Pilotage et supervision d'un robot en C#

AUTET Maeva CARPENTIER Lucas COURTOIS Thibault
SYSMER 2A

Responsable : M. Gies

27 octobre 2024

Introduction

Durant ces séances de travaux pratiques, nous avons pu aborder la programmation en orientée objet en C# avec des interfaces graphiques en WPF (Windows Presentation Foundation). Notre but était de réaliser un terminal permettant l'envoi et la réception de messages sur le port série du PC. Dans un premier temps, ce terminal nous servira de 'messagerie instantanée' entre deux PC reliés par un câble série. Dans un second temps, il nous servira à piloter un robot mobile tout en pouvant observer son comportement interne. Notre objectif à travers ce compte-rendu sera alors d'expliquer les étapes de cette réalisation en explicitant les solutions que nous avons choisies.

Table des matières

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

1.1 Développement d'une interface de messagerie instantanée

Notre première tâche consistait à élaborer un simulateur de messagerie instantanée au sein de notre interface utilisateur. Pour ce faire, nous avons exploité la ToolBox de Visual Studio afin d'intégrer deux GroupBox à notre interface. Ces dernières ont été destinées à afficher respectivement les messages envoyés et reçus. Par la suite, l'ajout d'un bouton "Envoyer" a été réalisé. Ce bouton, une fois programmé, permet l'envoi du contenu saisi dans la RichTextBox d'émission vers celle de réception en un simple clic. Il est à noter que le contenu de la RichTextBox d'émission est automatiquement effacé après chaque envoi. Nous avons également apporté des modifications au code de la fonction initiale pour que la couleur du bouton "Envoyer" se modifie après chaque action d'envoi. Pour optimiser l'expérience utilisateur, une fonction spécifique a été mise en place. Celle-ci autorise l'envoi du texte de l'émission vers la réception en utilisant directement la touche "Entrée" du clavier, rendant l'utilisation plus fluide et intuitive.

Voici un aperçu de notre interface finale, c'est elle qui nous permettra de faire le lien entre l'utilisateur et la machine :

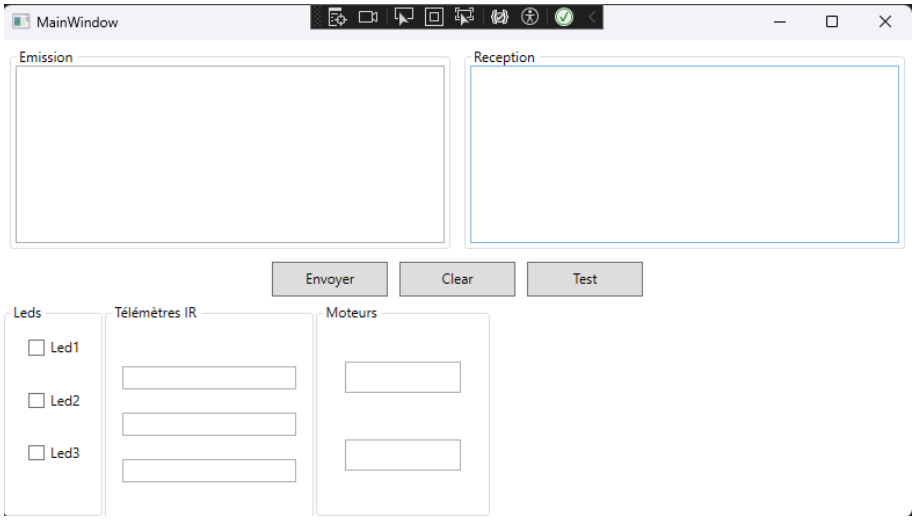

FIGURE 1 – Interface faites en WPF

1.2 Messagerie instantanée entre deux PC

Nous avons débuté par configurer la communication série entre deux PC à l'aide de modules FT232RL. Ces dispositifs convertissent les signaux USB des ordinateurs en flux série et inversement ce qui facilite notre connexion.

Face aux limitations de la librairie SerialPort standard dans l'environnement WPF, nous avons opté pour une librairie alternative compatible avec .NET6. Cette démarche a nécessité de rechercher et télécharger une librairie de remplacement adéquate, que nous avons ensuite intégrée à notre projet. Après avoir téléchargé la librairie, nous l'avons ajoutée à notre solution Visual Studio. Cette intégration nous a permis d'accéder aux fonctionnalités avancées de communication série nécessaires pour notre application. Nous avons ensuite instancié un objet ExtendedSerialPort dans notre code, en le configurant avec les paramètres nécessaires tels que le nom du port, la vitesse, la parité etc..

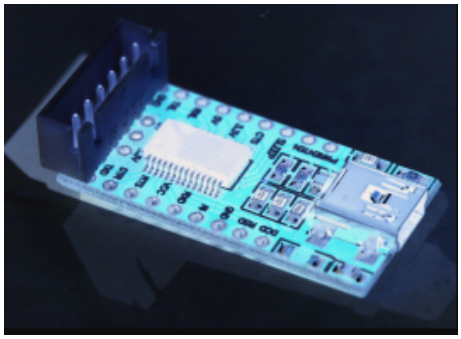

FIGURE 2 – Module FT232RL

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

Pour envoyer des messages via le port série, nous avons modifié notre fonction d'envoi initiale pour utiliser la méthode WriteLine de l'objet ExtendedSerialPort. Pour s'assurer de ce fonctionnement, nous pouvions observer la LED Rouge qui clignotait à chaque envoie d'un message sur le port Série. Cependant, nous voulions voir les données qui transitaient sur le port Série, l'utilisation d'un oscilloscope nous a permis de confirmer que les données transmises correspondaient aux données reçues par l'oscilloscope. Afin de valider la réception des données sans nécessiter un second ordinateur, nous avons mis en place un test en mode LoopBack, connectant la pin Tx du port série à la pin Rx de ce même module. Ce test nous permet alors de vérifier la réception de nos données sans avoir un second ordinateur.

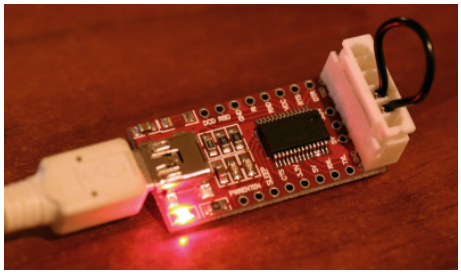

FIGURE 3 – Module FT232RL avec connecteur LoopBack

Pour gérer la réception de message sur le port série, nous enregistrons un callback. C'est une fonction qui sera appelée directement lorsqu'un message sera reçu

```
serialPort1 = new ReliableSerialPort("COM3", 115200, Parity.None, 8, StopBits.One);
serialPort1.DataReceived += SerialPort1_DataReceived;
serialPort1.Open();

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

Cependant, une erreur s'affiche lorsque nous utilisons cette méthode. Cette erreur nous indique qu'il est impossible de mettre à jour un objet (ici la TextBox) directement géré par un thread (celui de l'affichage) à partir d'un autre thread (celui du port série). Pour corriger cela, on crée une chaine de caractère dans laquelle on va venir ajouter les messages envoyer en attente de leur traitement. Pour vérifier si la chaîne de caractère contient quelque chose, on vérifie cela grâce à un *DispatcherTimer*. A présent nous pouvons alors communiquer avec un ordinateur voisin !

1.3 Structuration du code à l'aide d'une classe Robot

Pour améliorer la structure du code, nous créons une classe robot qui aura pour but de recueillir toutes les informations en provenance du robot. Nous instancions alors un objet robot dans la MainWindow.

```
public class Robot
{
    public string receivedText = "";
    public float distanceTelemetreDroit;
    public float distanceTelemetreCentre;
    public float distanceTelemetreGauche;

    public Robot()
    {
    }
}
```

1.4 Liaison série hexadécimale

Jusqu'à présent notre système de messagerie permet de communiquer par envois de chaîne de caractères mais pas avec n'importe quels caractères ASCII. Nous allons donc évoluer vers une liaison série qui permet l'envoi d'octet peu importe leur valeur. Nous créons alors une FIFO qui nous servira de buffer. Les premières données entrées seront les premières à sortir de la file d'attente.

```
public class Robot
{
    public string receivedText = "";
    public float distanceTelemetreDroit;
    public float distanceTelemetreCentre;
    public float distanceTelemetreGauche;

    public Robot()
    {
    }
}
```

```
public void SerialPort1_DataReceived(object sender, DataReceivedArgs e)
{
    for (int i = 0; i < e.Data.Length; i++)
    {
        robot.byteListReceived.Enqueue(e.Data[i]);
    }
}
```

Pour lire ces données, nous utilisons la fonction *ToString()* qui permet de convertir les données dans la base que l'on veut. Nous pouvons alors maintenant communiquer et interpréter le message reçu.

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

2.1 Principes de base de la liaison série en embarqué

Dans cette phase du projet, notre but était d'établir une communication série entre la carte principale de notre système embarqué et un ordinateur en sachant qu'il n'y a pas de liaison USB

directe ou de convertisseur USB-série sur la carte principale. Pour y parvenir, nous avons opté pour l'interconnexion de la carte principale à une carte de capteurs, elle-même connectée au PC via un dongle USB/Série précédemment utilisé. Nous avons crée un fichier 'UART.c'. Ce fichier contenait le code essentiel pour initialiser l'unité UART à une vitesse de 115200 bauds, en évitant l'usage des interruptions.

2.2 Échange de données entre le micro contrôleur et le PC

2.2.1 Émission UART depuis le micro contrôleur

Nous avons branché le câble à partir du convertisseur série-USB vers les pins Rx et Tx de l'UART1 du micro contrôleur. Avec l'aide du schéma de câblage du micro contrôleur, nous avons trouvé que le pin remappable correspondant à Rx était le RP24 et que celui correspondant à Tx était le RP36.

Nous voulons maintenant envoyer des messages et les visualiser, pour cela nous ajoutons une fonction d'envoi de message dans le fichier "UART.c" et nous appelons cette fonction dans la boucle infinie du 'main' de notre programme avec d'un délais. Cela nous permet alors d'observer notre message à l'oscilloscope.

```
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
    }
}
```

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

Nous avons oublié prendre une photo de l'oscilloscope à ce moment précis mais le résultat ressemble à ceci :

FIGURE 4 – Message vu à l'oscilloscope

2.2.2 Réception

Pour confirmer que la réception via le port série du micro contrôleur fonctionne correctement, nous avons décidé de le faire fonctionner en mode LoopBack. Pour ce faire, notre stratégie consistait à renvoyer immédiatement sur Tx tout caractère reçu en Rx. Nous avons utilisé l'interruption UART en réception pour détecter l'arrivée de caractères. Nous devons en premier autoriser les interruptions en réception sur l'UART et nous ajoutons ce code pour faire la routine d'interruption :

```
//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;
    }
}
```

Cette fonction sera automatiquement appelée dès qu'un caractère sera envoyé et pourra donc être traité. Nous pouvons tester ceci en envoyant des messages sur l'interface C#.

Voici ce que nous obtenons :

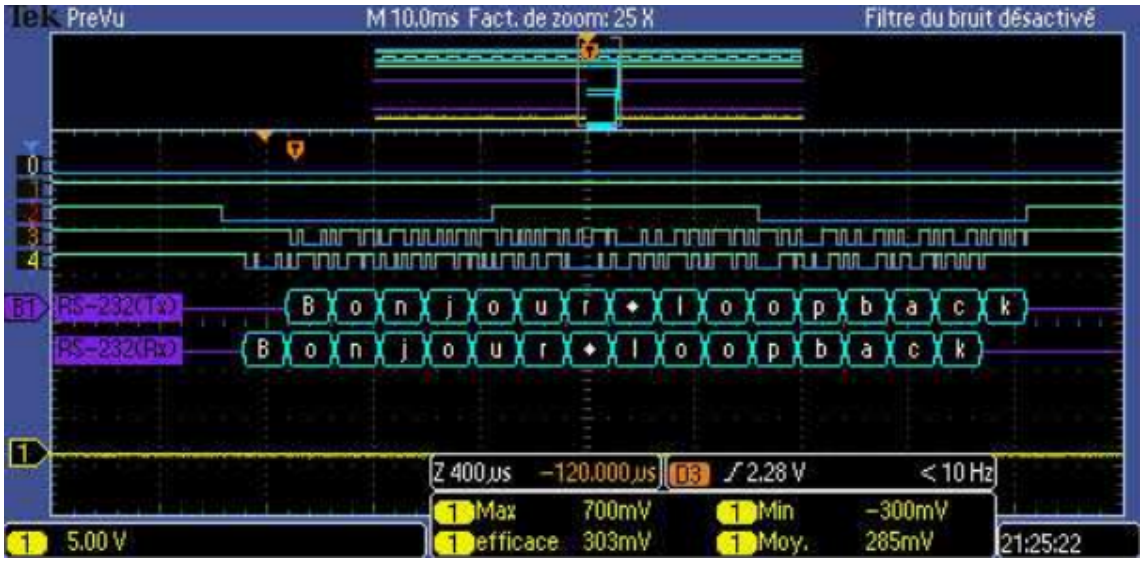

FIGURE 5 – Message vu à l'oscilloscope

On voit bien le "bonjour" en émission et le même "bonjour" en réception. La réception fonctionne, il faut maintenant que l'on puisse stocker les données envoyées le temps qu'elles puissent être traitées.

2.3 Liaison série avec FIFO intégré

2.3.1 Le buffer circulaire de l'UART en émission

Notre fonction qui s'occupe de l'envoi des messages fonctionne mais le gros inconvénient est qu'elle bloque la boucle principale du programme jusqu'à la fin de l'envoi du message. Cela est assez contraignant, pour palier ce problème nous allons implémenter un buffer circulaire qui stockera les données à envoyer sur le port Série en attendant leur envoi. La contrainte est que cet envoi doit se faire entièrement en mode interruption.

Voici un schéma d'un buffer circulaire :

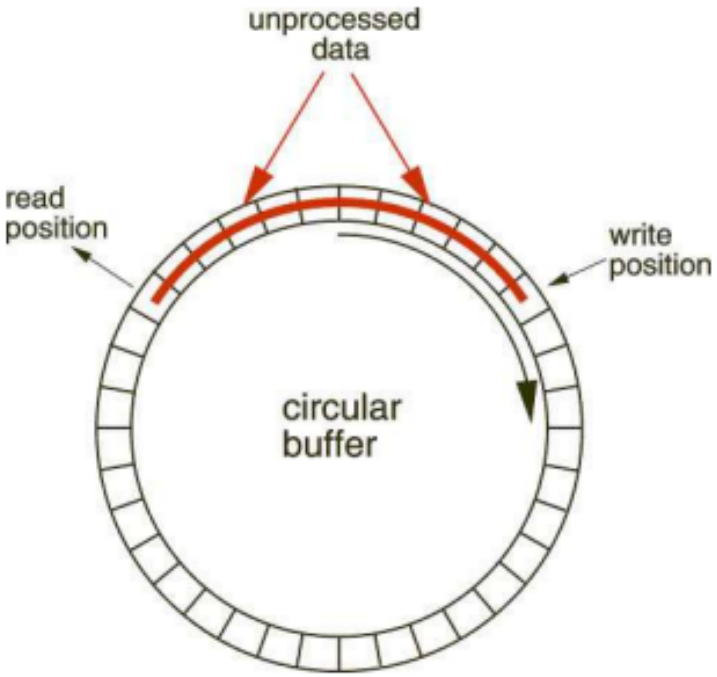

FIGURE 6 – Message vu à l'oscilloscope

Un buffer circulaire est une structure de données utilisée pour stocker et gérer les données de manière séquentielle dans des situations où les opérations d'écriture et de lecture se produisent à des vitesses différentes ou de manière asynchrone. Il s'agit enfaîte d'une file d'attente qui forme un cercle où l'élément suivant à remplir vient juste après le dernier élément, à condition qu'il n'y ait pas de donnée non lue à cet emplacement. Dans notre projet, lorsque des données arrivent au port série, elles sont stockées dans ce buffer circulaire jusqu'à ce qu'elles soient traitées. Cela permet à notre micro contrôleur de continuer à recevoir des données même pendant le traitement des données précédemment reçues. L'avantage d'un buffer circulaire est sa capacité à réutiliser l'espace de stockage de manière efficace. Une fois que les données ont été lues et traitées, cet espace peut être réutilisé pour de nouvelles données entrantes.

Les scripts de ces programmes étant très long, nous nous contenterons d'expliquer le fonctionnement général du code. L'entièreté des codes est évidemment accessible à l'adresse GitHub indiquée en Annexe.

Le buffer de transmission est configuré avec une capacité de 128 octets, avec les variables *cbTx1Head* et *cbTx1Tail* servant à marquer le début et la fin des données à transmettre. La fonction *SendMessage* joue un rôle clé en ajoutant des messages dans le buffer, à condition qu'il y ait

suffisamment d'espace, vérifié via *CB_TX1_GetRemainingSize*. Si l'espace est disponible, le message est ajouté dans le buffer par la fonction *CB_TX1_Add*, qui place chaque octet du message à l'indice actuel de *cbTx1Head* et incrémente cet indice. Si *cbTx1Head* dépasse la limite du buffer, il revient à zéro, facilitant ainsi une utilisation circulaire du buffer. La fonction *CB_TX1_Get* est utilisée pour extraire le prochain octet à transmettre du buffer, en lisant la valeur à l'indice actuel de *cbTx1Tail*, puis en incrémentant cet indice, avec une réinitialisation à zéro si nécessaire. L'interruption *_U1TXInterrupt* est configurée pour gérer la transmission des données. Elle vérifie si *cbTx1Tail* n'est pas égal à *cbTx1Head*, indiquant que des données sont prêtes à être envoyées. Si c'est le cas, la fonction *SendOne* est appelée pour transmettre un caractère. La variable *isTransmitting* indique si une transmission est en cours, permettant de gérer efficacement le flux de données. Les fonctions *CB_TX1_GetDataSize* et *CB_TX1_GetRemainingSize* calculent respectivement la quantité de données en attente dans le buffer et l'espace disponible restant, assurant une gestion optimale du buffer de transmission pour éviter tout débordement ou perte de données importantes.

2.3.2 Le buffer circulaire de l'UART en réception

Le buffer est défini avec une taille de 128 octets, les variables *cbRx1Head* et *cbRx1Tail* jouent un rôle crucial dans le suivi du début et de la fin des données dans le buffer. La fonction *CB_RX1_Add* est responsable de l'ajout de données dans le buffer. Elle vérifie d'abord s'il y a de l'espace disponible en utilisant *CB_RX1_GetRemainingSize*, puis stocke la valeur entrante à la position actuelle de *cbRx1Head* avant d'incrémenter *cbRx1Head*. Si *cbRx1Head* atteint la fin du buffer, il est réinitialisé à 0, permettant ainsi au buffer de fonctionner de manière circulaire. La fonction *CB_RX1_Get* extrait la prochaine donnée disponible du buffer. Elle lit la valeur à la position actuelle de *cbRx1Head*, incrémente *cbRx1Head*, et, comme pour l'ajout de données, réinitialise *cbRx1Head* à 0 si nécessaire. *CB_RX1_IsDataAvailable* vérifie si des données sont disponibles dans le buffer pour être lues, en comparant les positions de *cbRx1Head* et *cbRx1Tail*. Si ces deux positions ne sont pas égales, cela signifie qu'il y a des données en attente. *CB_RX1_GetDataSize* et *CB_RX1_GetRemainingSize* fournissent respectivement la taille des données stockées dans le buffer et l'espace restant.

Pour tester notre buffer, nous avons envoyé plusieurs message d'affiler afin de faire un tour du buffer. Cela a pu valider notre script et nous avons pu nous consacrer à la fiabilité de notre système de communication.

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

Notre méthode actuelle pour transférer des octets, qui permet la transmission de valeurs allant de 0x00 à 0xFF, s'est révélée fiable en termes de gestion du flux de données grâce à l'intégration de FIFO tant dans l'environnement embarqué que dans le code C#. Cette approche nous offre la possibilité d'envoyer des séquences d'octets, par exemple, pour contrôler un robot mobile. Cependant, le principal inconvénient de cette méthode réside dans le fait que les séquences d'octets échangées manquent de signification sémantique. De plus, dans un contexte industriel sujet à des interférences, il devient difficile, voire impossible, de détecter si les données ont été altérées durant la transmission, par exemple un 0 pourrait se transformer en 1 et inversement. Afin d'assurer une commande fiable et précise de notre robot, l'implémentation d'un protocole de communication devient cruciale. Ce besoin de structuration des données en paquets est aligné avec la couche 2 du modèle OSI.

3.1 Implantation en C# d'un protocole de communication

Pour intégrer un protocole de communication via la liaison série du PC, nous allons utiliser des messages structurés selon un format précis.

SOF	Command	Payload Length	Payload	Checksum
0xFE	2 octets	2 octets	n octets	1 octet

TABLE 1 – Structure d'un message dans le protocole de communication série

Ce format inclut un Start Of Frame (SOF) marqué par la valeur 0xFE, signifiant le début du message. Il est suivi d'une commande composée de deux octets, dont le premier est systématiquement fixé à 0x00. Après la commande, vient la partie Payload, qui contient les données ou arguments du message. La taille de cette section Payload est variable et déterminée par la Payload Length. Le message est conclu par un Checksum, qui est le résultat d'un XOR (Ou Exclusif) bit à bit de tous les octets du message, à l'exception du Checksum lui-même, afin d'assurer l'intégrité des données transmises.

3.1.1 Encodage des messages

Nous créons en premier une fonction qui calcul la *CheckSum* qui nous servira à vérifier si les données reçues sont correctes :

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

    byte checksum = 0;
    checksum ^= (byte)(0xFE);
    checksum ^= (byte)(msgFunction);
    checksum ^= (byte)(msgPayloadLength);

    for (int i = 0; i < msgPayloadLength; i++)
    {
        checksum ^= msgPayload[i];
    }

    return checksum;
}
```

Puis nous avons implémenté ce code qui encode les messages que nous envoyons :

Nous avons vérifié ce code en envoyant le message *Bonjour* et nous avons reçu une checksum qui valait *0x38*

3.1.2 Décodage des messages

Ce code définit une fonction *DecodeMessage* pour décoder des messages reçus byte par byte en se servant d'une machine à état. Initialement en attente (‘StateReception.Waiting‘), il cherche le byte de démarrage (‘0xFE‘) pour commencer le décodage, passe ensuite par plusieurs états pour extraire la fonction, la longueur du payload, et finalement le payload lui-même. À chaque étape, les données sont accumulées et l'état est mis à jour jusqu'à ce que le checksum soit vérifié à la fin du message. Si le checksum correspond, le message est considéré comme valide et traité. Sinon, le système revient en mode attente, en attendant de recevoir le prochain message.

3.1.3 Pilotage et supervision du robot

Avec la mise en place réussie du système de décodage des trames, nous voulons désormais pouvoir appliquer cela au contrôle et à la surveillance de notre robot. La supervision nous permettra d'avoir connaissance des divers paramètres de notre robot tels que les mesures des distances par les ADC, l'état des LEDs, les vitesses des moteurs, ou encore la position du robot. Ce mécanisme de pilotage et de supervision repose sur une bibliothèque de messages prédéfinis, chacun caractérisé par un identifiant de commande unique et une charge utile (payload) de taille spécifiée. Nous débutons par l'implémentation de fonctions initiales, présentées dans le tableau suivant.

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)

TABLE 2 – Fonctions de supervision

3.2 Implantation en électronique embarquée

Nous allons maintenant implanter un protocole de communication par-dessus la couche UART + Buffers circulaires. Pour ce faire nous créons un nouveau fichier nommé *UART_Protocol.c* avec son header correspondant et nous implémentons ces fonctions :

1. *UartCalculateChecksum* : Cette fonction calcule le checksum d'un message. Elle commence par initialiser la variable de checksum à 0, puis effectue un XOR bit à bit du Start Of Frame (0xFE) de la fonction du message puis de la longueur du payload et enfin de chaque octet du payload. Le résultat est ensuite retourné.
2. *UartEncodeAndSendMessage* : Cette fonction encode un message en ajoutant un Start Of Frame (0xFE), encode la fonction du message et sa longueur sur deux octets chacun, puis ajoute le payload et le checksum calculé. Le message encodé est ensuite envoyé via la liaison série.
3. *UartDecodeMessage* : Cette fonction décode les messages reçus. Elle utilise une machine à états pour traiter chaque partie du message reçu (Start Of Frame, fonction du message, longueur du payload, le payload lui-même, et le checksum). Elle valide le message en vérifiant le checksum et, si le message est valide elle le traite selon sa fonction.
4. *UartProcessDecodedMessage* : Cette fonction traite le message décodé. Selon la fonction du message (‘msgFunction‘) elle effectue différentes actions comme afficher un texte reçu, changer l'état des LEDs, afficher les distances mesurées par les télémètres IR, ajuster les vitesses des moteurs, ou afficher l'état et le timestamp. Elle utilise la valeur de la fonction du message pour déterminer l'action à exécuter.

3.2.1 Supervision

Pour tester ce code nous envoyons ce message depuis le main avec l'ajout d'une temporisation afin de ne pas surcharger le processus :

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

Nous voyons alors à l'oscilloscope le message qui s'affiche avec 'FE' au debut, le mot Bonjour et enfin le checksum qui vaut 8. Cela signifie que notre envoie fonctionne.

Nous voulons maintenant pouvoir afficher en temps réel les valeurs des télémètres ainsi que le stateRobot actuellement en cours. Nous ajoutons alors ces lignes de codes au *main.c* :

Celui-ci directement dans la boucle infinie

```
if (counter % 3 == 0) {
    unsigned char IR[] = {(unsigned char) robotState.distanceTelemetreGauche, (
    unsigned char) robotState.distanceTelemetreCentre, (unsigned char) robotState.
    distanceTelemetreDroit};
    UartEncodeAndSendMessage(0x0030, sizeof (IR), IR);
}
```

Celui-ci dans la fonction qui renvoie le prochain état de notre robot

```
    if (nextStateRobot != stateRobot - 1) {
        stateRobot = nextStateRobot;
        unsigned char State[] = {(unsigned char) stateRobot, (unsigned char)(
    timestamp>>24), (unsigned char)(timestamp>>16), (unsigned char)(timestamp>>8),
    (unsigned char)(timestamp>>0)};
        UartEncodeAndSendMessage(0x0050, sizeof (State), State);
    }
```

Nous complétons ainsi la fonction *DecodeMessage* afin de pouvoir afficher tout-ceci sur notre interface :

```
void ProcessDecodedMessage(int msgFunction, int msgPayloadLength, byte[] msgPayload
    )
{
    switch (msgFunction)
    {
        case 0x0080:
            textBoxReception.Text = Encoding.Default.GetString(msgPayload);
            break;

        case 0x0020:
            if (msgPayload[0] == 1)
            {
                CheckboxLed1.IsChecked = true;
            }
            else if (msgPayload[0] == 2)
                CheckboxLed2.IsChecked = true;
            else if(msgPayload[0] == 3)
                CheckboxLed3.IsChecked = true;
            break;

        case 0x0030:
            IRGauche.Text = "IRGauche :  " + msgPayload[0].ToString();
            IRCentre.Text = "IRCentre :  "  + msgPayload[1].ToString();
            IRDroit.Text =  "IRDroit :  " + msgPayload[2].ToString();
            break;

        case 0x0040:
            VitesseD.Text = "VitesseD :  " + msgPayload[0].ToString();
            VitesseG.Text = "VitesseG :  " + msgPayload[1].ToString();
            break;

```

```
31          case 0x0050:
32              State.Text = "";
33              int instant = (((int)msgPayload[1]) << 24) + (((int)msgPayload[2]) <<
    16) + (((int)msgPayload[3]) << 8) + ((int)msgPayload[4]);
34              State.Text += "Robot State : " + ((StateRobot)(msgPayload[0])).ToString
    () + "\n" + instant.ToString() + " ms";
35              break;
36      }
37  }
```

Nous avons finalement une interface graphique qui affiche en temps réel la valeur des télémètres, de la consigne moteur et de l'action en cours. Cette interface nous aurait été d'une grande aide lorsque nous avons premièrement implémenté les stratégies d'évitements lors du premier semestre.

Conclusion

Au terme de ce projet, nous avons traversé plusieurs étapes importantes dans la conception et la mise en œuvre d'un système de communication pour le pilotage et la supervision d'un robot en utilisant C. La création d'interfaces utilisateurs avec WPF, l'élaboration d'un protocole de communication efficace sur liaison série et l'étape de programmation en bas niveau avec le langage C nous ont permis de s'initier à la programmation embarquée. En intégrant des concepts tels que les buffers circulaires pour la gestion des données et en exploitant les possibilités offertes par la programmation embarquée, nous avons pu établir une fondation solide pour le contrôle fiable et la surveillance en temps réel de notre robot. Ce TP ouvre la voie à des applications plus sophistiquées dans le domaine de la robotique et au-delà.

4 Annexe

Cette annexe contient des liens vers des ressources externes utilisées ou mentionnées dans ce rapport.

1. GitHub - Projet SeaTech: Courtois & Carpentier