Projet
conception d'un robot mobile

Professeur : Valentin Gies

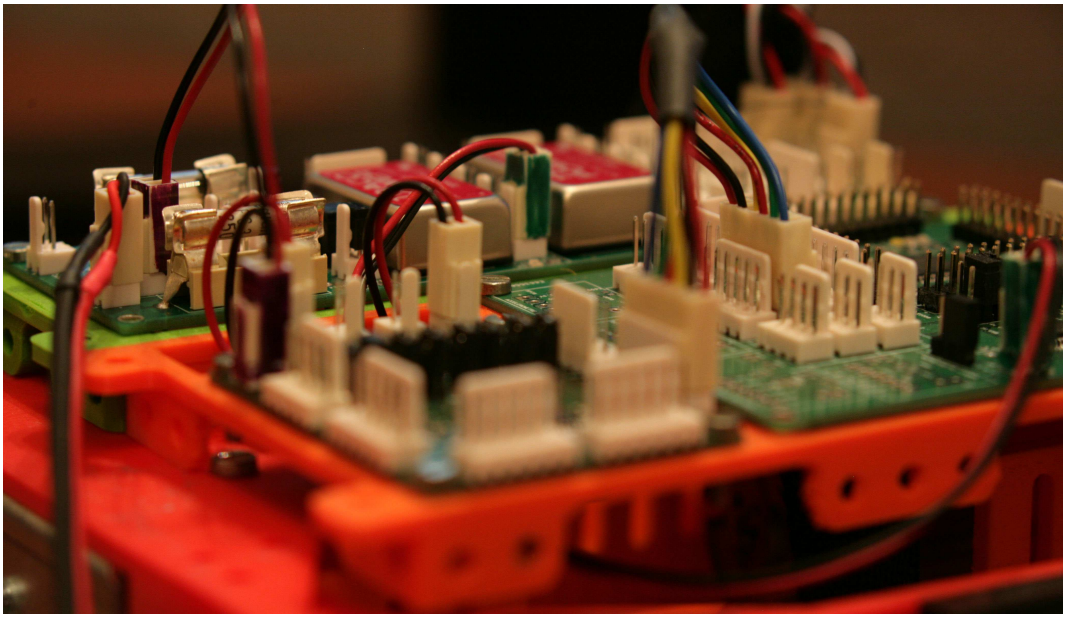

Table des matières

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 en *WPF* (Windows Presentation Foundation). 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*. Choisir *Application WPF (.NET Framework)*.
Avant de créer le projet, donnez lui un nom explicite tel que *RobotInterface* et spécifiez un chemin d'accès sur le disque dur tel que *C :/Projets/RobotWPF/*. Créez le projet, vous devriez avoir un écran similaire à celui de la figure 1.

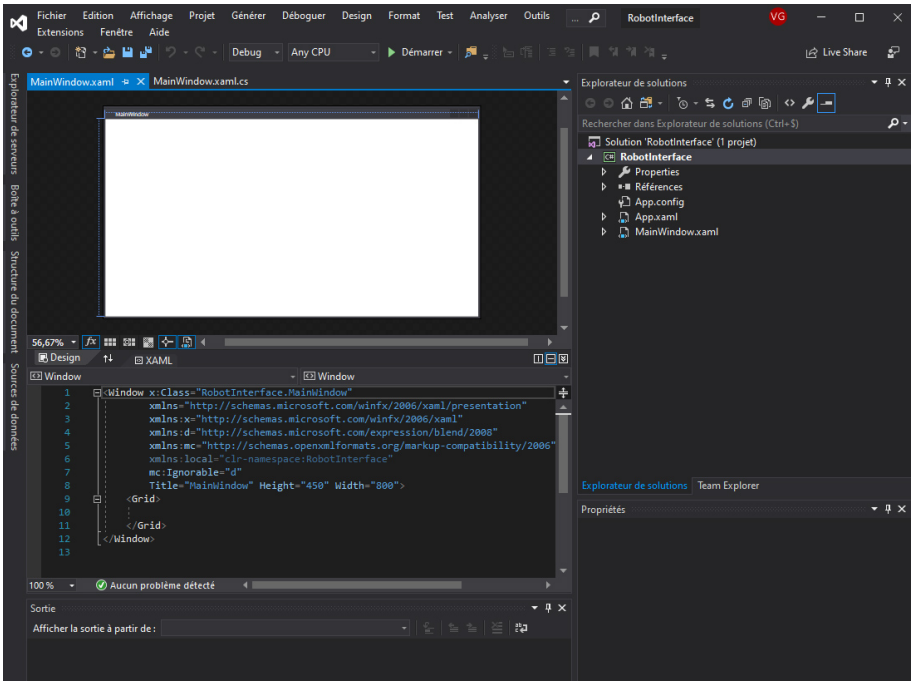

FIGURE 1 – Projet Visual Studio nouvellement créé

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 (fichiers *XAML*) en *C# WPF*. A la création du projet, un écran graphique dénommé *MainWindow.xaml* est créé par défaut. Il apparaît à gauche de l'écran en version graphique (en haut) et code XAML (en bas). Dans son état de base, il continet juste une grille (*Grid*) vide.

⟹ Ajoutez à présent à *Form1* deux objets de type *GroupBox* en les faisant glisser depuis la *Boite à outils* → *Conteneurs*. Une *GroupBox* est une boite destinée à contenir d'autre éléments graphiques. Par défaut sa bordure est transparente. En regardant le code XAML, vous pouvez vous apercevoir que les GroupBox ont été ajoutées au code comme indiqué à la figure Figure 2 dans la partie centrale à gauche. Il est important de noter que le XAML décrit totalement ce qui apparait graphiquement dans la partie du dessus.

⟹ En cliquant dans l'une des *GroupBox* dans la partie graphique, vous pouvez à présent modifier leurs propriétés. Celles-ci apparaissent en bas à droite de l'écran comme indiqué à la Figure 2. Vous pouvez par exemple mettre une couleur de fond (dans la catégorie Pinceau, définir une couleur uniforme de *Background* valent #FFDDDDDD par exemple) et une bordure noire (dans la catégorie Pinceau, mettre *BorderBrush* en couleur uniforme à #FF000000 par exemple). Faire de même pour la seconde *GroupBox*.

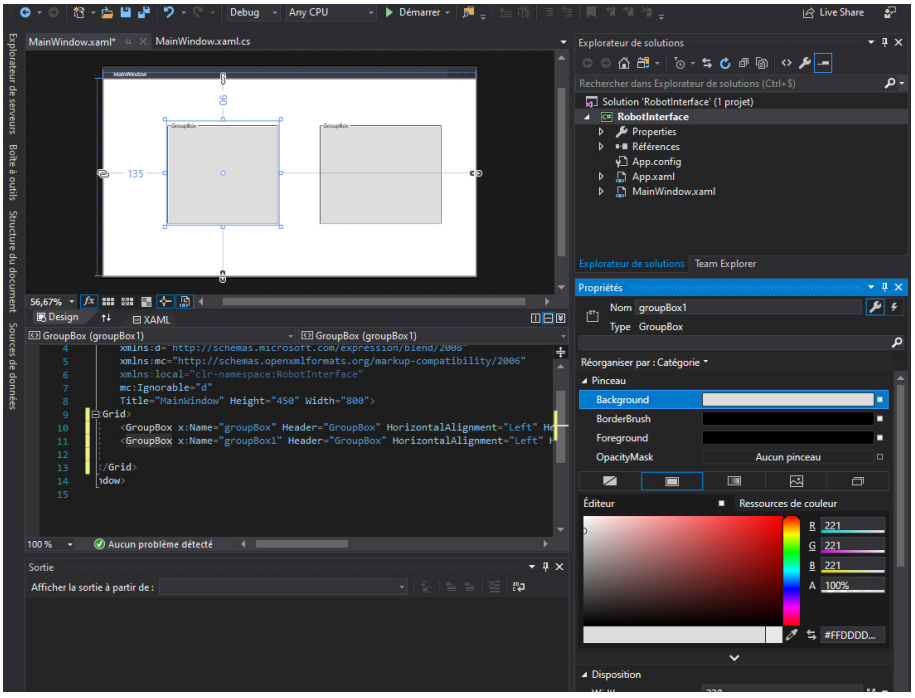

FIGURE 2 – Modification des propriétés de la GroupBox

⟹ A présent, vous devez avoir compris comment modifier les propriétés d'objets graphiques en WPF. Par vous même, renommez la *GroupBox* de gauche en *Emission* et celle de droite en *Réception*.

⟹ Arrivés à ce stade, vous pouvez voir à quoi ressemble votre application en la lançant. Pour cela appuyez sur *F5* ou *Démarrer* (flèche verte).

Si vous redimensionner votre fenêtre lors de l'exécution de l'application, vous devez constater que les GroupBox ne se redimensionnent pas et restent en position initiale. C'est regrettable et peu conforme aux standards des applications modernes. Le *WPF* est fait pour gérer simplement ces situations, ce qui lui confère un avantage indéniable par rapport au *WinForms* classiques. Pour ce faire, nous allons pousser plus loin le concept de grille introduit tout au début de cette partie.

⟹ Après avoir sélectionné la grille de fond d'application, cliquez sur le configurateur de Collones (*ColumnDefinitions*) dans les propriétés de la grille, onglet *Disposition* comme indiqué à la figure 3. Un éditeur de collections s'ouvre. Ajouter 5 ColumnDefinition (5 colonnes). Dans la première, la 3e et la 5e, spécifier une largeur de 30 pixels. Dans la 2e et la 4e, spécifiez une largeur de 1 Star. Vous devriez avoir une mise en forme des colonnes

ressemblant à la figure figure 3.

Quelques explications sont nécessaires : une largeur en *pixels* sera fixe quelque soit la taille de la fenêtre de l'application. Une largeur en *Star* sera dépendante de la taille de la fenetre de l'application : dans notre cas, si la fenêtre a une largeur de 590 pixels, une fois retiré les 3 colonnes de 30 pixels fixes, il reste 500pixels à répartir entre deux collonnes de même taille (1 *Star* chacune).

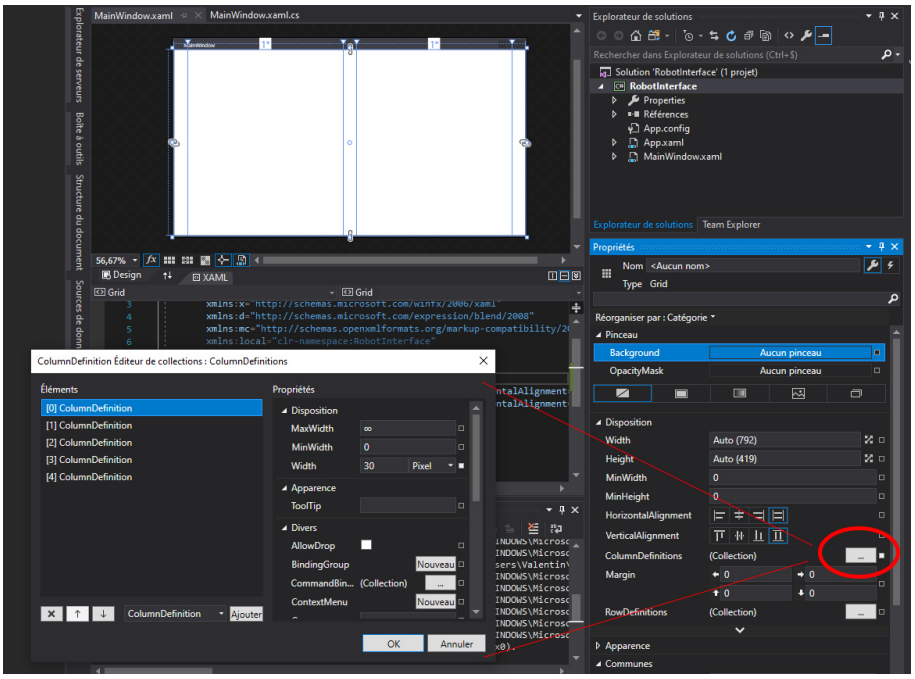

FIGURE 3 – Modification des propriétés des colonnes d'une *Grid WPF*

⟹ Maintenant que vous maitrisez les colonnes, faire de même avec les lignes (*Rows*), et définissant une première et une 3e ligne de 30 *pixels*, et une 2e ligne de 1 *Star*.

⟹ Il est temps à présent d'ancrer les GroupBox initialement créées dans les cases de la Grid. Pour cela sélectionnez l'une des GridBox (soit en passant avec la souris au dessus de l'endroit ou elle doit être, même si elle est cachée, soit en cliquant sur la ligne de la GroupBox dans le XAML). Glissez la dans la case de la grille ou vous voulez la mettre (ici une des grandes cases à redimensionnement automatique). Dans l'onglet *Disposition* des *Propriétés*, définissez la largeur et la hauteur en automatique, puis mettez les marges à 0 dans toutes les directions et les alignements horizontaux et verticaux à *Stretch*. Vous devriez avoir des propriétés proches de celles de la figure 4.

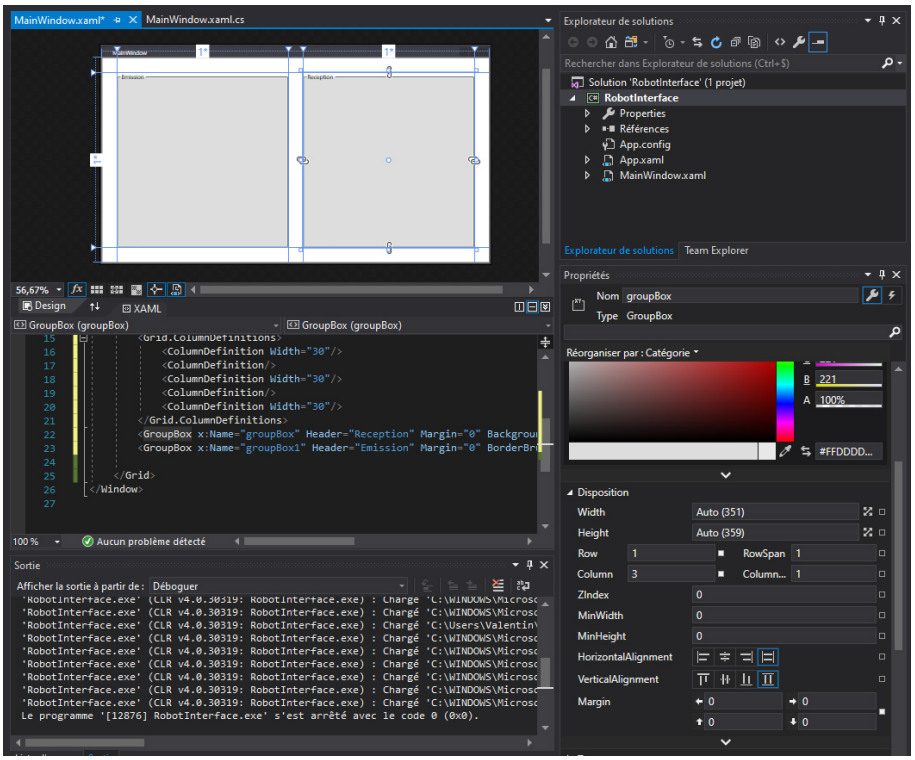

FIGURE 4 – GroupBox WPF en version redimensionnement automatique en fonction de la taille de l'application

Si vous exécutez le programme, vous pouvez vérifier que les *GroupBox* sont bel et bien redimensionnées dynamiquement si on change la taille de la fenêtre.

⟹ A présent, vous maitrisez le placement de composants dans une fenêtre de taille dynamique. Ajoutez à la *GroupBox Emission* une *TextBox* depuis la *Boite à Outils*. Faites en sorte que cette TextBox prenne toute la place possible dans la GroupBox (*Width* et *Height* en *Auto*, *Margins* à 0, et *Alignement* à *Stretch*), et supprimez sa couleur de fond et de bordure. Changez le nom de cette *TextBox* en *textBoxEmission*. Enfin, supprimez le texte par défaut *TextBox* dans les propriétés communes, et mettez la propriété *AcceptsReturn* à true (pour permettre des texte de plusieurs lignes).

⟹ Faites de même avec une *TextBox* de réception, en mettant en plus la propriété *IsReadOnly* à true afin d'empêcher l'utilisateur d'écrire dans cette *RichTextBox*. Exécutez le code, vous pouvez écrire dans la fenêtre d'envoi mais pas dans celle de réception.

⟹ Vous allez à présent rajouter un bouton à la grille pour envoyer les messages écrits dans la *TextBox* d'émission. Pour cela modifier la grille en rajoutant deux lignes de hauteur fixe égale à 30 pixels. Insérer en suite depuis la ToolBox un Bouton à l'avant dernière ligne sous la *RichTexBox* d'émission comme indiqué à la figure5. Redimensionner ce bouton de manière à ce qu'il fasse la hauteur de la case dans laquelle il est inséré et qu'il soit centré horizontalement avec une largeur fixe égale à 100 pixels. Renommer le bouton en *buttonEnvoyer* et changer son texte (*Content*) en *Envoyer*.

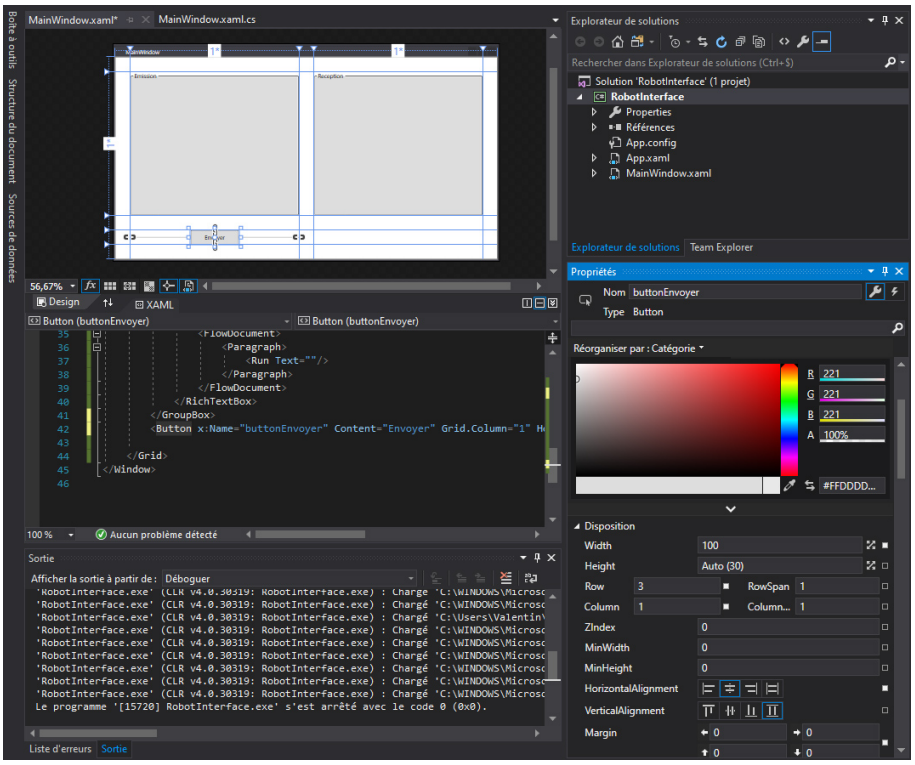

FIGURE 5 – Insertion du bouton d'envoi

⟹ Une fois les propriétés du bouton d'envoi définies, vous pouvez lui faire déclencher des actions. Pour cela, dans les propriétés, 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 vide située à droite de l'évènement *Click*. Une fenêtre dénommée *MainWindow.xaml.cs* a du s'ouvrir dans la fenêtre principale de *Visual Studio* comme indiqué à la figure 6. Cette fenêtre montre le code associé à l'écran graphique XAML *MainWindow* sur lequel nous avons travaillé jusqu'ici : ce code est dénommé *Code Behind* de la fenêtre.

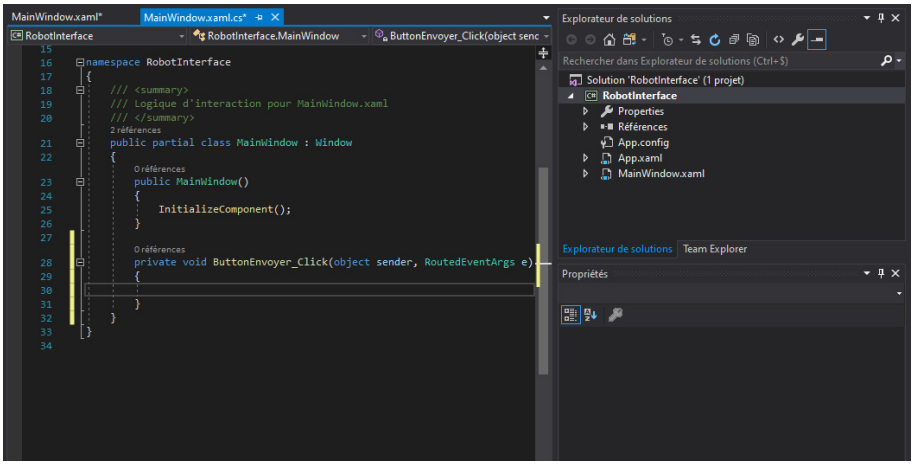

FIGURE 6 – Code Behind du bouton d'envoi

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, RoutedEventArgs e)
```

```
        {
            buttonEnvoyer.Background = Brushes.RoyalBlue;
        }
```

⟹ Exécutez le programme et 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 *TextBox* d'émission vers la *TextBox* de réception. Pour cela dans la fonction *buttonEnvoyer_Click*, rajouter du code permettant de récupérer le texte de la TextBox d'émission pour le placer dans la TextBox de réception, précédé d'un retour à la ligne et de la mention : "Recu : ". La *TextBox* d'emission doit également être vidée.

Le comportement doit être proche de celui de la figure 7 où 4 messages ont été envoyés successivement. Valider avec le professeur.

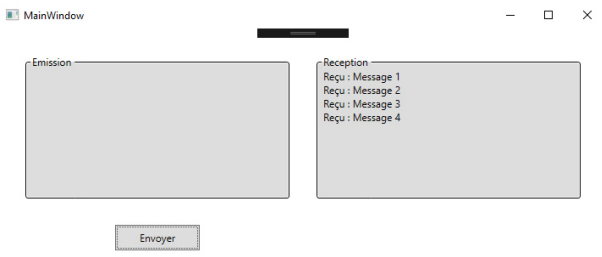

FIGURE 7 – Exemple d'exécution du simulateur de messagerie

⟹ 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 *TextBox* d'émission, en vérifiant que la source de l'appui soit le bouton *Entrée*. Rajoutez à la TextBox d'émission un *évènement (éclair) KeyUp*.

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

```
if (e.Key == Key.Enter)
{
    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 modules *FT232RL*. Ces modules permettent de faire sortir ou rentrer dans chacun PC un flux série, à l'aide d'une connexion USB au niveau du

PC.

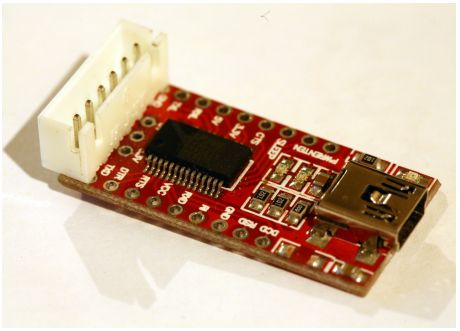

FIGURE 8 – Module FT232RL

L'envoi et la réception des données se fait par l'intermédiaire d'un objet de type *SerialPort* en *C#*.

Attention : l'implantation de *SerialPort* dans la librairie fournie par défaut (*System.IO.Ports*) est inutilisable en *C#* avec WPF. Il faudra donc télécharger une librairie de remplacement à l'adresse suivante, et ajouter une référence vers celle-ci au projet. Si besoin, demandez au professeur qui pourra en particulier vous expliquer son fonctionnement.

⟹ Créez un objet de type *ReliableSerialPort* dans le code behind du l'interface graphique. Cet objet doit être déclaré avant le constructeur de la classe *MainWindow*.

```
SerialPort serialPort1;
```

Initialisez le serial port dans le constructeur de l'interface graphique (*MainWindow*). Vous noterez que le constructeur du *ReliableSerialPort* à 5 arguments :

- le nom du port, *"COMX"* où *X* est le numéro du port correspondant à l'adaptateur USB/Série que vous trouverez dans le *Gestionnaire de périphériques* de *Windows*.
- la vitesse du port, ici *115200* bauds.
- la parité du port, ici *None*.
- le nombre de bits des datas, ici *8*.
- le type de StopBits, ici *One*.

Ouvrez ensuite le port pour qu'il soit utilisable.

```
serialPort1 = new ReliableSerialPort("COM3", 115200, Parity.None, 8, StopBits.One);
serialPort1.Open();
```

⟹ A présent, modifiez le code de la fonction SendMessage vue précédemment de manière à ce que les envois se fassent sur le port série en utilisant la méthode *WriteLine* de l'objet *SerialPort*. Les données envoyées sur le port série sont visualisables à l'aide d'un oscilloscope. La figure 9 montre les pins du module. La pin *Rx* est celle sur laquelle les données en provenance du PC sont reçues, la pin *Tx* est celle sur laquelle il faut écrire pour transmettre le flux série au PC.
Si tout est correct, quand vous envoyez un message sur le port série, une *LED* rouge doit clignoter sur les modules à chaque envoi. Vérifiez que des données passent effectivement.

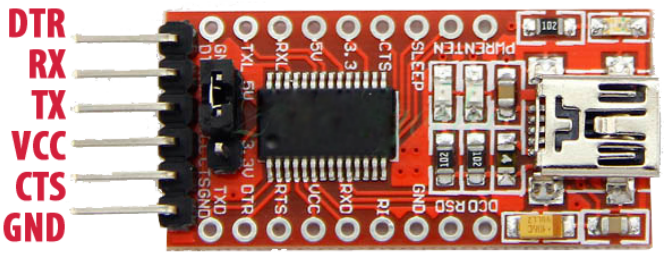

FIGURE 9 – Pins du module FT232RL

⟹ Voir le passage des données est une chose, voir quelles données passent est plus intéressant et vous allez le mettre en oeuvre. Configurez 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 (demander au professeur si besoin), vérifiez que les données envoyées correspondent aux données reçues par l'oscilloscope en observant la sortie *Tx* du module.

A présent, l'envoi des données depuis le PC étant testé, il reste à valider la réception des données sur le PC. Pour cela, nous allons renvoyer les données envoyées par la pin *Tx* du port série vers la pin *Rx* de ce même port série afin de recevoir sur le PC les données que nous envoyons de ce même PC. Ce mode de fonctionnement s'appelle le mode *LoopBack*, il permet de valider son code sans avoir besoin d'un second ordinateur.

⟹ Connectez un connecteur de loopback comme indiqué à la figure 10 en mode *Loopback*.

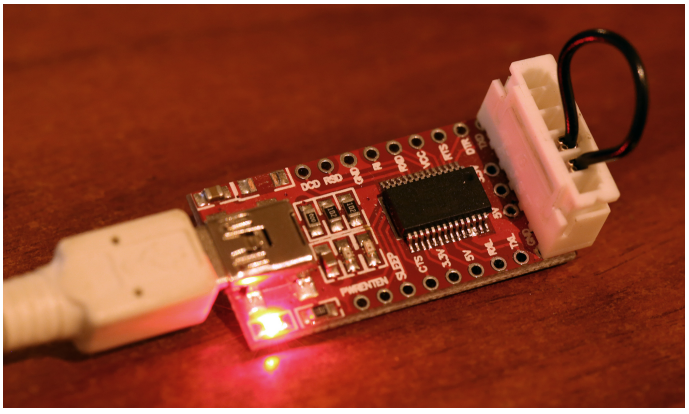

FIGURE 10 – Module FT232RL avec connecteur LoopBack

⟹ Afin de gérer la réception des données entrantes sur le port série, enregistrez un *callback SerialPort1_DataReceived* juste après l'initialisation du port série. L'enregistrement d'un *callback* est équivalent en code à l'activation d'un évènement depuis l'interface graphique comme déjà vu (avec les boutons par exemple). La fonction *callback* est appelée de manière automatique lorsque des données sont reçues sur le port série (évènement *DataReceived*). Il est à noter que le code de la fonction *callBack* appelée peut s'ajouter automatiquement dans le code en appuyant sur TAB lorsqu'on tape au clavier *+=* dans la ligne précédente. Laissez pour l'instant cette fonction vide (supprimer l'exception ajoutée par défaut).

```
serialPort1 = new SerialPort("COM4", 115200, Parity.None, 8);
serialPort1.DataReceived += SerialPort1_DataReceived;
serialPort1.Open();

public void SerialPort1_DataReceived(object sender, DataReceivedArgs e)
{  }
```

$\Longrightarrow$ En envoyant un message, *dongle USB* branché en mode loopback, vérifiez en ajoutant un point d'arrêt dans le code (*F9*), que vous passez dans cet évènement.

$\Longrightarrow$ Récupérez à présent les données disponibles dans l'évènement *SerialPort1_DataReceived* en ajoutant le code ci-dessous. Il est censé récupérer un tableau d'octet transmis dans l'évènement (en quelque sorte le contenu du paquet) et le convertir en string avant de l'afficher dans la *TextBox* de réception. Que se passe-t-il ? Les explications vous sont données ci-dessous.

```
public void SerialPort1_DataReceived(object sender, DataReceivedArgs e)
{
        textBoxReception += Encoding.UTF8.GetString(e.Data, 0, e.Data.Length);
}
```

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 threads apportent des contraintes dans la programmation, telles que celle que vous venez de rencontrer. Vous avez du lever l'exception suivante :

> *System.InvalidOperationException : Le thread appelant ne peut pas accéder à cet objet parce qu'un autre thread en est propriétaire.*

L'erreur déclenchée à l'exécution vous indique qu'il est impossible de mettre à jour un objet (en l'occurrence la *TextBox*) 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.

$\Longrightarrow$ Il est à présent nécessaire de regarder à intervalle régulier depuis l'interface graphique si la chaine *receivedText* contient quelque chose afin de l'afficher. Pour cela on utilisera un timer un peu particulier puisque lié au thread graphique : un *DispatcherTimer* que l'on nommera timerAffichage.

```
DispatcherTimer timerAffichage;
```

Le DispatcherTimer est inclus dans une librairie *C#* dénommée *System.Windows.Threading* que l'on appellera comme suit au début du code :

```
using System.Windows.Threading;
```

$\Longrightarrow$ A présent, initialisez le timerAffichage à l'aide du code suivant :

```
timerAffichage = new DispatcherTimer();
timerAffichage.Interval = new TimeSpan(0,0,0,0,100);
timerAffichage.Tick += TimerAffichage_Tick;
timerAffichage.Start();
```

L'intervalle de temps du *DispatcherTimer* est de type *TimeSpan*. Regarder sur internet à quoi correspond ce type qui permet de décrire des durées, et regardez en même temps à quoi correspond le type *DateTime* qui servira plus tard.

$\Longrightarrow$ La fonction *callback TimerAffichage_Tick* a du être ajoutée à votre code automatiquement si vous avez recopié le code précédent. Sinon, supprimez le *"+= TimerAffichage_Tick;"* et retapez manuellement += suivi de TAB pour faire l'ajout automatiquement. Demander au professeur en cas de souci. Dans la fonction obtenue et appelée périodiquement par le *Timer*, regardez si la chaîne *receivedText* contient quelque chose, et dans ce cas affichez ces données dans la TextBox de réception. 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* dans l'interface graphique permettant de vider la *textBox* 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 dans *e.Data* l'un après l'autre

dans la *Queue*.

$\Longrightarrow$ Sur les évènements *Timer*, récupérez les *bytes* de la *Queue* un par un et affichez les dans la *textBox* 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.