



*Dans cet exercice, nous allons faire évoluer l'application précédemment développée durant les exercices 3-4 en créant un DIGICODE capable de communiquer avec un PC. Depuis n'importe quel PC, l'opérateur sera par exemple capable de modifier le code secret ou encore le message d'accueil affiché sur l'afficheur LCD. Les communications entre le PC et le MCU seront réalisées via une liaison série asynchrone (norme RS232).*



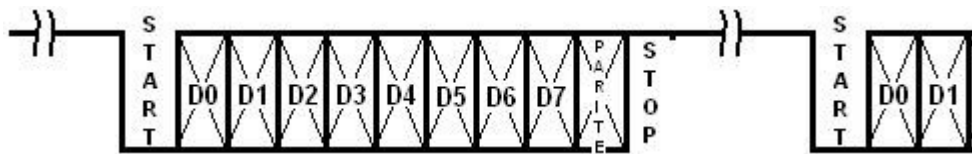
**figure 1 :** photo du DIGICODE et de l'interface série RS-232 pour une communication avec un PC (connecteur DB 9 )

La présentation de l'exercice est découpée en trois parties :

1. Présentation du protocole d'une liaison série asynchrone
2. Présentation de la norme RS-232, à ne pas confondre avec le protocole de la liaison série asynchrone
3. Présentation de l'EUSART (Enhanced Universal Synchronous Asynchronous Receiver Transmitter) du PIC18F4550. L'EUSART est un périphérique interne du MCU assurant l'interface entre une liaison série et le CPU du micro-contrôleur.

## 1. Qu'est-ce qu'une liaison série asynchrone ?

Une liaison série asynchrone est un protocole de communication série. **Il s'agit d'une "enveloppe" ajoutée aux données utiles à transférer (ajout de bits de contrôle)**. Les transferts sont de plus asynchrone, cela signifie que l'horloge n'est pas transmise avec les données (gain de fils ou de bits de synchronisation). Récepteur et émetteur doivent donc travailler chacun de leur côté avec une horloge de précision indépendante (typiquement réalisée à base de quartz). Il y a re-synchronisation du récepteur à l'arrivée de chaque bit de START. Ce protocole de communication série est relativement simple, notamment comparé au protocole USB :

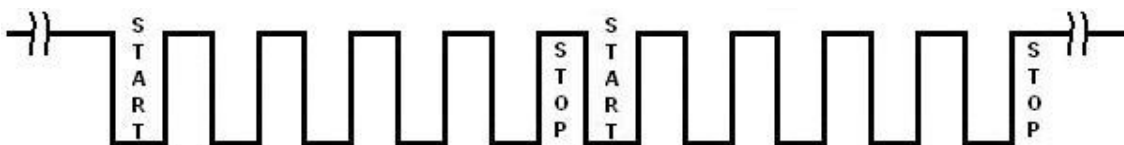


**figure 2 :** exemple de transfert asynchrone de données  
(8bits de donnée, 1bit de stop et 1 bit de parité)

- ➔ Avant d'envoyer la donnée utile, la trame commence par **UN bit de START**. Ce bit est toujours à l'état bas et permet au récepteur de se synchroniser.
- ➔ Le mot D0-D7 représente la **donnée utile à transférer (payload)**. La taille des données transférées est configurable et peut varier entre 7 et 8 bits selon l'UART. Le premier bit transmit est toujours le bit de poids faible.
- ➔ **Le bit de parité est optionnel**. Ce bit, quand il est utilisé, permet de détecter des erreurs de transmission. Par exemple, si nous travaillons en parité paire (après configuration récepteur/émetteur), à chaque transfert le nombre de **bits de donnée + parité à "1"** doit-être pair, sinon une erreur est détectée.
- ➔ **Le(s) bit(s) de stop** indique(nt) au récepteur la fin de la trame. Ce(s) bit(s) est(ont) toujours au niveau haut.



*Les différents bits de contrôle présentés ci-dessus représentent donc une "enveloppe" pour les données utiles. Les seuls bits obligatoires sont **UN bit de START** et **UN bit de STOP**, il s'agira de la configuration utilisée durant cet exercice.*



**figure 3 :** exemple de deux transferts consécutifs du caractère "U".  
Le code ASCII sur 8bits du caractère "U" est 0x55 = 0b01010101.  
(8bits de donnée, 1bit de stop et pas de bit de parité)

## 2. Qu'est-ce que la norme RS-232 ?

**RS-232 est une norme standardisant un bus de communication série asynchrone full duplex sur trois fils minimum.** Full duplex signifie que l'on peut transmettre et recevoir des données simultanément, contrairement à l'USB2.0 par exemple. Cette norme est communément rencontrée dans le monde du PC sous les appellations "port série" ou "port COM". Elle était encore rencontrée sur un grand nombre de PC jusqu'en 2000. Elle est depuis remplacée sur les marchés grand public par l'USB, proposant un protocole visant à remplacer la multitude d'anciens ports existants (ports série-COM, ports PS-2, ports parallèle-LPT, ...). **Cependant la norme RS-232 reste encore très rencontrée dans un grand nombre de domaines industriels, notamment pour sa simplicité et sa robustesse** (automates programmables, machines outils, appareillages de mesure ...).



Voie 1	DCD	Input Data Carrier Detect	Voie 6	DSR	Input Data Set Ready
Voie 2	Rx	Input Receive Data	Voie 7	RTS	Output Request To Send
Voie 3	Tx	Output Transmit Data	Voie 8	CTS	Input Clear To Send
Voie 4	DTR	Output Data Terminal Ready	Voie 9	RI	Input Ring Indicator
Voie 5	Ground	Masse électrique			

**figure 4 :** connecteur DB-9 mâle. Nous n'utiliserons que trois fils : Rx, Tx et la masse

Contrairement au protocole d'une liaison série asynchrone, **la norme RS-232 spécifie des contraintes électriques, mécaniques, physiques ...** Par exemple, la norme définit la longueur maximale du câble en fonction du débit utilisé :

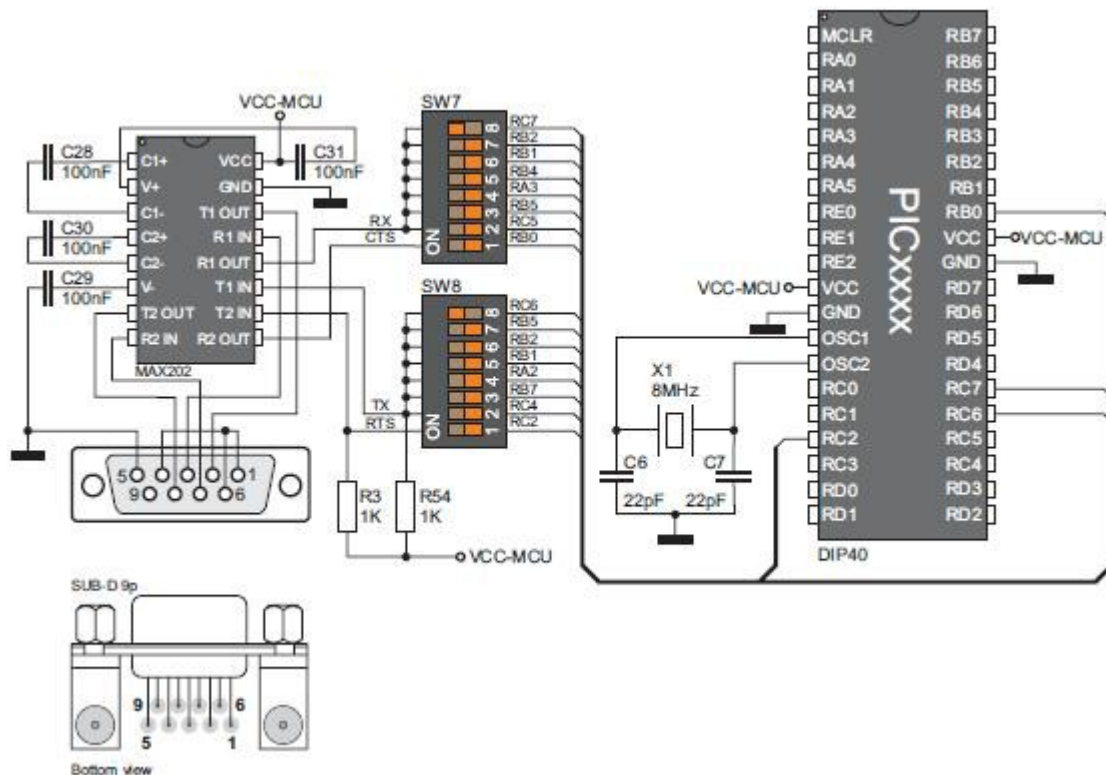
- |                      |       |                    |      |
|----------------------|-------|--------------------|------|
| ➔ <b>115,2Kbps :</b> | ~4m   | ➔ <b>9600bps :</b> | ~50m |
| ➔ <b>1200bps :</b>   | ~400m | ➔ ...              |      |

La norme RS-232 définit également les niveaux de tension sur les lignes :



**ATTENTION, la logique est inversée !**

Sur notre maquette, en sortie de l'EUSART du PIC18F4550, le composant assurant le passage entre le protocole de la liaison série asynchrone et la norme RS-232 est un MAX202 de chez Texas Instrument (cf. **figure 1** et **figure 5**).



**figure 5 :** câblage du MAX202 et de l'EUSART du PIC18F4550 (broches Tx=RC6 et Rx=RC7)

### 3. Comment utiliser l'EUSART du PIC18F4550 ?

**L'EUSART (Enhanced Universal Synchronous Asynchronous Receiver Transmitter) est un périphérique interne du MCU** assurant l'interface entre une liaison série synchrone ou asynchrone et le CPU de notre micro-contrôleur. Comme tout périphérique, nous allons devoir le configurer en découvrant au fil de la présentation les différents registres qui lui sont associés. Mais avant de passer à la suite il faut bien avoir assimilé ce qui suit :



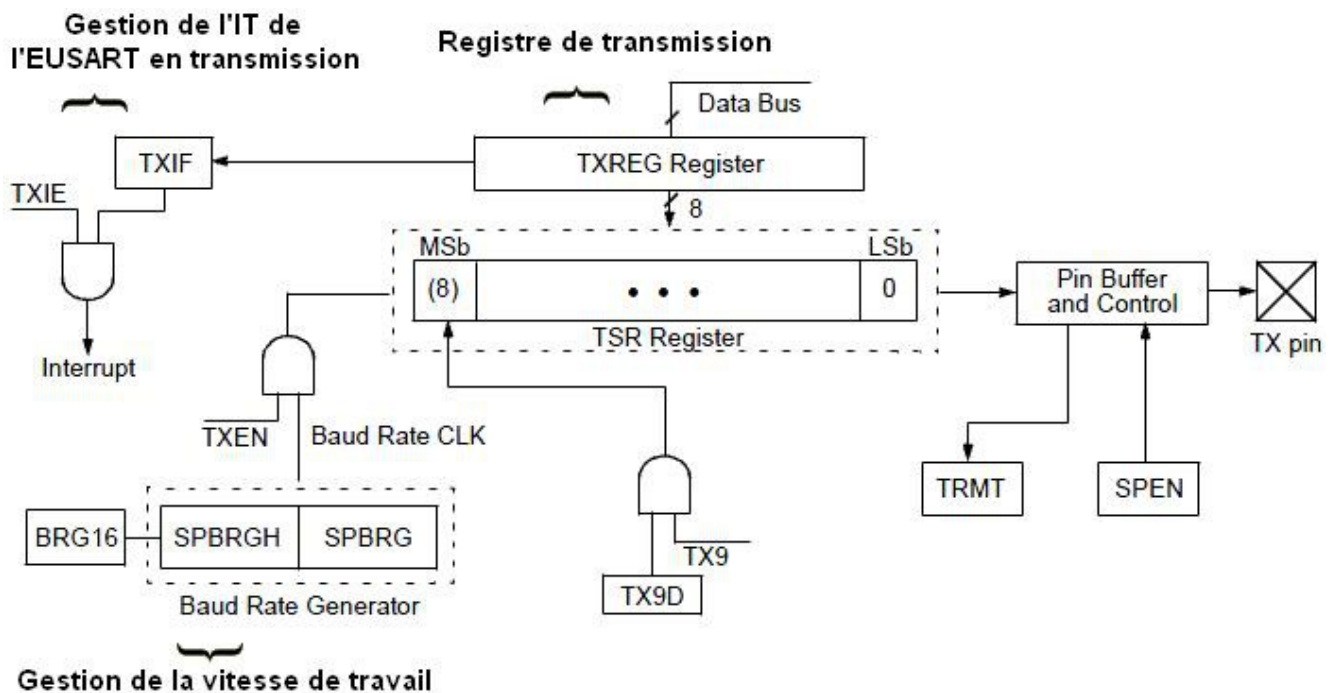
 De chaque côté du câble se trouve une UART (ou EUSART pour le PIC18F4550). Ces deux périphériques (côté PC et côté MCU) doivent absolument posséder la même configuration:

- ➡ Même vitesse de travail
- ➡ Même nombre de bits de donnée
- ➡ Même nombre de bits de stop
- ➡ Même convention pour un éventuel bit de parité



Afin de configurer l'EUSART, nous allons devoir nous intéresser aux sept registres qui lui sont associés. Pour bien assimiler le fonctionnement de ces registres, il faut découper leur étude en 3 parties : la **transmission (TXSTA, TXREG)**, la **réception (RCSTA, RCREG)** et la sélection de la **vitesse de travail (BAUDCON, SPBRGH, SPBRG)**.

### ➡ Commençons par étudier la chaîne de transmission (TXSTA, TXREG) !



**figure 6 :** schéma fonctionnel de la chaîne de transmission de l'EUSART



*En transmission, nous allons chercher à envoyer des données en série sur la broche Tx. La donnée utile à transférer devra être écrite dans le registre 8bits **TXREG**. L'EUSART se chargera ensuite de rajouter les bits de contrôle selon la configuration choisie (start, stop ...) puis d'envoyer les données en série sur la broche Tx.*



*TXSTA est à la fois un registre de contrôle et un registre d'état. Il nous permet donc à la fois de configurer l'EUSART et de connaître son état. Le seul bit d'état de TXSTA est TRMT !*

### TXSTA: TRANSMIT STATUS AND CONTROL REGISTER

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-1	R/W-0
CSRC	TX9	TXEN	SYNC	SENDB	BRGH	TRMT	TX9D
bit 7							bit 0

bit 7 **CSRC**: Clock Source Select bit

Asynchronous mode:

Don't care.

} Pour un fonctionnement  
en mode synchrone

Synchronous mode:

1 = Master mode (clock generated internally from BRG)

0 = Slave mode (clock from external source)

bit 6 **TX9**: 9-bit Transmit Enable bit

1 = Selects 9-bit transmission

0 = Selects 8-bit transmission

} Avec ou sans 9eme bit de donnée (bit  
de parité par exemple)

bit 5 **TXEN**: Transmit Enable bit

1 = Transmit enabled

0 = Transmit disabled

} Validation de l'EUSART en  
transmission

bit 4 **SYNC**: EUSART Mode Select bit

1 = Synchronous mode

0 = Asynchronous mode

} Sélection du mode synchrone ou  
asynchrone

bit 3 **SENDB**: Send Break Character bit

Asynchronous mode:

1 = Send Sync Break on next transmission (cleared by hardware upon completion)

0 = Sync Break transmission completed

Synchronous mode:

Don't care.

} Envoie d'un break. Non utilisé dans  
notre application

bit 2 **BRGH**: High Baud Rate Select bit

Asynchronous mode:

1 = High speed

0 = Low speed

Synchronous mode:

Unused in this mode.

} Gestion du mode "vitesse haute" ou  
"vitesse basse" pour la détermination  
de la vitesse de travail

bit 1 **TRMT**: Transmit Shift Register Status bit

1 = TSR empty

0 = TSR full

} Flag spécifiant si le registre de  
transmission TSR (cf. figure 6)  
est plein ou vide

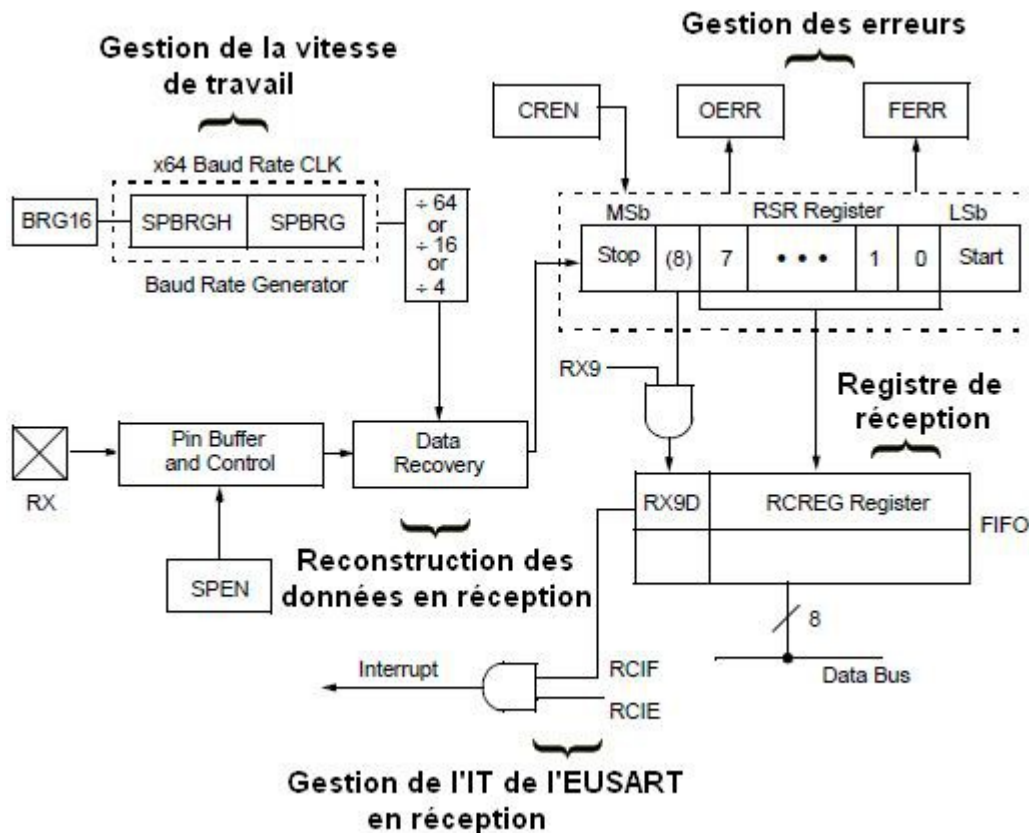
bit 0 **TX9D**: Ninth bit of Transmit Data

Can be address/data bit or a parity bit.

} Valeur du 9eme bit, si il est utilisé

Prenons l'exemple de notre application. Nous souhaitons travailler en mode asynchrone, sans bit de parité, en vitesse haute, **TXSTA=0x24** peut alors être une valeur possible. Nous remarquerons que les valeurs des bits TX9D et CSRC n'ont pas d'influences sur la configuration.

➡ Chaîne de réception (RCSTA, RCREG) :



**figure 7 :** schéma fonctionnel de la chaîne de réception de l'EUSART



*A chaque fois qu'une donnée sera reçue par l'EUSART, elle sera sauvée dans le registre de réception **RCREG** et une demande d'IT sera envoyée au CPU. Il nous suffira d'aller lire ce registre pour récupérer la donnée utile. Le travail de l'EUSART est donc de reconnaître la totalité des bits reçus puis d'extraire les bits de donnée des bits de contrôle (cf. **figure 7**).*

Comme TXSTA, le registre **RCSTA** est à la fois une registre de contrôle et d'état. Les trois bits d'état sont RX9D, FERR et OERR. Les deux derniers permettent de détecter des erreurs en réception (cf. **figure 7**). Nous ne détaillerons pas ce registre et nous vous laissons le soin de le configurer. Seul le bit SPEN (cf. **figure 6** et **figure 7**) est présenté. Il permet d'utiliser les broches TX et RX (RC6 et RC7) comme étant des broches de l'EUSART.

**RCSTA: RECEIVE STATUS AND CONTROL REGISTER**

R/W-0	R/W-0	R/W-0	R/W-0	R/W-0	R-0	R-0	R-x
SPEN	RX9	SREN	CREN	ADDEN	FERR	OERR	RX9D
bit 7							bit 0

bit 7 **SPEN:** Serial Port Enable bit

1 = Serial port enabled (configures RX/DT and TX/CK pins as serial port pins)

0 = Serial port disabled (held in Reset)

:

:



### ➡ Configuration de la vitesse de travail (BAUDCON, SPBRGH, SPBRG) :

Nous allons chercher à comprendre comment fixer la vitesse de travail en Baud de l'EUSART. Un baud correspond à un symbole par seconde, qui correspond à un bit par second dans notre cas. **Pour générer la vitesse de travail en Baud, l'EUSART utilise son propre timer interne. La valeur de pré-chargement utilisée par ce timer est celle comprise dans le(s) registre(s) SPBRG (et SPBRGH).** Comme pour le Timer0, nous pouvons sélectionner un mode 8bits ou 16bits. La sélection de ce mode se fait dans le registre BAUDCON. En mode 8bits, le timer de l'EUSART n'utilisera que le registre SPBRG. En mode 16bits, il utilisera les deux registres SPBRG et SPBRGH. En ce qui concerne le registre BAUDCON, seul le bit BRG16 nous intéresse. Les autres bits pourront être laissés à leurs états par défaut.

#### BAUDCON: BAUD RATE CONTROL REGISTER

R/W-0	R-1	U-0	R/W-0	R/W-0	U-0	R/W-0	R/W-0
ABDOVF	RCIDL	—	SCKP	BRG16	—	WUE	ABDEN
bit 7							bit 0

:

bit 3

**BRG16:** 16-bit Baud Rate Register Enable bit

1 = 16-bit Baud Rate Generator – SPBRGH and SPBRG

:

0 = 8-bit Baud Rate Generator – SPBRG only (Compatible mode), SPBRGH value ignored



*Dans le cadre de notre application, nous devons donc travailler en mode asynchrone (bit SYNC de TXSTA), en vitesse haute (bit BRGH de TXSTA) et en mode 16bits (bit BRG16 de BAUDCON). Sachant cela, il nous suffit maintenant d'appliquer une formule proposée dans la documentation technique de PIC (cf. **figure 8**). Dans ces formules, 'n' représente la valeur à charger dans les registres SPBRGH et SPBRG.*

Configuration Bits			BRG/EUSART Mode	Baud Rate Formula
SYNC	BRG16	BRGH		
0	0	0	8-bit/Asynchronous	$F_{osc}/[64 (n + 1)]$
0	0	1	8-bit/Asynchronous	$F_{osc}/[16 (n + 1)]$
0	1	0	16-bit/Asynchronous	
0	1	1	16-bit/Asynchronous	$F_{osc}/[4 (n + 1)]$
1	0	x	8-bit/Synchronous	
1	1	x	16-bit/Synchronous	

**Legend:** x = Don't care, n = value of SPBRGH:SPBRG register pair

**figure 8 :** formules permettant de calculer le BAUD RATE



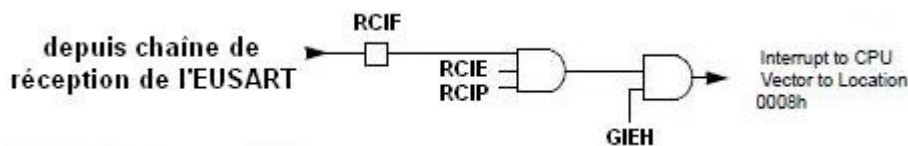
*Par exemple, si nous souhaitons travailler à 9600Bauds, la valeur à charger dans les registres SPBRGH:SPBRG vaut 1249=0x04E1. On aura donc SPBRGH=0x04 et SPBRG=0xE1.*

$$n = ((F_{OSC}/\text{Baud Rate})/4) - 1 \quad \text{avec : } F_{OSC}=48\text{MHz} \quad \text{Baud Rate} = 9600\text{Bauds}$$



## 4. Gestion de l'IT de l'EUSART en réception

Dans le cadre de notre application, nous n'allons utiliser que l'IT de l'EUSART en réception. **En effet, l'EUSART est capable d'envoyer deux demandes d'interruption distinctes au CPU.** Une en réception lorsqu'une donnée vient d'arriver dans le registre de réception RCREG (cf. **figure 7**). Une seconde en transmission lorsque le registre de transmission TSR (cf. **figure 6**) vient de se vider et que l'EUSART est prête à transmettre une nouvelle donnée.



**figure 9 :** mécanismes de démasquage de interruption de l'EUSART en réception pour le vecteur de haute priorité



*Les bits RCIE, RCIF et RCIP se trouvent respectivement dans les registres **PIE1**, **PIR1** et **IPR1**. Attention, la remise à '0' du flag RCIF se fait par une lecture du registre RCREG (cf. ci-dessous). Il vous faudra donc obligatoirement lire le registre RCREG dans le programme d'IT lié à l'EUSART en réception.*

### PIR1: PERIPHERAL INTERRUPT REQUEST (FLAG) REGISTER 1

R/W-0	R/W-0	R-0	R-0	R/W-0	R/W-0	R/W-0	R/W-0
SPPIF <sup>(1)</sup>	ADIF	RCIF	TXIF	SSPIF	CCP1IF	TMR2IF	TMR1IF
bit 7							bit 0

:

bit 5 **RCIF:** EUSART Receive Interrupt Flag bit

1 = The EUSART receive buffer, RCREG, is full (cleared when RCREG is read)  
0 = The EUSART receive buffer is empty

bit 4 **TXIF:** EUSART Transmit Interrupt Flag bit

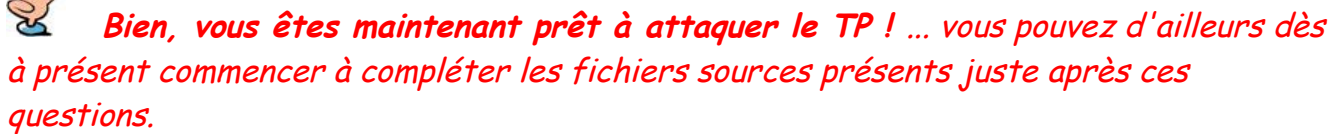
1 = The EUSART transmit buffer, TXREG, is empty (cleared when TXREG is written)  
0 = The EUSART transmit buffer is full

:

En ce qui concerne la transmission de données, nous utiliserons la technique du polling-scrutation qui consiste à scruter dans le "main()" le passage au niveau haut d'un flag (bit **TRMT** du registre **TXSTA**). Tant que ce bit est au niveau logique bas, l'EUSART n'est pas prête à transmettre une nouvelle donnée.



*Vous disposez maintenant de tous les éléments pour mener à bien cet exercice !*



1. (★) La norme RS-232 travail en FULL DUPLEX . De cela signifie-t-il ?

3. (★) Quel est le rôle du composant MAX202 présent sur la maquette ?

4. (★) Proposez une configuration pour les registres TXSTA, RCSTA, SPBRGH et SPBRG, sachant que nous souhaitons travailler à 115,2KBauds et que nous utiliserons les chaînes de réception et de transmission de l'EUSART. Aidez vous de la datasheet du PIC donnée en ANNEXE.

5. (★) Sachant que nous ne souhaitons utiliser que l'IT de l'EUSART en réception et que nous utiliserons le vecteur d'IT de priorité haute, proposée une configuration pour les registres PIE1, PIR1 et IPR1. Aidez-vous de la datasheet du PIC donnée en ANNEXE.



## Listing du programme *ex5a.c*

```

/*****
@file : ex5a.c
@brief : Communications PC via EUSART
@author :
last modification :
*****/

/**** Configuration bits - TCY = 83,2ns ****/
#pragma config PLLDIV=2, CPUDIV=OSC1_PLL2, FOSC=HSPLL_HS, BOR=OFF, WDT=OFF, MCLRE=ON

/**** Includes files ****/
#include <p18f4550.h>

/**** Définition des Macros ****/
#define          BUFFER_SIZE          40          /**** Taille du buffer circulaire ****/

/**** Déclaration des variables globales - Communications PC - EUSART ****/
volatile unsigned char indexWrite=0, indexRead=0, EUSARTbuffer[BUFFER_SIZE], flagCaracRecu = 0;
#pragma romdata
rom const unsigned char Menu_Message[] = "Entrez une commande a en .... \n\r";
rom const unsigned char Reset_Message[] = "Reset du Micro-contrôleur effectué !";
rom const unsigned char Error_Message[] = "Erreur de saisie - Il ne s'agit pas d'une commande !";
rom const unsigned char Error_Size_Code_Message[] = "Erreur de saisie - Entrez un code à 4 chiffres !";
rom const unsigned char Error_Size_LCD_Message[] = "Erreur de saisie - Entrez un ... aximum !";
rom const unsigned char Saisie_Message[] = "Saisie : ";
rom const unsigned char Choix_Message[] = " Choix : ";
rom const unsigned char crx2[] = "\n\r\n\r";
rom const unsigned char cr[] = "\n\r";

#pragma code
/****
/**** ISR : routine d'interruption de priorité haute ****/
/**** ATTENTION : l'EUSART envoie 2 demandes d'IT distinctes ****/
/****
#pragma interrupt high_isr
void high_isr(void)
{
    /**** à compléter ! ****/
}
/**** Configuration du vecteur d'interruption de priorité haute ****/
#pragma code high_vector = 0x08
void interrupt_at_high_vector(void)
{
    _asm goto high_isr _endasm
}

#pragma code
/****
/**** FONCTION : Configuration de l'EUSART et de l'IT associée ****/
/****
void EUSART_Init(void)
{

```

```
    /*** Initialisation de la valeur de division pour le BAUD RATE - 115,2KB    ***/
    /*** SYNC=0 (asynchrone) et BRGH=1 (high speed)    ... etc    ***/

    /*** à compléter ! ***/

    /*** Configuration du registre BAUDCON, SPBRG et SPBRGH pour la gestion du BAUD RATE    ***/
    /*** BRG16=1 (16bits mode) et BAUD RATE - 115,2KB avec : Fosc=48MHz    ***/
    /***

    /*** à compléter ! ***/

    /*** Configuration de l'interruption associée à l'USART en réception (démasquage) ***/

    /*** à compléter ! ***/
}

/*****
/*** FONCTION : Lecture d'un caractère entré depuis le PC    ***/
/*** @return keycode : code du caractère reçu    ***/
*****/
unsigned char EUSART_Read_Char_User(void){

    /*** à compléter ! ***/

    return 0;
}

/*****
/*** FONCTION : Lecture d'une chaîne de caractères entrée depuis le PC    ***/
/*** @param ptString : pointeur est en data memory    ***/
*****/
void EUSART_ReadRAM_String_User(unsigned char *ptString){

    /*** à compléter ! ***/

}

/*****
/*** FONCTION : Lecture d'une chaîne de caractères entrée depuis le PC    ***/
/*** @param ptString : pointeur est en program memory    ***/
*****/
void EUSART_ReadROM_String_User(rom unsigned char *ptString){

    /*** à compléter ! ***/

}

/*****
/*** FONCTION : Envoie d'un caractère vers le PC    ***/
/*** @param keyCode : caractère à envoyer    ***/
*****/
void EUSART_Write_Char_User(unsigned char keyCode){

    /*** à compléter ! ***/

}
```



```

/*****
*** FONCTION : Envoie d'une chaîne de caractères vers le PC ***
*** @param   ptString : pointeur est en data memory ***
*****/
void EUSART_WriteRAM_String_User(unsigned char *ptString){

    /*** à compléter ! ***/
}

/*****
*** FONCTION : Envoie d'une chaîne de caractères vers le PC ***
*** @param   ptString : pointeur est en program memory ***
*****/
void EUSART_WriteROM_String_User(rom unsigned char *ptString){

    /*** à compléter ! ***/
}

/*****
*** PROGRAMME PRINCIPAL ***
*****/
void main() {
    /*** variables locales pour la communication PC ***/
    unsigned char  keyCode;

    CMCON      |=    0x07;    // turn off comparators
    ADCON1     |=    0x0F;    // turn off analog inputs

    /*** Configuration de l'EUSART et de l'IT associée ***/

    /*** à compléter ! ***/

    /*** Démasquage global des IT ***/

    /*** à compléter ! ***/

    while(1){
        /*** Attente réception d'un caractère depuis le PC ***/
        if( ??? ) {

            /*** Lecture du caractère arrivé ***/

            /*** à compléter ! ***/

            /*** Envoie du caractère + saut de ligne au PC ***/



            /*** à compléter ! ***/

        }
    }
}
```

**Notez vos remarques :**

## 1. Travail pratique (ex5 : partie 1) : Envoi de caractères vers le PC

Cette première partie de l'exercice consiste respectivement à écrire les deux fonctions assurant l'envoi de caractères au PC, "**EUSART\_Write\_Char\_User**" puis "**EUSART\_Write\_String\_User**". Il est pour le moment inutile d'utiliser l'IT de l'EUSART en réception.

- ➡  *Copiez et complétez le listing du programme **ex5a.c**.*
- ➡  *Créez un projet **ex5** dans votre répertoire de travail (cf. **ANNEXE 1**)*
- ➡ Configurez l'EUSART
- ➡ Écrivez la fonction **EUSART\_Write\_Char\_User** permettant d'envoyer un caractère au PC
- ➡ Ouvrez la connexion **TP1A\_ex5.ht** de l'hyper terminal puis validez votre programme (cf. **ANNEXE 8**)
- ➡ Écrivez la fonction **EUSART\_WriteRAM\_String\_User** permettant d'envoyer une chaîne de caractères au PC. La chaîne de caractères sera en RAM (data memory).
- ➡ Écrivez la fonction **EUSART\_WriteROM\_String\_User** permettant d'envoyer une chaîne de caractères au PC. La chaîne de caractères sera en ROM (program memory).

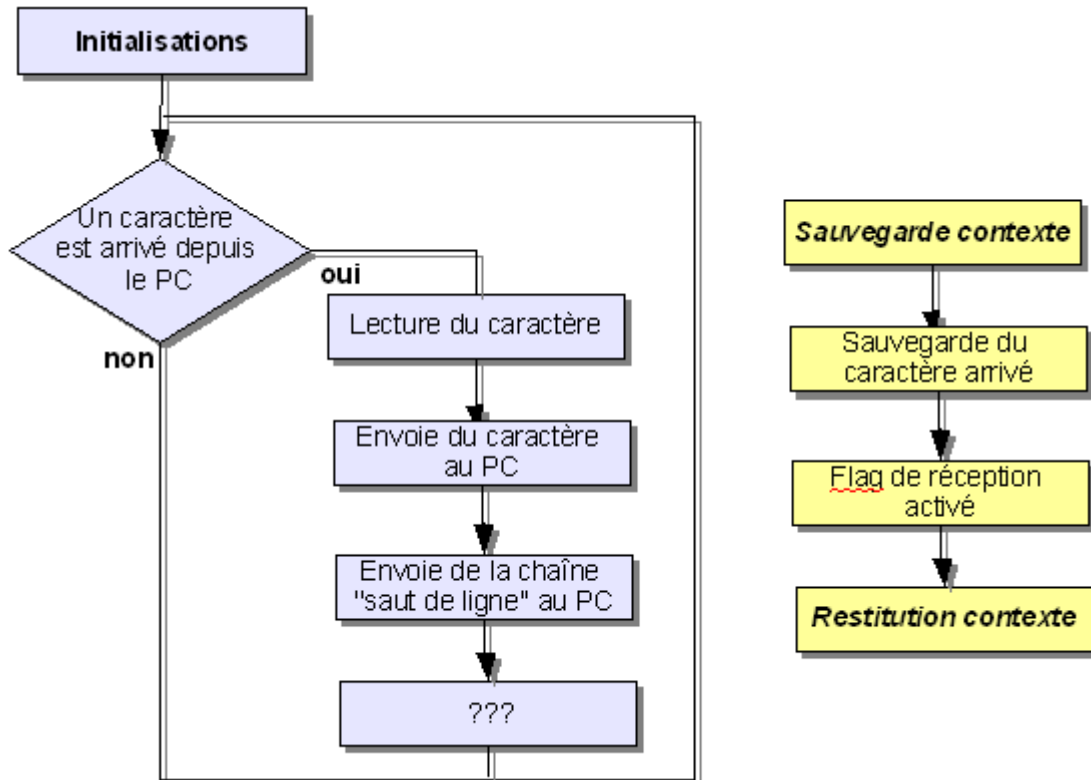
## 2. Travail pratique (ex5 : partie 2) : Réception de caractères

Dans cette seconde partie de l'exercice, nous allons effectuer la réception de caractères envoyés depuis le PC. A chaque fois qu'un caractère est reçu par le MCU, l'EUSART envoie une demande d'interruption au CPU. Vous allez donc devoir configurer l'IT de l'EUSART en réception et écrire le programme d'IT correspondant. N'oubliez pas de lire le registre RCREG dans la routine d'interruption. A chaque appel du programme d'IT, il vous faudra mettre un flag à "1". La valeur de ce flag sera ensuite constamment testée dans le "main()", technique du polling-scrutation. Si il passe à "1", alors un caractère vient d'arriver. Il vous faudra le récupérer puis le renvoyer au PC. Dans un premier temps vous n'aurez qu'à écrire la fonction "**EUSART\_Read\_Char\_User**".

- ➡ Configurez les IT liées à l'EUSART (vecteur de priorité haute)
- ➡ Complétez le programme d'IT de priorité haute
- ➡ Écrivez la fonction **EUSART\_Read\_Char\_User** permettant de lire un caractère envoyé depuis le PC
- ➡ Complétez le main() de façon à renvoyer au PC tout caractère reçu. Validez le fonctionnement de votre programme
- ➡ Envoyez le contenu d'un fichier texte à la maquette (sous l'**Hyper terminal >> Transfert >> envoyer un fichier texte ... >> TP1A\_ex5.txt** présent dans votre répertoire de travail) ... Que constatez-vous ?



*Aidez-vous de l'organigramme **figure 10** présentant le main() et le programme d'IT !*



**figure 10 :** organigramme présentant le main() et le programme d'IT

### 3. Travail pratique (ex5 : partie 3) : Buffer circulaire

Avec la technique précédemment utilisée, nous étions amenés dans certains cas à perdre des caractères. **Ces caractères étaient perdus car ils arrivaient plus vite qu'on ne pouvait les traiter.** Afin d'éviter ce problème, nous allons mettre en place un buffer circulaire. A chaque fois qu'un caractère arrive, nous allons directement dans le programme d'IT le sauver temporairement dans un tableau (buffer circulaire). Ce tableau est utilisé avec deux curseurs-index, un en écriture et l'autre en lecture permettant respectivement de spécifier où on écrit et où on lit dans le buffer circulaire. A chaque arrivée de caractère on incrémentera le curseur d'écriture et a chaque lecture de caractère depuis le buffer circulaire on incrémentera le curseur de lecture. Tant que les deux curseurs-index ont une valeur différente, un ou des caractères se trouvent dans le buffer circulaire. Dès qu'un curseur arrive à la taille limite du buffer, il repasse à zéro. D'où le nom de buffer circulaire. La taille du buffer dépend de l'application.

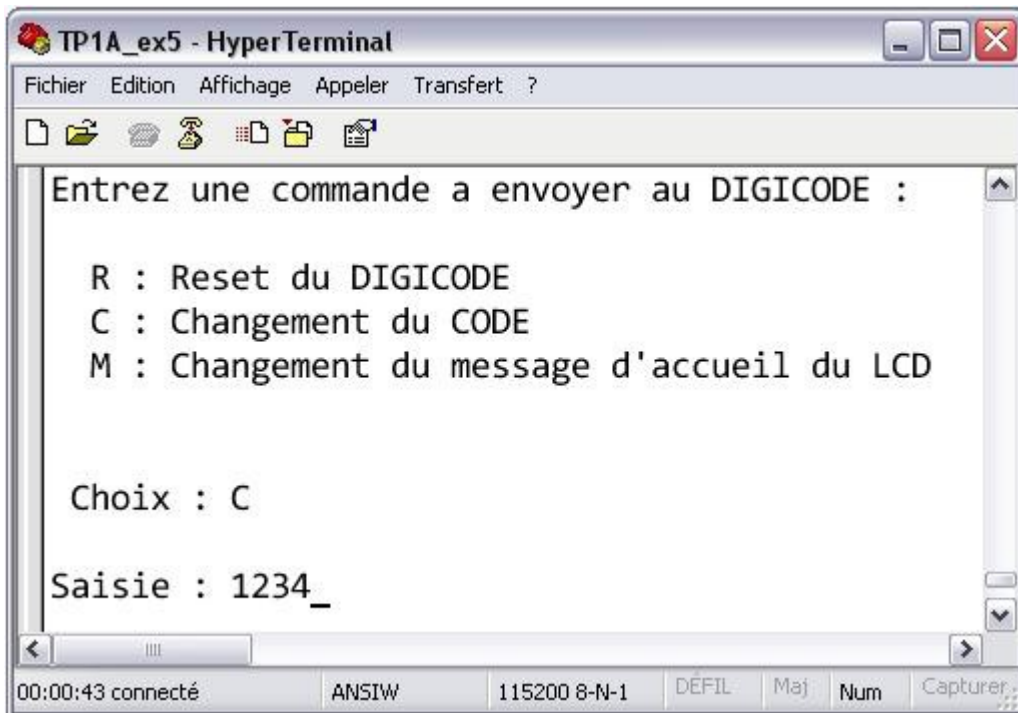
- ➔ Modifiez la fonction **EUSART\_Read\_Char\_User** ainsi que le programme d'IT de façon à implémenter le buffer circulaire
- ➔ Envoyez le contenu d'un fichier texte à la maquette (**Hyper terminal >> Transfert >> envoyer un fichier texte ... >> TP1A\_ex5.txt**) ... Que constatez-vous ?
- ➔ Écrivez la fonction **EUSART\_ReadRAM\_String\_User** permettant de lire une chaîne de caractères envoyée depuis le PC et de la stocker en mémoire donnée.



#### 4. Travail pratique (*ex5 : partie 4*) : commande depuis un PC du DIGICODE

➔  *Créer en partant de rien un nouveau projet **digicode** dont le fichier principal sera nommé **main.c**.*

Il ne vous reste maintenant plus qu'à réaliser un DIGICODE communicant. Contrairement à ce que l'on pourrait croire, il s'agit de la partie la plus simple. L'opérateur pourra par exemple changer le code secret à partir d'un PC quelconque équipé d'un port série. Afin de mieux comprendre le cahier des charges, aidez-vous du programme de démonstration **ex5b.hex**. Vous trouverez ci-dessous un exemple de capture d'écran effectuée dans le cadre de cet exercice.



```
TP1A_ex5 - HyperTerminal
Fichier  Edition  Affichage  Appeler  Transfert  ?
[Icons]
Entrez une commande a envoyer au DIGICODE :

R : Reset du DIGICODE
C : Changement du CODE
M : Changement du message d'accueil du LCD

Choix : C
Saisie : 1234_

00:00:43 connecté  ANSIW  115200 8-N-1  DEFIL  Maj  Num  Capturer...
```

Commencez par exemple par implémenter les actions correspondant aux commandes "C" et "M". Pour vous simplifier la tâche, utilisez les fonctions **strlen()** et **atoi()** accessibles en ajoutant les fichiers d'en-tête **stdlib.h** et **string.h**. La première fonction vous retourne la taille d'une chaîne de caractères et la seconde convertit une chaîne de caractères en entiers.



*Par exemple, la fonction **atoi()** est très pratique pour convertir en entier le nouveau code secret saisi par l'opérateur depuis le PC !*

Pour effectuer un Reset logiciel, rien de plus simple. Il vous suffit d'écrire "**\_asm goto \_startup \_endasm**" qui correspond à l'exécution de l'instruction assembleur goto depuis un fichier "C". Ceci amène quelques interrogations. Par exemple, pourquoi ne pas faire un "**goto main**" ? Et à quoi correspond l'étiquette "**\_startup**" ?



Il faut savoir qu'un programme en "C" ou en n'importe quel autre langage évolué (pascal, fortran, C++ ...) n'est jamais exécuter sans une phase d'initialisation. Par exemple, lorsque vous initialisez des variables globales dans un programme "C", c'est le fichier de **startup (c018i.c pour MPLAB)** qui se charge de leurs initialisations effectives en mémoire. Le code de ce programme doit être très performant en terme de vitesse d'exécution et de ressource mémoire. Il est donc très souvent développé directement en assembleur. Ce fichier n'existe pas lorsqu'on développe la totalité d'une application en assembleur. C'est alors au programmeur d'initialiser les variables dans son programme. Ce fichier de Startup se finit quant à lui par l'instruction "**goto main**".



*A vous de jouer maintenant !*

## **5.Travail pratique (ex5 : partie 5) :** librairie pour la gestion de périphérique de C18

Vous venez de conclure votre premier "gros" projet autour du domaine de l'embarqué. En une quinzaine d'heures et avec un peu d'aide vous avez pu développer un digicode communiquant et reconfigurable à distance depuis un PC. Seulement, contrairement à une démarche équivalente qui aurait pu être faite en entreprise, nous vous avons menti sur un point !!!

Dans le monde de l'embarqué, tout fabricant de MCU ou DSP fourni avec ses composants un compilateur (gratuit ou payant) ainsi que des librairies C pour piloter les périphériques internes voir certains externes. Nous allons donc vous demander de reconfigurer le timer0 en utilisant cette librairie de C18 ... et vous allez vous rendre compte qu'avec vos compétences, ce travail est maintenant enfantin !!

- Ouvrez la librairie (raccourci sur le bureau), sélectionnez notre MCU et ouvrez la documentation relative à la configuration des Timer.
- Masquez l'ancienne configuration du timer0 puis reconfigurez-le en utilisant les fonctions suivantes :

```
OpenTimer0( ???);  
writeTimer0( ??? );  
INTCON2bits.TMR0IP = 0;
```



*Maintenant, après avoir réalisé la trame de TP, vous pouvez avoir une idée très concrète du code caché derrière ces fonctions qui pourraient sembler magique au premier abord !!!*

- Pour les plus courageux, configurez également l'UART :

```
OpenUSART( ??? );  
baudUSART( ??? );
```



*Dans un contexte industriel, du moment que l'on sait ce que l'on fait, il ne faut pas hésiter à utiliser ces librairies qui sont ''le plus souvent'' assez robuste !!!*

