Projet
conception d'un robot mobile

Professeur : Valentin Gies

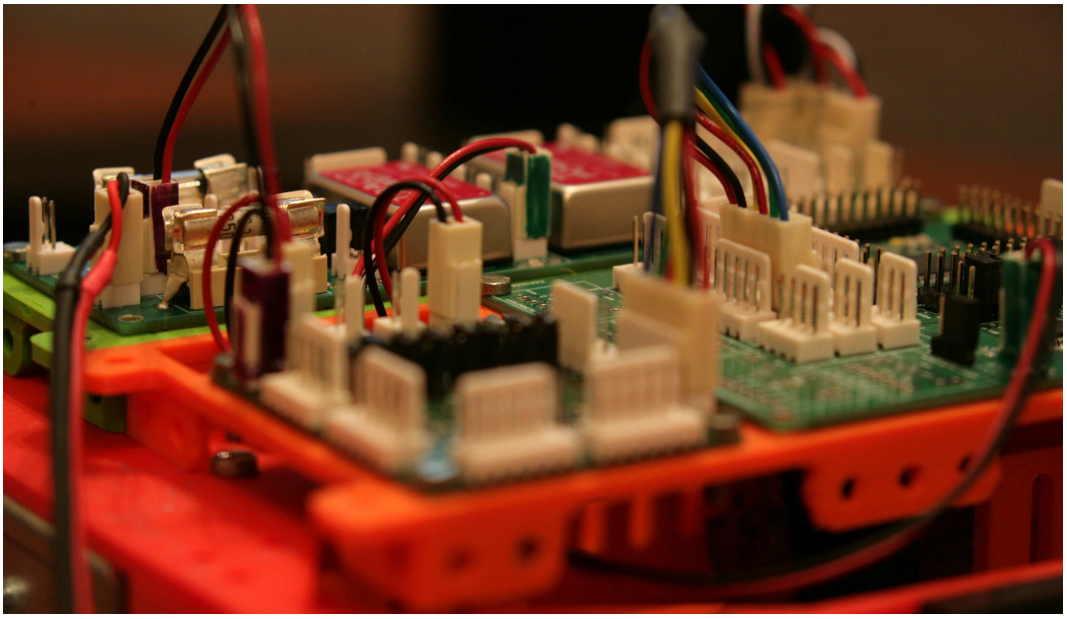


Table des matières

1 A la découverte de la programmation orientée objet en C#

Dans cette partie, vous allez apprendre à programmer en C# avec des interfaces graphiques. Le but est de réaliser un terminal permettant l'envoi et la réception de messages sur le port série du PC. Ce terminal servira dans un premier temps de messagerie instantanée entre deux PC reliés par un câble série, puis il servira ensuite à piloter un robot mobile tout en observant son comportement interne.
Pour commencer, vous apprendrez à travailler avec les objets de base des interfaces graphiques (*Button*, *RichTextBox*, ...). En particulier vous apprendrez à gérer les propriétés de ces objets et les évènements qui leurs sont associés.

1.1 Simulateur de messagerie instantanée

$\Longrightarrow$ Créez un projet C# dans Visual Studio. Pour cela lancer Visual Studio puis *Fichier* $\rightarrow$ *Nouveau* $\rightarrow$ *Projet*. Dans l'onglet *Modèles* $\rightarrow$ *Visual C#*, choisir *Application Windows Form*.
Avant de créer le projet, donnez lui un nom explicite tel que *Robot Interface* et spécifiez un chemin d'accès sur le disque dur tel que *C :/Visual Studio Projects/*. Assurez vous que la case, *Créer un répertoire pour la solution* est cochée. Créez le projet, vous devriez avoir un écran similaire à celui de la figure 1.

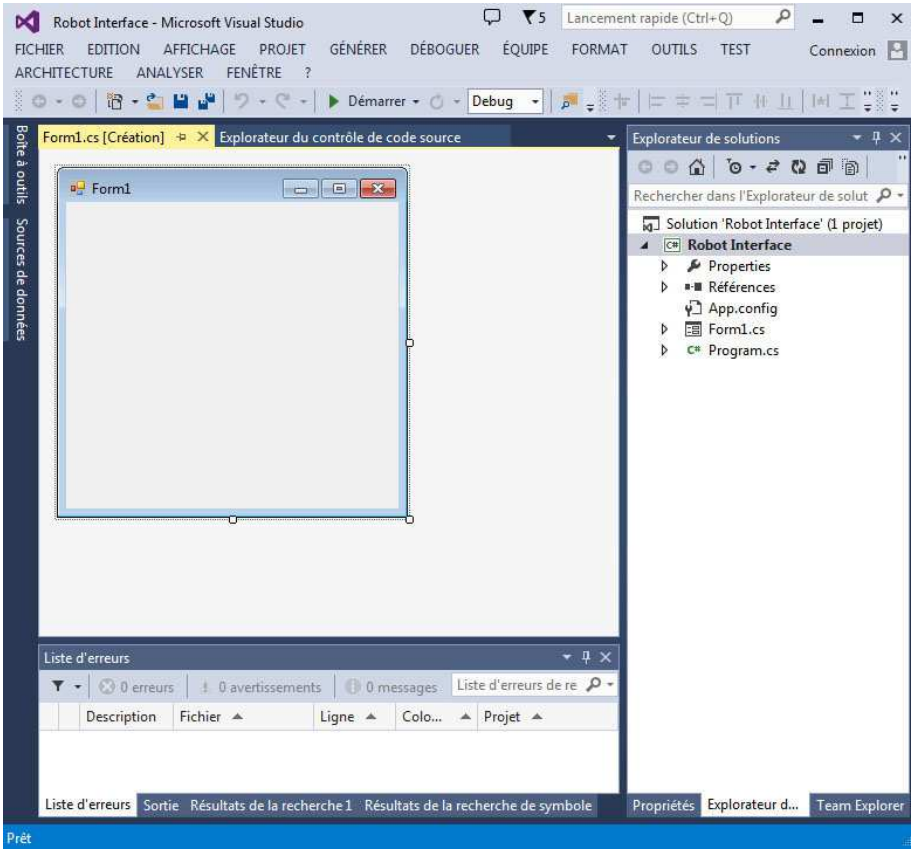


FIGURE 1 – Projet Visual Studio nouvellement créé

La partie droite de l'écran correspond à l'explorateur de solution, qui vous permet de voir les classes du projet, mais également les interfaces graphiques appelées *Form* en *C#*. A la création du projet, un écran graphique dénommé *Form1* est créé par défaut. Il apparaît à gauche de l'écran.

$\Longrightarrow$ Ajoutez à présent à *Form1* deux objets de type *GroupBox* en les faisant glisser depuis la *Boite à outils* $\rightarrow$ *Conteneurs*. Positionnez les comme indiqué à la figure 2. Sélectionnez une de ces *GroupBox* et cliquez ensuite sur l'onglets *Propriétés* dans la partie droite de l'écran. Vous avez accès à une liste importante d'attributs de

l'objet tels que par exemple le *Text* affiché dans le contrôle. Modifiez ce texte pour dénommer la *GroupBox* de gauche *Réception* et celle de droite *Emission*.

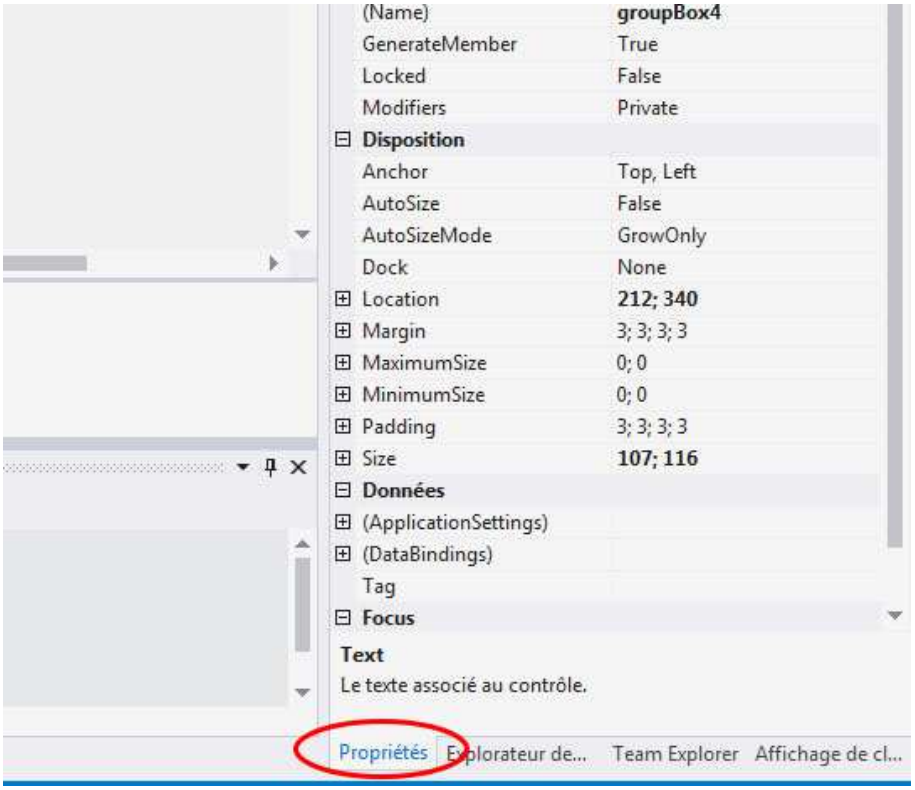


FIGURE 2 – Propriétés d'un objet de type GroupBox

Vous venez de modifier des propriétés d'objets : *une propriété, aussi nommée attribut, est une caractéristique propre à un objet donné.*

$\Longrightarrow$ Ajoutez à présent des objets de type *RichTextBox* depuis la boite à outils, dans chacune des deux *GroupBox*. Dans les propriétés des RichTextBox, passez l'attribut *Dock* à l'état *Fill*. La *RichTextBox* devrait remplir l'intérieur de la *GroupBox* la contenant.

$\Longrightarrow$ Modifiez ensuite l'attribut *Name* des *RichTextBox* en *rtbEmission* et *rtbReception*. Vous devriez avoir un projet ressemblant à celui de la figure 3.

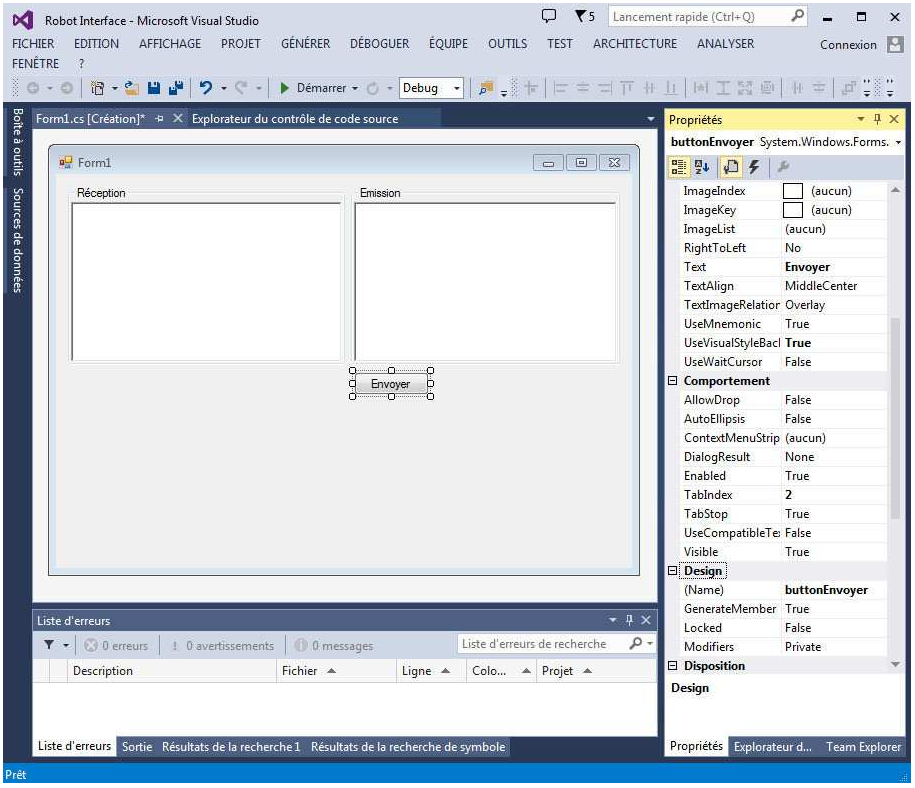


FIGURE 3 – État du projet après insertion du bouton d'envoi

⟹ Dans les propriétés du bouton d'envoi, cliquez sur l'icône en forme d'éclair. Vous ouvrez un autre onglet qui présente les évènements associés à l'objet bouton. Parmi ces évènements figure l'évènement *Click*. Double-cliquez dans la case blanche située à droite de l'évènement *Click*. Une fenêtre dénommée *Form1.cs* a du s'ouvrir dans la fenêtre principale de *Visual Studio*. Cette fenêtre montre le code associé à l'écran graphique *Form1* sur lequel nous avons travaillé jusqu'ici : ce code est dénommé *Code Behind* de la fenêtre.

⟹ Dans le *Code Behind*, une fonction *buttonEnvoyer_Click* a été automatiquement créée. Cette fonction est exécutée chaque fois que l'utilisateur clique sur le bouton envoyer. Pour illustrer son fonctionnement, ajouter dans cette fonction le code suivant :

```
private void buttonEnvoyer_Click(object sender, EventArgs e)
{
    buttonEnvoyer.BackColor = Color.RoyalBlue;
}
```

⟹ Exécutez le programme en appuyant sur *Démarrer* ou sur *F5*. Cliquez sur le bouton *Envoyer*. Que constatez-vous ? Que se passe-t-il si l'on appuie plusieurs fois sur le bouton *Envoyer* ? Commentez.

⟹ Modifier le code à votre convenance de manière à ce que la couleur de fond du bouton *Envoyer* évolue alternativement de la couleur *RoyalBlue* à *Beige* à chaque click. Valider avec le professeur.
Vous avez à présent fait connaissance avec les objets, les propriétés des objets et les évènements qui leurs sont associés.

⟹ A présent, nous souhaitons simuler l'envoi d'un message de la *RichTextBox* d'envoi vers la *RichTextBox* de réception. Pour cela dans la fonction *buttonEnvoyer_Click*, rajouter du code permettant de réaliser cette opération. Vous aurez à manipuler l'attribut *Text* des *RichTextBox*. Le comportement demandé est le suivant :

sur un appui sur le bouton *Envoyer*, le contenu de la *RichTextBox* d'envoi doit être ajouté à celui de la *RichTextBox* de réception, précédé d'un retour à la ligne et de la mention : "Recu : ". La *RichTextBox* d'envoi doit également être vidée.
Le comportement doit être proche de celui de la figure 4 où 4 messages ont été envoyés successivement. Valider avec le professeur.

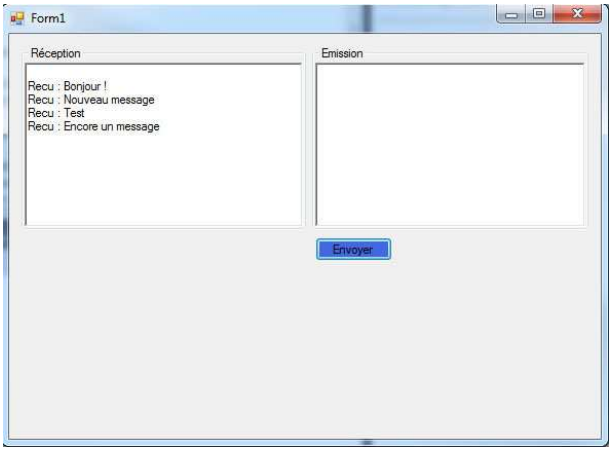


FIGURE 4 – Exemple d'exécution du simulateur d'envois

⟹ Le comportement de l'ensemble simule presque un service de messagerie instantanée, à ceci près que dans ce type de service les envois sont réalisés par appui sur la touche *Entrée*. Il faut pour cela gérer les évènements clavier dans la *RichTextBox* d'émission, en vérifiant que la source de l'appui soit le bouton *Entrée*. Dans la partie graphique de *Form1.cs*, allez dans les *évènements (éclair)* de l'onglet *Propriétés*, et ajoutez un évènement *KeyPress*.

Il est à noter qu'un évènement ne peut être ajouté que si le code n'est pas en cours d'exécution !

La fonction (ou méthode) associée à cet évènement et dénommée *rtbEmission_KeyPress*, possède un argument de type *KeyPressEventArgs*, qui permet de savoir si la touche appuyée est la touche *Entrée* en utilisant par exemple le code suivant :

```
if(e.KeyChar == '\r')
{
    SendMessage();
}
```

⟹ Implanter le code permettant l'envoi des messages sur appui sur la touche Entrée, ou sur le bouton d'envoi, en évitant les duplications de code. Valider le fonctionnement de votre simulateur de messagerie instantanée avec le professeur.

1.2 Messagerie instantanée entre deux PC

A présent vous allez faire communiquer deux PC entre eux, reliés par deux câbles *USB-Série* mis bout à bout. L'envoi et la réception des données se fera par l'intermédiaire d'un objet de type *Serial Port* en *C#*.

⟹ Ajoutez un objet de type *Serial Port* à votre projet, puis allez dans ses propriétés pour régler la vitesse de transmission à 115200 bauds et le nom du port à *"COMX"* où *X* est le numéro du port correspondant à l'adaptateur USB/Série que vous trouverez dans le *Gestionnaire de périphériques* de *Windows*.

⟹ Dans la fonction d'envoi codée précédemment, rajouter un envoi des données vers le port série en utilisant la méthode *WriteLine* de l'objet *SerialPort*. Appelez également à la création du formulaire principal la méthode

d'ouverture du port série, sans quoi les envois ne pourront se faire.

Les données envoyées sur le port série sont visualisables à l'aide d'un oscilloscope. Après avoir correctement configuré l'oscilloscope pour qu'il déclenche sur les arrivées de données série, et après avoir activé l'affichage en mode bus de la liaison série, vérifiez que les données envoyées correspondent aux données reçues par l'oscilloscope. Pour cela, vous brancherez un petit fil sur la pin *Transmit Data* du connecteur série comme indiqué à la figure 5 pour sortir vers l'oscilloscope.

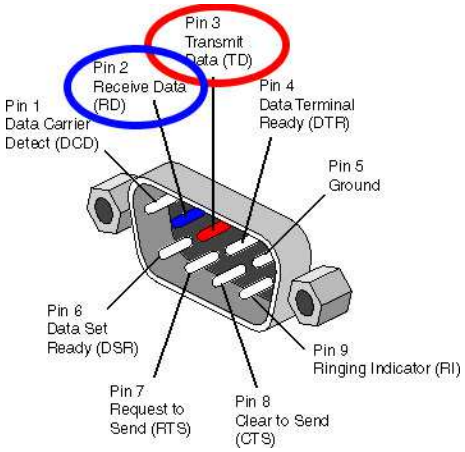


FIGURE 5 – Pinout du port RS232

A présent, nous allons renvoyer les données envoyées par la pin *Transmit Data* du port série vers la pin *Receive Data* de ce même port série afin de recevoir sur le port série les données que nous envoyons. Ce mode de fonctionnement s'appelle le mode *LoopBack*, il permet de valider son code sans avoir besoin d'un second ordinateur.

$\Longrightarrow$ Connectez le port *RS232* en mode *Loopback*.

$\Longrightarrow$ Ajoutez un évènement *DataReceived* au port série. Cet évènement est déclenché quand des données ont été reçues par le port série et sont disponibles. Vérifiez en ajoutant un point d'arrêt dans le code (*F9*), que vous passez dans cet évènement.

$\Longrightarrow$ Récupérez à présent les données disponibles grâce à la méthode *ReadExisting()* du port série, et envoyez les vers l'affichage de la *RichTextBox* de réception. Que se passe-t-il ?

Vous avez rencontré (sans vraiment le chercher !) un aspect important de la programmation sur OS : **le multithreading**. Dans un PC, de nombreuses tâches doivent s'effectuer en parallèle, alors que les processeurs effectuent les tâches séquentiellement. Pour ce faire, les tâches sont placées dans des *Threads* (processus indépendants) dont le fonctionnement est fractionné temporellement et mis à la file indienne de manière à ce que l'utilisateur ait la perception d'un fonctionnement parallèle de l'ensemble.

Dans notre application, nous avons pour l'instant deux threads : un qui gère l'interface graphique et le programme principal et un autre qui gère le port série. Ce second thread est nécessaire car le port série doit être en permanence à l'écoute de nouvelles données qui peuvent arriver de manière asynchrone sur l'entrée *Receive Data* du port. Il serait très impactant d'être obligé d'attendre que le programme principal ait terminé ses opérations avant de lire le port série, ce qui serait le cas si il n'était pas placé dans un thread distinct.

Les threads offrent donc des possibilités intéressantes en permettant d'éviter de bloquer l'application quand une partie du code est par exemple dans une boucle d'attente longue. Toutefois, les thread apportent des contraintes dans la programmation, telles que celle que vous venez de rencontrer. Vous avez du lever l'exception suivante :

> *Une exception non gérée du type System.InvalidOperationException s'est produite dans System.Windows.Forms.dll. Informations supplémentaires : Opération inter-threads non valide : le contrôle 'richTextBoxReception' a fait l'objet d'un accès à partir d'un thread autre que celui sur lequel il a été créé.*

L'erreur déclenchée à l'exécution vous indique qu'il est impossible de mettre à jour un objet (en l'occurrence la *RichTextBox*) directement géré par un thread (en l'occurrence le thread d'affichage) à partir d'un autre thread (en l'occurrence le thread du port série).

Il est nécessaire pour éviter cela, de passer par un objet non-graphique géré en dehors du code dans lequel on peut écrire les données et les lire : nous utiliserons pour l'instant une simple chaîne de caractère pour cela.

$\Longrightarrow$ Déclarez une chaîne de caractère nommée *receivedText* en dehors de la fonction de réception et ajoutez les données reçues à cette chaîne. Ces données sont en attente d'être récupérées par l'application et affichées graphiquement.

$\Longrightarrow$ Implanter un objet depuis la ToolBox permettant de regarder périodiquement si la chaîne *receivedText* contient quelque chose, et dans ce cas affichez ces données dans la RichTextBox de réception. N'oubliez pas de démarrer cet objet au lancement du programme ! Valider le fonctionnement avec le professeur.

$\Longrightarrow$ A présent, vous pouvez brancher votre câble série avec celui de vos voisins, en connectant le Tx de l'un sur le Rx de l'autre et vice-versa. Valider que les envois de messages fonctionnent bien... sans pour autant écrire n'importe quoi à vos voisins !

$\Longrightarrow$ Ajoutez un bouton *Clear* permettant de vider la *richTextBox* de réception, et écrivez le code correspondant.

1.3 Liaison série hexadécimale

Vous avez terminé votre système de messagerie instantanée entre deux PC. Ce système permet d'échanger des chaînes de caractère efficacement. Par contre, il ne permet pas de faire passer n'importe quel caractère et en particulier les caractères de contrôles de la table *ASCII*. Il est donc souhaitable d'évoluer vers une liaison série permettant d'envoyer des octets (*byte*), quelque soit leur valeur.

Dans cette partie vous travaillerez à nouveau en mode *LoopBack*.

$\Longrightarrow$ Pour mettre en évidence les problème existants avec le système actuel, ajouter un bouton *Test*, et sur l'évènement *Click*, effectuez les opérations suivante : construisez un tableau (nommé par exemple *byteList*) de 20 bytes et remplissez-le par exemple en y mettant les valeurs suivantes : $byteList[i] = (byte)(2 * i)$;. Envoyez ce tableau de bytes sur le port série à l'aide de la fonction *Write*.

$\Longrightarrow$ Testez l'applications et regardez le retour dans la console de réception en le comparant aux données circulant sur le bus et que vous pouvez afficher à l'aide de l'oscilloscope. Est-ce exploitable ?

Afin de visualiser correctement les données circulant sur la liaison série, il serait préférable de les traiter en tant qu'octet et non en tant que chaine de caractère, et il serait également mieux de les afficher en hexadécimal. La chaine de stockage temporaire de caractère utilisée précédemment (*receivedText*) n'est donc pas adaptée à notre problème. Il serait préférable de disposer d'un *buffer* d'octets pouvant être rempli lors de la réception de données sur le port série et vidé par le *Timer* permettant l'affichage des résultats dans la console. Un tel buffer est de type *FIFO (First In First Out)*, il est implanté en *C#* à l'aide d'une $Queue < byte >$.

$\Longrightarrow$ Déclarez une FIFO pour les octets reçus sur le port série et initialisez la à l'aide de la ligne de code suivante :

```
Queue<byte> byteListReceived = new Queue<byte>();
```

$\Longrightarrow$ Dans la fonction *DataReceived* du port série, placez les octets disponibles un par un et l'un après l'autre dans la *Queue*, jusqu'à ce qu'il n'y ait plus de données disponibles dans le port série. Pour cela, regardez dans les propriétés du attributs et les méthodes de l'objet *SerialPort*, celles qui vous serviront à mettre en oeuvre ce fonctionnement.

$\Longrightarrow$ Sur les évènements *Timer*, videz à présent la *Queue* et affichez les *bytes* récupérés dans la *RichTextBox* de réception. L'affichage se fera grâce à la méthode *byte.ToString()*. Cette méthode peut prendre des paramètres de formatage que vous allez tester pour les comprendre. Vous essayerez et commenterez les résultats :

- *ToString()*
- *ToString("X")*
- *ToString("X2")*
- *ToString("X4")*

$\Longrightarrow$ Vous implanterez pour terminer sur ce point un code permettant de renvoyer pour chaque octet arrivé, sa valeur au format *0xhh* ou *hh* est la valeur en hexadécimal sur *2* caractères. Chaque octet reçu sera séparé par un espace. Valider avec le professeur les résultats obtenus.

2 A la découverte de la communication point à point en embarqué

Cette partie utilise la carte principale. Toutefois, cette carte principale ne dispose pas de liaison USB ni de convertisseur *USB-série*. Il sera donc nécessaire, pour en disposer, d'utiliser la carte capteurs branchée sur un PC via une connexion USB (connecteur micro-USB). La liaison entre la carte principale et la carte capteurs se fera par deux fils placés de manière appropriée.

2.1 La liaison série en embarqué

$\Longrightarrow$ Créer un fichier *UART.c* contenant le code suivant qui permet d'initialiser l'UART à la vitesse de 115200 bauds, sans utiliser les interruptions :

```
#include <xc.h>
#include "UART.h"
#include "ChipConfig.h"

#define BAUDRATE 115200
#define BRGVAL ((FCY/BAUDRATE)/4)-1

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 = 0; // Disable UART Tx interrupt

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

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

$\Longrightarrow$ Créer également un fichier *UART.h* contenant le code suivant :

```
#ifndef UART_H
#define UART_H

void InitUART(void);

#endif  /* UART_H */
```

$\Longrightarrow$ Ajouter l'appel de la fonction d'initialisation dans le *main*, ainsi que l'include de *"uart.h"*.

$\Longrightarrow$ Le programme en l'état peut-il fonctionner ? Pourquoi ?

2.2 Échange de données entre le microcontrôleur et le PC

2.2.1 Validation du bon fonctionnement du convertisseur USB/série

$\Longrightarrow$ Ouvrir le projet *Visual Studio* que vous avez réalisé précédemment.

$\Longrightarrow$ Brancher le convertisseur USB-série fourni au PC via le câble mini-USB. Cette liaison *USB* va être utilisée comme support de la liaison *UART*.

$\Longrightarrow$ Vérifier que le port série est apparu dans la liste des ports de communication du PC, en allant dans le gestionnaire de périphériques. Si ça n'est pas le cas, appeler le professeur.

$\Longrightarrow$ Ajuster le numéro du port série du composant port série dans Visual Studio pour correspondre au numéro de port précédemment trouvé.

$\Longrightarrow$ Vous allez configurer le port en mode *LoopBack*, ce qui signifie que les informations arrivant de l'USB sur la pin Rx de la liaison série doivent être renvoyées vers l'USB sur la pin Tx. Placer un jumper sur le dongle USB-série entre Tx et Rx afin de réaliser le *LoopBack* physique.

$\Longrightarrow$ Lancer l'exécution du programme *C#* (*F5*).

$\Longrightarrow$ Entrer une chaîne de caractères dans la fenêtre d'envoi et appuyer sur *Envoyer*. La chaîne est envoyée via la liaison USB, elle arrive sur la carte en Rx, revient sur Tx et apparaît sur le moniteur de réception si tout se passe bien. Contrôler à l'oscilloscope le bon fonctionnement.

2.2.2 Émission UART depuis le microcontrôleur

$\Longrightarrow$ Brancher un cable (fourni) à partir du convertisseur série-USB, et faites le arriver sur les pins Rx et Tx du périphérique UART1 du microcontrôleur. Vous connecterez le fil Tx en provenance du convertisseur USB-série l'*USB* à la pin Rx de l'*UART*, puisque les données reçues depuis le PC par la liaison USB arrivent sur Tx *USB* et doivent donc être transmises et donc reçues par le microcontrôleur sur sa pin Rx *UART*.

$\Longrightarrow$ En analysant le schéma de câblage du dsPIC (fig. **??**), déterminez le numéro des pins remappables correspondantes.

$\Longrightarrow$ Configurez dans le code les pins remappables de la liaison série en ajoutant dans la partie correspondante du fichier *"IO.c"*, entre les appels des fonctions *lock* et *unlock*, le code suivant , en remplaçant les ... par les valeurs trouvées à la question précédente :

```
    _U1RXR = ...; //Remappe la RP... sur l'éentre Rx1
    _RP...R = 0b00001; //Remappe la sortie Tx1 vers RP...
```

$\Longrightarrow$ Ajoutez une fonction d'envoi de message au fichier "UART.c" et le header correspondant. Le code de cette fonction est le suivant :

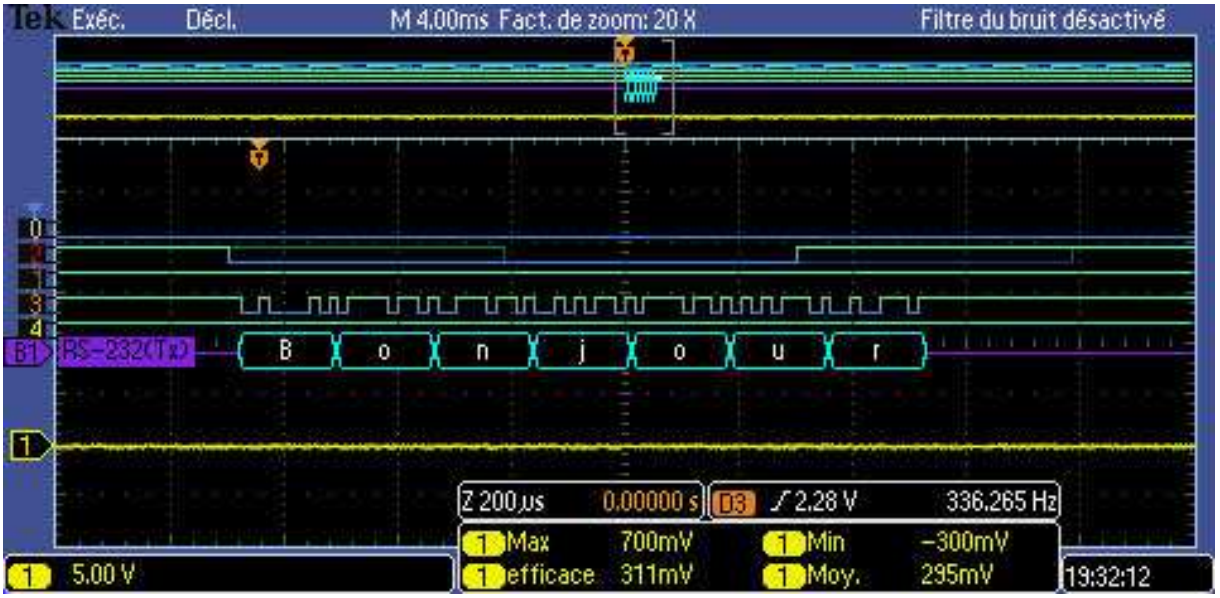
```
void SendMessageDirect(unsigned char* message, int length)
{
    unsigned char i=0;
    for(i=0; i<length; i++)
    {
        while ( U1STAbits.UTXBF); // wait while Tx buffer full
        U1TXREG = *(message)++; // Transmit one character
    }
}
```

$\Longrightarrow$ Ajoutez à la boucle infinie du main du code permettant d'envoyer à intervalle régulier une trame, par exemple "Bonjour". Ce peut être le suivant :

```
        SendMessageDirect((unsigned char*) "Bonjour", 7);
        __delay32(40000000);
```

Vous noterez que ce code est bloquant (un envoi chaque seconde, attente bloquante entre deux envois), mais vous l'utiliserez momentanément à des fins de test.

⟹ Vérifiez le bon fonctionnement des envois de trame à l'aide de l'oscilloscope et de son module d'analyse de bus série. Le résultat doit ressembler à ceci :



⟹ Vérifiez le bon fonctionnement des envois de trame à l'aide du logiciel de visualisation en *C#* :

2.2.3 Réception

Afin de valider la réception sur le port série du microcontrôleur, nous allons la faire fonctionner en mode *LoopBack* logiciel.

Pour cela, dès qu'un caractère arrive en Rx, il doit être renvoyé sur Tx. Nous utiliserons l'interruption UART en réception pour détecter les arrivées asynchrones de caractères.

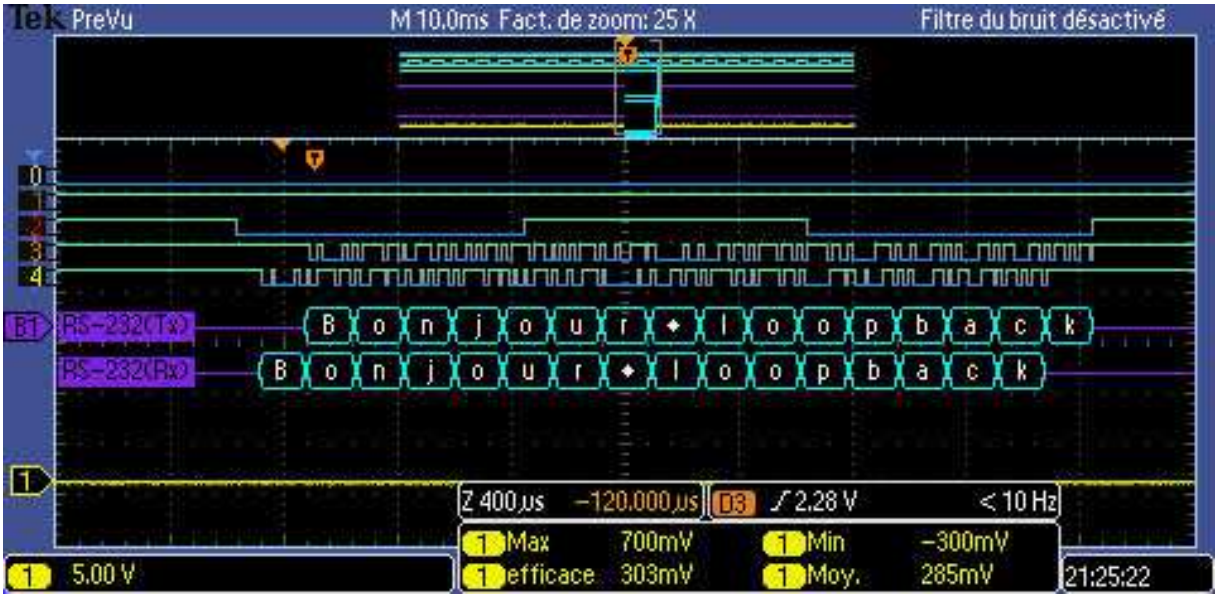
⟹ Autoriser les interruptions en réception sur l'UART en modifiant la fonction d'initialisation du port série.

⟹ Ajouter la routine d'interruption en utilisant le code suivant :

```
//Interruption en mode loopback
void __attribute__((interrupt, no_auto_psv)) _U1RXInterrupt(void) {
    IFS0bits.U1RXIF = 0; // clear RX interrupt flag
    /* check for receive 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 the data */
    while (U1STAbits.URXDA == 1) {
        U1TXREG = U1RXREG;
    }
}
```

⟹ Désactiver, l'envoi périodique et bloquant des messages depuis le *main*. Exécutez le programme sur le PIC.

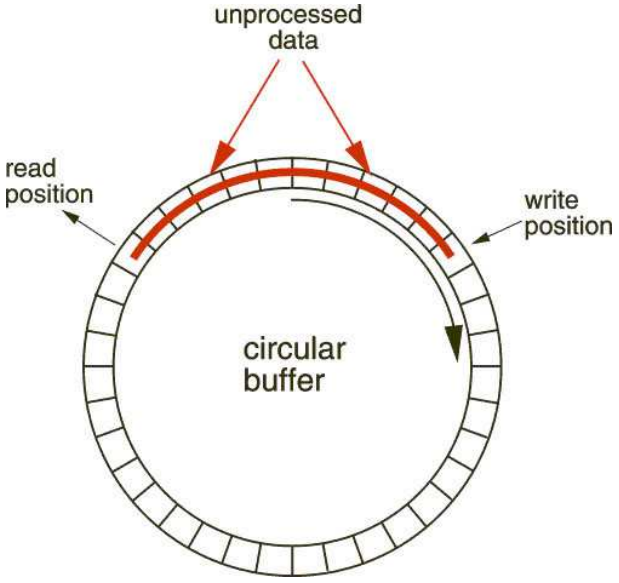
$\Longrightarrow$ Depuis l'interface *C#*, envoyez un message, celui-ci doit revenir sur la console de réception. Visualisez l'envoi et le retour de la trame sur l'oscilloscope. Vous devez obtenir un fonctionnement semblable à la capture d'écran ci-dessous.



2.3 Liaison série avec FIFO intégré

2.3.1 Le buffer circulaire de l'UART en émission

La fonction d'envoi développée précédemment fonctionne, mais elle bloque la boucle principale du programme jusqu'à la fin de l'envoi du message. Afin d'éviter cela, la suite du TP consiste à implanter un *buffer circulaire*, qui est une manière d'implanter une *FIFO* en embarqué, afin de stocker les messages à envoyer sur le port série en attente de leur envoi. Cet envoi doit se faire entièrement en mode interruption.



$\Longrightarrow$ Créer un fichier *CB_TX1.c* destiné à recevoir le code du buffer circulaire en transmission. Créer le header correspondant.

$\Longrightarrow$ L'objectif est à présent de faire fonctionner le buffer circulaire, sachant que la fonction *SendMessage* sert à initier les envois. Les caractères contenus dans le message doivent être insérés dans le buffer circulaire si la place restante le permet. Si la transmission n'est pas en cours, alors la fonction *SendMessage* doit l'initier en appelant la fonction *SendOne*.
A chaque caractère transmis par l'UART, une interruption est levée. Si le pointeur de queue n'est pas dans la même position que le pointeur de tête, c'est qu'il reste des caractères à envoyer et un nouvel envoi est effectué.
Le canevas est défini dans le code ci-dessous. Vous devez remplacer les "..." par votre code :

```
#include <xc.h>
#include <stdio.h>
#include <stdlib.h>
#include "CB_TX1.h"
#define CBTX1_BUFFER_SIZE 128

int cbTx1Head;
int cbTx1Tail;
unsigned char cbTx1Buffer[CBTX1_BUFFER_SIZE];
unsigned char isTransmitting = 0;

void SendMessage(unsigned char* message, int length)
{
    unsigned char i=0;

    if(CB_TX1_RemainingSize()>length)
    {
        //On peut écrire le message
        for(i=0;i<length;i++)
            CB_TX1_Add(message[i]);
        if(!CB_TX1_IsTranmitting())
            SendOne();
    }
}

void CB_TX1_Add(unsigned char value)
{
    ...
}

unsigned char CB_TX1_Get(void)
{
    ...
}


void __attribute__((interrupt, no_auto_psv)) _U1TXInterrupt(void) {
    IFS0bits.U1TXIF = 0; // clear TX interrupt flag
    if (cbTx1Tail!=cbTx1Head)
    {
        SendOne();
    }
    else
        isTransmitting = 0;
}

void SendOne()
{
    isTransmitting = 1;
    unsigned char value=CB_TX1_Get();
    U1TXREG = value; // Transmit one character
}

unsigned char CB_TX1_IsTranmitting(void)
```

```
{
    ...
}

unsigned char CB_TX1_RemainingSize(void)
{
    unsigned char rSize;
    ..
    return rSize;
}
```

⟹ Créez le header correspondant.

⟹ Le code fonctionnant sur interruption Tx, modifiez votre fonction d'initialisation de l'UART en conséquences.

⟹ Incluez le fichier *"CB_TX1.h"* dans le main et modifier le code de la boucle principale pour appeler la fonction *SendMessage*.

⟹ Compilez et testez le nouveau code, qui doit fonctionner à l'identique du précédent, mais qui n'est plus bloquant (durant l'envoi du message).

2.3.2 Le buffer circulaire de l'UART en réception

Afin de pouvoir mettre en oeuvre des traitements complexes sur les trames, la suite du projet consiste à implanter un second buffer circulaire pour stocker les données arrivant sur le port série en attente de leur utilisation. Ce stockage doit se faire entièrement en mode interruption à la réception.

⟹ Créez un fichier *CB_RX1.c* destiné à recevoir le code du buffer circulaire en réception. Créer le header correspondant.

⟹ L'objectif est à présent de faire fonctionner le buffer circulaire, sachant que chaque caractère reçu sur l'UART doit être stocké dans le buffer, et qu'à ce moment là le pointeur *head* de celui-ci doit être incrémenté. Le canevas est défini dans le code ci-dessous. Les "..." sont à remplacer par votre code :

```
#include <xc.h>
#include <stdio.h>
#include <stdlib.h>
#include "CB_RX1.h"

#define CBRX1_BUFFER_SIZE 128


int cbRx1Head;
int cbRx1Tail;
unsigned char cbRx1Buffer[CBRX1_BUFFER_SIZE];

void CB_RX1_Add(unsigned char value)
{
    if(CB_RX1_GetRemainingSize()>0)
    {
        ...
    }
}

unsigned char CB_RX1_Get(void)
{
    unsigned char value=cbRx1Buffer[cbRx1Tail];
    ...
```

```
    return value;
}

unsigned char CB_RX1_IsDataAvailable(void)
{
    if(cbRx1Head!=cbRx1Tail)
        return 1;
    else
        return 0;
}

void __attribute__((interrupt, no_auto_psv)) _U1RXInterrupt(void) {
    IFS0bits.U1RXIF = 0; // clear RX interrupt flag
    /* check for receive 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 the data */
    while(U1STAbits.URXDA == 1) {
        CB_RX1_Add(U1RXREG);
    }
}

unsigned char CB_RX1_GetRemainingSize(void)
{
    unsigned char rSizeRecep;
    ...
    return rSizeRecep;
}

unsigned char CB_RX1_GetDataSize(void)
{
    unsigned char rSizeRecep;
    ...
    return rSizeRecep;
}
```

⟹ Pensez à mettre à jour le header correspondant et à commenter le code de l'interruption *_U1RXInterrupt* des fichiers *UART.c* et *UART.h*.

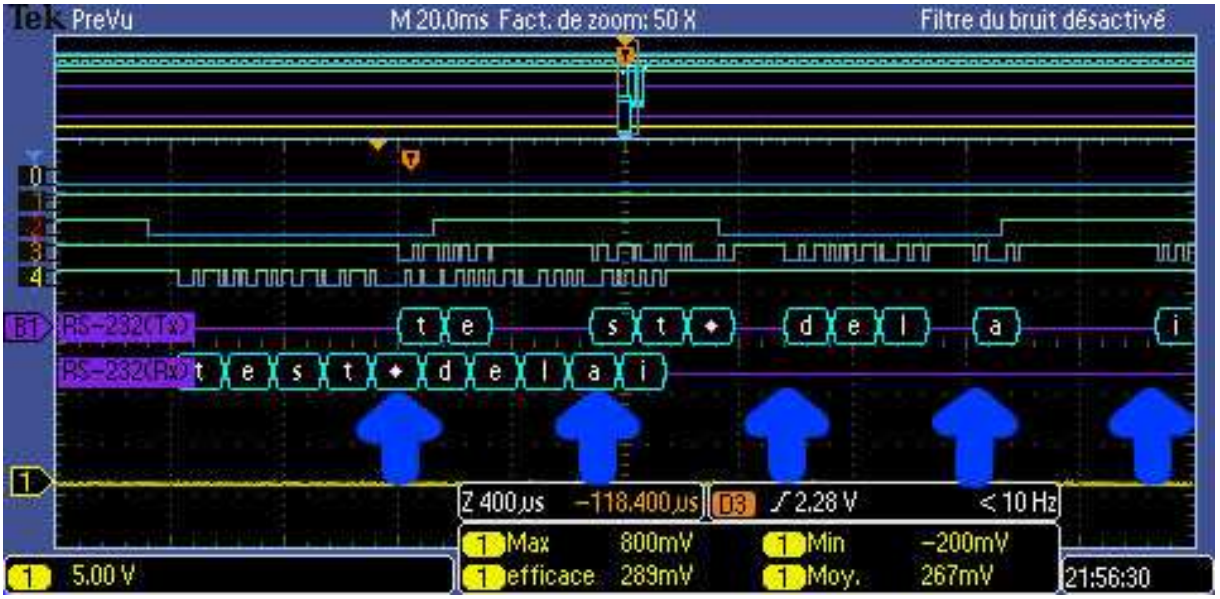
⟹ Insérez la boucle infinie du main par le code suivant, en n'oubliant pas de commenter au préalable le *SendMessage* précédent et son délai d'attente. Ce code regarde dans le buffer de réception si des caractères sont présents dans le buffer circulaire tRx, et les récupère avant de les envoyer dans le buffer circulaire Tx. Pensez à inclure les fichiers *.h* nécessaires.

```
        int i;
        for(i=0; i< CB_RX1_GetDataSize(); i++)
        {
            unsigned char c = CB_RX1_Get();
            SendMessage(&c,1);
        }
        __delay32(1000);
```

⟹ Tester le fonctionnement en envoyant un message depuis l'interface graphique.

⟹ Que ce passe-t-il si l'on augmente la valeur de la temporisation dans la boucle infinie placée dans l'instruc-

tion ___*delay32*. Essayez avec une valeur de 10000. Vous devriez obtenir un résultat proche de celui ci-dessous. Commentez-le.



3 A la découverte de la supervision d'un système embarqué

Le système d'échange d'octets que vous avez mis en oeuvre précédemment permet de faire passer des valeurs quelconques entre $0x00$ et $0xFF$. Son fonctionnement est robuste du point de vue du flux de données dans la mesure où il dispose de *FIFO* en embarqué et en *C#*. Il serait donc possible d'envoyer des suites d'octets pour par exemple piloter un robot mobile.

La limitation de ce système, est que les échanges se font via des suites d'octets qui n'ont pas de sens d'un point de vue sémantique. Pire, dans le cas d'un fonctionnement en environnement industriel bruité par exemple, il est impossible de savoir si des données transmises ont été corrompues ou pas (par exemple un *O* peut s'être transformé en 1 ou vice-versa). L'usage d'un protocole de communication est donc indispensable pour aller vers un pilotage vraiment fiable et efficace de notre robot. Ce formatage en trames correspond à la couche 2 du modèle *OSI* vu en cours.

3.1 Implantation en *C#* d'un protocole de communication avec messages

Vous allez donc implanter un protocole de communication utilisant la liaison série du PC. Ce protocole est basé sur des messages échangés avec le formatage suivant :

Start Of Frame (SOF)	Command	Payload Length	Payload	Checksum
0xFE	2 octets	2 octets	n octets	1 octet

Chaque message début par un *Start Of Frame* valant *0xFE*, il est suivi d'une *commande* sur 2 octets (le premier est fixé à 0x00), suivie d'arguments (*Payload*) de taille variable, la taille et étant spécifiée par la *Payload Length*. Un *Checksum*, termine la trame, il est calculé comme étant le *Ou Exclusif* bit à bit de tous les octets de la trame (incluant le *SOF*) à l'exception du checksum bien évidemment.

3.1.1 Encodage des messages

$\Longrightarrow$ La première des fonctions à coder est le calcul du checksum qui sera utilisé dans la génération de la trame. Proposer une implantation ayant le prototype suivant :

```
byte CalculateChecksum(int msgFunction,
        int msgPayloadLength, byte[] msgPayload)
```

$\Longrightarrow$ Implantez la fonction *UartEncodeAndSendMessage*, permettant de formatter et d'envoyer une trame de données sur la liaison série. Son prototype sera le suivant :

```
void UartEncodeAndSendMessage(int msgFunction,
        int msgPayloadLength, byte[] msgPayload)
```

$\Longrightarrow$ En utilisant votre bouton de test, valider cette fonction en envoyant un message dont le numéro de fonction est $0x0080$, la *payload* étant une chaîne de caractères convertie en byte[] et la *payload length* la taille de cette chaîne de caractère. La conversion de string en byte[] se fait grâce à la fonction :

```
byte[] array = Encoding.ASCII.GetBytes(s);
```

Vérifiez à l'oscilloscope et sur votre terminal de réception que la trame correspond bien à ce qui est attendu, et en particulier en ce qui concerne le *checksum* : si l'on envoie "Bonjour", on devrait avoir un *checksum* valant *0x38*. Validez les résultats avec le professeur.

3.1.2 Décodage des messages

A présent vous pouvez passer à un point plus complexe de ce projet : la fonction de décodage des trames reçues.

Cette fonction prend un unique octet en argument. Elle doit donc connaître au moment de l'arrivée de cet octet son état interne. Une machine à état est donc un bon moyen de décrire son fonctionnement. Cette machine a été sera utilisée en *C#* dans un premier temps, mais également en *C* ensuite, il est donc souhaitable de la coder de manière à ce qu'elle puisse être réutilisée en *C*. Pour cette raison, nous utiliserons la structure *Switch Case* pour décrire notre machine à état en *C#*. Afin de clarifier le code, un *enum* est utilisé pour donner des noms logiques aux états.

Le canevas du code à compléter est le suivant. Notez que l'allocation de msgPayload devra se faire lorsque l'on connaîtra la taille du message.

```
public enum StateReception
{
    Waiting,
    FunctionMSB,
    FunctionLSB,
    PayloadLengthMSB,
    PayloadLengthLSB,
    Payload,
    CheckSum
}

StateReception rcvState = StateReception.Waiting;
int msgDecodedFunction = 0;
int msgDecodedPayloadLength = 0;
byte[] msgDecodedPayload;
int msgDecodedPayloadIndex = 0;

private void DecodeMessage(byte c)
{
    switch(rcvState)
    {
        case StateReception.Waiting:
            ...
            break;
        case StateReception.FunctionMSB:
            ...
            break;
        case StateReception.FunctionLSB:
            ...
            break;
        case StateReception.PayloadLengthMSB:
            ...
            break;
        case StateReception.PayloadLengthLSB:
            ...
            break;
        case StateReception.Payload:
            ...
            break;
        case StateReception.CheckSum:
            ...
            if (calculatedChecksum == receivedChecksum)
                {
                    //Success, on a un message valide
                }
            ...
            break;
        default:
            rcvState = StateReception.Waiting;
            break;
    }
```

```
    }
```

⟹ Programmez la fonction *DecodeMessage*, et testez là en l'appelant sur chaque lecture de caractère en sortie de la *FIFO*. Quand votre fonction marchera, vous devriez valider la condition *if (calculatedChecksum == receivedChecksum)*. Veillez à ce que tout se passe bien plusieurs messages d'affilée. Vous pouvez également générer un message avec une erreur dedans pour valider le rejet du message et la synchronisation ultérieure de la machine à état (pour cela pensez à tester la taille de payload acceptable). Validez avec le professeur cette partie.

3.1.3 Pilotage et supervision du robot

Le décodage des trames étant à présent opérationnel, nous allons utiliser ce système de messagerie pour piloter le robot et le superviser. La **supervision** permet de rendre observable des variables internes au robot telles que les distances mesurées par les ADC, l'état des LEDs, les vitesses moteurs ou la position du robot...

Le pilotage et la supervision sont basés sur un ensemble de message définis à l'avance et qui forment une bibliothèque devant être connue du robot et de la plate-forme de supervision. Chacun de ces messages a un numéro de fonction unique, et une *payload* de taille définie à l'avance. Les premières fonction que nous allons implanter sont définies dans le tableau ci-dessous (fig. 6. Celui-ci sera complété au fur et à mesure.

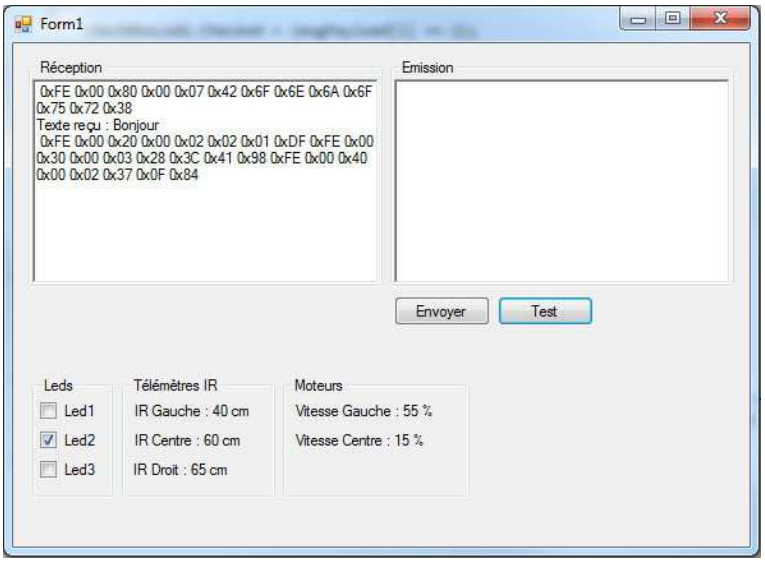
Command ID (2 bytes)	Description	Payload Length (2 bytes)	Description de la payload
0x0080	Transmission de texte	taille variable	texte envoyé
0x0020	Réglage LED	2 bytes	numéro de la LED - état de la LED (0 : éteinte - 1 : allumée)
0x0030	Distances télémètre IR	3 bytes	Distance télémètre gauche - centre - droit (en cm)
0x0040	Consigne de vitesse	2 bytes	Consigne vitesse moteur gauche - droit (en % de la vitesse max)

FIGURE 6 – Fonctions de supervision

⟹ A l'aide du bouton de *Test*, simulez l'envoi successif de chacun de ces messages et implantez dans l'interface graphique des éléments de votre choix permettant de visualiser les valeurs reçues en mode *loopback*. Pensez à regroupez les éléments relatifs à une même fonctionnalité dans une *groupBox*. Pour plus de clarté, vous pouvez utiliser un *enum* couplant le nom des fonctions et leur *ID*, en cas de besoin demandez au professeur. Les messages seront traités (après leur réception et leur validation) par une fonction *ProcessDecodedMessage* dont le prototype est le suivant :

```
void ProcessDecodedMessage(int msgFunction,
            int msgPayloadLength, byte[] msgPayload)
```

⟹ Vous pouvez vous inspirer de la figure 3.1.3 pour l'interface graphique. Valider le résultat avec le professeur.



Votre interface de supervision et commande est pour l'instant terminée, reste à implanter une version équivalente du code en embarqué et vous pourrez faire communiquer votre robot et votre interface ensemble.

3.2 Implantation en électronique embarquée

L'UART du dsPIC n'ayant plus de secret pour vous, vous allez implanter un protocole de communication par-dessus la couche *UART + Buffers circulaires* que vous avez implanté précédemment.

$\Longrightarrow$ Pour cela, créez un nouveau fichier *"UART_Protocol.c"* et son header, qui servira à implanter ce protocole.

$\Longrightarrow$ Insérez dans votre fichier le squelette de code ci-dessous.

```
#include <xc.h>
#include "UART_Protocol.h"

unsigned char UartCalculateChecksum(int msgFunction,
        int msgPayloadLength, unsigned char* msgPayload)
{
    //Fonction prenant éentre la trame et sa longueur pour calculer le checksum
    ...
}

void UartEncodeAndSendMessage(int msgFunction,
        int msgPayloadLength, unsigned char* msgPayload)
{
    //Fonction d'encodage et d'envoi d'un message
    ...
}

int msgDecodedFunction = 0;
int msgDecodedPayloadLength = 0;
unsigned char msgDecodedPayload[128];
int msgDecodedPayloadIndex = 0;

void UartDecodeMessage(unsigned char c)
{
    //Fonction prenant en éentre un octet et servant à reconstituer les trames
    ...
}

void UartProcessDecodedMessage(unsigned char function,
```

```
        unsigned char payloadLength, unsigned char* payload)
{
    //Fonction éappele èaprs le édcodage pour éexcuter l'action
    //correspondant au message çreu
    ...
}


//****************************************************************/
//Fonctions correspondant aux messages
//****************************************************************/
```

3.2.1 Supervision

La supervision permet de faire remonter les informations de fonctionnement du robot vers l'interface graphique. Elle est définie dans l'implantation de la norme OSI relative aux bus terrain.

⟹ Écrire les fonctions *UartEncodeAndSendMessage* et *UartCalculateChecksum* en vous inspirant de la version en *C#*. **Pour l'instant commentez les 2 autres fonctions non implantées.**

⟹ Testez la fonction *UartEncodeAndSendMessage* depuis le main avec comme arguments *fonction = 0x0080*, *payload length* la taille de la *payload* à envoyer, et *payload*, le tableau d'octets représentant la chaîne de caractère à envoyer. On initialisera la payload comme suit :

```
unsigned char payload[] = {'B', 'o', 'n', 'j', 'o', 'u', 'r'};
```

N'oubliez pas de remettre une temporisation d'envoi entre les messages (par exemple *___delay32(40000000);*), sans quoi le code de réception en *C#* ne pourra pas absorber le flux.

⟹ Le résultat de vos envois doit apparaître dans la console de réception. En premier lieu viennent les caractères du message puis le contenu de la payload est affiché. Vérifier ce fonctionnement.

Si seul les caractères du message s'affichent, le message est mal constitué.

⟹ A présent, désactivez l'appel de la fonction précédente et la temporisation bloquante qui a été ajoutée. A la fin de la fonction de conversion des valeurs issues des télémètres infrarouge, envoyez un message permettant de visualiser les valeurs sur l'interface graphique en *C#*. Le format de la trame envoyée doit être en accord avec le tableau de la figure 6.

⟹ A ce point d'avancement, vous pouvez désactiver l'affichage de caractères reçus dans la console de réception en *C#* afin de ne pas surcharger l'affichage.

⟹ Implantez à présent une fonction de supervision supplémentaire permettant de savoir quand le robot passe d'une étape de déplacement à l'autre. Les étapes correspondent aux moments où de nouvelles valeurs sont envoyées aux consignes de vitesse moteur, il ne faut rien renvoyer durant les attentes de transitions sinon la console va être inondée de messages. Cette fonction aura pour identifiant 0x0050, et une *payload* de 5 octets : le numéro d'étape, suivi de l'instant courant en millisecondes codé sur 4 octets.

⟹ Testez le fonctionnement en rajoutant en *C#* une fonction permettant d'afficher dans la console de réception l'étape en cours et son instant de déclenchement. Pour cela on rajoutera à la fonction *ProcessDecodedMessage* en *C#* un case implanté par exemple comme suit :

```
    case MsgFunction.RobotState:
        int instant = (((int)msgPayload[1])<<24) + (((int)msgPayload[2])<<16)
                    + (((int)msgPayload[3])<<8) + ((int)msgPayload[4]);
        rtbReception.Text += "\nRobot State : " +
                    ((StateRobot)(msgPayload[0])).ToString() +
```

```
                    "␣-␣" + instant.ToString() + "␣ms";
            break;
```

Dans ce code, *StateRobot* est un *enum* implanté comme suit :

```
        public enum StateRobot
        {
            STATE_ATTENTE = 0,
            STATE_ATTENTE_EN_COURS = 1,
            STATE_AVANCE = 2,
            STATE_AVANCE_EN_COURS = 3,
            STATE_TOURNE_GAUCHE = 4,
            STATE_TOURNE_GAUCHE_EN_COURS = 5,
            STATE_TOURNE_DROITE = 6,
            STATE_TOURNE_DROITE_EN_COURS = 7,
            STATE_TOURNE_SUR_PLACE_GAUCHE = 8,
            STATE_TOURNE_SUR_PLACE_GAUCHE_EN_COURS = 9,
            STATE_TOURNE_SUR_PLACE_DROITE = 10,
            STATE_TOURNE_SUR_PLACE_DROITE_EN_COURS = 11,
            STATE_ARRET = 12,
            STATE_ARRET_EN_COURS = 13,
            STATE_RECULE = 14,
            STATE_RECULE_EN_COURS = 15
        }
```

3.2.2 Pilotage

Après avoir implanté les fonctions de supervision du robot, vous allez à présent passer à l'implantation des fonctions de pilotage distant du robot. Pour cela , le microcontrôleur doit être capable de décoder les messages arrivant en provenance de l'interface en *C#*.

⟹ Analysez le code de réception et décodage des messages en *C#* et codez à votre tour la fonctions *UartDecodeMessage* en embarqué.

⟹ Appelez la fonction de décodage à chaque octet reçu dans la boucle principale du main

⟹ Supprimez la temporisation du *main* et le renvoi des trames reçues en mode *loopback*.

⟹ Validez son fonctionnement en envoyant des messages avec l'interface graphique et en vérifiant avec des points d'arrêts que le décodage se fait bien. Validez le résultat avec le professeur.

⟹ Ajoutez dans la fonction *UartProcessDecodedMessage* le code suivant :

```
void UartProcessDecodedMessage(unsigned char function,
                               unsigned char payloadLength, unsigned char payload[])
{
    //Fonction éappele èaprs le édcodage pour éexcuter l'action
    //correspondant au message çreu
    switch (msgFunction)
    {
        case SET_ROBOT_STATE:
            SetRobotState(msgPayload[0]);
            break;
        case SET_ROBOT_MANUAL_CONTROL:
            SetRobotAutoControlState(msgPayload[0]);
            break;
        default:
            break;
    }
```

```
}
```

⟹ Ajoutez au fichier *UART_Protocol.h*, les définitions suivantes :

```
#define SET_ROBOT_STATE 0x0051
#define SET_ROBOT_MANUAL_CONTROL 0x0052
```

⟹ Implantez en embarqué les fonctions *SetRobotState* et *SetRobotAutoControlState* qui doivent permettre le fonctionnement suivant :

- Par défaut le robot est en mode automatique, il réagit donc aux valeurs lues sur le télémètre.
- La fonction *SetRobotAutoControlState* permet de passer en mode manuel si la payload (1 octet) vaut *0* en provenance du *PC*. Elle permet de repasser en mode automatique si la payload vaut *1*. Une variable interne au *dsPIC* sera donc nécessaire pour stocker cet état, idem en *C#* où nous utiliserons un *bool* dénommé *autoControlActivated*. La fonction *SetNextRobotStateInAutomaticMode*, appelée dans la machine à état ne le sera désormais que si le robot est en mode automatique. Vous devez modifier le code en conséquence.
- La fonction *SetRobotState*, quant à elle, permet de forcer la machine à état du robot dans un état particulier, ce qui est utile en pilotage manuel du robot depuis l'interface.

3.2.3 Pilotage à l'aide d'un clavier

Vous allez à présent utiliser une bibliothèque externe vous permettant d'implanter des évènements clavier. Il est à noter que la gestion des évènements clavier existe en *C#* mais qu'elle ne fonctionne pas très bien car les évènements sont associés à un objet graphique qui doit être sélectionné pour que les évènements se déclenchent ! Ce mode étant très restrictif nous ferons ici appel à une bibliothèque externe, que vous allez apprendre à importer et à utiliser.

⟹ Téléchargez et copiez dans votre dossier de projet *C#* la bibliothèque suivante :
http://www.vgies.com/downloads/univ/ressources/projetrobot/keyboardHook/MouseKeyboardActivityMonitor.dll.

⟹ Dans l'onglet *Références* de l'*Explorateur de solutions*, cliquez-droit et ajoutez une Référence. Pour cela allez dans parcourir, et sélectionnez le fichier *MouseKeyboardActivityMonitor.dll* que vous venez d'ajouter à votre projet. A ce stade, *MouseKeyboardActivityMonitor* doit apparaître dans la liste des références du projet.

⟹ Pour utiliser la bibliothèque référencée précédemment, il faut à présent le dire explicitement au programme. Pour cela, ajoutez au code de *Form1.cs* les appels aux bibliothèques suivants :

```
using MouseKeyboardActivityMonitor.WinApi;
using MouseKeyboardActivityMonitor;
```

⟹ Ajoutez à présent à la classe *Form1*, un objet chargé de surveiller les appuis sur le port série. Le code à insérer est le suivant, et doit se placer juste avant le constructeur de la classe *Form1()* :

```
    private readonly KeyboardHookListener m_KeyboardHookManager;
```

⟹ A la fin du constructeur de la classe *Form1*, ajoutez et expliquez en détails ce que fait le code suivant :

```
    m_KeyboardHookManager = new KeyboardHookListener(new GlobalHooker());
    m_KeyboardHookManager.Enabled = true;
    m_KeyboardHookManager.KeyDown += HookManager_KeyDown;
```

⟹ Ajoutez à présent une méthode permettant de gérer les évènements clavier à l'aide du code suivant, en expliquant ce que fait ce code en détails :

```
private void HookManager_KeyDown(object sender, KeyEventArgs e)
    {
    if (autoControlActivated == false)
    {
        switch (e.KeyCode)
        {
            case Keys.Left:
                UartEncodeAndSendMessage(0x0051, 1, new byte[] {
                    (byte)StateRobot.STATE_TOURNE_SUR_PLACE_GAUCHE });
                break;
            case Keys.Right:
                UartEncodeAndSendMessage(0x0051, 1, new byte[] {
                    (byte)StateRobot.STATE_TOURNE_SUR_PLACE_DROITE });
                break;
            case Keys.Up:
                UartEncodeAndSendMessage(0x0051, 1, new byte[]
                    { (byte)StateRobot.STATE_AVANCE });
                break;
            case Keys.Down:
                UartEncodeAndSendMessage(0x0051, 1, new byte[]
                    { (byte)StateRobot.STATE_ARRET });
                break;
            case Keys.PageDown:
                UartEncodeAndSendMessage(0x0051, 1, new byte[]
                    { (byte)StateRobot.STATE_RECULE });
                break;
        }
    }
}
```

$\Longrightarrow$ Validez que vous entrez bien dans la fonction précédente quand vous appuyez sur une des flèches du clavier, que vous soyez dans l'application *C#* ou pas. Validez ensuite que le robot effectue bien les mouvements voulus.

$\Longrightarrow$ Insérer dans le fichier *UART_Protocol.c* le code correspondant à la réception de l'ordre précédent et à l'envoi du message de test.

Nous sommes arrivés au terme de la mise en oeuvre du bus UART utilisant des messages formatés et robustes. Ce bus est utilisé pour superviser le robot et pour le piloter en temps réel depuis le *PC*.