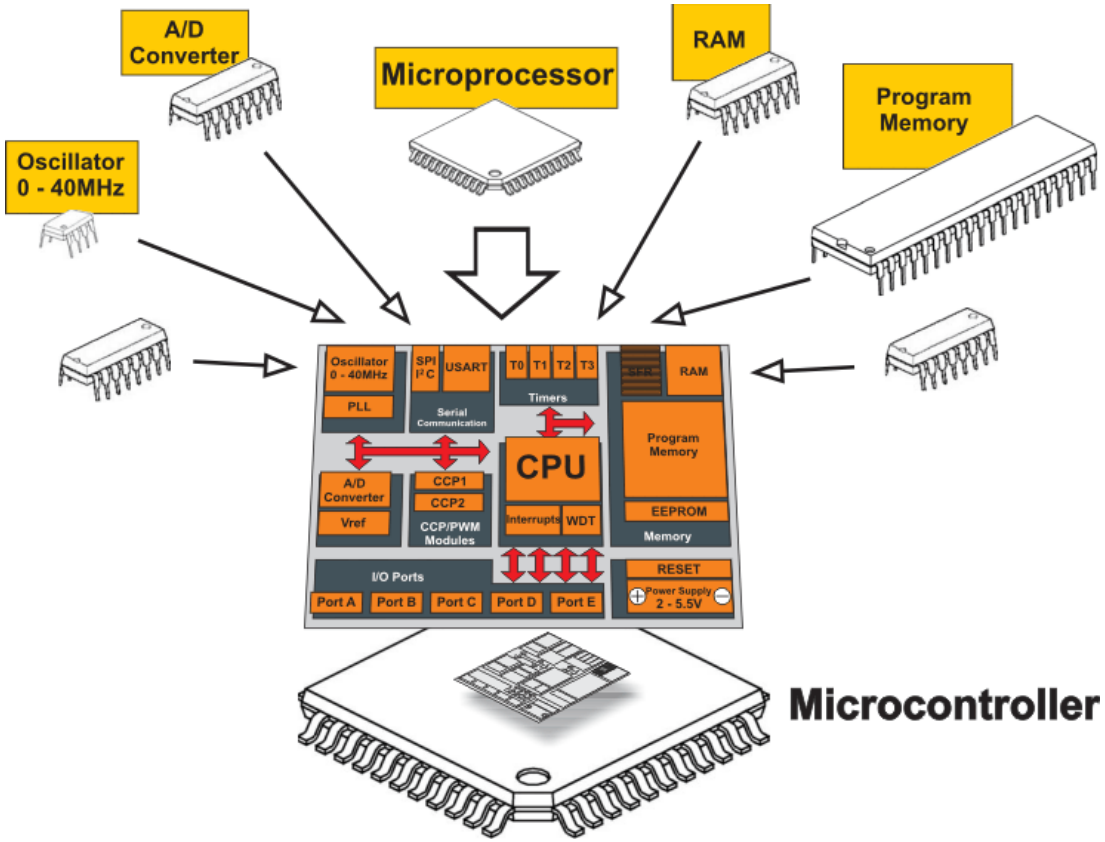
Plan du cours

1. Introduction aux microcontrôleurs
   - Contexte d'utilisation des processeurs
   - Structure des microcontrôleurs
2. La programmation de microcontrôleurs en C
   - Une programmation à ressources limitées
   - Structure d'un programme microcontrôleur
   - Les périphériques des microcontrôleurs

Plan

1. Introduction aux microcontrôleurs
   - Contexte d'utilisation des processeurs
   - Structure des microcontrôleurs
2. La programmation de microcontrôleurs en C
   - Une programmation à ressources limitées
   - Structure d'un programme microcontrôleur
   - Les périphériques des microcontrôleurs

Le microcontrôleur : un système intégré



- Unité de calcul + mémoires + périphériques **dans un seul composant**
- **Simplicité et rapidité** de développement

Le microcontrôleur : un système intégré

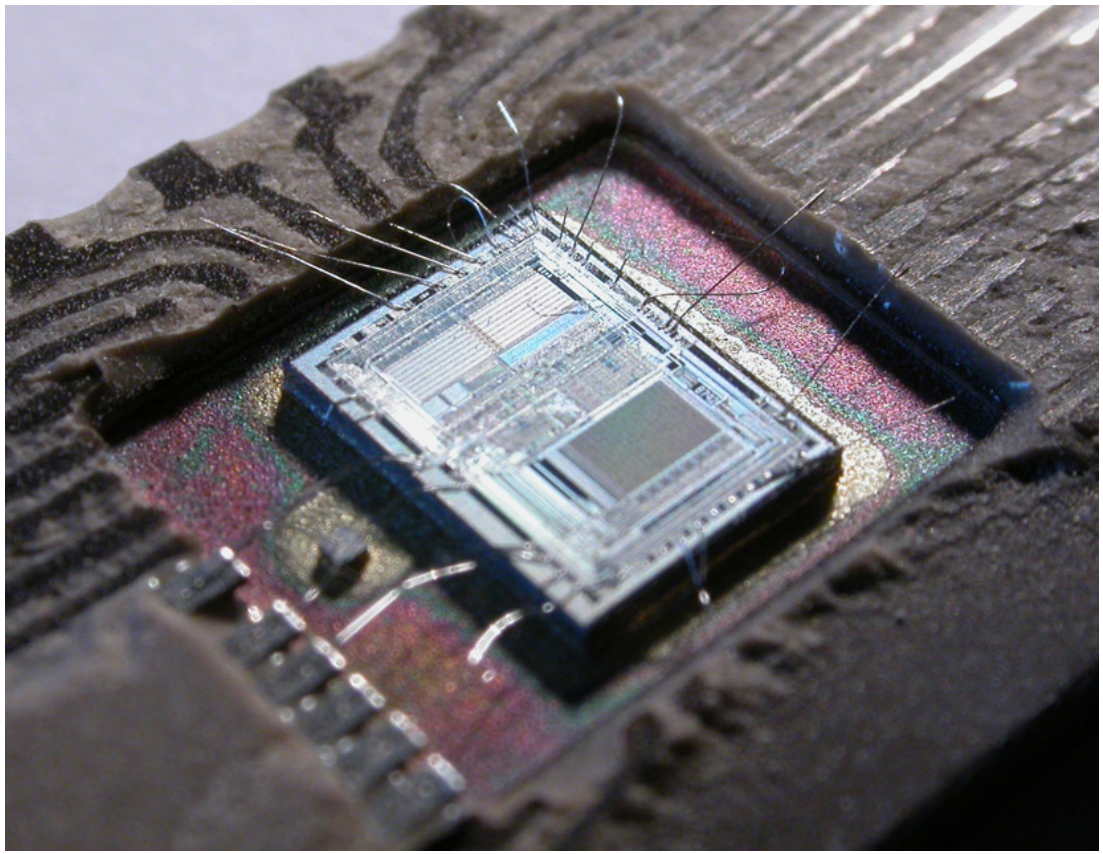


FIGURE: Silicium : Processeur Intel dans son boitier ouvert

Le microcontrôleur : un système intégré

Le microcontrôleurs intègre les fonctionnalités nécessaires au fonctionnement de systèmes complexes :

- Unité de calcul : **ALU**
- Base de temps : **Oscillateur**
- Temporisations et compteurs : **Timers**
- Stockage d'informations : **Mémoires programme et données**
- Interfaces externes :
  - **Ports d'entrées-sorties**
  - **Périphériques de communication** : UART, I2C, SPI...
  - **Convertisseurs analogique-numérique**

Architecture des microcontrôleurs

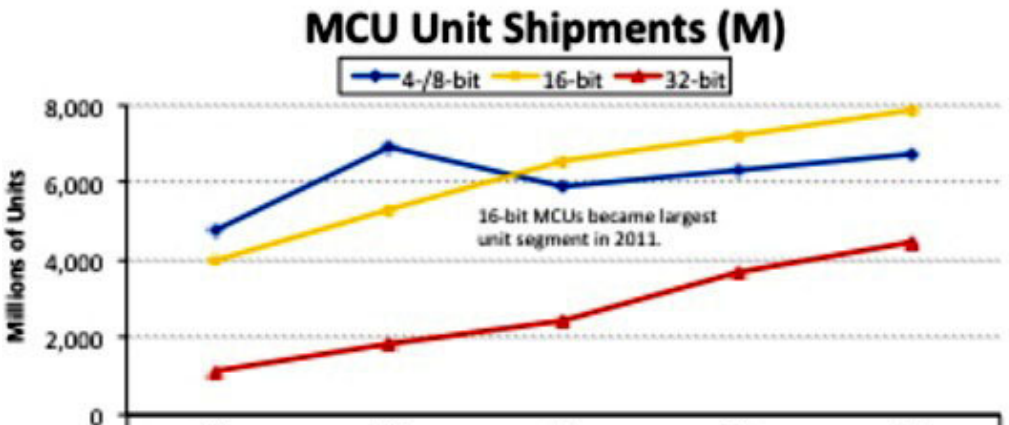
L'architecture d'un microcontrôleur est construite autour d'un bus commun :

- Bus : fils placés en parallèle.
- La taille du bus défini le type de microcontrôleur :
  - Bus 8 bits : permet de transmettre un *Byte* (octet)
    $\Rightarrow \mu C$ basiques
  - Bus 16 bits : permet de transmettre un *Word*
    $\Rightarrow \mu C$ performants
  - Bus 32 bits : $\Rightarrow \mu C$ avancés

Contexte d'utilisation des processeurs

Elements économiques :

Type de processeur	CA annuel	Nb unités vendues	Prix moyen
MCU 4-8 bits	4.4 Md$	6.7 Md	0.65$
MCU 16 bits	4.2 Md$	7.9 Md	0.53$
MCU 32 bits	6.9 Md$	4.45 Md	1.55$

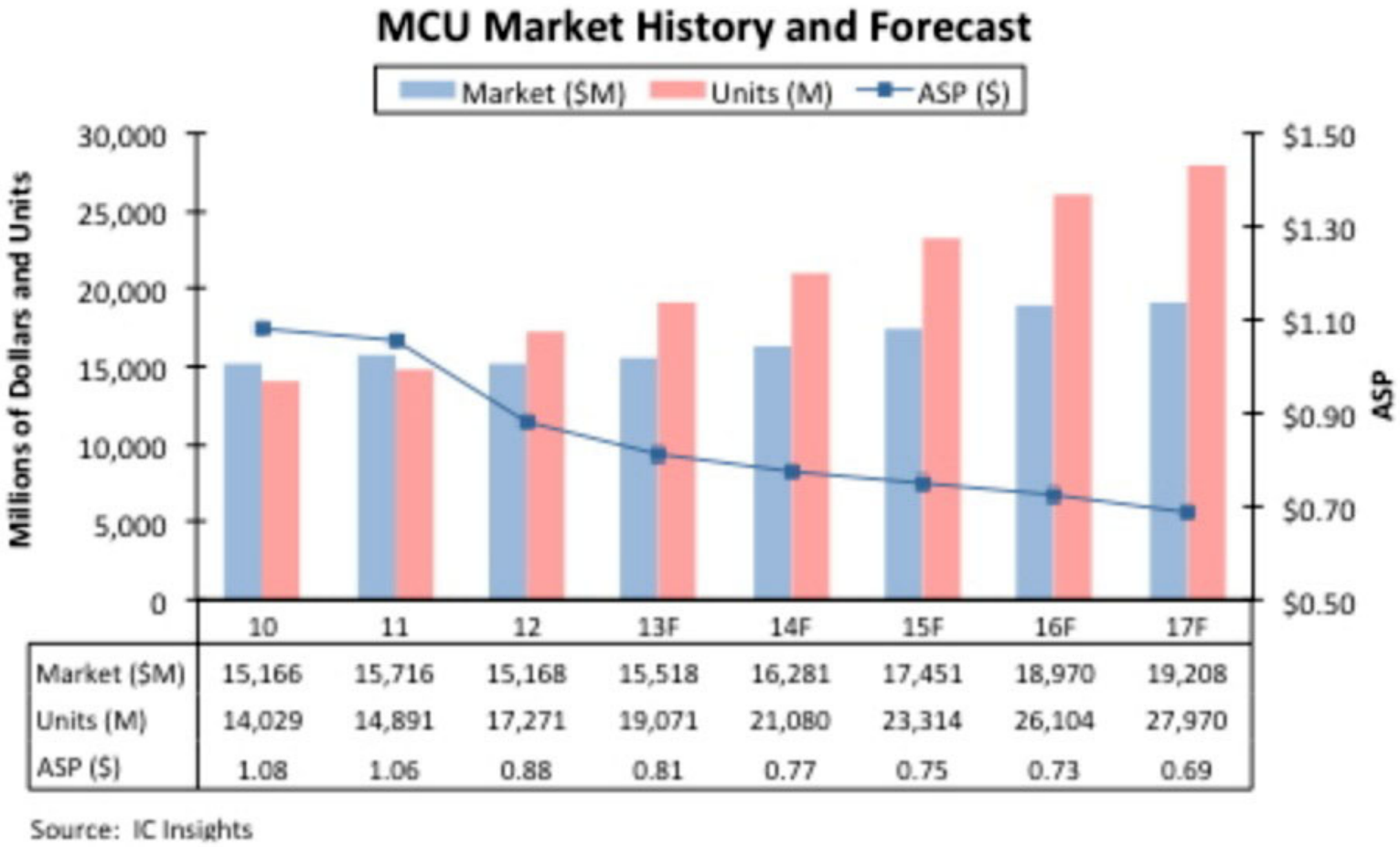


	09	10	11	12	13F
4-/8-bit	4,768	6,913	5,911	6,343	6,744
16-bit	3,982	5,298	6,528	7,227	7,870
32-bit	1,131	1,818	2,451	3,700	4,457

Source: IC Insights

Contexte d'utilisation des processeurs

Evolution des ventes annuelles moyennes de microcontrôleurs



	10	11	12	13F	14F	15F	16F	17F
Market ($M)	15,166	15,716	15,168	15,518	16,281	17,451	18,970	19,208
Units (M)	14,029	14,891	17,271	19,071	21,080	23,314	26,104	27,970
ASP ($)	1.08	1.06	0.88	0.81	0.77	0.75	0.73	0.69

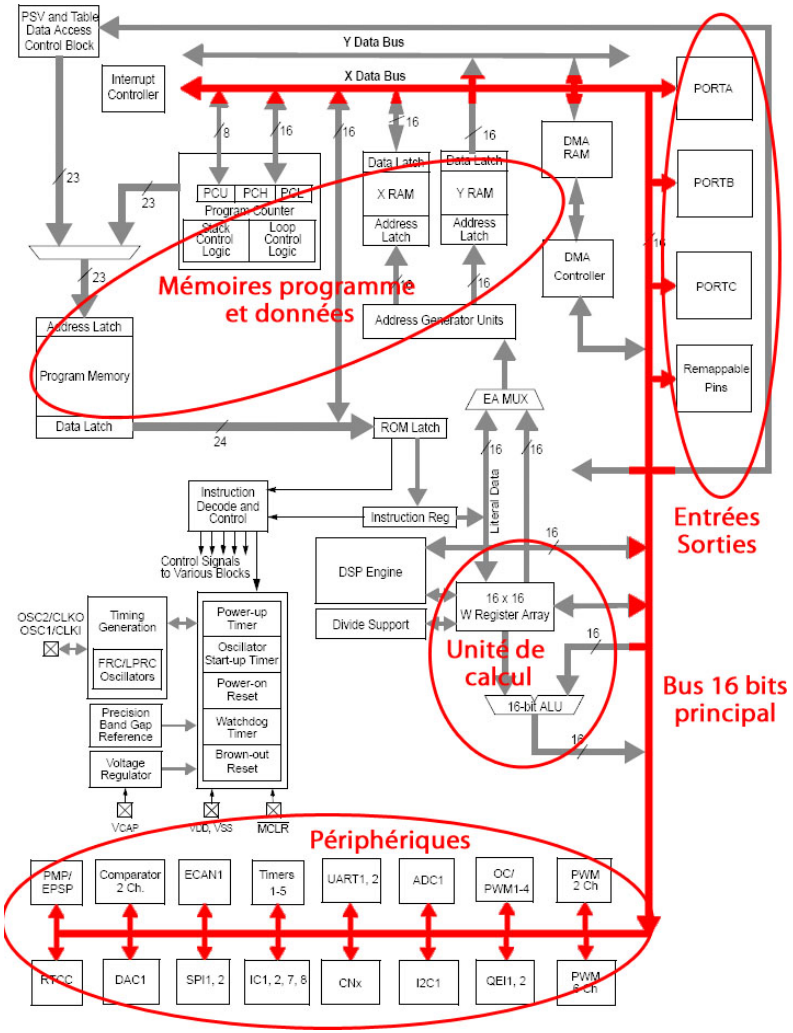
Source: IC Insights

Un exemple de microcontrôleur 16 bits

DSPIC33FJ128MC804

Microcontrôleur 16 bits doté de fonctionnalités DSP.

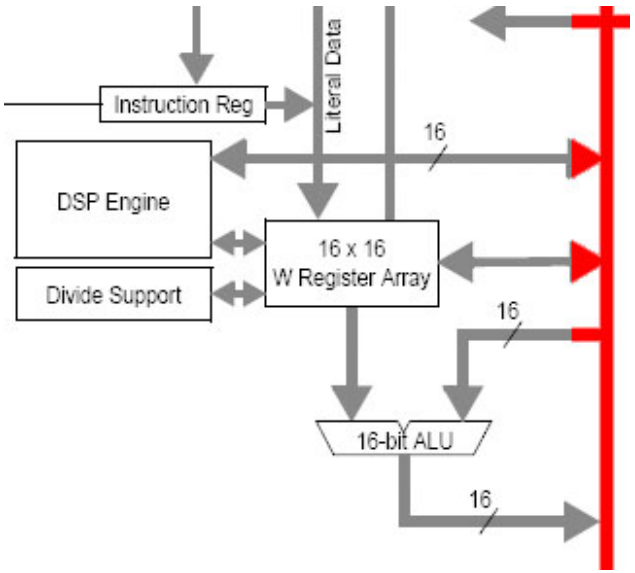
Le bus 16 bits est la colonne vertébral du microcontrôleur.



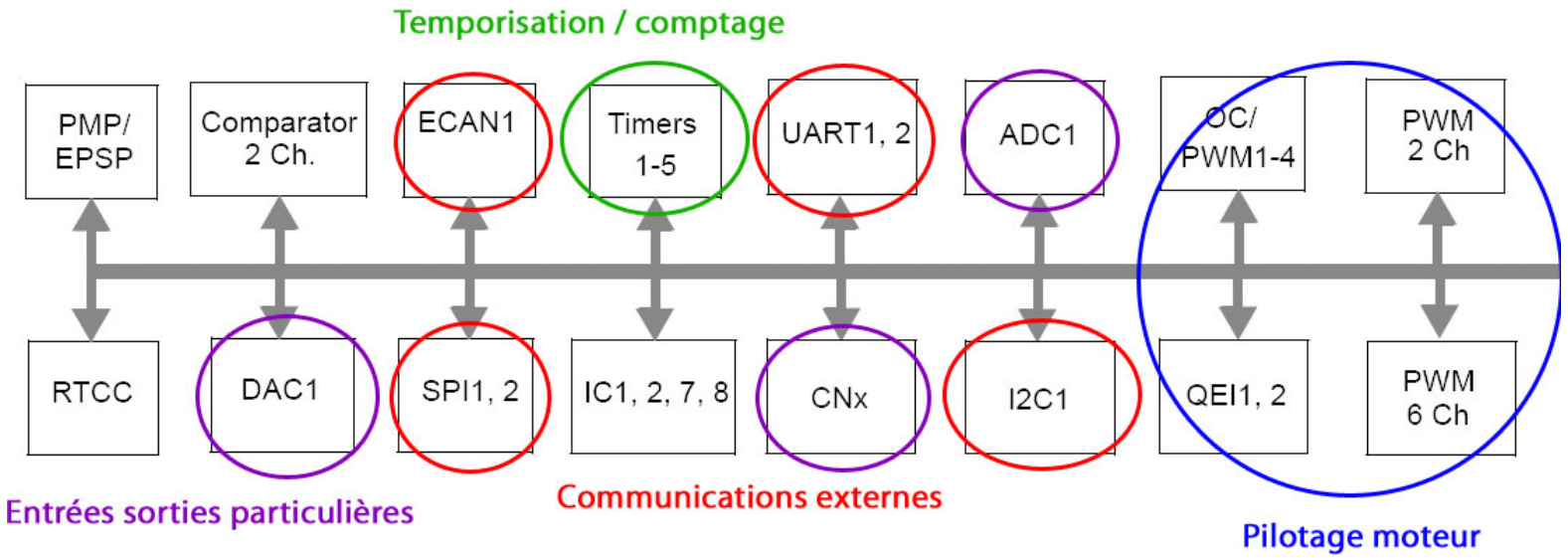
Un exemple de microcontrôleur 16 bits : cœur

Le cœur du processeur permet d'effectuer les opérations et calculs :

- Fréquence opératoire : 35 MIPS maximum
- Processeur RISC : 74 instructions (1 ou 2 cycles sauf division en 18 cycles)
- Présence d'un multiplieur/diviseur combinatoire
- Présence d'une unité DSP : 14 instructions (1 cycle)



Un exemple de microcontrôleur 16 bits : périphériques



- Les périphériques permettent d'exécuter de nombreuses tâches :
  - Temporisation / Comptage
  - Gestion de capteurs et pilotage distant : communications externes
  - Pilotage moteur : génération de PWM
  - Interfaçage analogique

Plan

1 Introduction aux microcontrôleurs
- Contexte d'utilisation des processeurs
- Structure des microcontrôleurs

2 La programmation de microcontrôleurs en C
- Une programmation à ressources limitées
- Structure d'un programme microcontrôleur
- Les périphériques des microcontrôleurs

Les spécificités en programmation sur microcontrôleur

- Pas d'OS (dans la plupart des cas).
- Contrôle total et bas niveau de tous les périphériques.
- Faible capacité mémoire $\Rightarrow$ allocation statique.
- Possibilités de debuggage limitées (1 à 5 points d'arrêt).

Les conséquences

- Programmation très proche de l'électronique.
- Pour simplifier la programmation : recours possible à des librairies de macro-fonctions.
- Utilisation d'outils de debuggage connectés à la carte.

Le C : un compromis entre abstraction et bas-niveau

Microcontrôleur = proche de l'électronique

Besoin d'un langage de bas-niveau

- Contrôle direct des composants électroniques : registres, mémoires, ADC, timers...
- Programmation optimale : ressources très limitées (CPU, mémoire...)

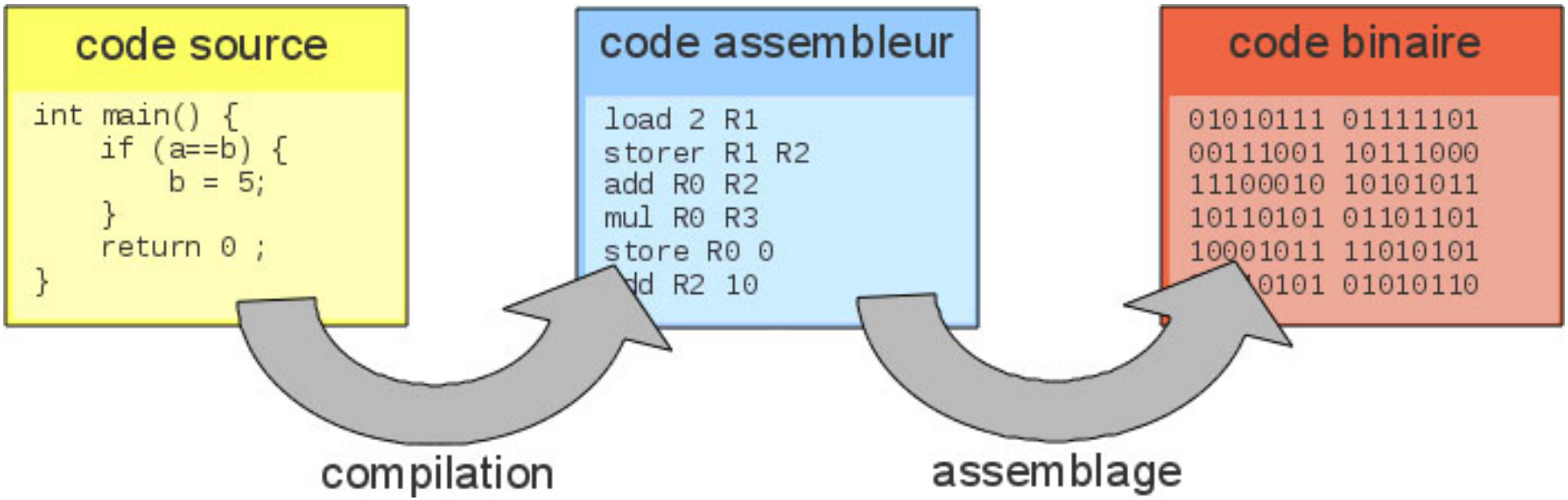
Besoin d'un certain niveau d'abstraction

- Pour implanter des concepts de moyen niveau (machines à état...).
- Pour simplifier la programmation : macro-fonctions.

Un langage répond à ces contraintes $\Rightarrow$ le C.

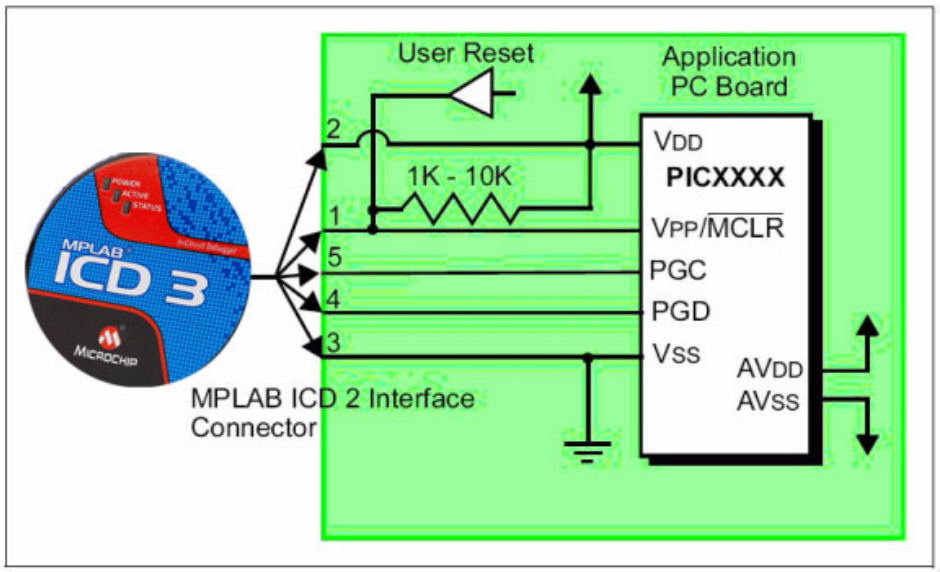
La chaine de programmation PIC24F

- **MPLAB X : environnement de développement**
  - Utilise Java NetBean : complétion automatique...
- **XC16 : compilateur pour DSPIC33**
  - XC16 (compilateur C) : génère du code assembleur à partir du code C



La chaine de programmation PIC24F (suite)

- **ICD 3 : programmateur in-situ**
  - Programmateur EEPROM de PIC sur carte (sans démontage du PIC).
  - Possibilités de debuggage (limitées) : 2 E/S en moins.



Structure d'un programme microcontrôleur

- **Initialisations**
  - Effectuées une seule fois au démarrage
- **Boucle infinie**
  - Démarre à la fin des initialisations.
  - Ne se termine jamais (sauf en cas de *Reset*).
- **Interruptions**
  - Evènements extérieurs ou internes interrompant momentanément la boucle.
  - niveaux de priorité différents : jusqu'à 8 niveaux utilisateur

Structure d'un programme : initialisations

Inclusions de bibliothèques :

- Standard : *#include* $< \ldots .h >$
- Spécifique au projet : *#include* " $\ldots .h$"

Inclusions de bibliothèques : exemple

```
#include <stdio.h>          //Standard Input/Output Header
#include <p33FJ128MC804.h>  //Propre au microcontrôleur choisi
#include <libpic30.h>       //Librairie pour famille de microcontrôleurs
#include "IO.h"             //Include custom
```

Registres de configuration : paramétrage du chip

Registres de configuration du chip : exemples

```
_FOSCSEL(FNOSC_PRI & IESO_OFF) // Configuration du Quartz sans PLL
_FOSC(FCKSM_CSECMD & OSCIOFNC_ON ) // Clock Switching Disables
_FWDT(FWDTEN_OFF) // Watchdog disabled
_FGS(GWRP_OFF & GSS_OFF & GCP_OFF) // Disable Code Protection
_FICD(JTAGEN_OFF) // JTAG disabled
```

Structure d'un programme : initialisations (suite)

Initialisations des entrées sorties et des périphériques :

- A l'intérieur de la fonction *main*
- Appelé une fois à la mise sous tension ou lors d'un *Reset*

Inclusions de librairies : exemple

```
int main (void){
    /*************************************************************/
    // Configuration des entrées sorties
    /*************************************************************/
    InitIO();

    /*************************************************************/
    // Configuration des périphériques
    /*************************************************************/
    InitTimer1();    //
    InitPWM();       //
    InitUART();       //
}
```

Structure d'un programme : la boucle infinie

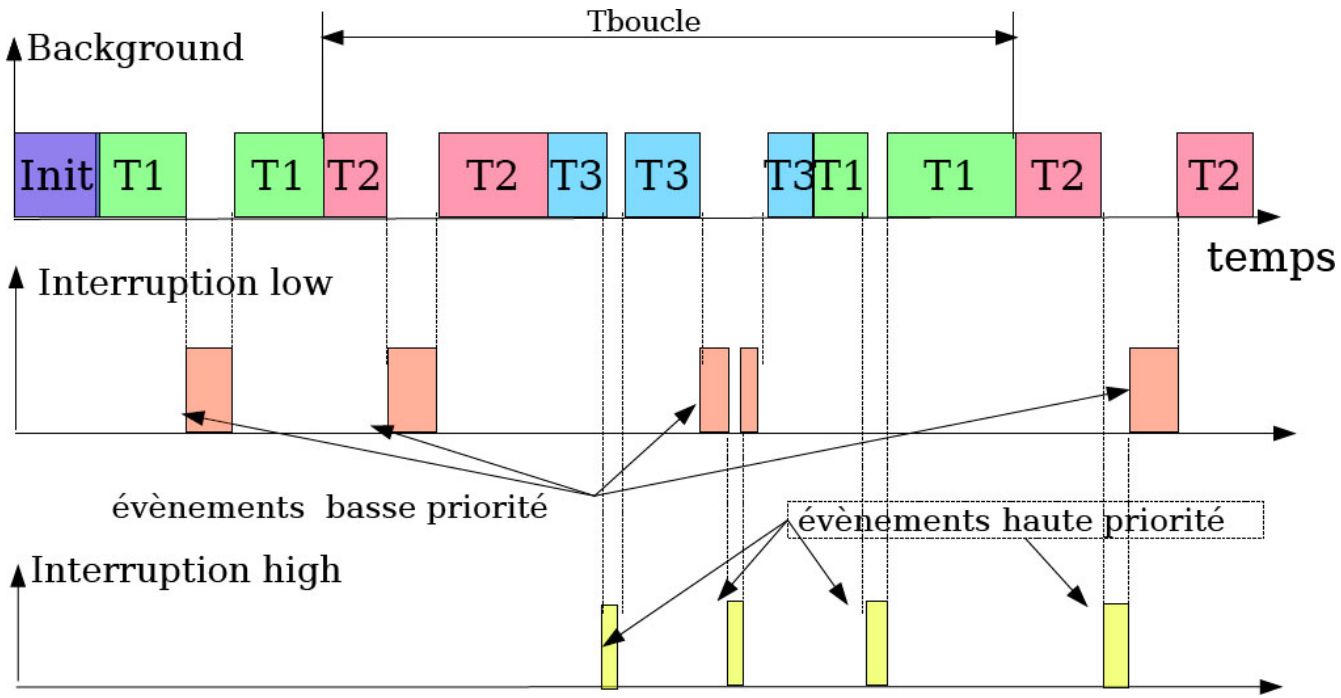
- **Priorité minimale** : dédiée aux tâches de fond
  - Ecritures en mémoire de masse
  - Gestion USB
- **Permet l'implantation de machines à état**
  - Utilisation de *switch-case*
  - OS temps réel : meilleure solution

Exemple de machine à états

```
int etape = 0;
while(1){
switch(etape){
    case 0 :
        ActionEtape0();  //actions étape 0
        etape = 1;
        break;
    case 1 :
        ActionEtape1();  //actions étape 1
        etape = 2;
        break;
    case 2 :
        ... //actions étape 2
        break;
    }
}
```

Les interruptions

- **Principe** :



- **Utilisation** :
  - Actions urgentes, courtes et asynchrones.

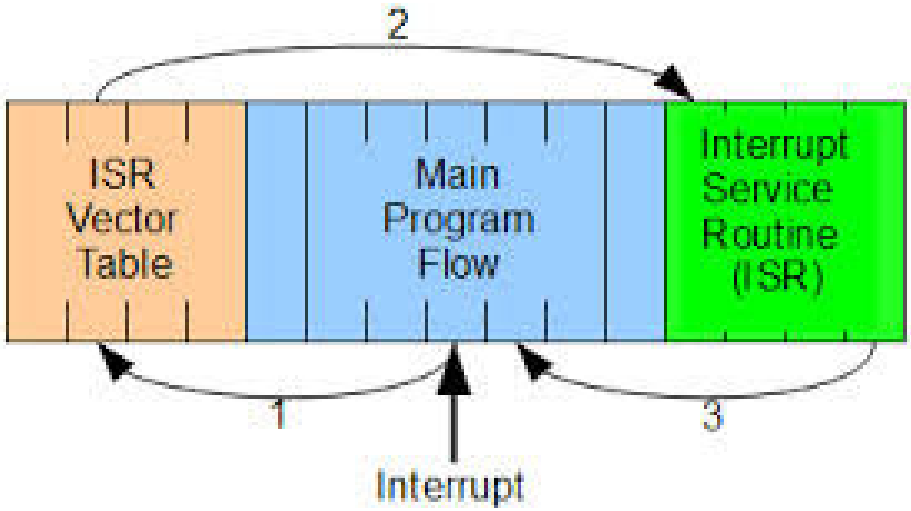
Les interruptions

Traitement des interruptions

- **Traitement immédiat**
  Action effectuée dans la routine d'interruption : utilisé pour les actions à temps de traitement court (arrêt d'urgence, mesure temporelle, temporisation...)
- **Traitement différé**
  Un drapeau est levé dans la routine d'interruption : l'action est effectuée ensuite dans la boucle principale qui consulte l'état du drapeau périodiquement : utilisé pour les traitements lourds.
- **Traitement mixte**
  Une partie immédiate et une partie différée.

Les interruptions

- **Principe** :



- **Performance** :
  - Les flags d'interruptions sont vérifiées après chaque cycle d'horloge.
  - L'accès à l'interruption se fait très rapidement : 5 cycles CPU chez Microchip (0.3μs à 16 MIPS).

Les interruptions

Interruptions : configuration des périphériques

Exemple de configuration des interruptions du Timer3

```
void InitTimer3Interrupt(void) {
    T3CONbits.TON = 0;      // Stop any 16-bit Timer3 operation
    ...
    IPC2bits.T3IP = 5;      // Set Timer3 Interrupt Priority Level
    IFS0bits.T3IF = 0;      // Clear Timer3 Interrupt Flag
    IEC0bits.T3IE = 1;      // Enable Timer3 interrupt
    T2CONbits.TON = 1;      // Start 32-bit Timer
}
```

- **Initialisation**
  - Autoriser la source à émettre des interruptions : $PxIE = 1$
  - Régler le niveau de l'interruption (entre 0 et 7) : $PxIP = level$
  - Acquitter le flag d'interruption pour éviter de rentrer directement dans la routine : $PxIF = 0$;

Les interruptions

Interruptions : routine d'interruption

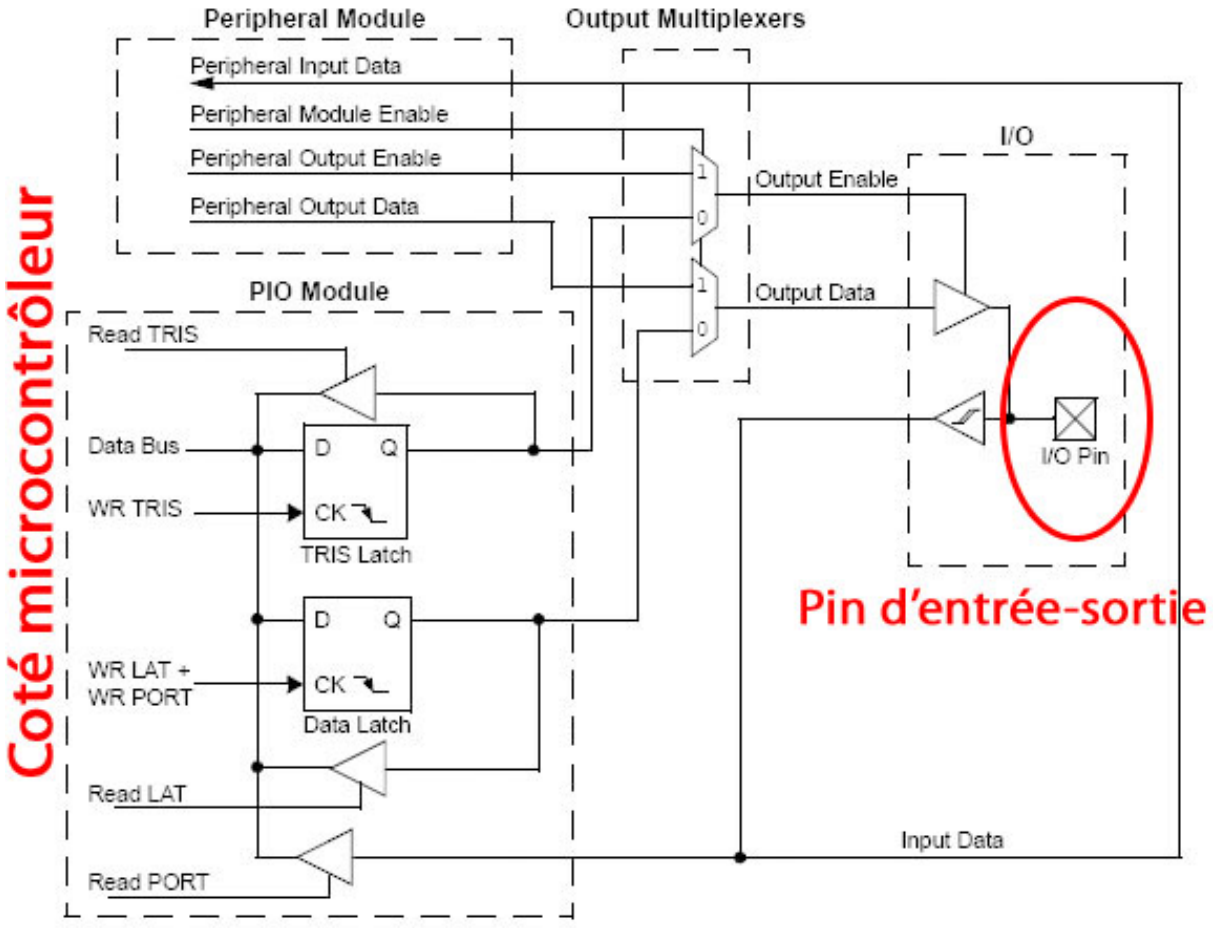
Exemple de configuration d'une PWM

```
/* Example code for Timer3 ISR */
void __attribute__((__interrupt__, no_auto_psv)) _T3Interrupt(void) {
    IFS0bits.T3IF = 0;                  // Clear Timer3 Interrupt Flag
    LED1 = !LED1;                       // inverse l'état de la led
}
```

- Les routines d'interruptions ont des prototypes particuliers
- Remarque : il faut absolument effacer la notification d'interruption pour en sortir ensuite

Les entrées-sorties logiques

Les entrées-sorties logiques



- I/O partagées avec les périphériques.
- Possibilité de désactiver des I/O

Les entrées-sorties logiques

Les entrées-sorties logiques

- **Registres utilisés par les ports E/S**
  - *PORTx* : bits représentants l'état réel des lignes d'E/S.
  - *LATx* : bits représentants l'état souhaité des sorties.
  - *TRISx* : bits de configurations :
    - 1 : entrée (valeur au reset)
    - 0 : sortie
- En principe, $LATx = PORTx$ sauf court-circuit ou sur-consommation.

Les entrées-sorties logiques

Utilisation en entrée logique

Configuration

```
TRISBbits.TRISB0 = 1; //inutile au reset
```

Lecture

```
if(PORTBbits.RB0 == 0){
..... ;} //Action si entrée = 0
else {
..... ;} //Action si entrée = 1
```

Les entrées-sorties logiques

Utilisation en sortie logique

Configuration

```
LATBbits.LATB3 = 0; //met RB3 à 0, on fixe la valeur avant de
                    //définir la broche en sortie
TRISBbits.RB3 = 0;  //configure RB3 en sortie
```

Lecture

```
LATBbits.RB3 = 1; //Met la sortie à 1
LATBbits.RB3 = 0; //Met la sortie à 0
```

Remarque : on **tente d'imposer à la sortie un état** logique, sa valeur réelle peut différer si la sortie est physiquement reliée à une source de tension différente

Les Timers

Les Timers : compteurs 16 ou 32 bits préchargeables et dont on peut spécifier l'horloge.

- **Principe** :
  - Le compteur s'incrémente à chaque période.
  - Lorsque le compteur sature, il émet un signal (interruption) et revient à 0.
  - Il est possible de précharger le compteur ou de lire sa valeur.
- **Horloge** :
  - Dérivée de l'horloge principale du PIC : période T_{osc} ou $K_p T_{osc}$ avec $K_p = 1, 2, 4, 8, 16, 32...$ appelé prescaler.
- **Utilisation** :
  - Temporisation.
  - Comptage d'événements.
  - Mesure du temps entre 2 événements.

Les Timers

Les Timers : exemple - clignotement d'une LED

Initialisation

```
//Configure le Timer 1 en mode 16 bits avec un prescaler de 1/8
OpenTimer1(T1_ON & T1_GATE_OFF & T1_PS_1_8 & T1_SYNC_EXT_OFF &
T1_SOURCE_INT, 0xFF);
```

Boucle infinie : clignotement de la LED outLed

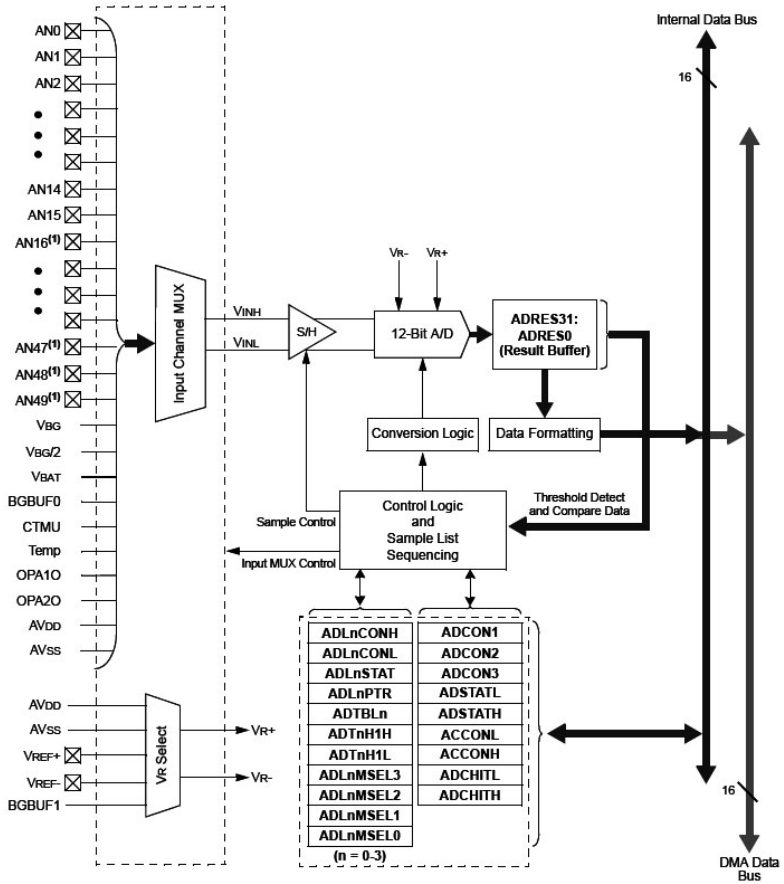
```
T_up = 5000;                    //On définit le temps à 1
T_down = 10000;                 //On définit le temps à 0
while(1){
    outLed = 1;                 //Sortie passe à 1
    WriteTimer0(0);             //Timer mis à 0;
    while(ReadTimer0()<T_up);   //Attente timer = T_up

    outLed = 0;                 //Sortie passe à 0
    WriteTimer0(0);             //Timer mis à 0;
    while(ReadTimer0()<T_down); //Attente timer = T_down
}
```

Les entrées analogiques

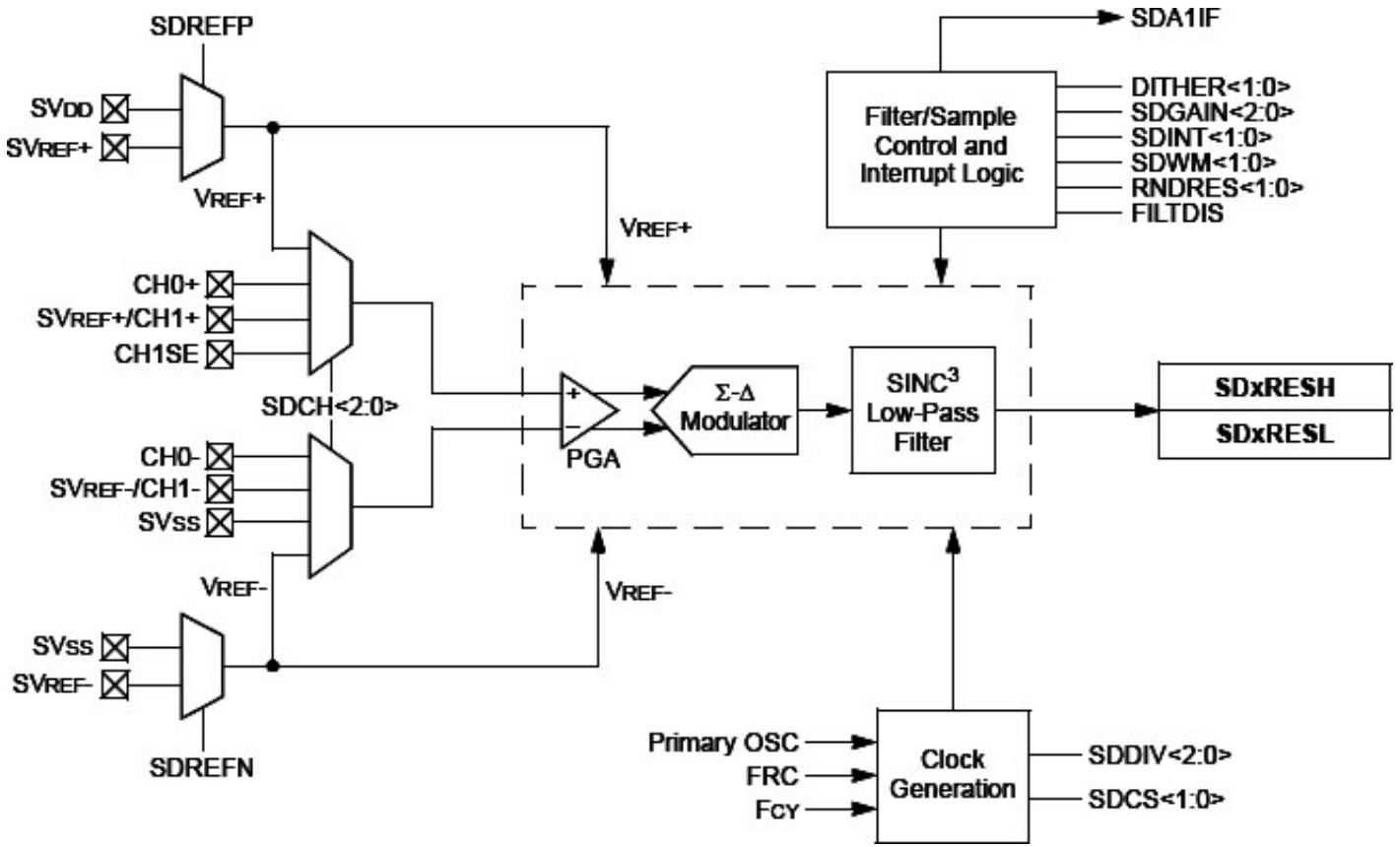
2 types de convertisseurs analogique-numérique (CAN)

- CAN 12 bits High Speed
  - Précision moyenne : 12 bits
  - 50 entrées possibles (séquentielles)
  - Acquisition simultanées possibles
  - **Fréquence de conversion très élevée : jusqu'à 10Msps**
- CAN 16 bits Sigma Delta
  - **Bonne précision : 16 bits** Sigma Delta
  - Entrées différentielles
  - Fréquence de conversion : 1ksps $\longleftrightarrow$ 62ksps

CAN 12 bits High-Speed



CAN 16 bits Sigma-Delta



CAN 12 bits : Initialisation

Configuration en mode 12 bits mono canal ADC sur AN4

```
void InitADC1(void)
{
    AD1CON1bits.ADON = 0;          // ADC module OFF - pendant la config
    AD1CON1bits.AD12B = 1;         // 0 : 10bits - 1 : 12bits
    AD1CON1bits.FORM = 0b00;       // 00 = Integer (DOUT = 0000 dddd dddd dddd)
    AD1CON1bits.ASAM = 0;          // 0 = Sampling begins when SAMP bit is set
    AD1CON1bits.SSRC = 0b111;      // 111 = auto-convert

    AD1CON2bits.VCFG = 0b000;      // 000 : Voltage Reference = AVDD AVss
    AD1CON2bits.CSCNA = 1;         // 1 : Enable Channel Scanning
    AD1CON2bits.SMPI = 0;          // 0+1 conversions successives

    AD1CON3bits.ADRC = 0;          // ADC Clock is derived from Systems Clock
    AD1CON3bits.SAMC = 0;          // Auto Sample Time = 0 * TAD
    AD1CON3bits.ADCS = 1;          // ADC Conversion Clock TAD = TCY * (ADCS + 1)

    //Configuration des ports
    AD1PCFGLbits.PCFG4 = 0; // AN4 as Analog Input
    AD1CSSLbits.CSS4=1; // Enable AN4 for scan

    IFS0bits.AD1IF = 0; // Clear the A/D interrupt flag bit
    IEC0bits.AD1IE = 1; // Enable A/D interrupt
    AD1CON1bits.ADON = 1; // Turn on the A/D converter
}
```

CAN 12 bits : conversion et récupération du résultat

Les conversions ADC peuvent :

- Fonctionner en boucle (pas forcément souhaitable)
- Etre lancées par exemple via un *timer* régulièrement.

Lancement d'une conversion

```
AD1CON1bits.SAMP = 1 ; //Lance une acquisition ADC
```

Le résultat peut être récupéré :

- En polling du flag de fin de conversion (à éviter)
- Sur interruption
- Dans la mémoire DMA (complexe mais efficace).

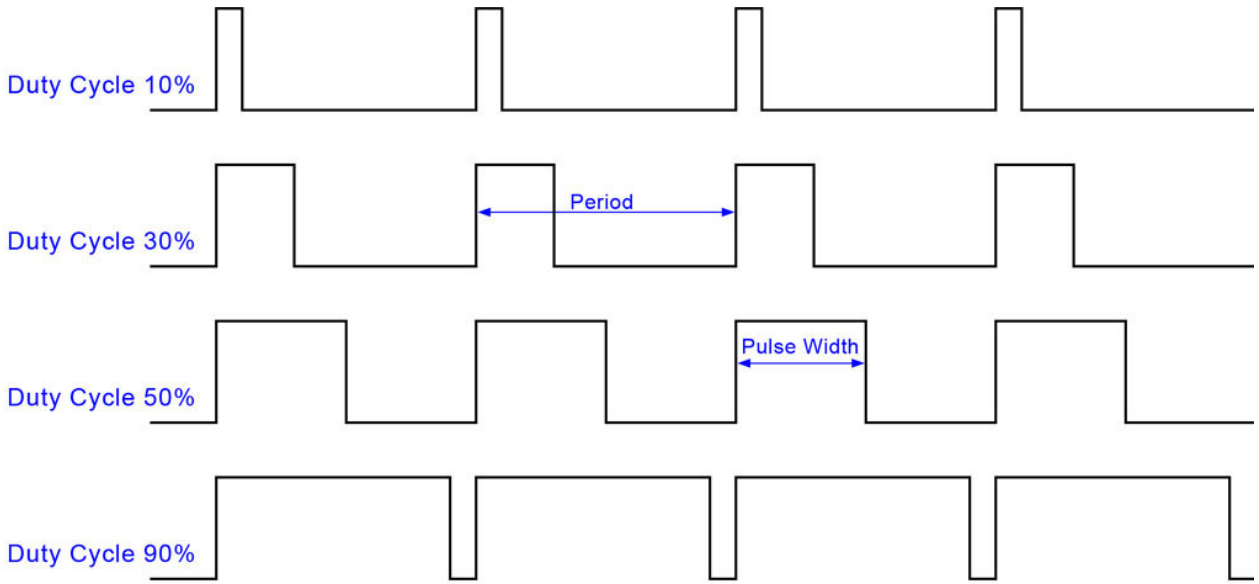
Lecture du résultat de la conversion ADC sur interruption

```
void __attribute__((interrupt, no_auto_psv)) _ADC1Interrupt (void){
IFS0bits.AD1IF = 0;
    unsigned int result = ADC1BUF0;
}
```

Les PWM : Pulse Width Modulation

Permet de contrôler les hacheurs et onduleurs.

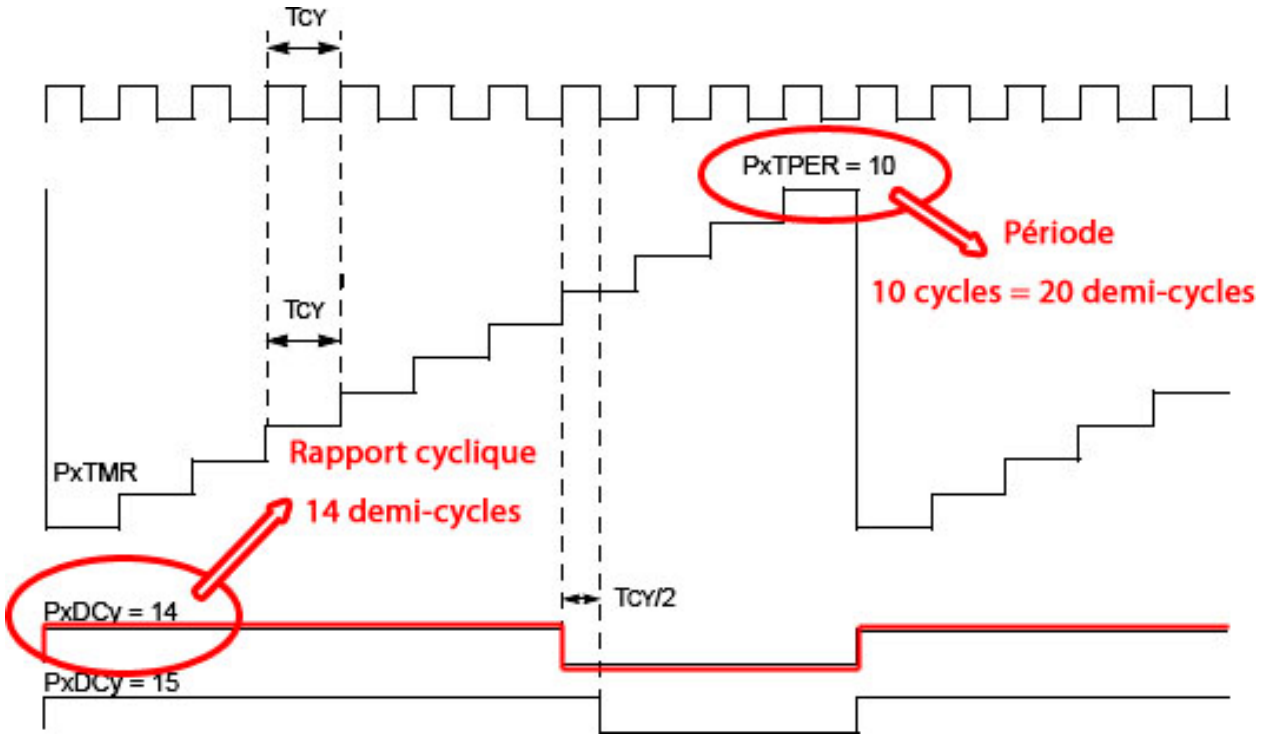
- Pulse Width Modulation = Modulation de Largeur d'Impulsion (MLI)
- Générateur de signal créneau à rapport cyclique variable



Les PWM : Pulse Width Modulation

Structure et fonctionnement du module :

- Réglage de la période : PxTER (en cycles)
- Réglage du rapport cyclique : PxDC1 (en demi-cycles)



Les PWM : Pulse Width Modulation

Configuration de la PWM 1

```
void InitPWM(void){
P1TCONbits.PTMOD = 0b00; //PWM time base operates in a Free Running mode
P1TCONbits.PTCKPS = 0b00; //PWM Prescaler – 0b00 : 1:1 – 0b01 1:4 ...
P1TCONbits.PTOPS = 0b00; //PWM Postcaler – 0b0000 : 1:1 – 0b0001 1:2 ...
P1TPER = 1000; //PWM period
P1DC1 = 500; //PWM duty cycle

PWM1CON1bits.PEN1H = 1; //PWM1H pin is enabled for PWM output
P1OVDCONbits.POVD1H = 1; //PWM I/O pin controlled by PWM Generator

PWM1CON2bits.IUE = 1; // Updates of Duty Cycle registers are immediate
P1TCONbits.PTEN = 1; //PWM time base is on
}
```

- La fréquence de la PWM est égale à

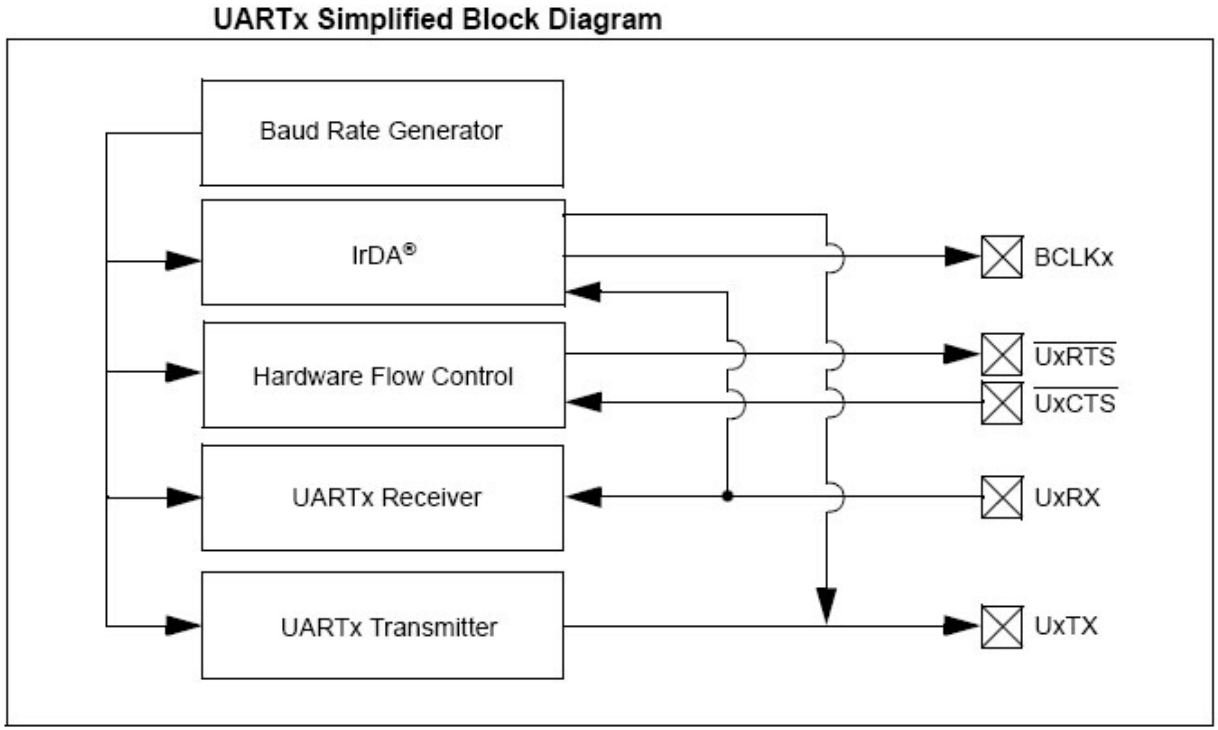
$$F_{PWM} = \frac{F_{CY}}{Prescaler * Postscaler * P1PTER}$$

- Le rapport cyclique de la PWM est égal à

$$DC_{PWM} = \frac{P1DC1}{P1TER * 2}$$

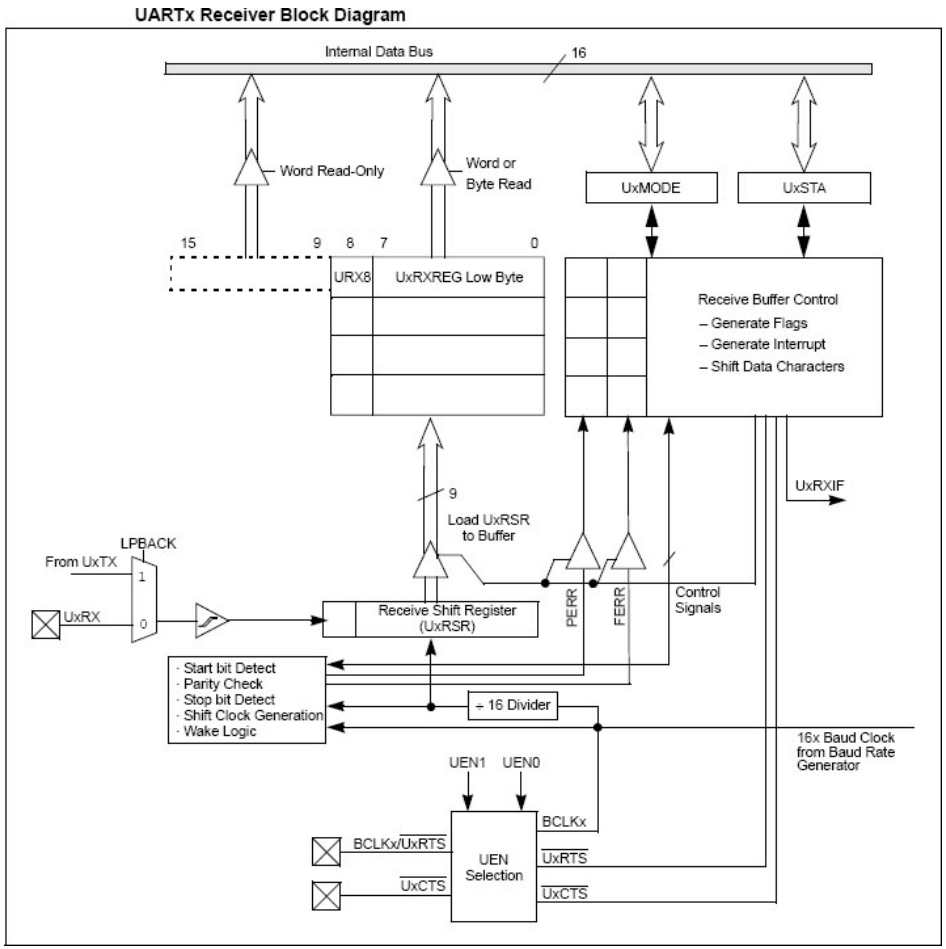
Le bus série UART

- Communication avec un **unique circuit**
- Communications full duplex et asynchrones
- **Débit limité** : $> 1Mb/s$ avec contrôle de flux
- **Débit limité** : $57.6Kb/s$ sans contrôle de flux



Le bus série UART

Schéma de détaillé du récepteur (Rx) :

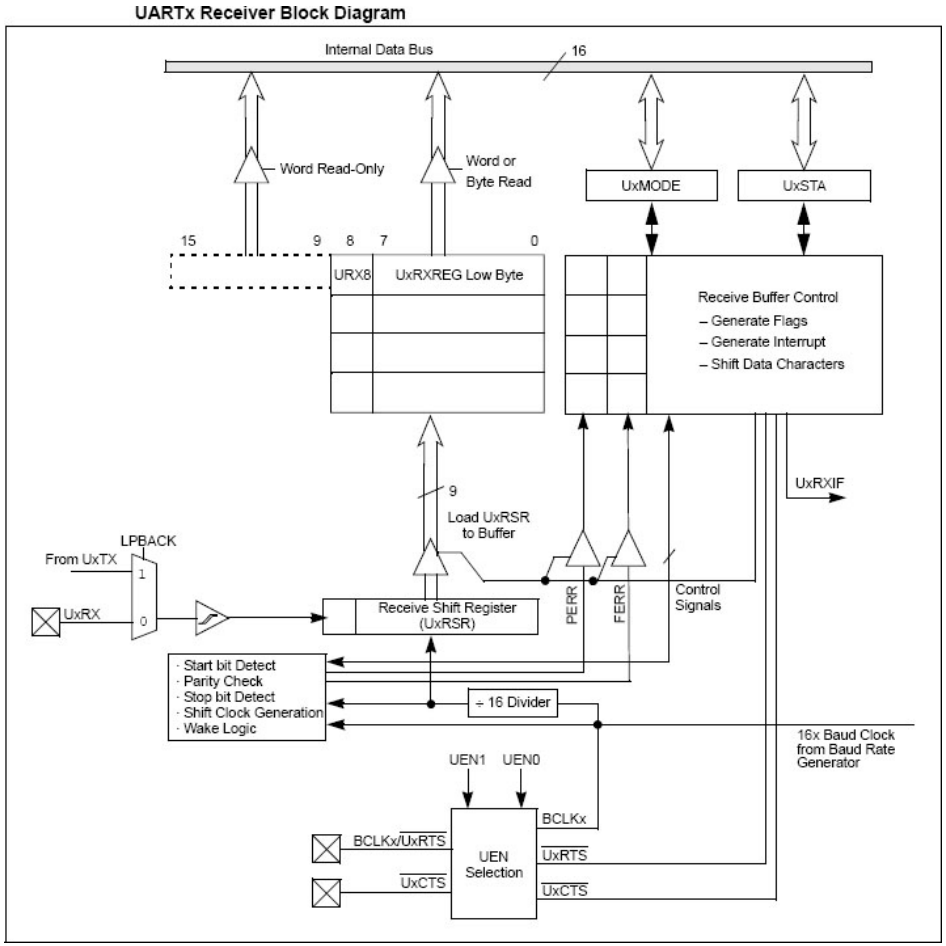


Le bus série UART : Initialisation

Initialisation en mode interruption sur Rx et Tx

```
void InitUART(void) {
    U1MODEbits.STSEL = 0; // 1-stop bit
    U1MODEbits.PDSEL = 0; // No Parity, 8-data bits
    U1MODEbits.ABAUD = 0; // Auto-Baud Disabled
    U1MODEbits.BRGH = 1; // Low Speed mode
    U1BRG = BRGVAL; // BAUD Rate Setting

    U1STAbits.UTXISEL0 = 0; // Interrupt after one Tx character is transmitted
    U1STAbits.UTXISEL1 = 0;
    IFS0bits.U1TXIF = 0; // clear TX interrupt flag
    IEC0bits.U1TXIE = 1; // Enable UART Tx interrupt

    U1STAbits.URXISEL = 0; // Interrupt after one RX character is received;
    IFS0bits.U1RXIF = 0; // clear RX interrupt flag
    IEC0bits.U1RXIE = 1; // Enable UART Rx interrupt

    U1MODEbits.UARTEN = 1; // Enable UART
    U1STAbits.UTXEN = 1; // Enable UART Tx

    /* wait at least 104 usec (1/9600) before sending first char */
    int i=0;
    for (i = 0; i < 4160; i++) {
        Nop();
    }
}
```

Le bus série UART : Schéma du récepteur (Rx)



Le bus série UART : Réception

- Attention à vérifier les éventuelles erreurs à la réception
- L'interruption est générée à chaque fois qu'une data est disponible dans le buffer (voir initialisation)

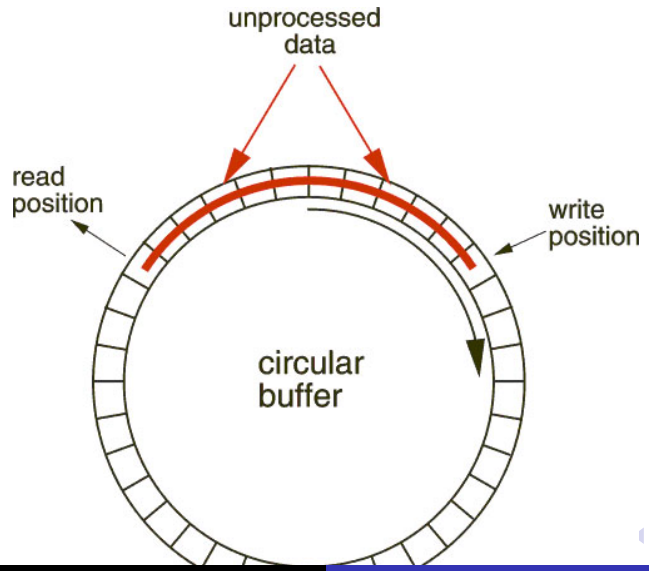
Réception sur interruption

```
void __attribute__((__interrupt__,__auto_psv__)) _U1RXInterrupt(void) {
    IFS0bits.U1RXIF = 0; // clear RX interrupt flag
    /* check for framing errors */
    if (U1STAbits.FERR == 1)
        U1STAbits.FERR = 0;
    /* must clear the overrun error to keep uart receiving */
    if (U1STAbits.OERR == 1)
        U1STAbits.OERR = 0;
    /* get and process data */
    if (U1STAbits.URXDA == 1)
        Process(U1RXREG);
}
```

- Attention : il est préférable de stocker les données dans un buffer circulaire traité avec une priorité faible.

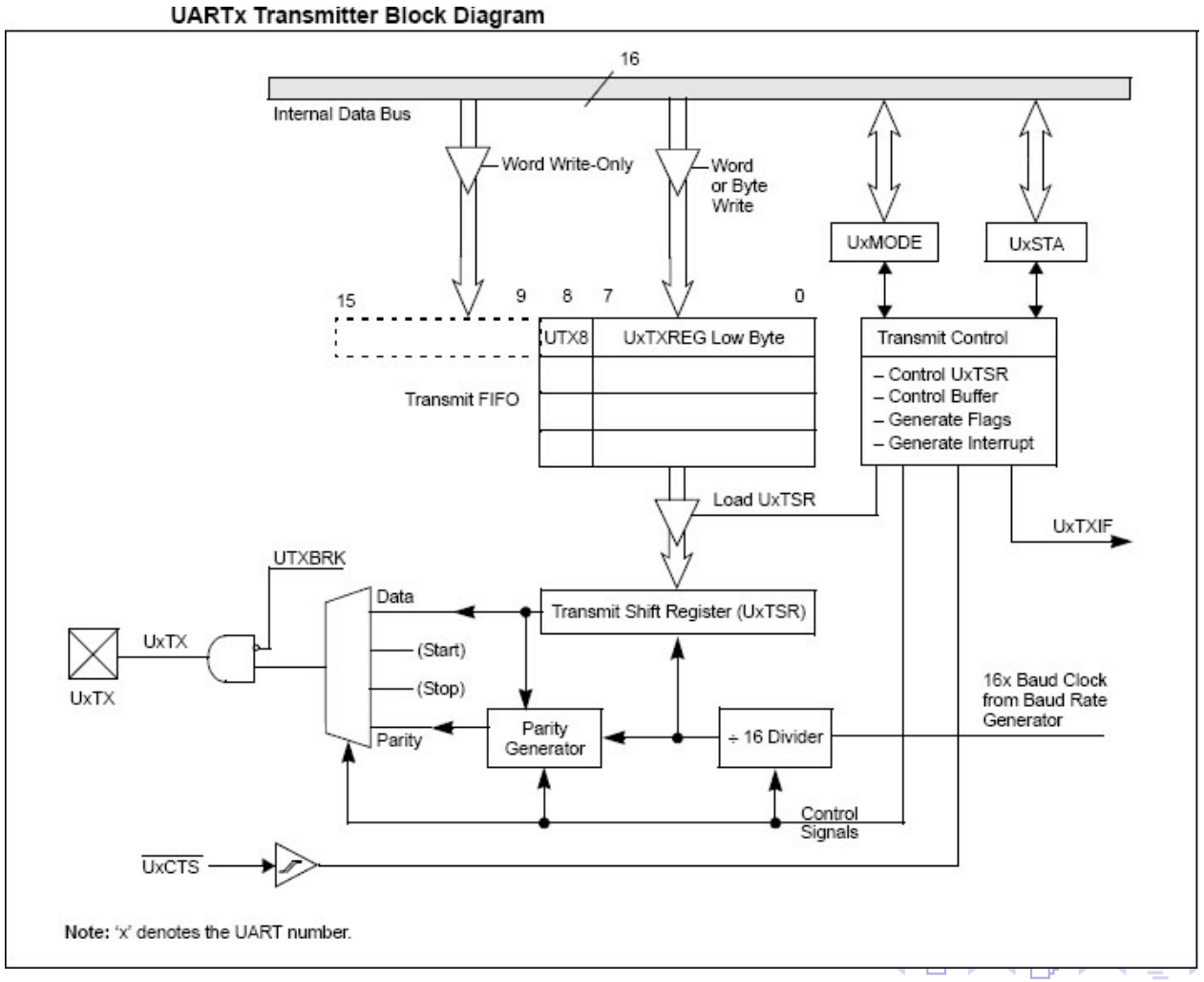
Le bus série UART : Buffer circulaire

- Permet de stocker en mémoire des données en attente de traitement. Celui-ci se fait dans la boucle principale ou dans l'OS temps réel
- Utilité : si des opérations de priorité supérieure à l'UART durent longtemps.
- Inconvénient : utilise de la place en mémoire (typiquement 128 octets)



Le bus série UART : Schéma du transmetteur (Tx)

Schéma de détaillé du transmetteur (Tx) :



Le bus série UART : Emission Tx

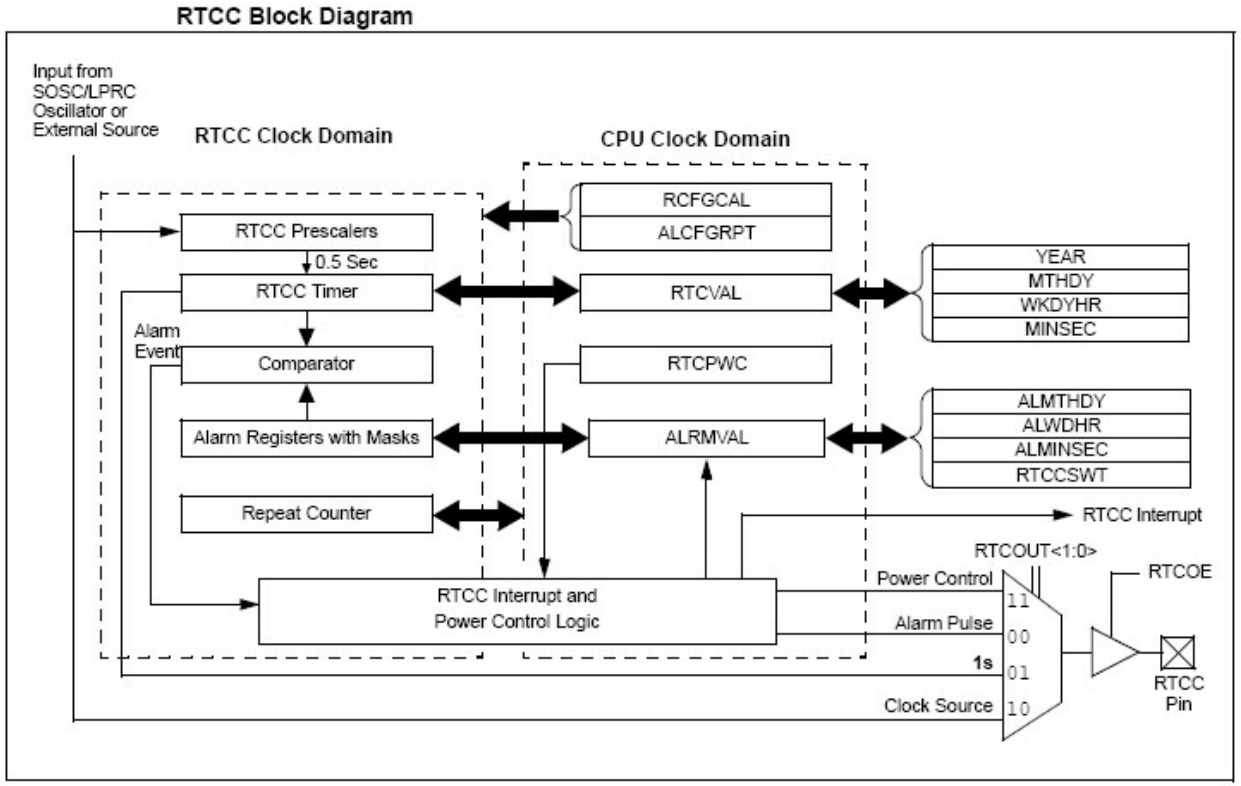
- L'envoi se fait sur interruption (non bloquant pour le reste)
- Un buffer circulaire peut être utilisé $\Rightarrow$ point d'envoi unique

Envoi sur interruption

```
void SendMessage(unsigned char* message, int length){
    if(!CB_TX1_IsTranmitting())
        SendOne();              // On initie la transmission avec le premier octet
}

void SendOne(){
    isTransmitting = 1;         // On lève un flag : transmission en cours
    unsigned char value=CB_TX1_Get();    // On récupère un caractère dans le buffer
    U1TXREG = value;            // Transmet un caractère
}

void __attribute__((__interrupt__,__auto_psv__)) _U1TXInterrupt(void){
    IFS0bits.U1TXIF = 0; // clear TX interrupt flag
    if (cbTx1Tail!=cbTx1Head) //Si il y a des octets restant dans le buffer
          circulaire
        SendOne();              //On relance une transmission
    else
        isTransmitting = 0;     //Sinon on arrête
}
```

Périphériques annexes : Horloge temps réel (RTCC)

- **Réglage et mise à jour de la date et de l'heure**
- **Nécessite un quartz à 32.768 kHz**
- Peut servir de timer à très longue période.



Questions ?

- Questions
- Contact : contact@vgies.com
- Site internet : **www.vgies.com**