Pilotage et supervision d'un robot en C#

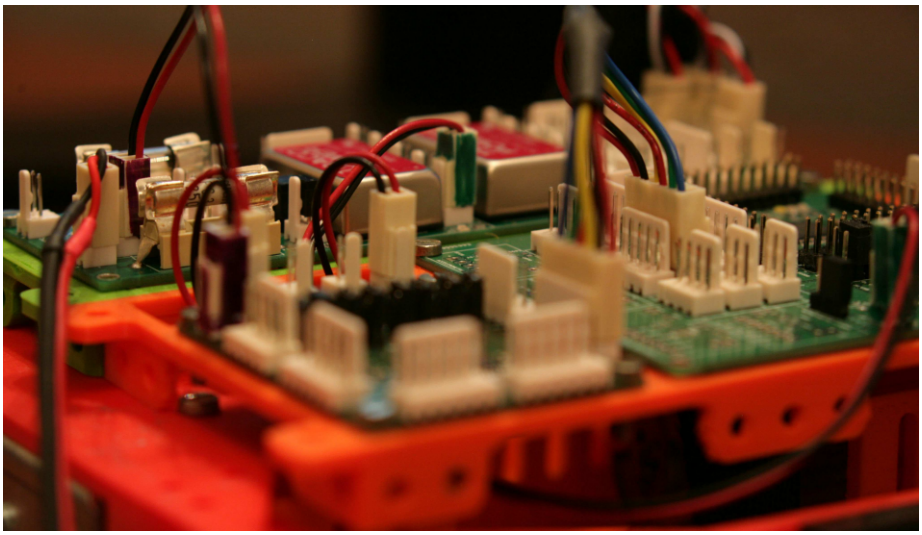
RIE

Travail réalisé par :

Camille BERTA et Enzo CHERIF

Étudiants de Seatech Promo 2025

2A, filière SYSMER



Cours supervisé par :

Mr Valentin GIES

Dans le cadre du cours

RIE

2023-2024

Table des matières

Introduction

Ce TP vise à explorer et à maîtriser des concepts fondamentaux en électronique appliquée à travers la conception et le développement d'un système de supervision et de pilotage pour un robot mobile, utilisant la communication série pour interagir entre le robot et un PC.

Au-delà de l'application des connaissances théoriques, nous allons mettre en pratique ces dernières dans un contexte réel. Nous souhaitons réaliser un système fonctionnel capable d'une communication efficace via un protocole série, de répondre aux commandes de pilotage envoyées depuis un ordinateur, et d'être supervisé en temps réel grâce à une interface utilisateur graphique développée en C# avec WPF (Windows Presentation Foundation).

Cet exercice nous a permis d'aborder divers aspects de l'électronique et de la programmation, notamment la gestion des interfaces graphiques, la communication série en embarqué, et le développement de protocoles de communication. Il a également souligné l'importance de l'intégration et du test des systèmes, nécessaires pour la réussite de projets d'ingénierie complexes.

Ainsi, les objectifs de nos TP englobent la compréhension et la mise en œuvre d'une communication série entre un PC et un microcontrôleur, le développement d'une interface de supervision et de pilotage intuitive et réactive, ainsi que la conception d'un système embarqué fiable et performant.

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

1.1 Simulateur de messagerie instantanée

Dans cette première partie, nous allons nous focaliser sur la création d'un simulateur de messagerie instantanée par le biais d'une interface graphique en utilisant le langage C#. L'objectif principal est de développer un terminal permettant l'envoi et la réception de messages via le port série d'un PC.

Lors de la création d'un nouveau projet C# dans Visual Studio, nous sommes accueillis par une fenêtre de conception. Cette interface nous permet de personnaliser notre application en y ajoutant les éléments souhaités. Dans notre cas, nous avons choisi d'intégrer deux GroupBox à notre fenêtre en les faisant glisser depuis la Boîte à outils. Elles peuvent être personnalisées en ajustant leurs propriétés. En utilisant des GroupBox, nous pouvons ainsi organiser les éléments de notre interface utilisateur de manière claire et structurée pour faciliter la compréhension et la'utilisation pour l'utilisateur.

Afin de faciliter la création de notre interface, nous avons utilisé une grille et défini des colonnes et des lignes avec des largeurs et hauteurs proportionnelles à la taille de la fenêtre. Cela permet aux éléments de s'adapter automatiquement lors du redimensionnement de la fenêtre.

Nous avons intégré des TextBox pour la saisie et l'affichage des messages, ainsi qu'un bouton "Envoyer" pour transmettre les messages. En associant un événement Click au bouton, nous avons pu exécuter du code lorsque l'utilisateur clique dessus.

Afin de créer un simulateur de messagerie instantanée, nous avons écrit dans notre code la fonction buttonEnvoyer_Click pour simuler l'envoi de messages de la TextBox d'émission vers la TextBox de réception. Ce code récupère le texte saisi dans la TextBox d'émission, le place dans la TextBox de réception précédé de "Reçu : ", puis vide la TextBox d'émission pour permettre la saisie d'un nouveau message.

Enfin, nous avons ajouté la gestion de l'événement KeyUp dans la TextBox d'émission pour simuler un service de messagerie instantanée où les messages sont envoyés en appuyant sur la touche Entrée. La fonction TextBoxEmission_KeyUp vérifie si la touche pressée est Entrée et, si oui, envoie le message.

1.2 Messagerie instantanée entre deux PC

Maintenant que notre simulateur est terminé, nous allons l'utiliser pour créer une messagerie instantanée entre 2 PC. Pour faire ça, nous allons utiliser le module FT232RL. Il convertit les données série en signaux USB et vice versa et nous permet de connecter des périphériques série, tels que des microcontrôleurs, des capteurs, des modules GPS, etc., à un ordinateur via un port USB.

Nous avons tout d'abord crée un objet SerialPort sur notre code C#. Pour garantir son bon fonctionnement, nous avons dû télécharger une librairie compatible avec .NET6, l'ajouter à notre solution puis la référencer dans notre interface graphique.

Pour initialiser le port série avec les paramètres adéquats, nous avons saisi le nom du port, la vitesse de transmission, la parité, le nombre de bits des données et le type de StopBits pour ensuite l'ouvrir afin de rendre la communication possible.

Pour garantir la transmission des messages via le port série, nous avons modifié la fonction SendMessage en lui ajoutant la commande WriteLine

```
1 private void SendMessage()
2     {
3         textBoxReception.Text += "Recu : " + textBoxEmission.Text + Environment.NewLine;
```

```
        serialPort1.WriteLine(textBoxEmission.Text);
        textBoxEmission.Text = "";
    }
```

Nous avons pu observer que la LED du robot s'allume désormais à chaque fois qu'un message est envoyé depuis l'ordinateur. Cette LED nous permet de valider le bon fonctionnement de l'envoi des données.

Nous allons maintenant utiliser le mode Loopback pour vérifier la réception des données. Pour se faire, nous allons connecter un connecteur sur le module FT232RL pour connecter le pin Tx (qui transmet les données) aux pin Rx (qui reçoit les données). Après avoir configuré le serialPort1, nous utilisons un callback dans une fonction appelée SerialPort1_DataReceived pour gérer la réception des données entrantes sur le port série. Cela permet de recevoir et de traiter les données envoyées depuis le même PC. En envoyant un message depuis l'interface graphique vers le port série, les données seront envoyées sur la pin Tx du module, puis reçues par la pin Rx du même module en raison de la connexion Loopback.

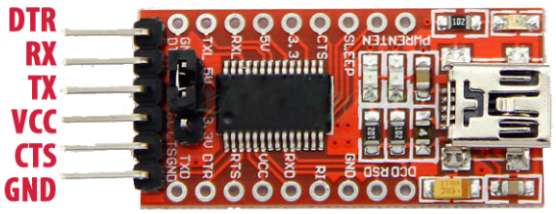


Figure 1 – Pins du module FT232RL

Cependant, nous sommes confrontés à un problème : les threads. Le programme ne peut pas gérer plusieures séquences d'instructions simultanement. Le multithreading permet d'accomplir plusieurs tâches en parallèle, ce qui peut conduire à une utilisation plus efficace des ressources du processeur. Cependant, cela introduit également des problèmes où plusieurs threads peuvent accéder aux mêmes ressources partagées simultanément, ce qui peut entraîner des résultats imprévisibles. Nous avons ici deux threads différents, un qui gère le port série et l'autre qui gère l'interface graphique. Nous utilisons donc une chaîne de caractère pour lier les deux threads et éviter le problème.

Enfin, nous mettons en place un mécanisme pour afficher les données reçues à intervalles réguliers en utilisant un DispatcherTimer lié au thread graphique. En connectant les câbles série entre les PC, nous avons pu vérifier que les envois de messages fonctionnent correctement.

1.3 Structuration du code avec une classe Robot

Nous avons ensuite entrepris d'améliorer la structuration de notre code en introduisant une nouvelle classe appelée Robot. Cette classe a pour objectif de centraliser toutes les informations en provenance du robot, telles que les messages, les données des capteurs et les vitesses des moteurs.

1.4 Liaison série hexadécimale

Notre messagerie instantée est faite et nous avons vérifié son fonctionnement. Cependant, nous ne pouvons communiquer que des chaînes de caractères. Nous allons donc utiliser une liaison série hexadécimale pour envoyer des octets bruts afin de transmettre n'importe quelle valeur binaire, y compris les caractères de contrôle de la table ASCII et d'autres données non textuelles. Pour ce faire, nous allons travailler en mode Loopback.

Nous avons ajouté à notre interface un bouton "Test" pour évaluer notre système. Lorsque l'on clique dessus, le programme construit un tableau de 20 octets et l'envoie sur le port série à l'aide de la fonction Write. Afin de pouvoir interpréter ces données reçues, nous allons créer une buffer de type FIFO.

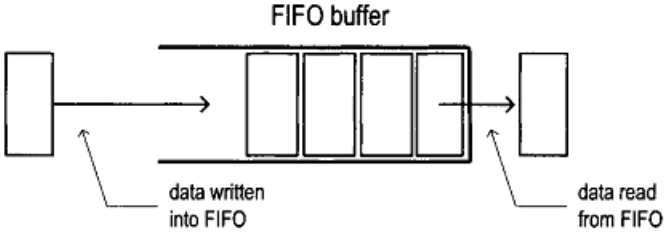


Figure 2 – Schéma du buffer FIFO

Le buffer FIFO est une structure de données utilisée pour stocker et gérer des éléments de manière séquentielle, où le premier élément ajouté est également le premier élément à être retiré.

Dans notre fonction DataReceived du port série, les octets disponibles seront ajoutés un par un à la Queue.

```
private void SerialPort1_DataReceived1(object sender, DataReceivedArgs e)
        {
            foreach (byte b in e.Data)
            {
                robot.byteListReceived.Enqueue(b);
            }
        }
```

Grâce au timer, nous allons récupérer les bytes de la Queue pour les afficher dans la textBox de réception. Puis, nous utilisons la méthode de formatage ToString() avec différents paramètres, notamment "X", "X2" et "X4", pour tester l'affichage des octets en hexadécimal.

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

2.1 Configuration de la liaison série en embarqué

Dans le prolongement de notre exploration de la programmation orientée objet en C#, nous nous sommes aventurés dans la phase majeure de notre projet : la configuration de la liaison série en embarqué. Cette étape s'avère essentielle pour la communication entre notre PC et le robot mobile, impliquant l'utilisation d'une carte capteurs connectée à l'ordinateur via un dongle USB/Série. Cette configuration permet de pallier l'absence de liaison USB directe sur la carte principale.

Mise en Place de l'UART

Pour réaliser cette configuration, nous avons débuté par la création d'un fichier `UART.c` dédié à l'initialisation de l'UART. Le code suivant a été intégré pour configurer l'UART à une vitesse de 115200 bauds, tout en désactivant l'utilisation des interruptions, conformément aux exigences de notre projet :

```
#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; // High 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
}
```

La précision du paramétrage, telle que la définition du baud rate, du nombre de bits de stop et de données, ainsi que l'activation du mode haute vitesse, ont été des étapes clés pour garantir une communication fluide et sans erreur.

Déclaration dans UART.h

Afin de faciliter l'intégration de notre module UART dans l'ensemble du projet, un fichier d'en-tête `UART.h` a été créé. Ce fichier contient la déclaration de la fonction `InitUART`, permettant son appel depuis n'importe quelle partie du code :

```
#ifndef UART_H
#define UART_H

void InitUART(void);
```

```

#endif /* UART_H */
```

Intégration et Tests

L'intégration de la configuration UART dans notre TP a été réalisée en ajoutant l'appel de la fonction `InitUART` dans la fonction principale `main` de notre code, ainsi qu'en incluant le fichier `UART.h`.

```
#include "UART.h"

int main(void) {
    InitUART();
    // Le reste du code du programme
    return 0;
}
```

Ce code configure un canal de communication série UART sur un microcontrôleur avec des paramètres spécifiques : 115200 bauds comme vitesse de communication, 8 bits de données par caractère, pas de bit de parité, 1 bit de stop, et désactive les interruptions pour une gestion simplifiée des données envoyées et reçues. Ce type de configuration est essentiel pour assurer une communication fiable et conforme aux attentes des dispositifs communicants dans un projet embarqué, tel que le pilotage et la supervision d'un robot mobile.

2.1.1 Échange de données entre le microcontrôleur et le PC

Dans le cadre de notre travail pratique, nous avons franchi une étape supplémentaire en établissant une communication directe entre le microcontrôleur et le PC via une liaison série-USB. Cette configuration nous a permis de transmettre efficacement des données du microcontrôleur vers le PC, une composante essentielle pour le pilotage et la supervision de notre système embarqué.

2.1.2 Émission UART depuis le microcontrôleur

La connexion entre le microcontrôleur et le PC a été réalisée grâce à un câble fourni, connecté du convertisseur série-USB aux pins Rx et Tx de l'UART1 du microcontrôleur. La connexion correcte nécessitait d'attacher le fil Tx du convertisseur au pin Rx de l'UART1, permettant ainsi la réception par le microcontrôleur des données émises par le PC.

Après l'analyse du schéma de câblage du dsPIC, nous avons identifié les numéros de pins remappables correspondants. Les configurations suivantes ont été ajoutées dans le fichier `IO.c`, spécifiquement dans la section de configuration des pins :

```
_U1RXR = 24; // Remappe la RP24 sur l'entr e Rx1
_RP36R = 0b00001; // Remappe la sortie Tx1 vers RP36
```

Cette configuration a permis de rediriger correctement les données entrantes et sortantes de l'UART vers les bons pins du microcontrôleur, établissant ainsi les fondations pour une communication série efficace.

Fonction d'Envoi de Messages

Pour faciliter l'envoi de données depuis le microcontrôleur vers le PC, une fonction `SendMessageDirect` a été implémentée dans le fichier `UART.c`, avec son en-tête déclaré dans `UART.h`. Cette fonction était chargée de l'envoi de chaînes de caractères au PC :

```
void SendMessageDirect(unsigned char* message, int length) {
    unsigned char i = 0;
    for(i = 0; i < length; i++) {
        while (U1STAbits.UTXBF); // Attente de la disponibilit du tampon de
    transmission
        U1TXREG = *(message)++; // Transmission d'un caract re
    }
}
```

En parcourant et en envoyant chaque caractère de la chaîne, tout en surveillant la disponibilité du tampon de transmission UART, cette fonction a joué un rôle clé dans la communication du microcontrôleur vers le PC.

Tests de Communication

Pour vérifier la communication établie, nous avons utilisé `SendMessageDirect` dans la boucle principale de notre programme pour envoyer le message "Bonjour" à des intervalles réguliers :

```
while(1) {
    SendMessageDirect((unsigned char*)"Bonjour", 7);
```

```
    __delay32(40000000); // Intervalle d'une seconde
}
```

2.1.3 Réception

Dans la suite de notre exploration de la communication série, nous avons focalisé notre attention sur la réception des données au niveau du microcontrôleur. L'objectif était de valider la capacité de réception sur le port série du microcontrôleur à travers un mode LoopBack logiciel, qui consiste à renvoyer immédiatement les données reçues en émission, permettant ainsi un test efficace de la communication série.

Configuration des Interruptions en Réception

Une étape initiale cruciale a été d'adapter la fonction d'initialisation du port série pour activer les interruptions en réception sur l'UART. Cette adaptation nous a permis de capturer de manière asynchrone les caractères entrants, offrant une réactivité essentielle pour le traitement des données reçues.

Gestion des Interruptions en Mode LoopBack

L'élément central de notre mise en place était la gestion des interruptions, configurée pour être déclenchée à chaque réception d'un caractère. Le code suivant illustre comment cette gestion a été mise en œuvre :

```
// Gestion des interruptions en mode loopback
void __attribute__((interrupt, no_auto_psv)) _U1RXInterrupt(void) {
    IFS0bits.U1RXIF = 0; // Réinitialise le drapeau d'interruption de réception

    // Traitement des erreurs de réception
    if (U1STAbits.FERR == 1) {
        U1STAbits.FERR = 0;
    }

    // Correction de l'erreur d'overrun pour continuer la réception
    if (U1STAbits.OERR == 1) {
        U1STAbits.OERR = 0;
    }

    // Lecture et renvoi immédiat des caractères reçus
    while (U1STAbits.URXDA == 1) {
        U1TXREG = U1RXREG; // Renvoi des caractères reçus
    }
}
```

Ce code joue un rôle multiple : il réinitialise le drapeau d'interruption pour indiquer que l'interruption a été traitée, gère les erreurs telles que les erreurs de framing et d'overrun, et lit puis renvoie les caractères reçus, assurant ainsi le fonctionnement du mode LoopBack.

Tests de la Communication

Pour valider notre configuration, nous avons temporairement désactivé l'envoi de messages périodiques et bloquants depuis le programme principal, et nous avons lancé le programme sur le PIC. En utilisant notre interface C# pour envoyer des messages au microcontrôleur, nous avons observé que ces messages étaient renvoyés vers la console de réception, confirmant la réussite de notre configuration LoopBack.

Le suivi de l'envoi et de la réception des trames via l'oscilloscope a fourni une confirmation visuelle claire de la communication réussie. Les signaux capturés ont montré un écho immédiat des trames envoyées, illustrant parfaitement la réception et le renvoi des données.

2.2 Liaison série avec FIFO intégré

2.2.1 Détails sur le buffer circulairede l'UART en émission

Fonctionnement d'un Buffer Circulaire

Un buffer circulaire fonctionne comme un conteneur cyclique pour les données, où la capacité de stockage est fixe et prédéterminée. Pour illustrer son fonctionnement, considérons un buffer circulaire avec une capacité de 7 éléments.

Initialisation et Ajout d'Éléments

Au départ, le buffer est vide. Imaginons que nous commençons par ajouter l'élément "1" dans le buffer :

–	1	–	–	–	–	–

Ensuite, nous ajoutons deux éléments supplémentaires, "2" et "3", après le "1" :

–	1	2	–	3	–	–

Retrait d'Éléments

Si nous décidons de retirer deux éléments du buffer, les éléments retirés sont ceux qui ont été ajoutés en premier, conformément au principe FIFO (First In, First Out). Après le retrait de "1" et "2", le buffer ne contiendrait plus que l'élément "3" :

–	–	–	3	–	–	–

Remplissage Complet du Buffer

Lorsque nous continuons à ajouter des éléments jusqu'à remplir complètement le buffer, il pourrait ressembler à ceci, complètement rempli :

7	8	9	3	4	5	6

Gestion de l'Écrasement

Si le buffer est plein et qu'un nouvel élément est ajouté, il y a deux possibilités. La première est que les données les plus anciennes soient écrasées par les nouvelles. Par exemple, en ajoutant "A" et "B", les éléments "3" et "4" seraient écrasés :

7	8	9	A	B	5	6

La seconde possibilité est que la routine de gestion du buffer empêche l'écrasement en signalant une erreur ou en augmentant automatiquement la capacité du buffer, bien que cette dernière option sorte du cadre classique d'un buffer circulaire.

Retrait Après Écrasement

Si après l'écrasement, nous retirons deux éléments, les éléments obtenus ne seraient pas "3" et "4" (car ils ont été écrasés), mais "5" et "6". Le buffer serait alors dans cet état :

7	8	9	A	B	–	–

Le buffer circulaire est un outil puissant pour la gestion des données dans des environnements où l'espace est limité et où les données doivent être traitées de manière ordonnée. Sa conception garantit une utilisation efficace de l'espace de stockage, mais requiert une gestion soigneuse pour éviter la perte de données importantes par écrasement. Dans le contexte des communications série, le buffer circulaire permet de maintenir un flux de données constant et efficace, en évitant les blocages tout en gérant la production et la consommation de données à des vitesses variables.

2.2.2 Le buffer circulaire de l'UART en émission

Dans le cadre de notre TP, l'introduction d'un buffer circulaire pour la gestion de l'émission UART a constitué une évolution significative dans notre manière d'aborder les communications série. Ce mécanisme a été conçu pour optimiser l'envoi des données en permettant une gestion asynchrone et non bloquante des transmissions. Voici comment nous avons implémenté cette solution en utilisant le code fourni.

Structure du Buffer Circulaire

Le buffer circulaire a été implémenté pour stocker temporairement les messages avant leur transmission par l'UART, en s'assurant que le programme principal continue de s'exécuter sans interruption. Le code initial définit la taille du buffer (`CBTX1_BUFFER_SIZE`), initialise les indices de tête (`cbTx1Head`) et de queue (`cbTx1Tail`), et prépare le buffer pour stocker les données :

```
#define CBTX1_BUFFER_SIZE 128
int cbTx1Head;
int cbTx1Tail;
unsigned char cbTx1Buffer[CBTX1_BUFFER_SIZE];
unsigned char isTransmitting = 0;
```

Fonction d'Envoi des Messages

La fonction `SendMessage` a été conçue pour ajouter des messages au buffer circulaire. Elle vérifie d'abord si l'espace disponible est suffisant pour le message. Si tel est le cas, le message est ajouté au buffer, caractère par caractère. Si aucune transmission n'est en cours, l'envoi du premier caractère est initié :

```
void SendMessage(unsigned char* message, int length) {
    if (CB_TX1_GetRemainingSize() > length) {
        for(int i = 0; i < length; i++) {
            CB_TX1_Add(message[i]);
        }
        if(!CB_TX1_IsTranmitting()) {
            SendOne();
        }
    }
}
```

Gestion du Buffer Circulaire

La fonction `CB_TX1_Add` est responsable de placer un nouveau caractère dans le buffer, tout en ajustant l'indice de tête (`cbTx1Head`). Si l'indice atteint la fin du buffer, il revient au début, créant ainsi un cycle. Voici comment cette fonction est implémentée :

```
void CB_TX1_Add(unsigned char value) {
    cbTx1Buffer[cbTx1Head] = value;
    cbTx1Head = (cbTx1Head + 1) % CBTX1_BUFFER_SIZE;
}
```

La fonction `CB_TX1_Get` est responsable d'extraire le caractère le plus ancien du buffer pour l'envoi, tout en ajustant l'indice de queue (`cbTx1Tail`). Voici comment cette fonction est implémentée :

```
unsigned char CB_TX1_Get(void) {
    unsigned char value = cbTx1Buffer[cbTx1Tail];
    cbTx1Tail = (cbTx1Tail + 1) % CBTX1_BUFFER_SIZE;
    return value;
}
```

Cette fonction récupère simplement la valeur du caractère à l'indice actuel de la queue du buffer, puis incrémente l'indice de la queue en utilisant l'opérateur modulo pour assurer le comportement cyclique du buffer. Ainsi, lorsque la queue atteint la fin du buffer, elle revient automatiquement au début, assurant une extraction continue et cyclique des caractères du buffer.

Gestion des Interruptions et Transmission

Lorsqu'une interruption d'émission (`_U1TXInterrupt`) est déclenchée, elle vérifie si des caractères restent à envoyer. Si tel est le cas, la fonction `SendOne` est appelée pour continuer l'envoi :

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

La fonction `SendOne` déclenche l'envoi d'un caractère et marque le début de la transmission si ce n'était pas déjà le cas :

```
void SendOne() {
    isTransmitting = 1;
    U1TXREG = CB_TX1_Get();
}
```

Vérification de l'État de Transmission et Calcul de l'Espace

La fonction `CB_TX1_IsTranmitting` permet de vérifier si une transmission est en cours, facilitant la décision d'initier ou non un nouvel envoi :

```
unsigned char CB_TX1_IsTranmitting(void) {
    return isTransmitting;
}
```

Les fonctions `CB_TX1_GetDataSize` et `CB_TX1_GetRemainingSize` fournissent respectivement la taille des données stockées dans le buffer et l'espace restant, essentielles pour la gestion optimale du buffer circulaire :

```
int CB_TX1_GetDataSize(void) {
    int dataSize = cbTx1Head >= cbTx1Tail ? cbTx1Head - cbTx1Tail : CBTX1_BUFFER_SIZE +
    cbTx1Head - cbTx1Tail;
    return dataSize;
```

```
}

int CB_TX1_GetRemainingSize(void) {
    return CBTX1_BUFFER_SIZE - CB_TX1_GetDataSize();
}
```

L'intégration de ces fonctions dans notre système de buffer circulaire pour l'UART en émission illustre une gestion avancée des communications série. Grâce à ce mécanisme, notre application est capable de gérer efficacement l'envoi de données de manière asynchrone et non bloquante, reflétant notre capacité à mettre en œuvre des solutions complexes pour optimiser les systèmes embarqués.

2.2.3 Le buffer circulaire de l'UART en réception

Pour améliorer la gestion des données reçues via l'UART dans notre TP, nous avons implémenté un buffer circulaire dédié à la réception. Cette structure, similaire à celle utilisée pour l'émission, permet de stocker temporairement les données entrantes jusqu'à ce qu'elles soient traitées par le programme. Ce mécanisme assure une réception fluide des données sans risque de perte, même lors de pics d'activité où les données arrivent plus rapidement que le programme ne peut les traiter.

Mise en place du Buffer Circulaire en Réception

Le buffer circulaire est conçu pour stocker les caractères reçus sur le port série, en attente de leur traitement ultérieur. Le code commence par définir la taille du buffer (`CBRX1_BUFFER_SIZE`) et initialise les indices de tête (`cbRx1Head`) et de queue (`cbRx1Tail`), ainsi que le buffer lui-même :

```
#define CBRX1_BUFFER_SIZE 128
int cbRx1Head;
int cbRx1Tail;
unsigned char cbRx1Buffer[CBRX1_BUFFER_SIZE];
```

Fonctions du Buffer Circulaire

La fonction `CB_RX1_Add` est appelée pour ajouter un caractère reçu au buffer. Elle vérifie d'abord s'il y a de l'espace disponible. Si c'est le cas, le caractère est ajouté et l'indice de tête est incrémenté. Si l'indice atteint la limite du buffer, il revient à zéro :

```
void CB_RX1_Add(unsigned char value) {
    if (CB_RX1_GetRemainingSize() > 0) {
        cbRx1Buffer[cbRx1Head] = value;
        cbRx1Head++;
        if (cbRx1Head >= CBRX1_BUFFER_SIZE)
            cbRx1Head = 0;
    }
}
```

`CB_RX1_Get` permet de récupérer le prochain caractère à traiter du buffer, en incrémentant l'indice de queue. Comme pour la tête, si cet indice dépasse la taille du buffer, il est réinitialisé à zéro :

```
unsigned char CB_RX1_Get(void) {
    unsigned char value = cbRx1Buffer[cbRx1Tail];
    cbRx1Tail++;
    if (cbRx1Tail >= CBRX1_BUFFER_SIZE)
        cbRx1Tail = 0;
    return value;
}
```

La fonction `CB_RX1_IsDataAvailable` vérifie si des données sont présentes dans le buffer, facilitant la décision de lire ou de traiter les données reçues.

Gestion des Interruptions pour la Réception

Dans le contexte de notre système embarqué, la routine d'interruption `_U1RXInterrupt` est essentielle pour gérer efficacement la réception des données via UART. Cette routine est automatiquement déclenchée chaque fois qu'un caractère est reçu sur le port série, ce qui permet de réagir en temps réel sans nécessiter de vérification active ou de polling de la part du programme principal.

Voici une explication détaillée de son fonctionnement :

```
void __attribute__((interrupt, no_auto_psv)) _U1RXInterrupt(void) {
    IFS0bits.U1RXIF = 0; // Efface le drapeau d'interruption de r ception pour indiquer
    que l'interruption est  g  r  e

```

```
    // Vrification et nettoyage des erreurs de rception
    if (U1STAbits.FERR == 1) {
        U1STAbits.FERR = 0; // Efface l'erreur de framing
    }
    if (U1STAbits.OERR == 1) {
        U1STAbits.OERR = 0; // Efface l'erreur d'overrun pour permettre la continuation de la rception
    }

    // Rcupration et stockage des donnes
    while (U1STAbits.URXDA == 1) {
        CB_RX1_Add(U1RXREG); // Ajoute chaque caractre reu au buffer circulaire
    }
}
```

Dans cette routine, le premier pas est de nettoyer le drapeau d'interruption de réception (U1RXIF), indiquant que l'interruption est en cours de traitement. Ensuite, les erreurs potentielles de réception, comme les erreurs de framing (FERR) et d'overrun (OERR), sont vérifiées et nettoyées. Ces étapes assurent que la liaison UART est prête à recevoir de nouvelles données.

Finalement, tant qu'il y a des données disponibles (URXDA), chaque caractère reçu est ajouté au buffer circulaire par CB_RX1_Add. Cette gestion par interruption permet une réception fluide des données, sans interruption du fonctionnement normal du programme.

Calcul de la Taille des Données et de l'Espace Restant

La gestion précise de l'espace dans le buffer circulaire est cruciale pour éviter l'écrasement des données non traitées ou la perte de nouvelles données entrantes. Les fonctions CB_RX1_GetDataSize et CB_RX1_GetRemainingSize permettent de gérer cet aspect :

```
int CB_RX1_GetDataSize(void) {
    int dataSize;
    if (cbRx1Head >= cbRx1Tail) {
        dataSize = cbRx1Head - cbRx1Tail;
    } else {
        dataSize = CBRX1_BUFFER_SIZE - (cbRx1Tail - cbRx1Head);
    }
    return dataSize; // Retourne la taille actuelle des donnes stockees dans le buffer
}

int CB_RX1_GetRemainingSize(void) {
    return CBRX1_BUFFER_SIZE - CB_RX1_GetDataSize(); // Calcule l'espace restant dans le buffer
}
```

CB_RX1_GetDataSize calcule la quantité de données actuellement stockées dans le buffer, en tenant compte de la circularité du buffer et de la position relative des indices de tête et de queue. CB_RX1_GetRemainingSize, quant à elle, utilise cette information pour déterminer combien d'espace reste disponible pour de nouvelles données. Ensemble, ces fonctions assurent que les données sont ajoutées au buffer uniquement s'il y a suffisamment d'espace, prévenant ainsi tout risque d'écrasement.

Ces mécanismes de gestion des interruptions et de calcul de l'espace dans le buffer circulaire sont essentiels pour maintenir l'intégrité des données dans des systèmes embarqués complexes, où la gestion efficace et en temps réel des flux de données est primordiale.

Avantages et Conclusion

L'utilisation d'un buffer circulaire pour la réception UART présente plusieurs avantages clés :

- **Réduction du risque de perte de données** : En stockant temporairement les données entrantes, le buffer permet au programme de traiter les données à son rythme sans risque de les perdre.
- **Amélioration de la fiabilité** : Le buffer circulaire garantit que les données sont traitées dans l'ordre de leur arrivée, préservant ainsi la séquence des informations.
- **Flexibilité dans le traitement des données** : Le programme peut traiter les données par lots, améliorant potentiellement l'efficacité du traitement.

En somme, le buffer circulaire en réception renforce la robustesse de notre système de communication série, en assurant une gestion efficace et ordonnée des données entrantes, essentielle pour les applications nécessitant une haute fiabilité et performance de communication.

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

Cette dernière partie introduit la nécessité d'implémenter un protocole de communication pour améliorer la fiabilité et la signification des échanges de données sur la liaison série entre deux systèmes embarqués.

Le système actuel d'échange d'octets est robuste en termes de flux de données, mais il manque de sémantique et ne permet pas de détecter les erreurs de transmission. Or, dans un environnement bruité, il est important de pouvoir vérifier l'intégrité des données transmises.

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

Pour remédier à ce problème, nous allons implanter le protocole de communication ci-dessous :

Start of Frame	Command	Payload Length	Payload	Checksum
OxFE	2 octets	2 octets	n octets	1 octet

Table 1 – Protocole de communication

Dans ce protocole de communication, tous les messages suivent la même structure. Le Start of Frame (SOF) est le premier octet de chaque message. Il marque le début d'une nouvelle trame. Il est suivi de la commande constituée de deux octet. Elle contient l'identifiant de la commande à exécuter. Nous avons ensuite les arguments (Payload) qui contient les données spécifiques à la commande à exécuter. Sa longueur peut varier en fonction de la commande, mais elle est spécifiée par le champ "Payload Length" pour que le récepteur puisse l'interpréter correctement. Pour finir, nous avons le checksum qui est un octet calculé à partir de tous les octets de la trame, à l'exception du checksum lui-même. Son but est de permettre au récepteur de vérifier l'intégrité du message reçu. Si le checksum calculé ne correspond pas au checksum reçu, le message est considéré comme corrompu et ignoré. Cependant, il est important de retenir que le checksum ne détecte que les erreurs, il ne les corrige pas.

3.1.1 Encodage des messages

Pour encoder les messages, nous allons d'abord devoir calculer le checksum grâce à la fonction CalculateCheksum. Elle prend en entrée le numéro de fonction, la longueur de la charge utile (payload) et la charge utile elle-même, et renvoie le checksum calculé. Pour calculer le checksum, nous effectuons un OU exclusif bit à bit de tous les octets de la trame (y compris le SOF, la commande et la charge utile).

```
public byte CalculateChecksum(int msgFunction, int msgPayloadLength, byte[] msgPayload)
{
    byte checksum = 0;
    checksum ^= 0xFE;
    checksum ^= (byte)(msgFunction >> 8);
    checksum ^= (byte)(msgFunction >> 0);
    checksum ^= (byte)(msgPayloadLength >> 8);
    checksum ^= (byte)(msgPayloadLength >> 0);
    for(int i = 0; i< msgPayloadLength; i++)
    {
        checksum ^= msgPayload[i];
    }
    return checksum;
}
```

Après cette étape, nous allons pouvoir implanter une fonction permettant de formater les messages en suivant le modèle de notre protocole de communication.

```
void UartEncodeAndSendMessage(int msgFunction, int msgPayloadLength, byte[] msgPayload)
{
    byte checksum= UartCalculateChecksum(msgFunction, msgPayloadLength, msgPayload);

    byte[] Trame = new byte[6 + msgPayloadLength];
    int pos=0;
    Trame[pos++] = 0xFE;
    Trame[pos++] = (byte)(msgFunction >> 8);
    Trame[pos++] = (byte)(msgFunction >> 0);
    Trame[pos++] = (byte)(msgPayloadLength >> 8);
    Trame[pos++] = (byte)(msgPayloadLength >> 0);
    Array.Copy(msgPayload, 0, Trame, 5, msgPayloadLength);
    Trame[Trame.Length - 1] = checksum;

    serialPort1.Write(Trame, 0, Trame.Length);
}
```

3.1.2 Décodage des messages

L'objectif maintenant va être de décoder la trame lors de sa réception à l'aide d'une machine à état. Nous allons utiliser un enum StateReception pour définir les différents états possibles. Chaque état correspond à une phase de décodage de la trame, comme la lecture de la fonction, de la longueur de la charge utile, de la charge utile elle-même, etc. Nous initialisons les variables nécessaires au décodage, telles que l'état courant (rcvState), la fonction décodée (msgDecodedFunction), la longueur de la charge utile décodée (msgDecodedPayloadLength), le tableau de la charge utile décodée (msgDecodedPayload) et l'indice de position dans la charge utile (msgDecodedPayloadIndex).

Nous allons ensuite implémenter la fonction DecodeMessage, qui prend un octet en argument et réalise le décodage en fonction de l'état courant. À chaque état, nous effectuons les opérations nécessaires, comme la lecture de la fonction, la longueur de la charge utile, la charge utile elle-même, etc. Une fois le checksum atteint, nous vérifions si le checksum calculé correspond au checksum reçu. Si c'est le cas, le message est considéré comme valide.

3.1.3 Pilotage et supervision du robot

Maintenant, nous pouvons utiliser notre système de messagerie que nous avons mis en place pour piloter le robot et le superviser. Chaque message est identifié par un numéro de fonction unique et possède une charge utile de taille définie, définis 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)

FIGURE 3 – Fonctions de supervision

À l'aide du bouton de test, nous pouvons simuler l'envoi successif de chaque message défini. Dans notre interface graphique, nous allons ajouter des éléments permettant de visualiser les valeurs reçues en mode Loopback pour chaque type de message. Une fois les messages reçus et validés, ils sont traités par une fonction appelée ProcessDecodedMessage. Cette fonction prend en argument le numéro de fonction, la longueur de la charge utile et la charge utile elle-même.

Finalement, nous obtenons l'interface suivante :

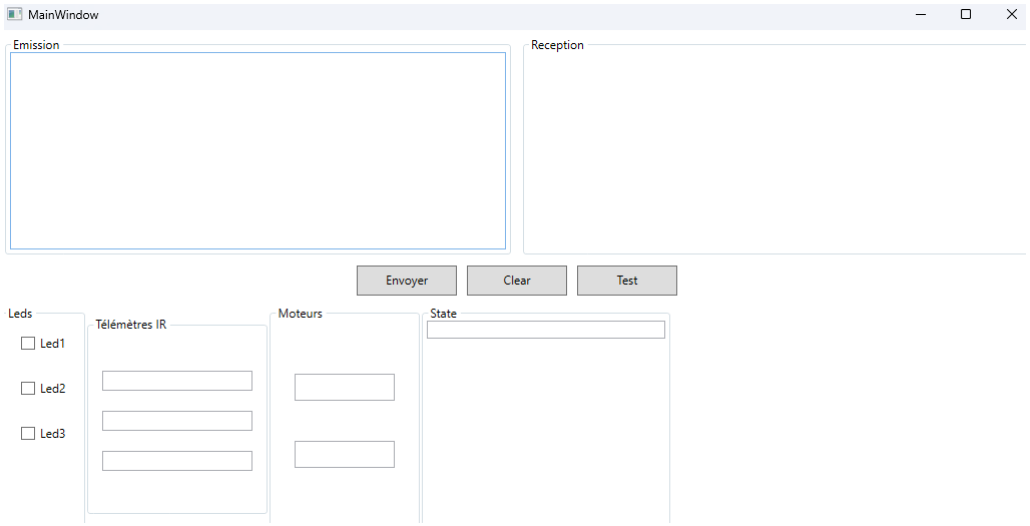


FIGURE 4 – Capture d'écran de l'interface

3.2 Implantation en électronique embarquée

Nous allons maintenant implanter notre code en C# dans le code de notre robot en C. Cette transition est essentielle pour intégrer le protocole de communication que nous avons conçu à l'écosystème logiciel de notre robot, permettant ainsi une interaction fluide et structurée entre les différents composants du système. Voici comment nous procédons pour transposer et intégrer les fonctions clés de notre protocole dans l'environnement C du robot.

Transposition du Calcul du Checksum

Le calcul du checksum est un élément fondamental de notre protocole pour garantir l'intégrité des données. La fonction C# suivante est transposée en C pour être utilisée côté robot :

```
unsigned char UartCalculateChecksum(int msgFunction, int msgPayloadLength, unsigned char*
     msgPayload) {
    unsigned char checksum = 0;
    checksum ^= 0xFE;
    checksum ^= (unsigned char)(msgFunction >> 8);
    checksum ^= (unsigned char)(msgFunction & 0xFF);
    checksum ^= (unsigned char)(msgPayloadLength >> 8);
    checksum ^= (unsigned char)(msgPayloadLength & 0xFF);
    for (int i = 0; i < msgPayloadLength; i++) {
        checksum ^= msgPayload[i];
    }
    return checksum;
}
```

Cette fonction en C reprend la logique de celle en C#, effectuant un XOR sur les différentes parties du message ainsi que sur chaque byte du payload.

Encodage et Envoi des Messages

Cette fonction est essentielle pour encoder les messages selon notre protocole défini et les envoyer à travers l'interface UART. L'objectif est de construire une trame de message contenant la fonction du message, la longueur de la charge utile (*payload*), le *payload* lui-même, et finalement un checksum pour assurer l'intégrité des données. Voici un exemple d'utilisation pour envoyer un message simple :

```
void UartEncodeAndSendMessage(int msgFunction, int msgPayloadLength, unsigned char*
    msgPayload) {
    unsigned char checksum = UartCalculateChecksum(msgFunction, msgPayloadLength,
    msgPayload);
    unsigned char Trame[6 + msgPayloadLength];
    int pos = 0;
    Trame[pos++] = 0xFE;
    Trame[pos++] = (unsigned char)(msgFunction >> 8);
    Trame[pos++] = (unsigned char)(msgFunction & 0xFF);
    Trame[pos++] = (unsigned char)(msgPayloadLength >> 8);
    Trame[pos++] = (unsigned char)(msgPayloadLength & 0xFF);
    for (int i = 0; i < msgPayloadLength; i++) {
        Trame[pos++] = msgPayload[i];
    }
    Trame[pos++] = checksum;
    SendMessageDirect(Trame, pos);
}
```

Cette fonction montre comment les données sont préparées et envoyées en respectant le format défini par notre protocole.

3.2.1 Supervision

Pour tester notre protocole de communication en intégrant à la fois des messages simples et une supervision plus complexe des étapes de déplacement du robot, nous combinons deux approches dans notre code en C. D'abord, l'envoi d'un message simple pour vérifier la capacité de notre système à transmettre et recevoir des données via l'interface graphique en C#. Ensuite, nous mettons en œuvre une fonction de supervision qui communique les transitions d'étapes de déplacement du robot, offrant une visibilité accrue sur son fonctionnement en temps réel.

Tests de Supervision

Nous commençons par envoyer un message simple, tel que "Bonjour", pour valider le bon fonctionnement de notre protocole de communication UART. Ce test initial nous permet de confirmer que l'encodage, l'envoi, et la réception des messages sont bien gérés par notre système.

```
unsigned char payload[] = {'B', 'o', 'n', 'j', 'o', 'u', 'r'};
UartEncodeAndSendMessage(0x0080, sizeof(payload), payload);
__delay32(40000000); % Temporisation pour permettre la r ception du message
```

L'utilisation de la temporisation `__delay32` sert à espacer les envois de messages pour s'assurer que l'interface graphique en C# ait suffisamment de temps pour traiter et afficher les informations reçues.

Fonction de Supervision des Étapes de Déplacement

Après avoir validé le mécanisme d'envoi avec un message simple, nous désactivons temporairement cet envoi de test pour intégrer une fonctionnalité de supervision plus avancée. Cette fonction, destinée à signaler les changements d'étapes de déplacement du robot, envoie des messages contenant le numéro d'étape et l'instant courant, codé sur 4 octets.

```
void SendStepTransitionMessage(int stepNumber) {
    // Obtenir l'instant courant en millisecondes
    clock_t currentTime = clock();
    double currentTimeMillis = ((double)currentTime / CLOCKS_PER_SEC) * 1000;

    // Cr er la payload du message de supervision
    unsigned char payload[5];
    // Copier l'instant courant en millisecondes dans les 4 premiers octets de la payload
    payload[0] = (unsigned char)(currentTimeMillis >> 24);
    payload[1] = (unsigned char)(currentTimeMillis >> 16);
    payload[2] = (unsigned char)(currentTimeMillis >> 8);
    payload[3] = (unsigned char)currentTimeMillis;
    // Copier le num ro de l' tape  dans le dernier octet de la payload
    payload[4] = (unsigned char)stepNumber;

    // Envoyer le message de supervision avec l'identifiant 0x0050 et la payload
    UartEncodeAndSendMessage(0x0050, 5, payload);
}
```

Cette fonction envoie un message avec l'identifiant 0x0050, incorporant à la fois le numéro d'étape actuelle et l'instant précis de cette transition. Cela permet une supervision détaillée et en temps réel des actions du robot, facilitant le diagnostic et l'analyse de son comportement.

Intégration avec l'Interface Graphique en C#

Pour exploiter au mieux les informations transmises par la fonction SendStepTransitionMessage, nous avons développé côté interface graphique en C# une fonction capable d'interpréter et d'afficher ces données dans la console de réception. Cette fonction analyse le payload des messages reçus pour extraire le numéro d'étape et le timestamp, qu'elle présente ensuite de manière claire et lisible. Ainsi, chaque changement d'étape du robot est consigné avec son instant précis de déclenchement, offrant une vue d'ensemble détaillée de son comportement au fil du temps.

```
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;
```

Ce code, intégré dans la fonction ProcessDecodedMessage, enrichit l'interface utilisateur, transformant des données brutes en informations utiles et facilement interprétables. Il améliore non seulement la compréhension en temps réel des actions du robot mais aussi la capacité à détecter et à analyser tout comportement inattendu ou tout retard dans l'exécution des tâches.

3.2.2 Pilotage

Après avoir mis en place les fonctions de supervision du robot, notre prochaine étape consiste à développer les fonctions de pilotage du robot. Dans cette phase du projet, notre objectif est de permettre au microcontrôleur embarqué de décoder les messages envoyés depuis l'interface graphique. Nous reprenons les fonctions que nous avons précédemment implémentées dans l'interface graphique en C et nous les réécrivons pour les adapter à l'environnement embarqué.

Conclusion

En conclusion, ce travail pratique a constitué une exploration approfondie et technique de la communication série via UART, complétée par l'implémentation de buffers circulaires et le développement d'un protocole de communication spécifique. À travers les différentes phases de ce projet, de la mise en place initiale de la communication série à l'implémentation de fonctions avancées pour la supervision des étapes de déplacement du robot, nous avons étudié la communication entre dispositifs électroniques ainsi que le développement d'une application concrète et fonctionnelle.

La première partie de notre travail a consisté à établir une communication série fiable entre le robot et une interface graphique, utilisant l'UART et des buffers circulaires pour une gestion efficace des flux de données. Cette base nous a permis de transmettre avec succès des messages simples et de vérifier la robustesse de notre système de communication.

Par la suite, l'introduction d'un protocole de communication a ajouté une structuration à nos échanges de données, permettant la transmission de messages complexes, incluant des commandes et des informations d'état du robot. La fonction `UartEncodeAndSendMessage`, en particulier, a joué un rôle central dans l'encodage et l'envoi de ces messages, assurant une communication claire et précise entre le robot et son système de contrôle.

Le développement de la fonction `SendStepTransitionMessage` a marqué une étape significative dans notre projet, offrant une visibilité en temps réel sur le comportement et les actions du robot. Cette capacité de supervision a ouvert de nouvelles voies pour le diagnostic, le monitoring, et l'optimisation du robot, en fournissant des informations détaillées sur son état et ses performances.

Enfin, l'intégration et le test de notre protocole de communication dans l'interface graphique en C# ont démontré la flexibilité et l'efficacité de notre approche dans la gestion des communications entre différents langages de programmation et plateformes.

Ce travail pratique a non seulement mis en lumière l'importance de la communication série dans les systèmes embarqués mais a également illustré le potentiel des protocoles de communication personnalisés dans la création de systèmes complexes et interactifs.

Annexes

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.IO.Ports;
using System.Linq;
using System.Text;
using System.Threading;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;
using System.Windows.Threading;
using ExtendedSerialPort_NS;
using RobotInterface;

namespace WpfApp1
{
    /// <summary>
    /// Logique d'interaction pour MainWindow.xaml
    /// </summary>
    public partial class MainWindow : Window
    {
        ExtendedSerialPort serialPort1;
        DispatcherTimer timerAffichage;

        Robot robot = new Robot();
        Queue<byte> byteListReceived = new Queue<byte>();


        public MainWindow()
        {
            InitializeComponent();
            serialPort1 = new ExtendedSerialPort("COM3", 115200, Parity.None, 8, StopBits.One);
```

```
            serialPort1.DataReceived += SerialPort1_DataReceived1;
            while (!serialPort1.IsOpen)
            {
                serialPort1.Open();
                Thread.Sleep(100);
            }

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



        }

        private void SerialPort1_DataReceived1(object? sender, DataReceivedArgs e)
        {
            foreach (byte b in e.Data)
            {
                robot.byteListReceived.Enqueue(b);

            }
        }

        private void TimerAffichage_Tick(object? sender, EventArgs e)
        {
            while (robot.byteListReceived.Count > 0)
            {
                byte b = robot.byteListReceived.Dequeue();
                DecodeMessage(b);
                //textBoxReception.Text += Encoding.ASCII.GetString(new byte[] { b });
            }

        }

        private void textBoxEmission_TextChanged(object sender, TextChangedEventArgs e)
        {

        }

        private void textBox_TextChanged(object sender, TextChangedEventArgs e)
        {

        }

        private void buttonEnvoyer_Click(object sender, RoutedEventArgs e)
        {
            if (buttonEnvoyer.Background == Brushes.RoyalBlue)
            {
                buttonEnvoyer.Background = Brushes.Beige;
            }
            else
            {
                buttonEnvoyer.Background = Brushes.RoyalBlue;
            }

            SendMessage();

        }

        private void textBoxEmission_KeyUp(object sender, KeyEventArgs e)
        {
            if (e.Key == Key.Enter)
            {
                SendMessage();
            }

        }

        private void SendMessage()
        {
            textBoxReception.Text += "Re u : " + textBoxEmission.Text + Environment.NewLine;
            serialPort1.WriteLine(textBoxEmission.Text);
            textBoxEmission.Text = "";
        }
```

```
        private void buttonClear_Click(object sender, RoutedEventArgs e)
        {
            textBoxReception.Text = "";
        }

        private void buttonTest_Click(object sender, RoutedEventArgs e)
        {
            //byte[] byteList = new byte[20];
            //for (int i = 0; i<byteList.Length; i++)
            //{
            //byteList[i] = (byte)(2*i);
            //}
            //serialPort1.Write(byteList, 0, byteList.Length);

            int msgfunction = 0x0080;
            string payload = "Bonjour";
            int msgPayloadLength = payload.Length;
            byte[] array = Encoding.ASCII.GetBytes("Bonjour");
            UartEncodeAndSendMessage(msgfunction, msgPayloadLength, array);
        }
        public byte CalculateChecksum(int msgFunction, int msgPayloadLength, byte[] msgPayload)
        {
            byte checksum = 0;
            checksum ^= 0xFE;
            checksum ^= (byte)(msgFunction >> 8);
            checksum ^= (byte)(msgFunction >> 0);
            checksum ^= (byte)(msgPayloadLength >> 8);
            checksum ^= (byte)(msgPayloadLength >> 0);
            for(int i = 0; i< msgPayloadLength; i++)
            {
                checksum ^= msgPayload[i];
            }
            return checksum;

        }

        void UartEncodeAndSendMessage(int msgFunction,int msgPayloadLength, byte[] msgPayload)
        {
            byte checksum= CalculateChecksum(msgFunction, msgPayloadLength, msgPayload);

            byte[] Trame = new byte[6+ msgPayloadLength];
            Trame[0] = 0xFE;
            Trame[1] = (byte)(msgFunction >> 8);
            Trame[2] = (byte)(msgFunction >> 0);
            Trame[3] = (byte)(msgPayloadLength >> 8);
            Trame[4] = (byte)(msgPayloadLength >> 0);
            Array.Copy(msgPayload, 0, Trame, 5, msgPayloadLength);
            Trame[Trame.Length - 1] = checksum;

            serialPort1.Write(Trame, 0, Trame.Length);
        }
        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:
                    if (c == 0xFE)
                        rcvState = StateReception.FunctionMSB;
                        break;
                case StateReception.FunctionMSB:
```

```
                    msgDecodedFunction = (c << 8);
                    rcvState = StateReception.FunctionLSB;
                        break;
                case StateReception.FunctionLSB:
                    msgDecodedFunction |= (c << 0);
                    rcvState = StateReception.PayloadLengthMSB;
                        break;
                case StateReception.PayloadLengthMSB:
                    msgDecodedPayloadLength = (c << 8);
                    rcvState = StateReception.PayloadLengthLSB;
                        break;
                case StateReception.PayloadLengthLSB:
                    msgDecodedPayloadLength |= (c << 0);
                    if (msgDecodedPayloadLength == 0)
                    {
                        rcvState = StateReception.CheckSum;
                    }
                    else if (msgDecodedPayloadLength >=256)
                    {
                        rcvState = StateReception.Waiting;
                    }
                    else
                    {
                        rcvState = StateReception.Payload;
                        msgDecodedPayload = new byte[msgDecodedPayloadLength];
                        msgDecodedPayloadIndex = 0;
                    }
                        break;
                case StateReception.Payload:
                    msgDecodedPayload[msgDecodedPayloadIndex] = c;
                    msgDecodedPayloadIndex++;
                    if (msgDecodedPayloadIndex >= msgDecodedPayloadLength)
                    {
                        rcvState = StateReception.CheckSum;
                    }
                        break;
                case StateReception.CheckSum:
                    byte receivedChecksum = c;
                    byte calculatedChecksum = CalculateChecksum(msgDecodedFunction,
    msgDecodedPayloadLength, msgDecodedPayload);
                    if (calculatedChecksum == receivedChecksum)
                    {
                        serialPort1.WriteLine("Correct");
                        ProcessDecodedMessage(msgDecodedFunction, msgDecodedPayloadLength,
    msgDecodedPayload);
                    }
                    else
                    {
                        serialPort1.WriteLine("Incorrect");
                    }
                    rcvState = StateReception.Waiting;
                    break;
                default:
                    rcvState = StateReception.Waiting;
                    break;
            }
        }
        public enum SupervisionCommand
        {
            TransmissionTexte = 0x0080,
            ReglageLED = 0x0020,
            DistancesTelemetreIR = 0x0030,
            ConsigneVitesse = 0x0040
        }

        void ProcessDecodedMessage(int msgFunction,int msgPayloadLength, byte[] msgPayload)
        {
            switch ((SupervisionCommand)msgFunction)
            {
                case SupervisionCommand.TransmissionTexte:
                    string texte = Encoding.ASCII.GetString(msgPayload);
                    textBoxReception.Text = texte;
                    break;
                case SupervisionCommand.ReglageLED:
                    if (msgPayloadLength == 2)
                    {
```

```
                        int ledNumber = msgPayload[0];
                        bool ledValue = msgPayload[1]==1;
                        switch(ledNumber)
                        {
                            case 1:
                                LED1CheckBox.IsChecked = ledValue;
                                break;
                            case 2:
                                LED2CheckBox.IsChecked = ledValue;
                                break;
                            case 3:
                                LED3CheckBox.IsChecked = ledValue;
                                break;
                            default:
                                break;
                        }
                    }
                    break;
                case SupervisionCommand.DistancesTelemetreIR:
                    if (msgPayloadLength == 3)
                    {
                        IRGauche.Content = msgPayload[0].ToString() + "cm";
                        IRCentre.Content = msgPayload[1].ToString() + "cm";
                        IRDroit.Content = msgPayload[2].ToString() + "cm";
                    }
                    break;
                case SupervisionCommand.ConsigneVitesse:
                    if (msgPayloadLength == 2)
                    {
                        int vitesseGauche = msgPayload[0];
                        int vitesseDroite = msgPayload[1];
                        VitesseGauche.Content = vitesseGauche.ToString() + "%";
                        VitesseDroit.Content = vitesseDroite.ToString() + "%";
                    }
                        break;
                default:
                    break;
            }
        }


    }
}
```

Listing 1 – Code de l'interface graphique en C#