Réseaux et informatique embarquée

5. Bus terrains dans les microcontrôleurs

Valentin Gies

Seatech - 4A
Université de Toulon (UTLN)

Plan du cours

Le bus terrains dans les microcontrôleurs

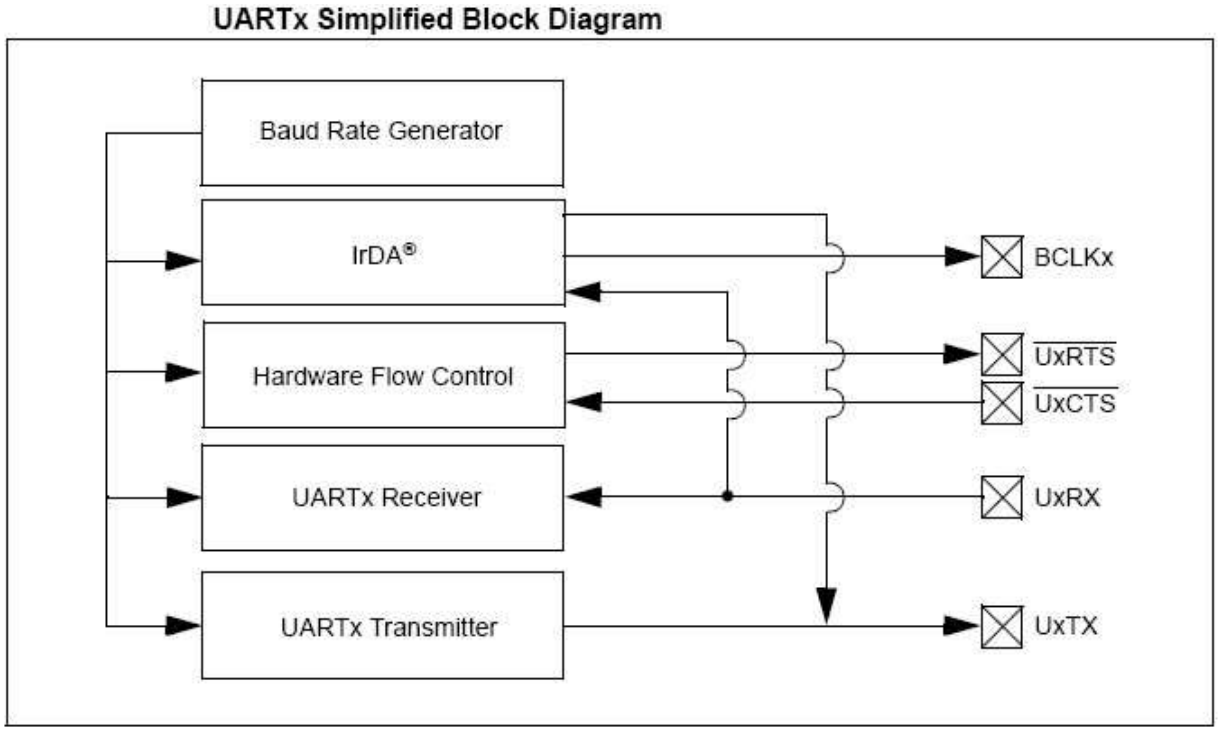
Des modules additionnels avec du hardware dédié :

- Interfaçage capteur local : I2C / SPI / UART
  - Distance de communication réduite
  - Fréquence de communication élevée
  - Impact logiciel embarqué réduit
- Interfaçage système : USB
  - Fréquence de communication très élevée
  - Impact logiciel embarqué important
- Interfaçage bus externes : CAN
  - Fréquence limitée
  - Hardware très spécifique et non présent dans tous les processeurs

Plan

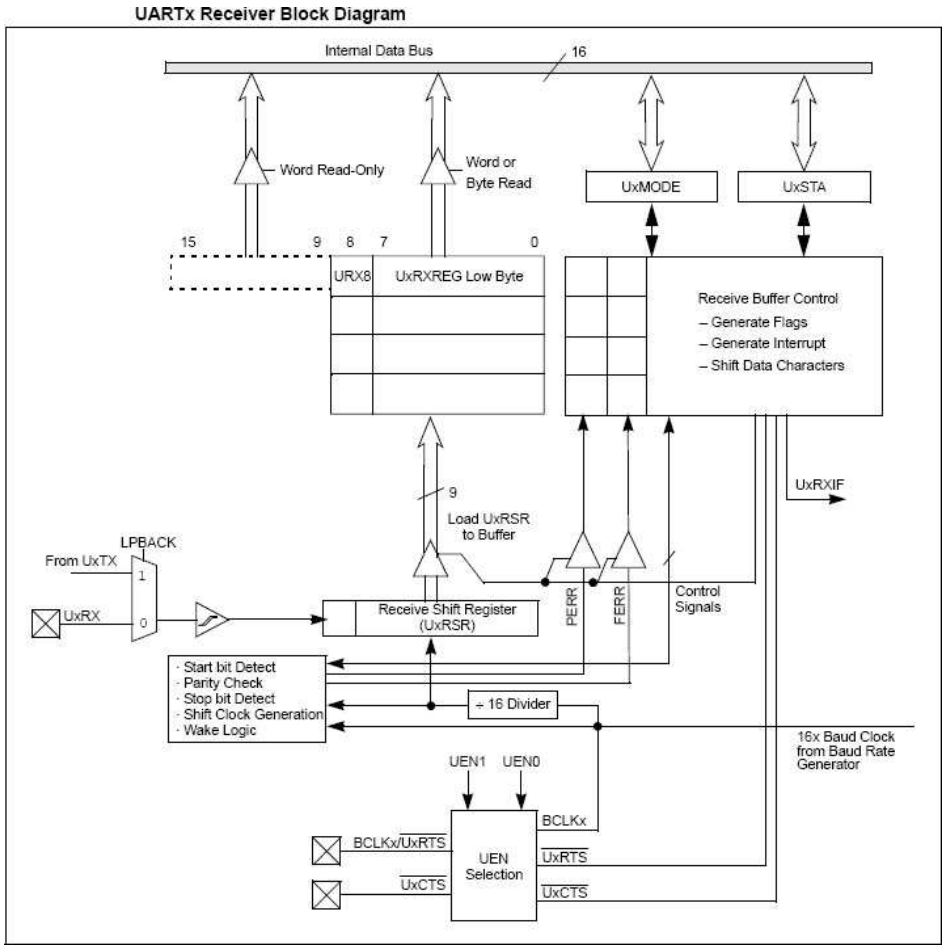
Le bus série UART

- 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) :

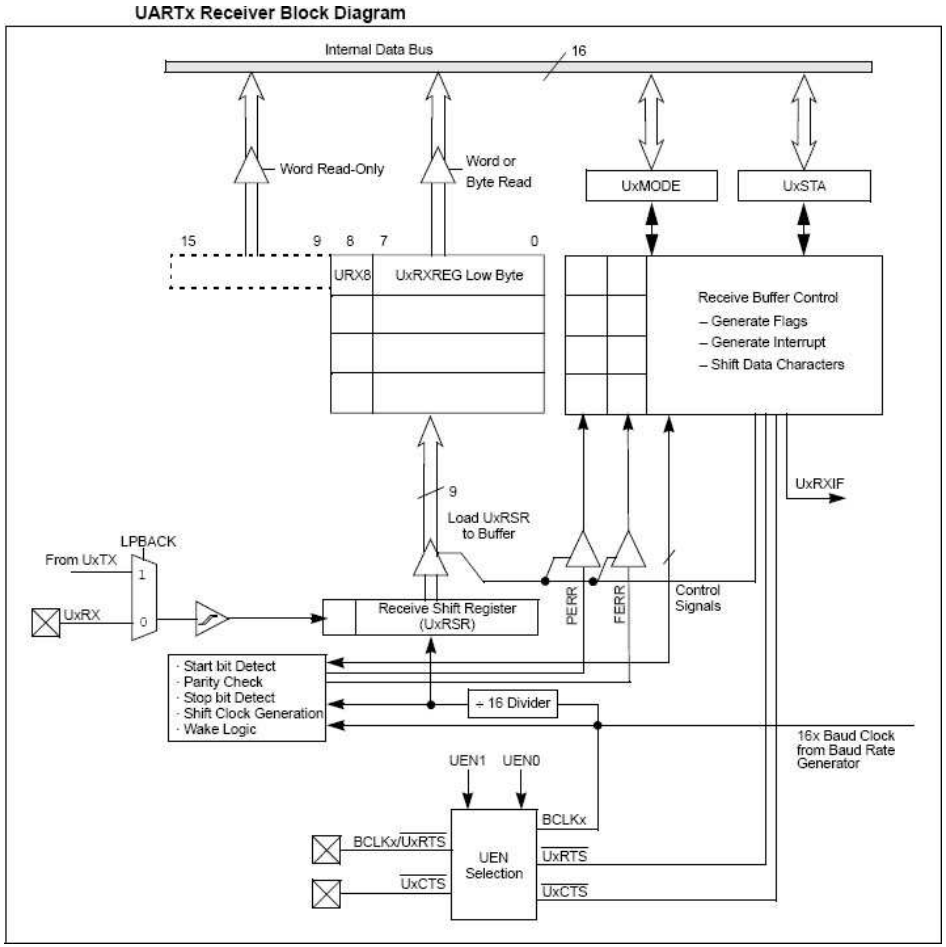


Le bus série UART : Initialisation

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)

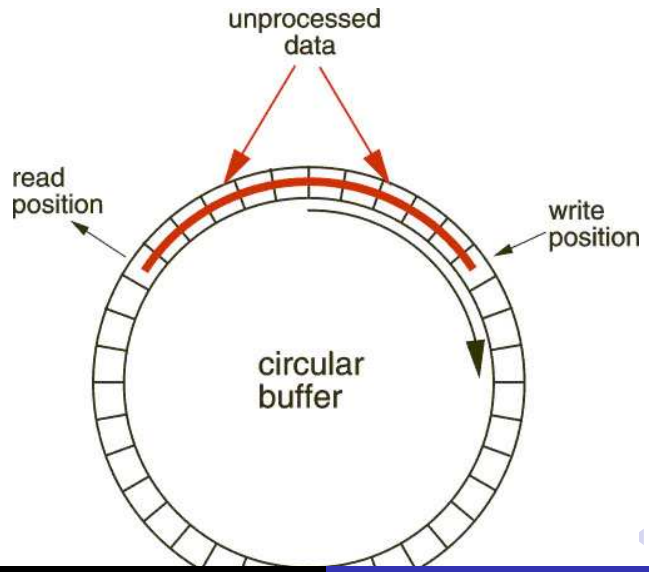
Réception sur interruption

```
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.

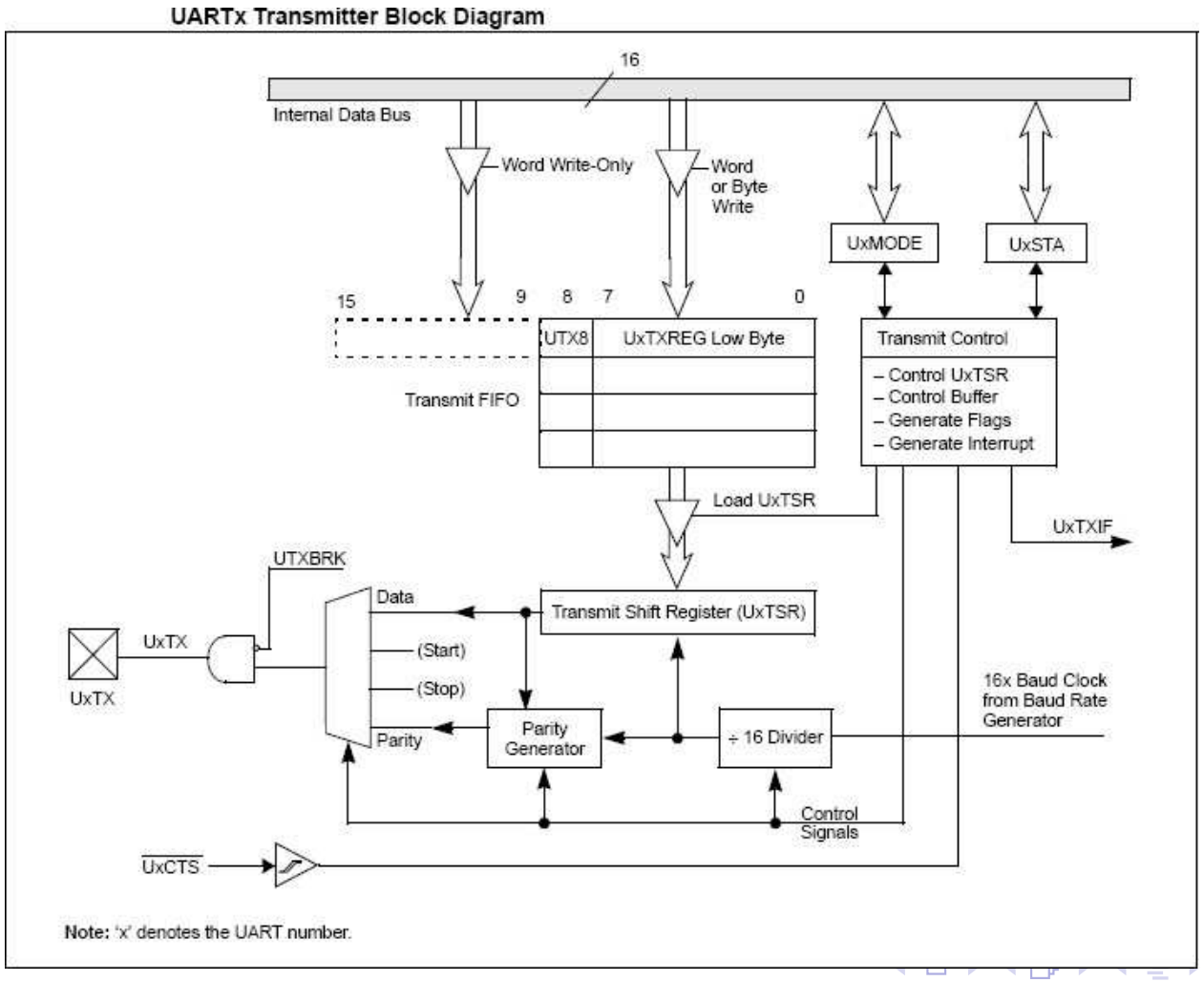
Le bus série UART : Buffer circulaire

- 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

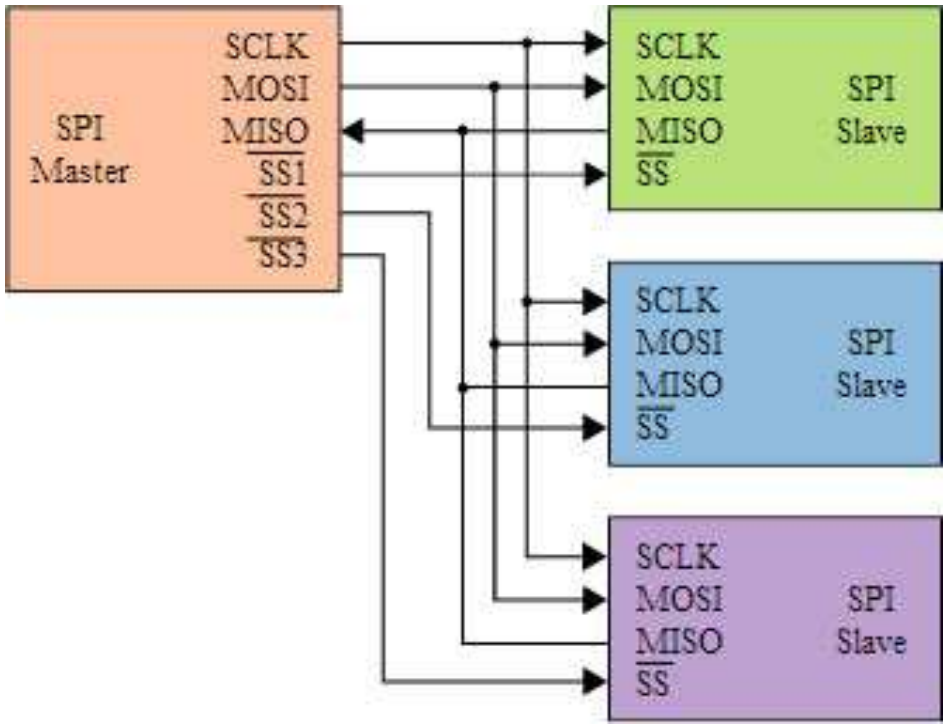
- L'envoi se fait sur interruption (non bloquant pour le reste)
- Un buffer circulaire peut être utilisé $\Rightarrow$ point d'envoi unique

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
}
```

Les bus de communication : le bus SPI

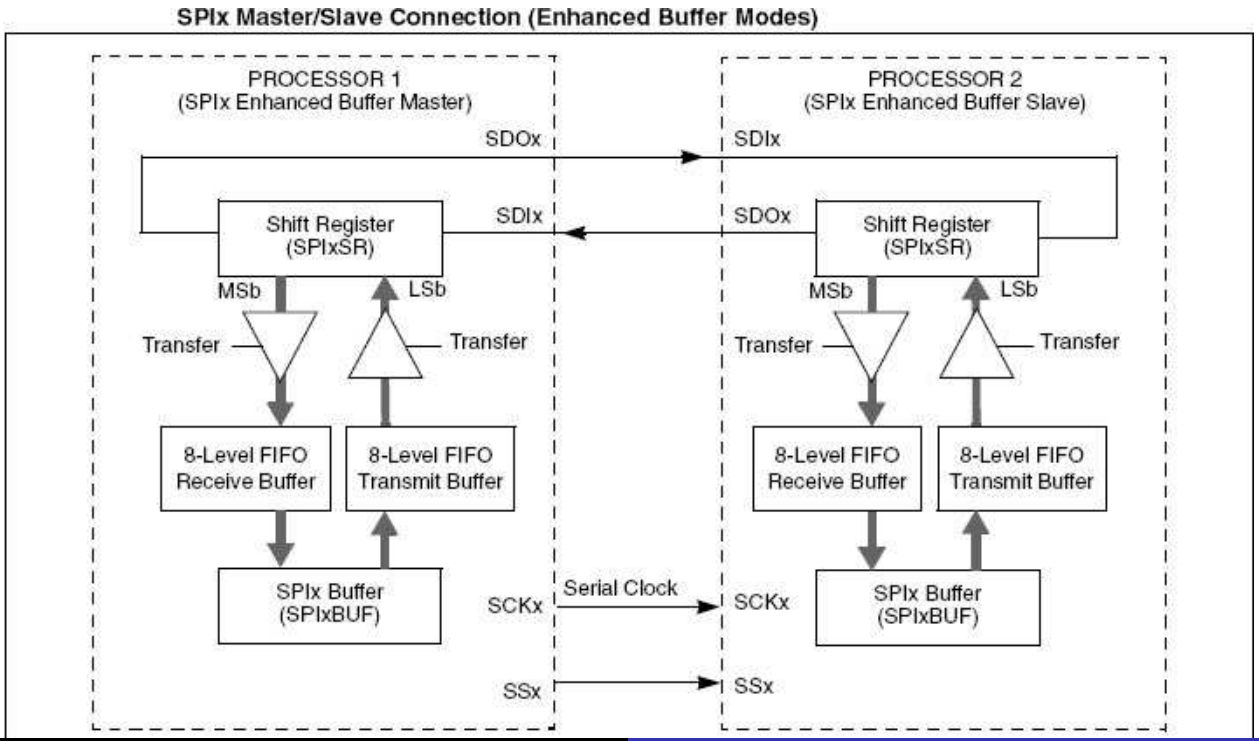
- **Bus trois fils + un *slave select* par circuit relié**
- **Communications maitre-esclave initiées par le processeur**



Le bus SPI : Principe

Principe du SPI

- Echange de données de type plateau tournant entre master et slave.
- Echange initié par le master.
- Le slave doit toujours être à l'écoute.



Le bus SPI : Avantages et inconvénients

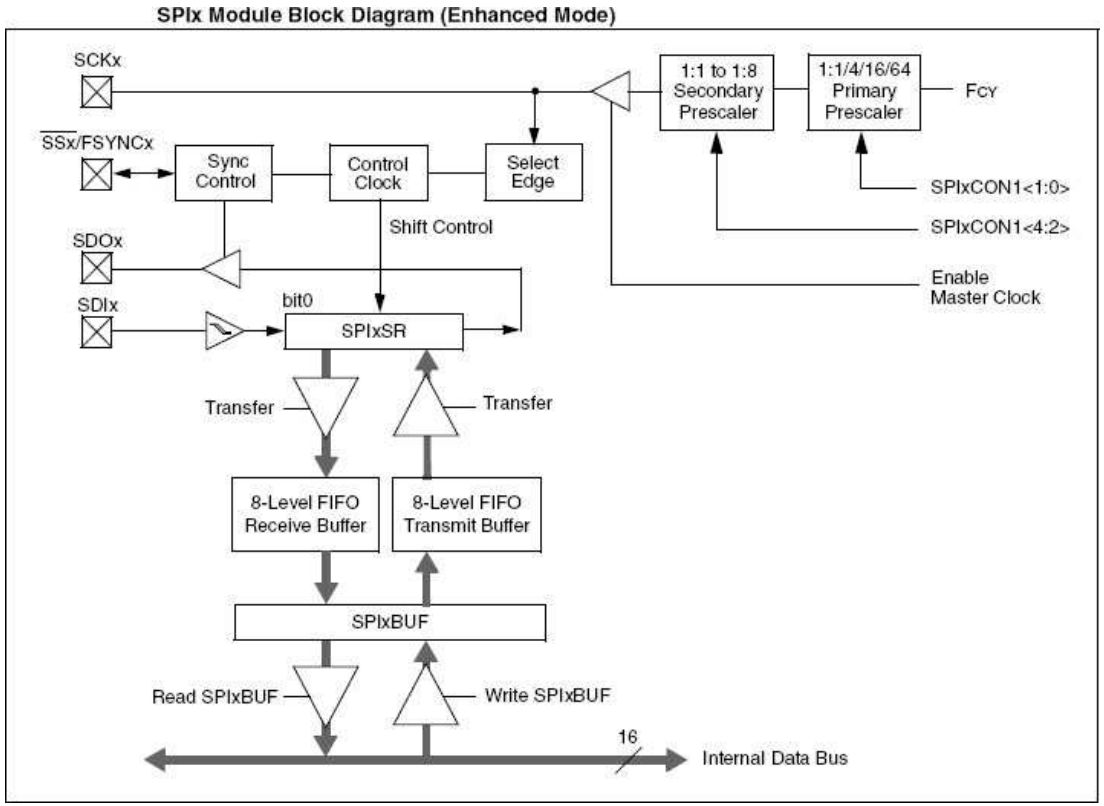
Avantages :

- Débit important : $> 10 Mbits$ sur microcontrôleur.
- Pas d'adressage ($\Rightarrow$ économie sur le flux transmis).
- Interface robuste et simple : plateau tournant

Inconvénients :

- Le master initie les communications $\Rightarrow$ besoin d'interruptions *data ready (DRDY)* en provenance des capteurs.
- Plus couteux en broches que l'I2C.
- Utilisable sur des distances réduites.
- Protocole non normalisé :
  - Attention aux réglages de *CKP* et *CKE*
  - Attention aux spécificités par rapport au SPI "classique".

Schéma interne du périphérique SPI sur PIC24F



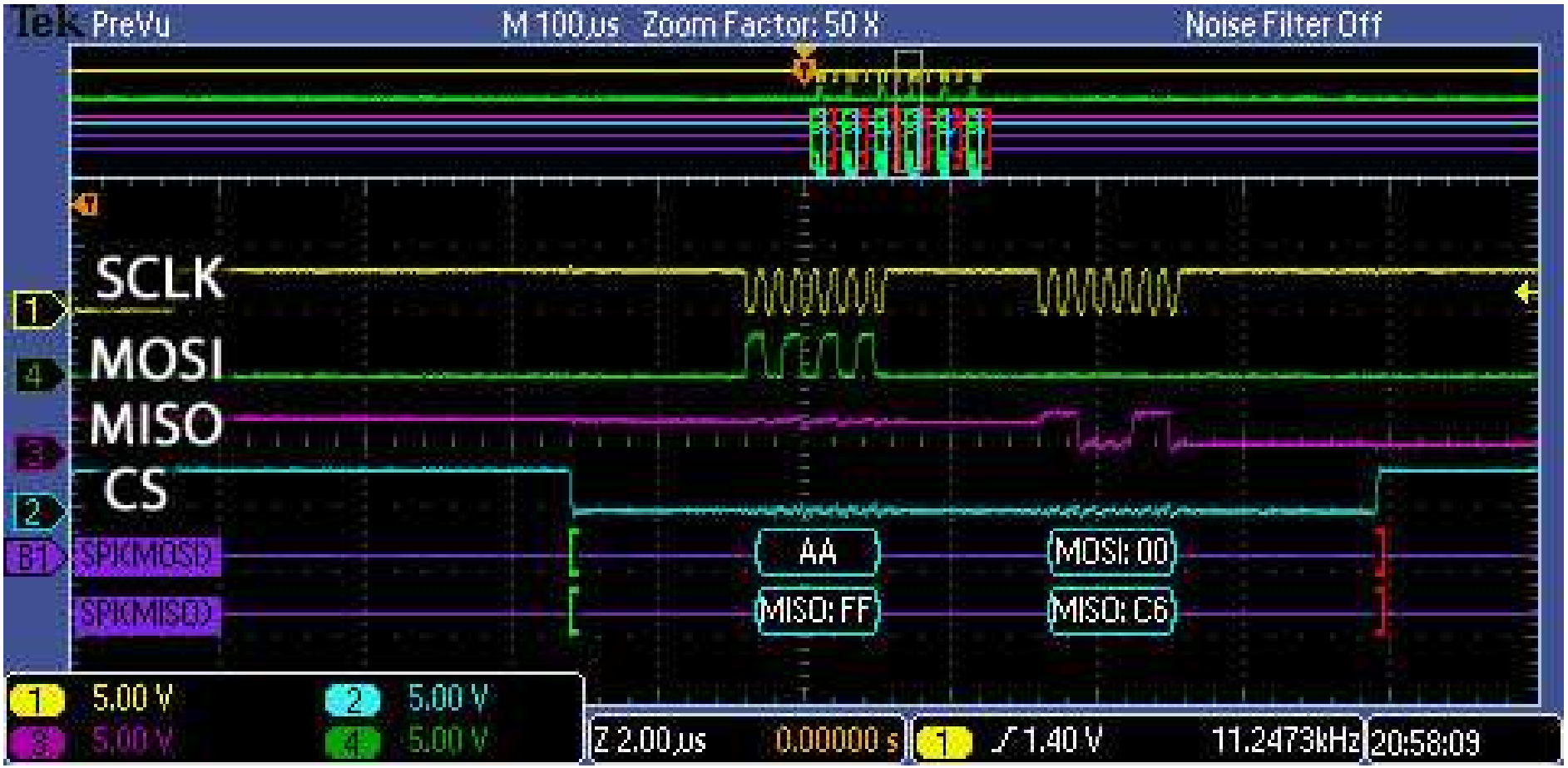
- Présence de FIFO 8 octets en réception et transmission.

Le bus SPI (Master) : Initialisation

Initialisation en mode interruption (sur les fins d'envoi)

```
void initSPI1(void){              // Init en mode master
    // SPI1CON1 Register Settings
    SPI1CON1bits.DISSCK = 0;// Internal Serial Clock is Enabled
    SPI1CON1bits.DISSDO = 0;// SDOx pin is controlled by the module
    SPI1CON1bits.MODE16 = 0;// 1 : Communication is word-wide (16 bits)
    SPI1CON1bits.SMP = 0; // Input data is sampled at the middle of data output time
    SPI1CON1bits.CKE = 0; // Serial output data changes on active to idle transition
    SPI1CON1bits.CKP = 0; // Clock Idle state : low level; active state : high level
    SPI1CON1bits.MSTEN = 1; // Master mode Enabled
    SPI1CON1bits.SPRE = 0b111; // 0b111 : 1:1
    SPI1CON1bits.PPRE = 0b10; // 0b00 : 64:1

    // SPI1CON2 Register Settings
    SPI1CON2bits.FRMEN = 0; // 0 : framing disabled
    SPI1CON2bits.SPIFSD = 0;// 0 : Frame sync pulse output Master
    SPI1CON2bits.FRMPOL = 0;// 0 : sync pulse active low

    // Interrupt Controller Settings
    IFS0bits.SPI1IF = 0; // Clear the Interrupt Flag
    IEC0bits.SPI1IE = 0; // Disable the Interrupt
    SPI1STATbits.SPIEN = 1; // Enable SPI module

    while ( SPI1STATbits.SPITBF ) ;
}
```

Le bus SPI (Master) : Opérations de lecture - écriture



Le bus SPI (Master) : Opérations de lecture - écriture

Commande SPI de base : écriture multiple suivie d'une lecture multiple

```
void WriteMultipleCommandMultipleReadSpi1(...){
    int i =0;
    ChipSelectSpi1SetValue(0, device);     //on déclenche le SS

    for (i=0; i<nbCommands; i++)     //On envoie les commandes
    {
            SPI1BUF=(unsigned int)commands[i];
            while ( SPI1STATbits.SPITBF ) ;
            dummy = SPI1BUF;
            while ( !SPI1STATbits.SPIRBF ) ;
            dummy = SPI1BUF;
    }
    for (i=0; i<nbResults; i++)
    {
            SPI1BUF=0x0000;          //On envoie un signal nul
            //quand TBF = 0 : fin de transmission
            while ( SPI1STATbits.SPITBF ) ;
            dummy = SPI1BUF;
            //quand RBF = 1 : données reçues sont dispo dans le buffer
            while ( !SPI1STATbits.SPIRBF ) ;
            results[i] = SPI1BUF;
    }

    ChipSelectSpi1SetValue(1, device);     //on termine le SS
}
```

Le bus SPI (Master) : Opérations de lecture - écriture

Des subtilités peuvent être nécessaires :

- Inversion de la polarité entre la lecture et l'écriture
- Ajout de délais pour laisser le temps au slave de réagir

Inversion de la polarité (CKE)

```
//on permute la valeur de CKE si le flag d'inversion de CKE est active
if(invertReadCKE)
    invertSPI1_CKE();
```

Ajout d'un delai

```
//Delay pour laisser le temps au slave de traiter les infos
__delay32(SPI_DELAY);
```

Le bus SPI (Slave) : Initialisation

Initialisation en mode interruption (sur les fins d'envoi)

```
void initSPI1(void){            // Init en mode slave
    // SPI1CON1 Register Settings
    SPI1CON1bits.DISSCK = 0;// Internal Serial Clock is Enabled
    SPI1CON1bits.DISSDO = 0;// SDOx pin is controlled by the module
    SPI1CON1bits.MODE16 = 0;// 1 : Communication is word-wide (16 bits)
    SPI1CON1bits.SMP = 0; // Input data is sampled at the middle of data output time
    SPI1CON1bits.CKE = 0; // Serial output data changes on active to idle transition
    SPI1CON1bits.CKP = 0; // Clock Idle state : low level; active state : high level
    SPI1CON1bits.MSTEN = 0; // Master mode Enabled
    SPI1CON1bits.SPRE = 0b111; // 0b111 : 1:1
    SPI1CON1bits.PPRE = 0b10; // 0b00 : 64:1

    // SPI1CON2 Register Settings
    SPI1CON2bits.FRMEN = 0; // 0 : framing disabled
    SPI1CON2bits.SPIFSD = 0;// 0 : Frame sync pulse output Master
    SPI1CON2bits.FRMPOL = 0;// 0 : sync pulse active low

    // Interrupt Controller Settings
    IFS0bits.SPI1IF = 0; // Clear the Interrupt Flag
    IEC0bits.SPI1IE = 0; // Disable the Interrupt
    SPI1STATbits.SPIEN = 1; // Enable SPI module

    while ( SPI1STATbits.SPITBF ) ;
}
```

Le bus SPI (Slave) : particularités

Le récepteur esclave : à l'écoute en permanence

- Le SPI Slave doit répondre très rapidement aux demandes du Master (horloge $> 4Mbps$) : fonctionnement sur interruption.
- $\Rightarrow$ Toutes les données doivent être disponibles dans des registres.
- $\Rightarrow$ Les interruptions SPI Slave doivent être prioritaires sur tout.
- La machine à état de gestion du SPI doit être performante :
  - Eviter les tests : le Master doit savoir a priori la taille des informations à écrire ou récupérer.
  - Utilisation de pointeurs obligatoire : pas de duplication de variable

Le bus SPI (Slave) : Interruption en fin de transfert d'un octet

Interruption du SPI slave

```
//On entre dans l'interruption à chaque fois que l'on recoit un octet complet.
//(8 coups d'horloge sur SCLK)
void __attribute__((interrupt, no_auto_psv)) _SPI1Interrupt(void)
{
    IFS0bits.SPI1IF = 0;                //Clear the interrupt flag
    if(SPI1STATbits.SPIROV != 0)
    {
        //Clear any errors
        SPI1STATbits.SPIROV = 0;
        IFS0bits.SPI1EIF = 0;
    }
    SpiSlaveStateMachine();             //Processing dans la machine à état
    while(SPI1STATbits.SPITBF);         //Attente du départ dans le buffer d'envoi
}
```

Le bus SPI (Slave) : Machine à état

Machine à état de gestion du SPI slave

```
inline void SpiSlaveStateMachine(){
    switch (cIndex)
    {
        case STATE_SELECT_READ_WRITE:              //On recupere la commande (READ/WRITE)
            spiReadWrite = SPI1BUF;                //Read in SPI1 buffer
            cIndex = STATE_READ_REGADDR;           //passe a l'etape suivante
            SPI1BUF = 0x01;                        //dummy read
            break;
       case STATE_READ_REGADDR:                    //On recupere l'adresse du registre
            spiRegister = SPI1BUF;                 //registre impacté
            spiRegisterPointer = spiGetRegisterPointer(spiRegister);
            if(spiReadWrite == READ_CMD){          //Si mode lecture
                SPI1BUF = *spiRegisterPointer++; //On renvoie le premier octet
                cIndex = STATE_READ;}
            else{                                  //Si mode écriture
                SPI1BUF = 0x02;                    //dummy read
                cIndex = STATE_WRITE;}
            break;
        case STATE_READ:                    //On envoie la suite du registre
            SPI1BUF = *spiRegisterPointer++;
            break;
        case STATE_WRITE:                   //On écrit dans le registre
            *spiRegisterPointer++ = SPI1BUF;
            break;
    } // end switch
    return;
```

Le bus SPI : Avantages et inconvénients

Avantages :

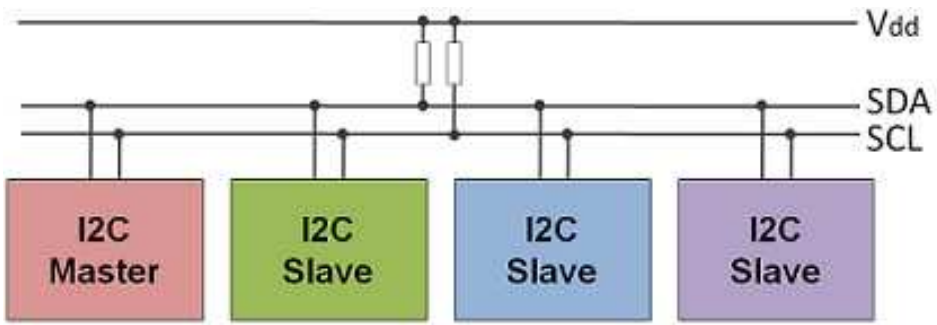
- Débit important : $> 10 Mbits$ sur microcontrôleur.
- Pas d'adressage ($\Rightarrow$ économie sur le flux transmis).
- Interface robuste et simple : plateau tournant

Inconvénients :

- Le master initie les communications $\Rightarrow$ besoin d'interruptions *data ready (DRDY)* en provenance des capteurs.
- Plus couteux en broches que l'I2C.
- Utilisable sur des distances réduites.
- Protocole non normalisé :
  - Attention aux réglages de *CKP* et *CKE*
  - Attention aux spécificités par rapport au SPI "classique".
- Mode slave complexe à implanter

Les bus de communication : le bus I2C

- **Bus deux fils**
- **Communications maitre-esclave initiées par le processeur**



Le bus I2C : Avantages et inconvénients

Avantages :

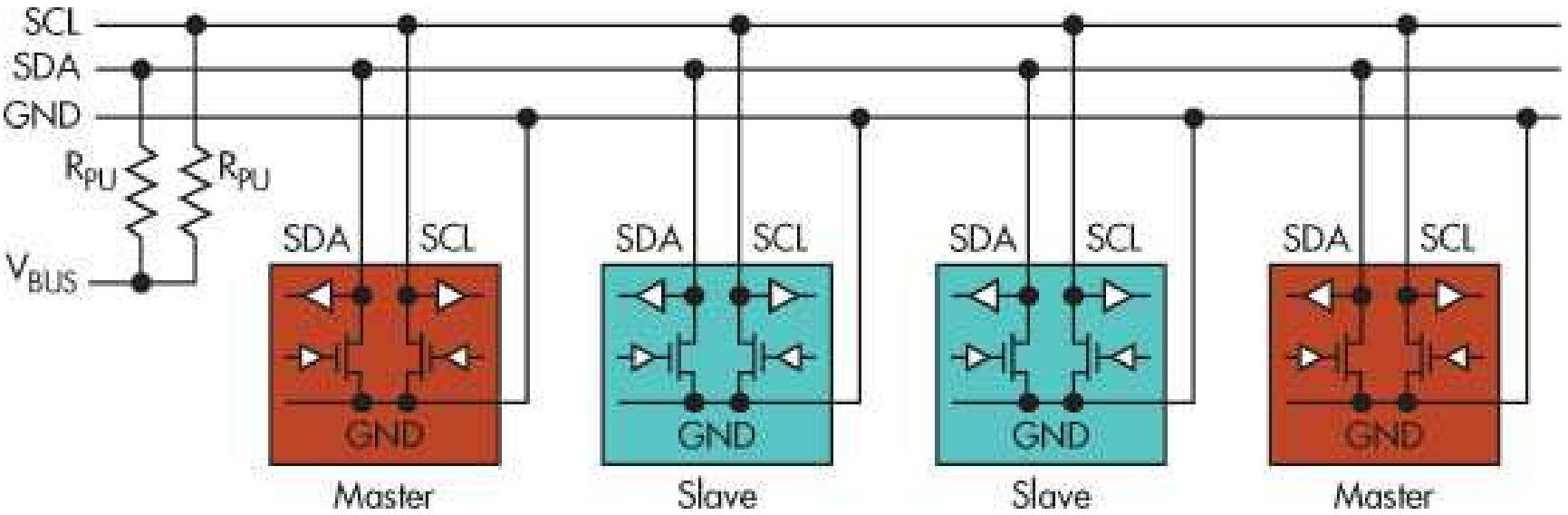
- Adressage des modules $\Rightarrow$ pas de fil de Chip Select
- Coût fixe : 2 fils.
- Protocole normalisé par NXP.

Inconvénients :

- Débit faible : 400*kbits* sur microcontrôleur
- Adressage des modules $\Rightarrow$ perte de temps dans les communications
- Le master initie les communications $\Rightarrow$ besoin d'interruptions *data ready (DRDY)* en provenance des capteurs.
- Utilisable sur des distances réduites.

Le bus I2C : un bus partagé

- Electronique permettant des conflits sur le bus : Open Drain
- Présence de deux résistances de pull-up



Le bus I2C : un bus partagé

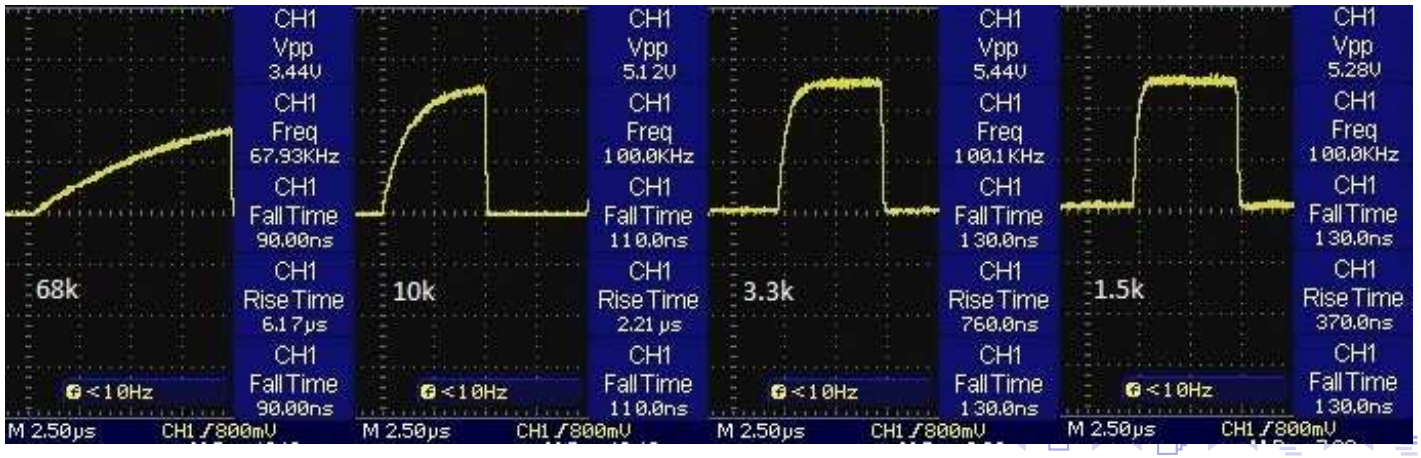
Les résistances de pull-up doivent être dimensionnées avec précautions

- Valeur minimum limitée par le *sink current* des pins :

$$R_{min} = \frac{V_{dd}}{I_{OL}} = \frac{3.3V}{4mA} = 825\Omega$$

- Valeur maximum limitée par le temps de montée t_R maximum :

$$R_{max} = \frac{t_R}{C_{bus} * 3} = \frac{300n}{30pF * 3} = 3.33k\Omega(400kHz) = \frac{1000n}{30pF * 3} = 11k\Omega(100kHz$$



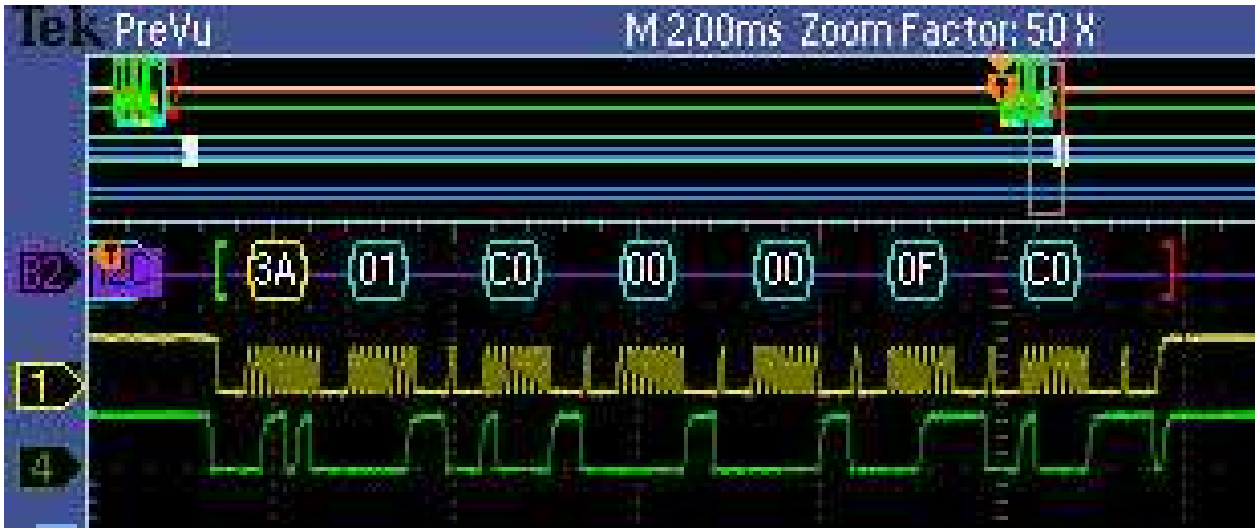
Le bus I2C (Master) : Initialisation

Initialisation en mode interruption

```
void InitI2C1(void)
{
    IdleI2C1();
    I2C1CONbits.I2CEN = 0;          //Module I2C arrêté

    I2C1CONbits.I2CSIDL = 0;        // 0 = Continue module operation in Idle mode
    I2C1CONbits.IPMIEN = 0;         // 0 = IPMI Support mode disabled
    I2C1CONbits.A10M = 0;           // 0 = I2CxADD is a 7-bit slave address
    I2C1CONbits.DISSLW = 0;         // 0 = Slew rate control enabled
    I2C1CONbits.SMEN = 0;           // 0 = Disable SMBus input thresholds
    I2C1CONbits.GCEN = 0;           // 0 = General call address disabled
    I2C1CONbits.STREN = 0;          // 0 = Disable software or receive clock stretching
    I2C1CONbits.ACKDT = 1;          // 1 = Send NACK during Acknowledge

    I2C1BRG = 20;                   // Valeur donnant réellement 1MHz avec du 32MHz
    //Attention : un minimum de 14 avec des pull-up de 1.2KOhm

    I2C1CONbits.I2CEN = 1;          // Module I2C démarré
    IEC1bits.MI2C1IE = 1;           //On active les interruptions I2C
}
```

Le bus I2C : Ecriture avec adressage 7 bits

- Start
- Ecriture en 0x3A=0b00111010 de 0x01
  $\Leftrightarrow$ Reg 0x01 actif (adresse device : 0b0011101)
- Ecriture des data sans restart
- Stop



Le bus I2C : Ecriture

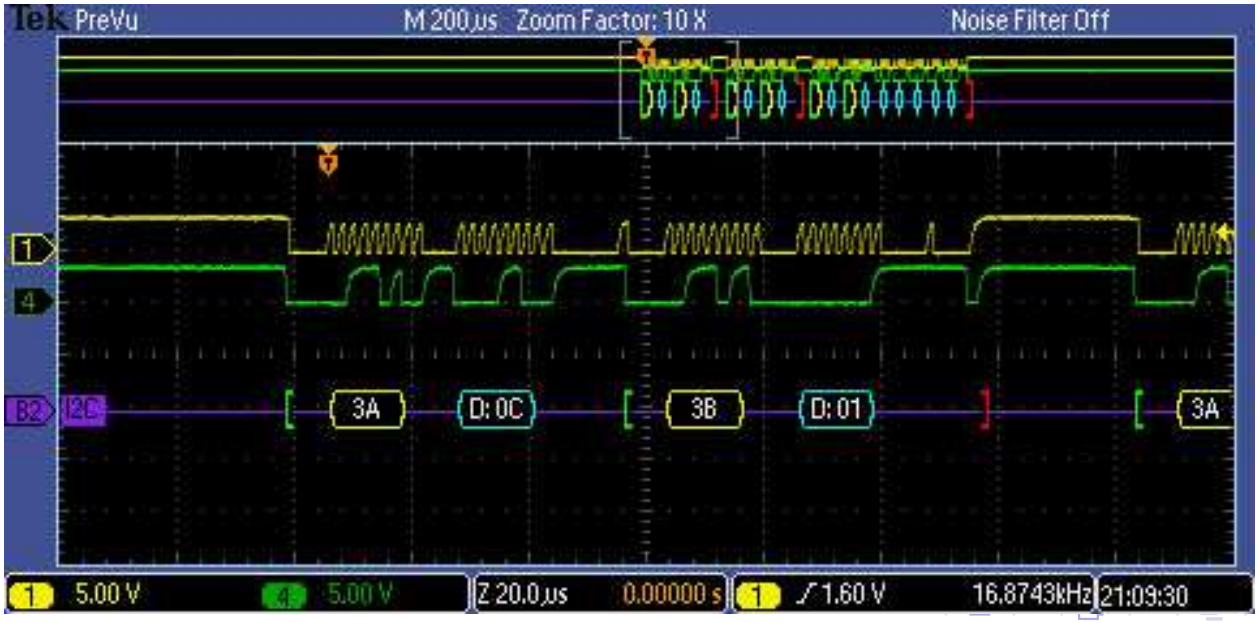
Les écritures peuvent être réalisées sous interruption en stockant au préalable les données à écrire dans un buffer.

Ecriture avec buffer de préchargement et séquence lancée en mode interrupt

```
void I2C1WriteNInterrupt( unsigned char slaveAddress, unsigned char registerAddress,
                          unsigned char* data, unsigned int length ) {
    I2CData msgCommandData;

    msgCommandData.RW = I2C_WRITE;
    msgCommandData.data[0] = slaveAddress & 0xFE;
    msgCommandData.data[1] = registerAddress;

    for (i=0;i<length+2;i++)
        msgCommandData.data[2+i] = data[i];

    msgCommandData.length = 2+length;

    I2C1WriteToBuffer(msgCommandData); // On place les messages dans le buffer

    // On lance la tranmission si aucune transmission n'est en cours
    if(I2C1IsTransmissionActive()==FALSE)
        StartI2C1Message();
}
```

Le bus I2C : Lecture avec adressage 7 bits

- Start
- Ecriture en 0x3A=0b00111010 de 0x0C
  $\Leftrightarrow$ Reg 0x0C actif (adresse device : 0b0011101).
- Restart
- Lecture en 0x3B = 0b00111011 du registre actif
- Stop



Le bus I2C : Lectures

Les Lectures peuvent être aussi réalisées sous interruption.

Lecture avec buffer de préchargement et séquence lancée en mode interrupt

```
void I2C1ReadNInterrupt( unsigned char slaveAddress, unsigned char registerAddress,
                         volatile unsigned char* data, unsigned int length )
{
    I2CData msgCommand, msgData;
    msgCommand.RW = I2C_WRITE;
    msgCommand.data[0] = slaveAddress & 0xFE;
    msgCommand.data[1] = registerAddress;
    msgCommand.length = 2;

    msgData.RW = I2C_READ;
    msgData.data[0] = slaveAddress  | 0x01;
    msgData.length = length;

    I2C1WriteToBuffer(msgCommand);   // On place les messages dans le buffer
    I2C1WriteToBuffer(msgData);

    // On lance la tranmission si aucune transmission n'est en cours
    if(I2C1IsTransmissionActive()==FALSE)
        StartI2C1Message();

    for (i=0; i<length; i++)
        data[i] = currentI2CMsg.data[i];
}
```

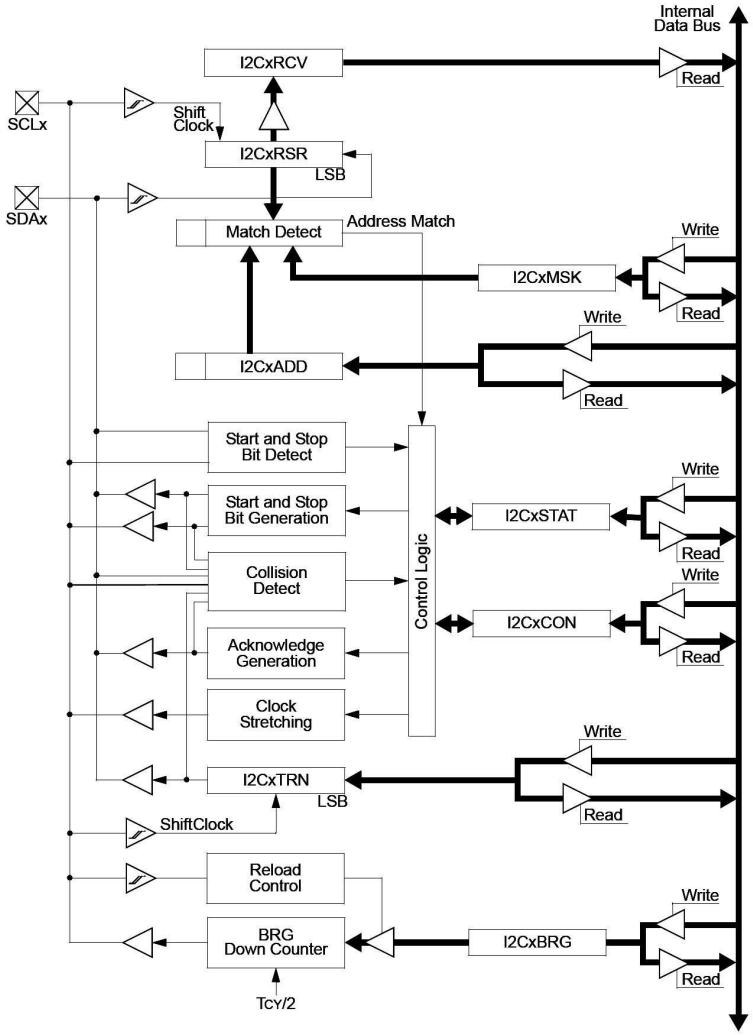
Le bus I2C : Interruptions

La machine à état du bus I2C peut être **gérée sous interruptions** afin d'éviter de bloquer le processeur.
La fonction *I2C1TransmissionOperation* effectue les envois depuis le buffer de stockage

Lecture avec buffer de préchargement et séquence lancée en mode interrupt

```
void __attribute__((interrupt, no_auto_psv)) _MI2C1Interrupt(void)
{
    IFS1bits.MI2C1IF = 0; // Clear CN interrupt
    switch(I2C1State)
    {
        case I2C_START:                          //On vient de terminer le start
            I2C1TrnCounter = 0;
            I2C1State = I2C_MASTER_TRANSMIT;     //On passe en transmission
            I2C1TransmissionOperation(); break;
        case I2C_MASTER_TRANSMIT:
            I2C1TransmissionOperation(); break;
        case I2C_RESTART:                        //On vient de terminer le restart
            I2C1TrnCounter = 0;
            I2C1State = I2C_MASTER_TRANSMIT;
            I2C1TransmissionOperation(); break;
        case I2C_STOP:                           //Transmission terminée
            I2C1State = I2C_IDLE; break;
    }
}
```

Schéma du périphérique I2C

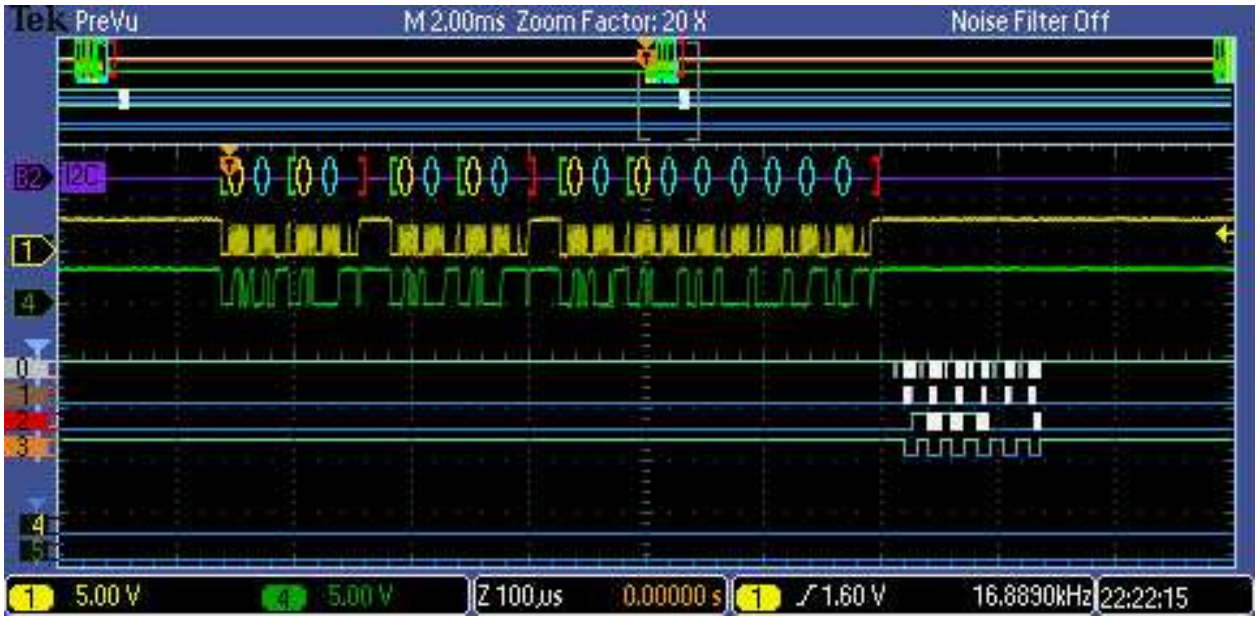


Le bus I2C : extensions

- Adressage sur 10 bits
- Réponse à tous les messages (mode IPMI) $\Rightarrow$ répéteurs I2C.
- Mode multi-masters : présence d'un arbitrage intégré au périphérique I2C.
- Compatibilité possible avec le SMBus (System Management Bus).
- Clock stretching : permet d'interrompre une requête temporairement.

Comparaison I2C - SPI

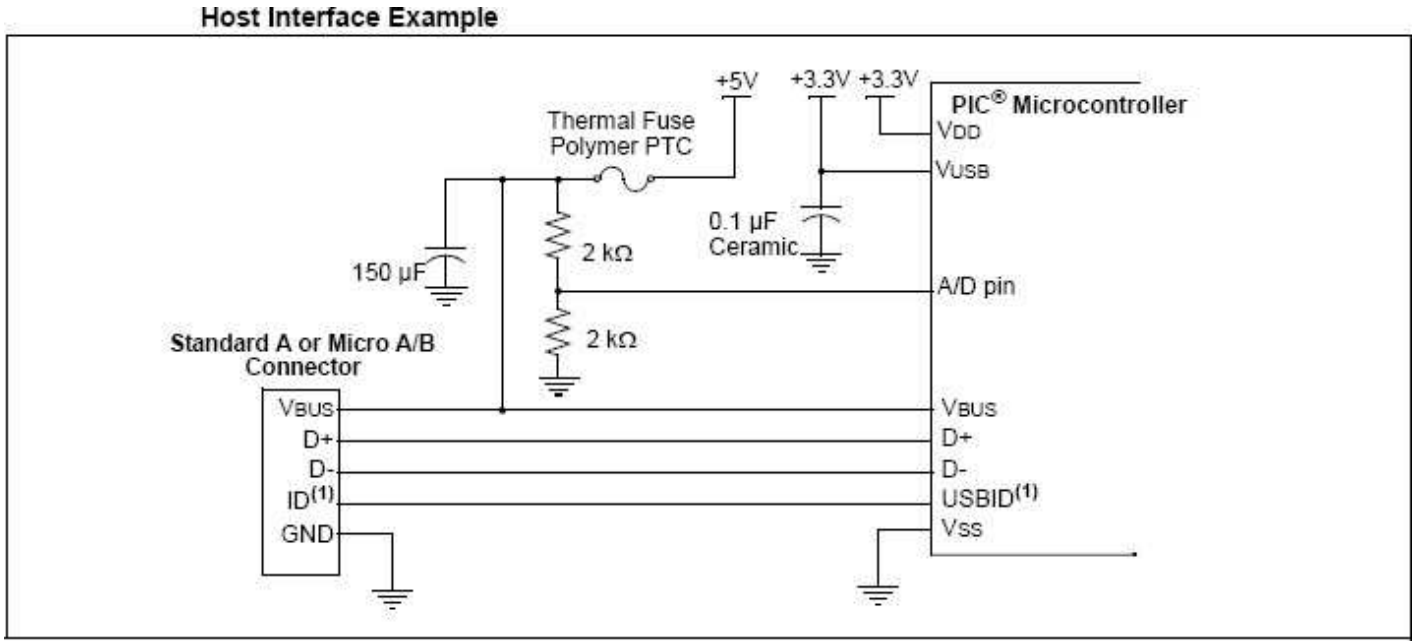
- Débit : **SPI** 4*Mbps* - I2C 400*kbps*
- Pins : **I2C** 2 **pins** - SPI $3 + n$ pins
- Consommation : **faible en SPI** - modérée en I2C (en raison de l'Open Drain)
- Topologie : **Vrai bus en I2C** - bus dégradé en SPI



Plan

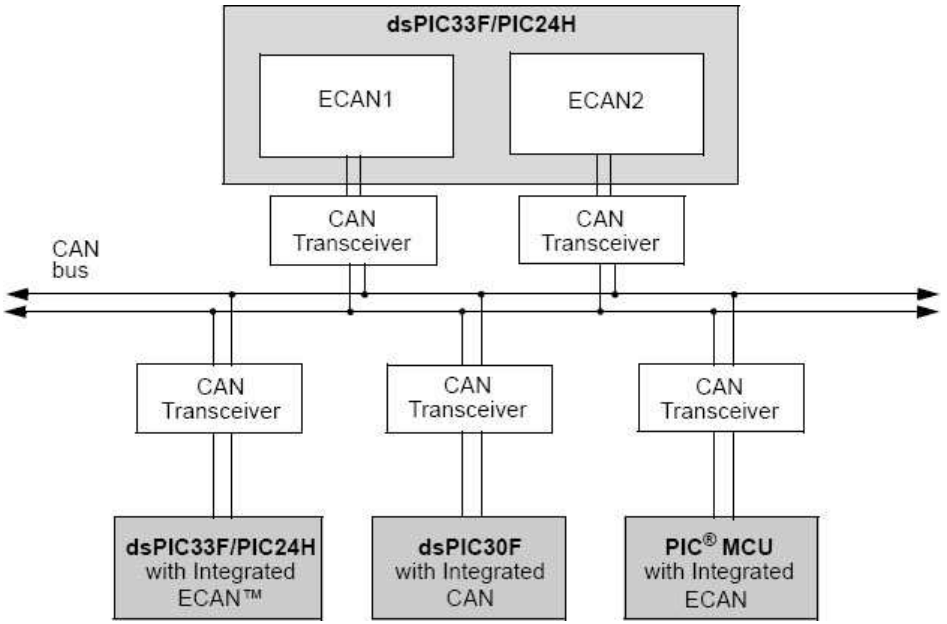
Interfaçage système : le bus USB

- Communication avec un **unique circuit**
- Mode de communication fixé (device, host ou OTG)
- Communications en USB 2.0 : *jusqu'à 48 Mb/s*
- Impact important sur la mémoire programme (20 kBytes)



Les bus de communication : le bus CAN

- Compatible avec la norme 2.0B
- Débit : jusqu'à 1Mbps
- Buffer FIFO de 32 messages en réception
- Buffer FIFO de 8 messages en émission
- 16 filtres d'acceptation de messages



Plan

Les contraintes d'un système temps réel

Un système temps réel embarqué est soumis à des fortes contraintes.

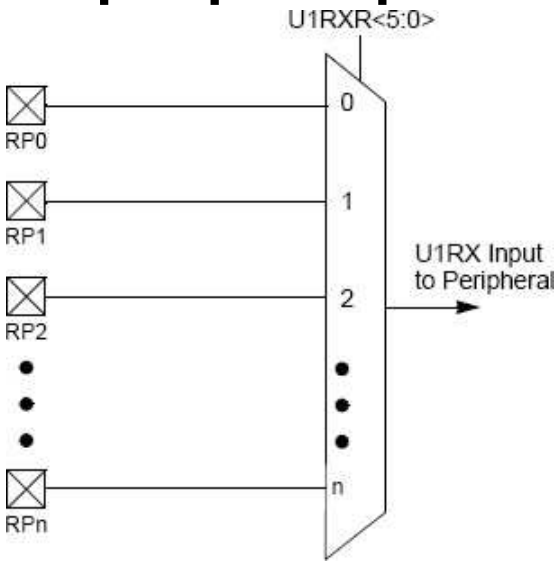
- Contraintes de conception :
  - Forte densité de routage
  - Besoin de placer certains capteurs dans des endroits précis
  - Isolement de certaines alimentations pour les parties analogiques
- Contraintes de fonctionnement
  - Nombreux échanges des données entre les capteurs, les carte périphériques et la mémoire.
  - Algorithmes embarqués et devant fonctionne en temps réel

Peripheral Pin Select : pins remappables

- **Principe** :
  - Le routage des pistes de PCB vers des pins fixées est parfois difficile.
  - Le Peripheral Pin Select permet de choisir les pins d'entrées-sortie de certains périphériques.
  - **Le choix des pins doit être effectué, sinon le périphérique ne fonctionne pas !**
- Les périphériques concernés sont principalement : Timers, UART, SPI
- Certains périphériques ne sont pas remappables :
  - I2C
  - Entrées - sorties analogiques
  - USB

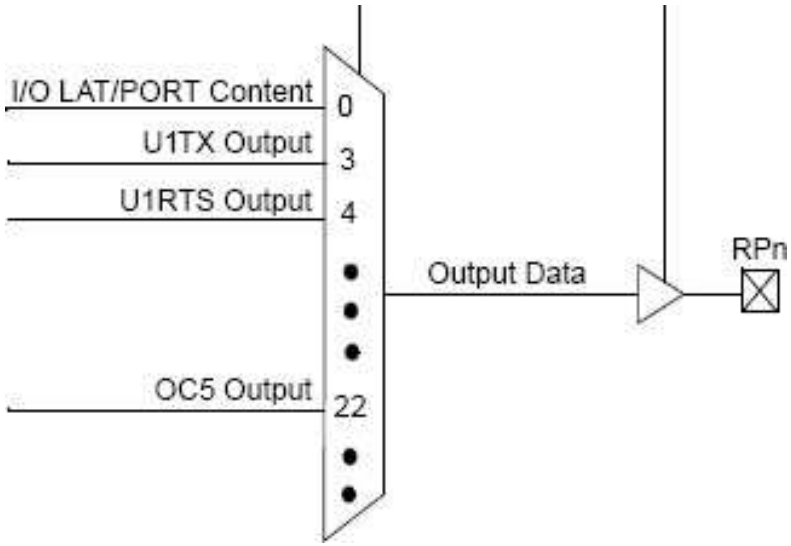
Peripheral Pin Select : pins remappables

Entrée d'une pin externe vers un périphérique



Exemple : RP23 vers UART1 RX
RPINR18bits.U1RXR = 23 ;

Sortie d'un périphérique vers une pin externe



Exemple : UART1 RX vers RP2
RPOR1bits.RP2R = 3 ; // 3 signifie U1TX

Peripheral Pin Select : pins remappables

Fonction assembleur de verrouillage la config des entrées-sorties

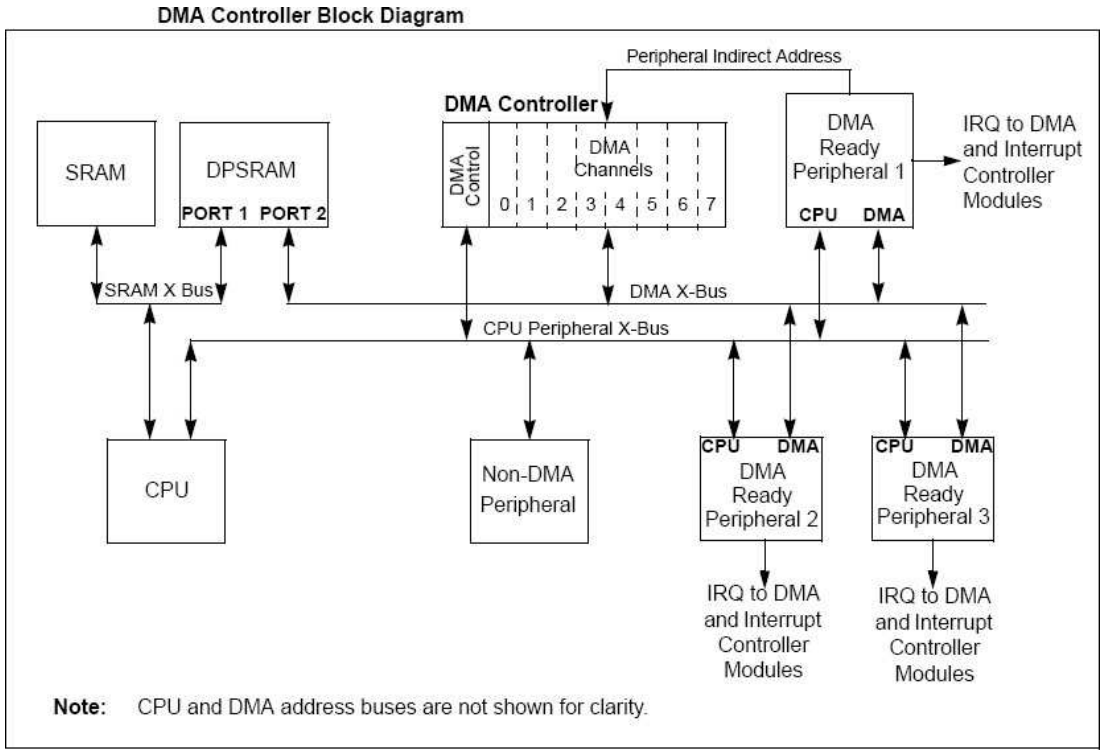
```
void LockIO()
{
    asm volatile ("mov #OSCCON,w1 \n"
                  "mov #0x46, w2 \n"
                  "mov #0x57, w3 \n"
                  "mov.b w2,[w1] \n"
                  "mov.b w3,[w1] \n"
                  "bset OSCCON, #6" : : : "w1","w2","w3");
}
```

Fonction assembleur de déverrouillage la config des entrées-sorties

```
void UnlockIO()
{
    asm volatile ("mov #OSCCON,w1 \n"
                  "mov #0x46, w2 \n"
                  "mov #0x57, w3 \n"
                  "mov.b w2,[w1] \n"
                  "mov.b w3,[w1] \n"
                  "bclr OSCCON, #6" : : : "w1","w2","w3");
}
```

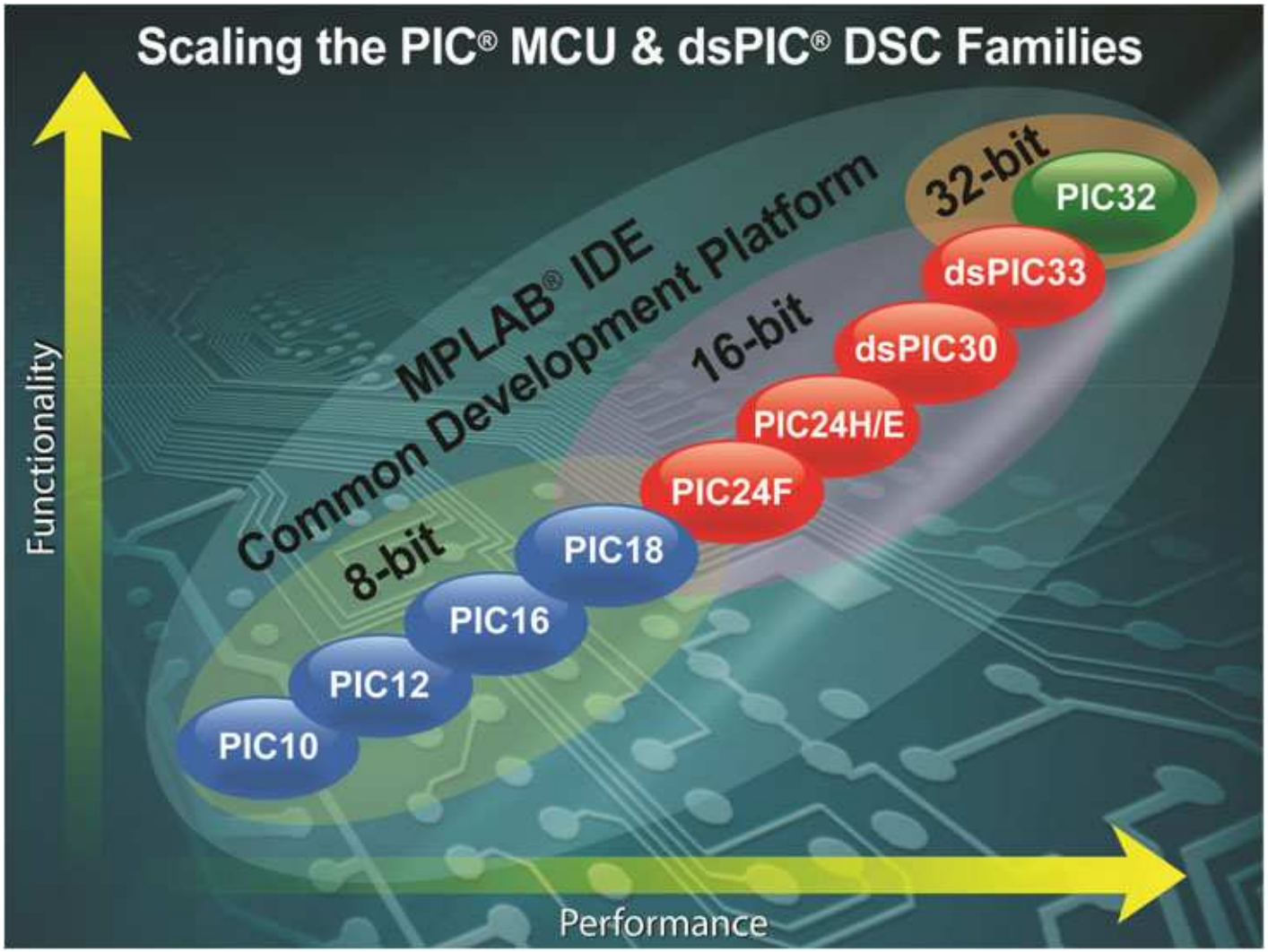
Le contrôleur DMA

- **Permet de réduire la charge du bus processeur**
- **Opérations autonomes entre I/O et mémoire DMA**



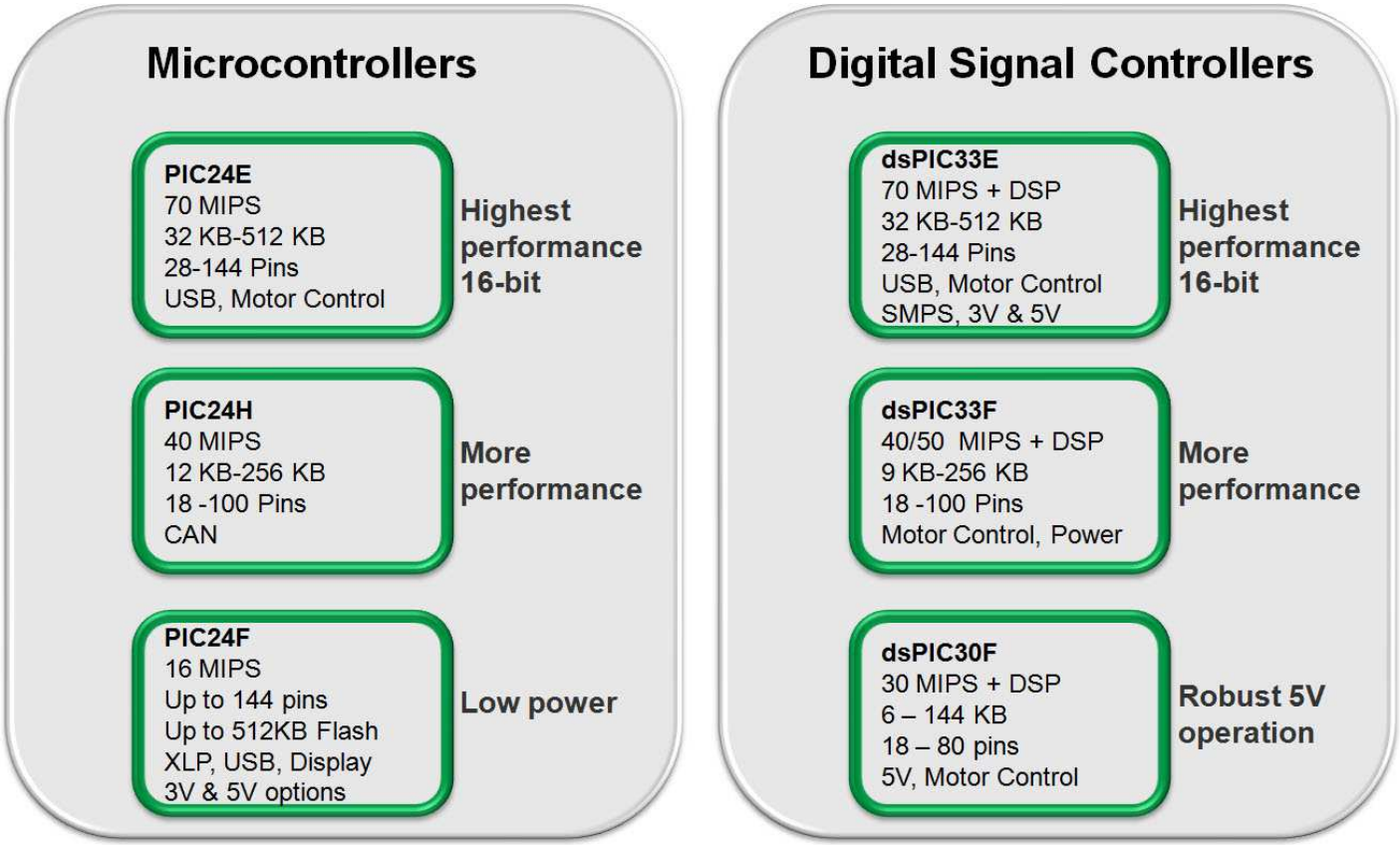
Le calcul en virgule fixe

Exemple de gamme de microcontrôleurs

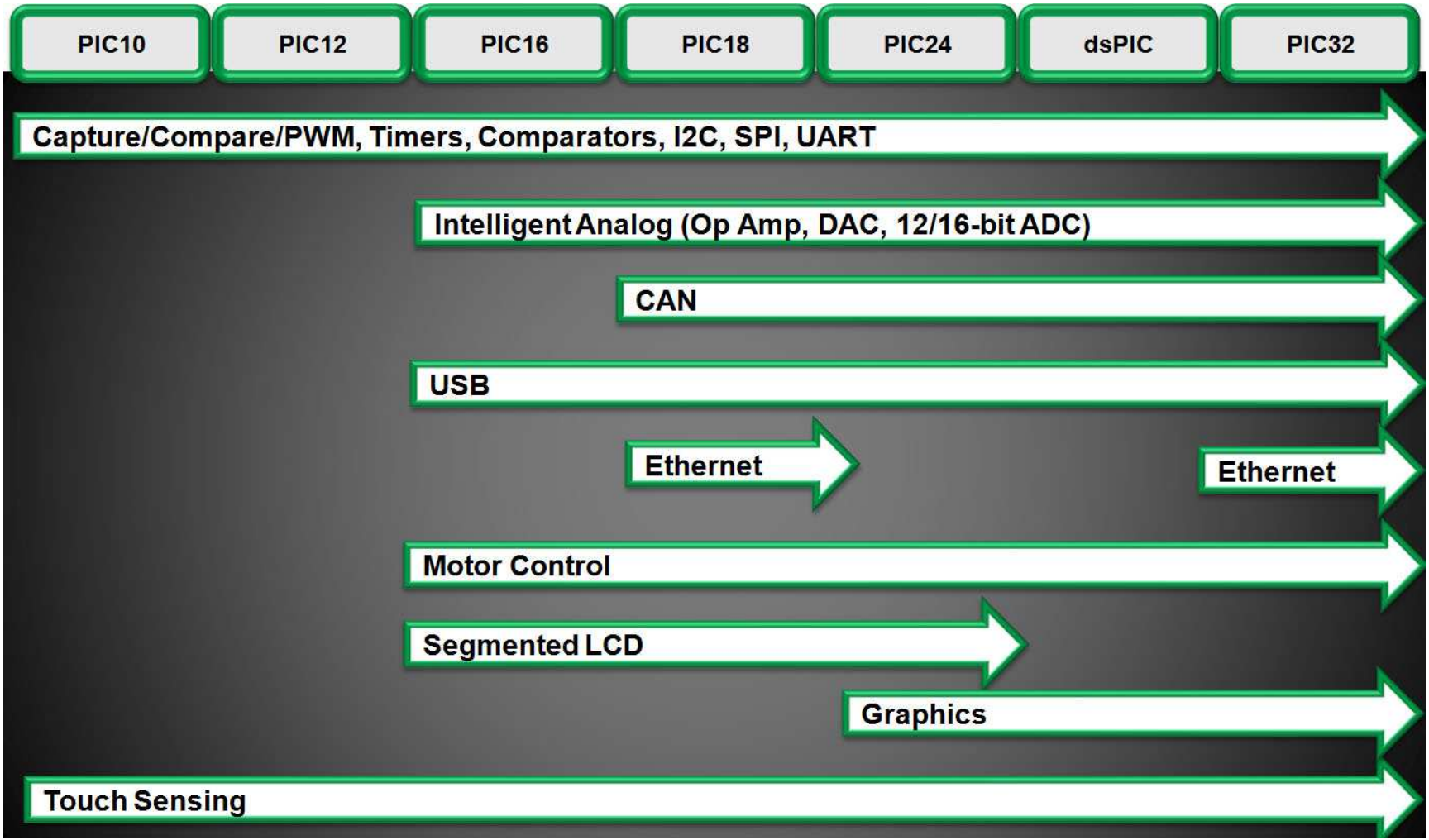


Les usages des familles 16 bits de Microchip

Un marché divisé en deux : μC et DSP



Les périphériques disponibles selon les familles de μC



Les périphériques orientés "connectivité"

- UART : 2 à 6 par PIC24F
- I2C : 1 ou 2 par PIC24F
- SPI : 1 à 4 par PIC24F
- CAN : supporte le 1.2, 2.0A, 2.0B
- USB : Device / Host / On the Go.

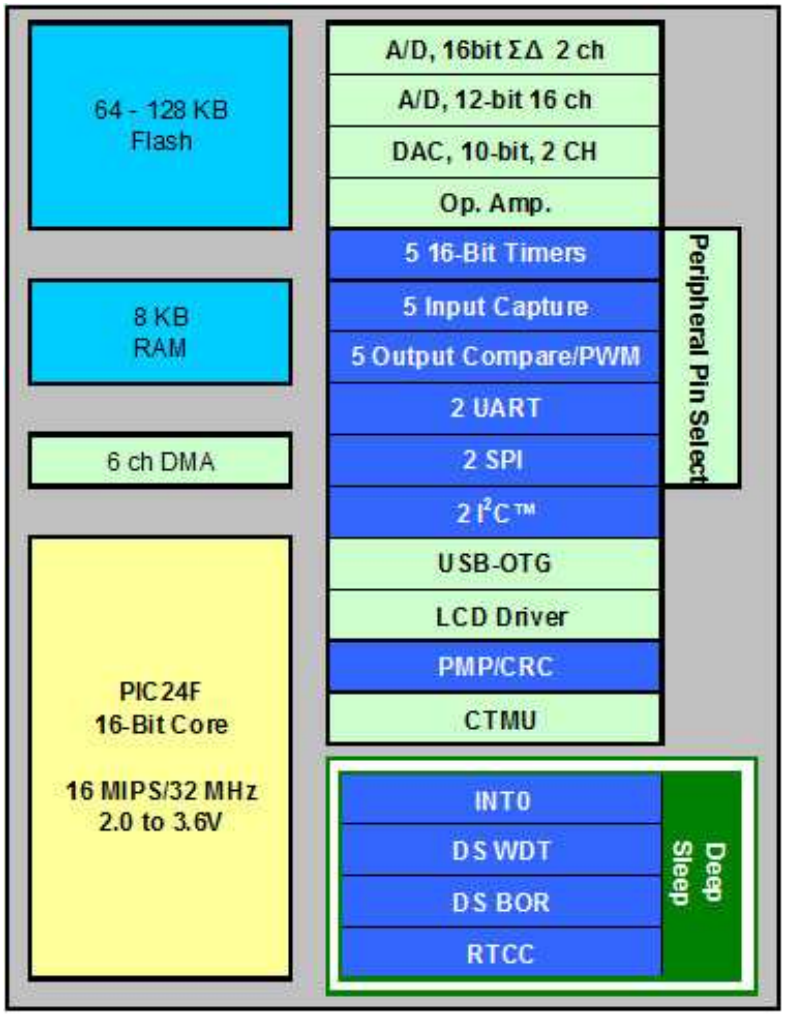
Les périphériques orientés "interface homme-machine"

- Controleur graphique pilotant directement le LCD jusqu'à 480 segments.
- mTouch : gestion des claviers capacitifs (avec slider).
- Controleur audio disponible en librarie gratuite.

Les périphériques orientés "ultra low power"

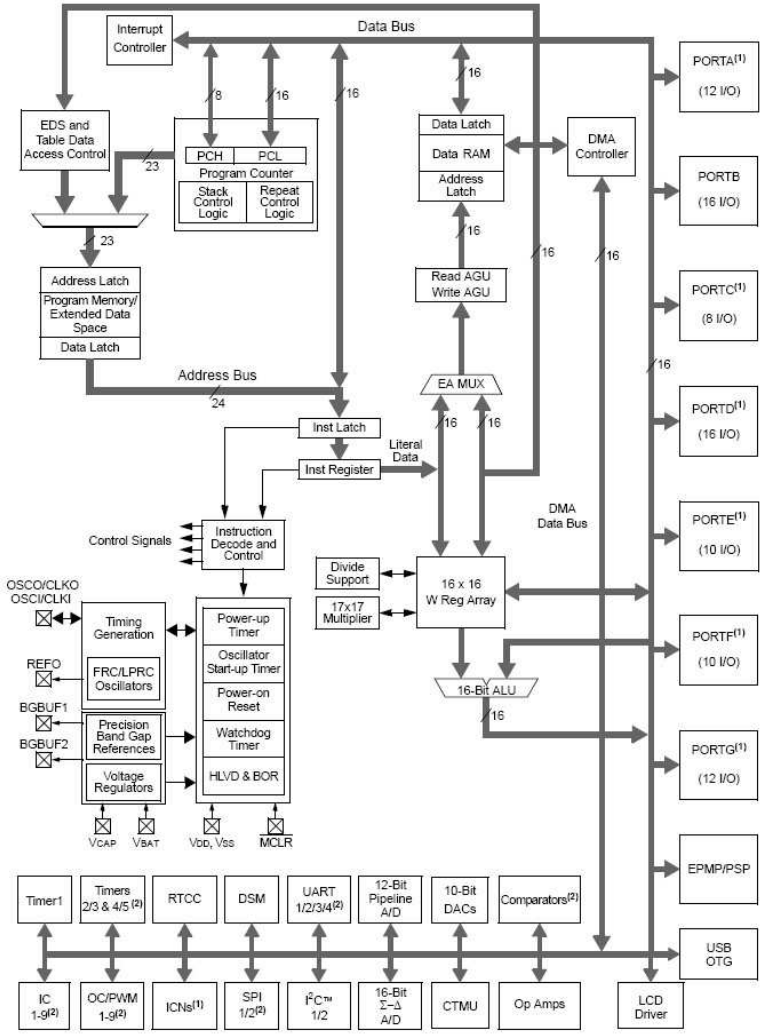
- Mode ultra low power nanoWatt XLP
- Deep Sleep : 9nA $\Rightarrow$ 280 ans sur une batterie 200mAh
- RTCC ou WDT : 200nA $\Rightarrow$ 11 ans sur une batterie 200mAh
- mode actif : à partir de 35 uA/MHz.

Un exemple de microcontrôleur 16 bits : le PIC24FJ128GC010 (μC pour objets connectés)

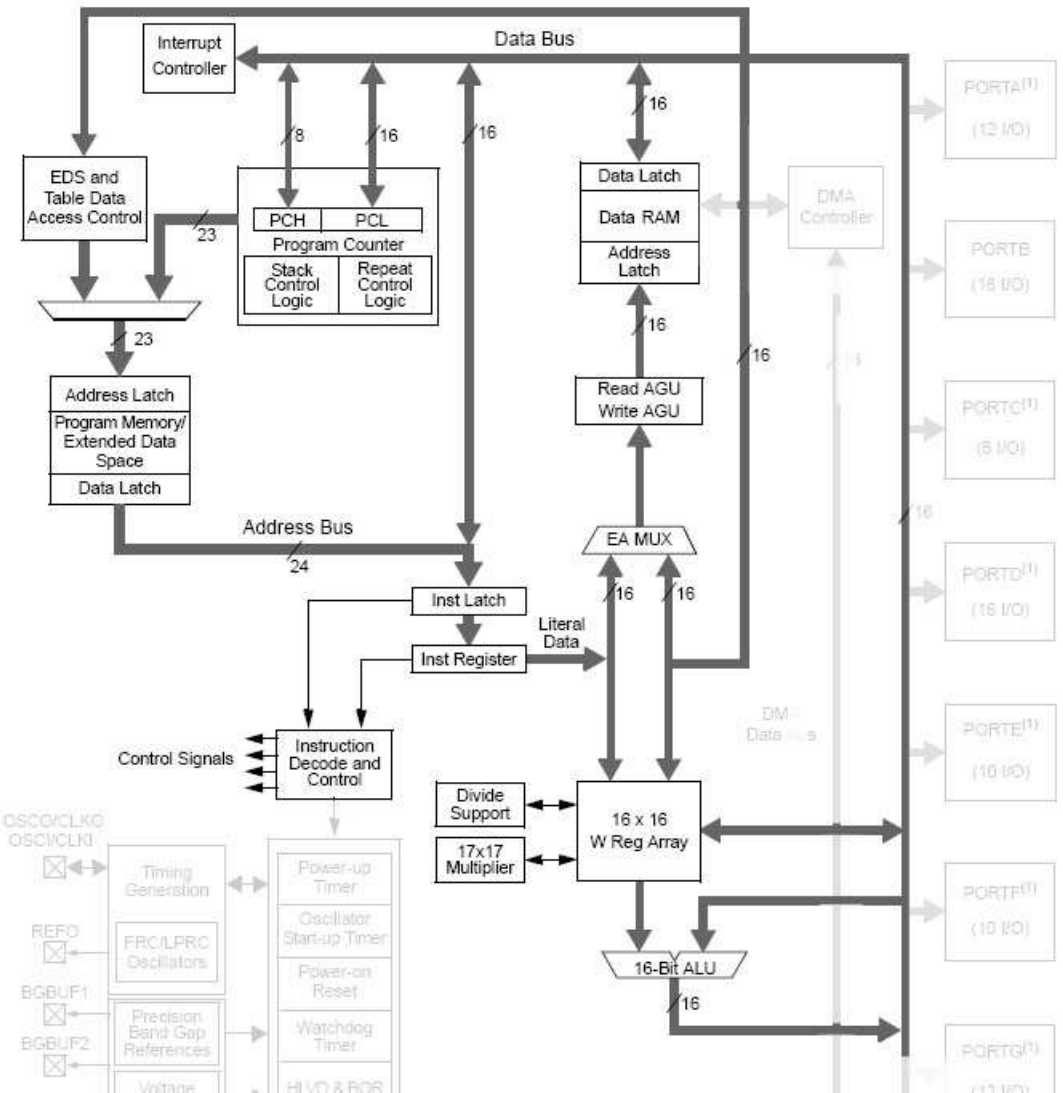


- Ultra Low Power
- Connectivité complète
  - USB on the Go
  - 2 UART
  - 2 I2C
  - 2 SPI
- DMA 6 canaux
- Conversion analogique 10M samples/sec
- RAM 8KB - Flash 128KB
- Boitier : jusqu'à 100 pins
- 3.6$ en petites séries

Un exemple : le PIC24FJ128GC010

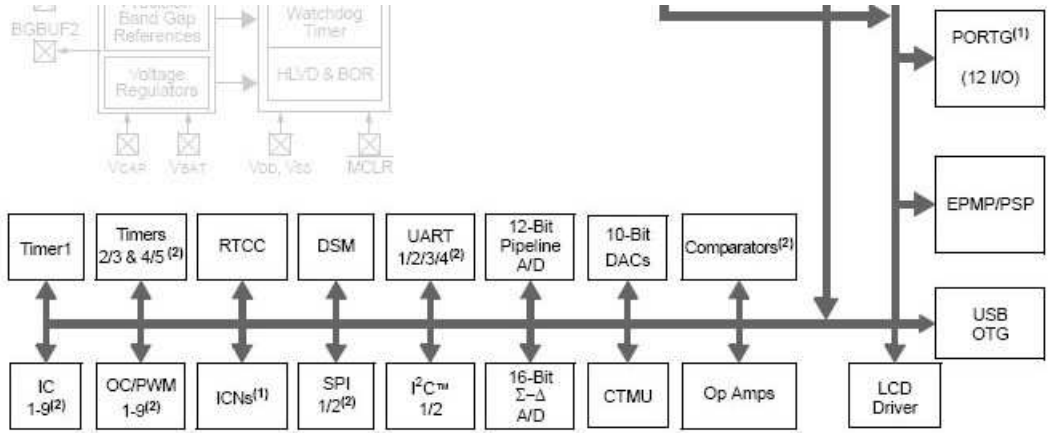


PIC24FJ128GC010 : partie coeur CPU



- Architecture 16 bits classique.
- Présence d'un multiplieur hardware 17×17

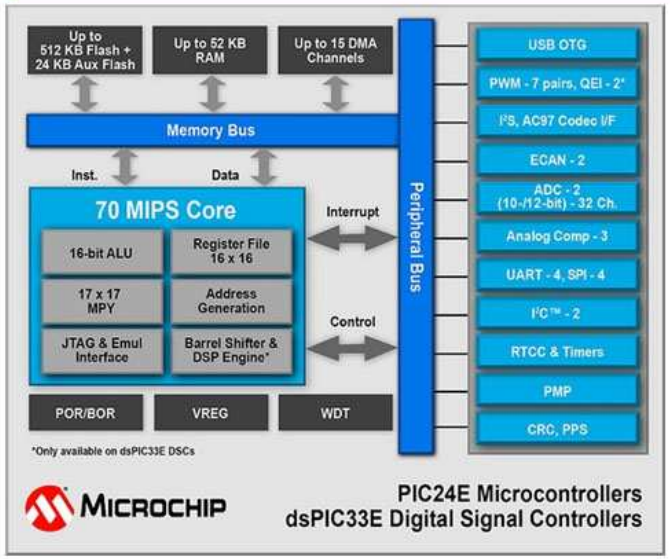
PIC24FJ128GC010 : périphériques



- Les périphériques partagent le même bus de données.
- 6 canaux DMA permettent d'accélérer certains transferts.

Un exemple de DSP 16 bits : le DSPIC33EP (*DSP* pour pilotage moteur)

dsPIC33EPXXXMU806/10/14
GP Family (64-144 Pin)



Click for more information

- Single-cycle multiply (16X16) & 32/16 and 16/16 divide
- Industrial & Extended - Temp Operation
- 64-pin QFN, 64/100/144-pin TQFP, 121-pin BGA & 144-pin LQFP
- Voltage Range : 2.7 to 3.6V

Motor Control Peripherals

- Motor Control PWM (up to 14 channels)
- Dual 32 bit, Quadrature Encoder Interface module
- Timer/Counters, up to five 16-bit timers
- Input Capture (up to 16channels)
- Output Compare (up to 16channels)
- Direct Memory Access (DMA)
- Audio Digital-to-Analog Converter (DAC)
  - 16-bit Dual Channel DAC module
  - 100 Ksps maximum sampling rate
- ADC Module
  - 10-bit, 1.1 Msps or 12-bit, 500 ksps conversion
  - Up to 9 input channels with auto-scanning
- 4 x UART w/LIN and IrDA® interfaces
- 2 x I²C™ and 4 x 4-wire SPI
- 2 x Enhanced CAN (ECAN™ module)
- USB
- Peripheral Pin Select (PPS)
- Hardware RTCC
- Parallel Master Port and 32 bit CRC

Questions ?

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