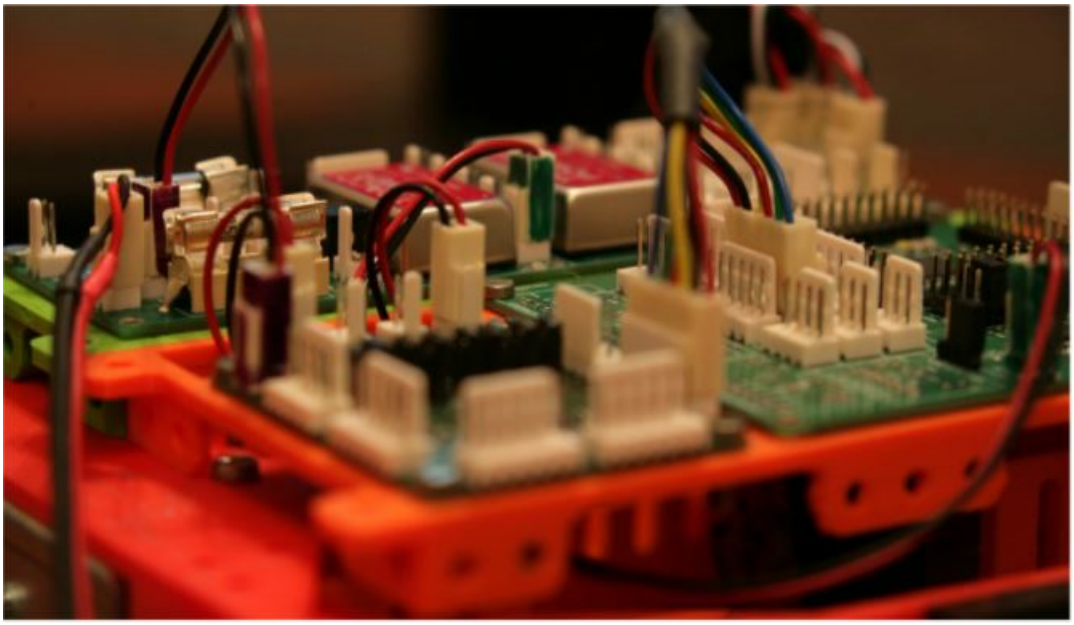
SeaTech
ÉCOLE D'INGÉNIEURS

ELECTRONIQUE



Supervision et pilotage d'un robot mobile autonome

PROVOST MATHIS
ROMENSKY VLADIMIR
Groupe A - Table 6

Enseignant : V.GIES
Année : 2020-2021
SYSMER 2A

Table des matières

Partie I
Introduction

La programmation orientée objet est un modèle de langage de programmation qui s'articule autour d'objets et de données. Notre objectif est de faire interagir des objets entre eux. La première étape consiste à identifier tous les objets que l'on souhaite manipuler et leurs interactions.
Après s'être focalisés sur le microcontrôleur et sa programmation, notre objectif est, dans cette seconde partie, de constituer une communisation en C# entre le microcontrôleur et l'ordinateur.
Nous cherchons donc à transmettre des consignes à notre robot mobile et à recevoir ses données en retour. Pour ce faire, nous allons utiliser l'interface de Visual Studio.

Partie II
A la découverte de la programmation orientée objet en C#

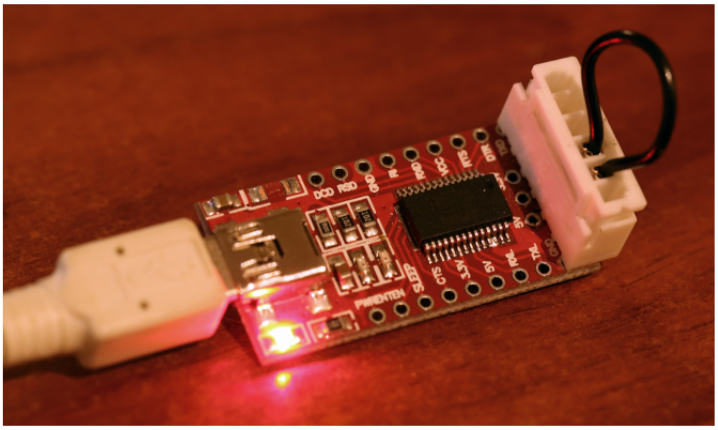
1 Simulateur de messagerie instantanée

Dans cette partie, nous allons nous initier à la programmation en C#. Pour cela nous allons utiliser l'interface de Visual Studio afin de transmettre des consignes au robot et d'en recevoir des données. Nous avons créé un simulateur de messagerie instantanée sur Visual Studio avec lequel nous avons pu envoyer et recevoir des messages envoyés en interne sur le PC. L'interface de Visual Studio permet de manipuler l'IHM et d'accéder à son code, dans la fenêtre CodeBehind, afin de retravailler l'interface graphique et de pouvoir facilement corriger les erreurs du code. Nous avons ensuite écrit les différentes fonctions associées aux boutons de l'interface (envoyer, effacer, test) afin qu'ils effectuent les actions voulues associées à chacun de ces boutons.

```
bool cpt=true;
private void buttonEnvoyer_Click(object sender, RoutedEventArgs e)
{
    if( cpt)
        buttonEnvoyer.Background = Brushes.RoyalBlue;
    else
        buttonEnvoyer.Background = Brushes.Beige;
    cpt=!cpt ;
    SendMessage();
}
```

2 Messagerie instantanée entre deux PC

Une fois la prise en main terminée, nous avons travaillé sur la conception d'une messagerie instantanée entre deux appareils, avec un ordinateur .Pour cela, nous avons eu recours à un module, le FT232RL. Ce dernier permet d'échanger des informations via le port série du microcontrôleur et de l'ordinateur. L'objectif est ici de transmettre un message à une autre machine pour qu'elle le récupère et le lise. Nous avons donc simulé une transmission de données entre notre ordinateur et un autre avec un fil reliant notre pin de sortie à notre pin d'entrée de donnée. Ainsi ce que nous envoyons nous est directement retourné (loopback) . L'objectif ici est de relier la commande à un port série afin que les données envoyées passent pour ensuite les récupérer et les transmettre. Lors de l'affichage dans la RichTextBox des données récupérées par la méthode ReadExisting(), l'exécution du programme plante. Cette erreur est due au multithreading. L'ordinateur sépare sa charge de travail en plusieurs tâches réalisées les unes après les autres. Pour une fois, l'exécution, on ait l'impression des différentes tâches sont exécutées en même temps ce qui n'est pas réellement le cas. Un thread ne doit pas agir sur les autres threads. Pour éviter cela, il faut passer par des variables de passage qui sont en dehors de ces derniers. Ici pour régler le problème nous utilisons une chaîne de caractères « receivedText ». À présent l'envoi de données et la réception fonctionnent parfaitement en LoopBack. L'ajout d'un bouton Clear va nous permettre de vider la RichTextBox contenant les dernières données reçues. Un DispatcherTimer regardera alors à intervalles réguliers si des données sont reçues. Il est alors possible d'envoyer des messages à d'autres ordinateurs.



3 Liaison série hexadécimale

Dans cette partie nous allons apprendre à transmettre n'importe quel type de caractère. Pour tester cela, nous allons ajouter un bouton « Test » qui envoie une série de bytes enregistrée dans un tableau « byteList ». Pour l'instant les données reçues sont une succession de bytes sans interprétation possible, ainsi il faudrait regrouper les paquets de bytes en octet pour les afficher en hexadécimal. Pour cette partie, nous allons rester en mode LoopBack avec le module. Pour ce faire nous allons créer un FIFO (First In First Out). Un FIFO est une pile de données, où l'on superpose ces dernières par le haut et on les récupère par le bas. Ainsi la première à entrer est la première à sortir. Cette pile nous permet ainsi de récupérer les paquets de bytes et de les convertir en caractères grâce à la fonction "byte.ToString()". Nous pouvons désormais envoyer des octets et les afficher pour une interprétation.

Partie III
A la découverte de la communication embarquée

4 La liaison série embarquée

Dans cette partie, la communication se fera à l'aide d'une connexion micro-USB/USB. Pour démarrer, le sujet nous fournit un code permettant de paramétrer la carte suite à l'ajout des fichiers UART dans le microchip, nous allons réutiliser la carte principale reliée à une carte capteurs où les capteurs sont directement branchés sur l'ordinateur car la carte principale ne possède pas de liaison USB. On crée donc un fichier source que l'on nomme UART.c dans lequel figure le paramétrage de la carte. On initialise l'UART à la vitesse de 115200 baud, l'unité "baud" correspondant à un symbole par seconde, et donc la vitesse de transmission de données.

5 Échange de données entre le MC et le PC

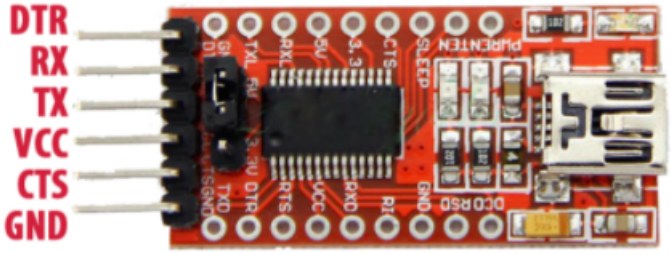
5.1 Validation du bon fonctionnement du convertisseur USB/série

Pour s'assurer d'un bon fonctionnement du convertisseur, réalise physiquement un Loopback en plaçant un jumper sur le dongle USB-série. Cela va relier par un fil les PIN Rx et Tx. En plaçant correctement les voies de l'oscilloscope, notre objectif est maintenant d'envoyer des messages et de les visualiser. On place l'oscilloscope en mode BUS afin de pouvoir visualiser ce qui arrive sur la Pin Rx ou Tx. Pour cela, on envoie des messages de manière périodique (toutes les secondes par exemple). Tant que l'on n'envoie pas un nouveau message, l'attente est bloquante.

5.2 Émission UART depuis le microcontrôleur

Grâce au schéma de câblage du microcontrôleur on peut trouver les numéros de pin remappables correspondant à Rx et à Tx. Le remappage de Pin permet de relier de manière plus pratique les pins. En effet certaines pins peuvent être remappées, c'est-à-dire associées à d'autres pins grâce à quelques lignes de code. On aboutit au remappage suivant :

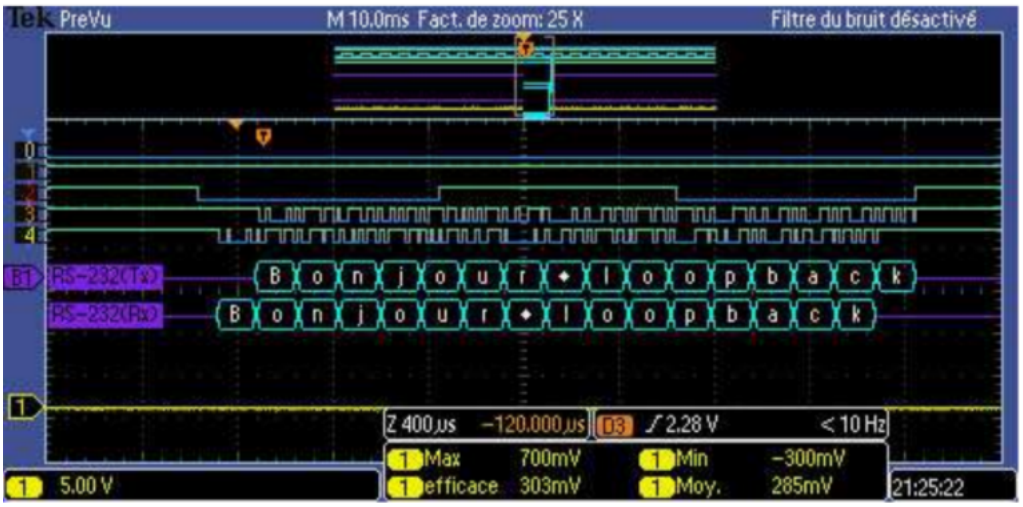
- renvoi de Rx sur RP24
- renvoi de TX ur RP36



Une fois le remappage configuré dans le code “IO.c”, on peut commencer à envoyer des données via la carte. Notre objectif est maintenant d’envoyer des messages et de les visualiser. Pour cela, on envoie des messages de manière périodique (toutes les secondes par exemple). Tant que l’on n’envoie pas un nouveau message, l’attente est bloquante. En branchant l’oscilloscope, nous pouvons observer l’envoi du message ”Bonjour”. Le message à donc bien été envoyé.

5.3 Réception

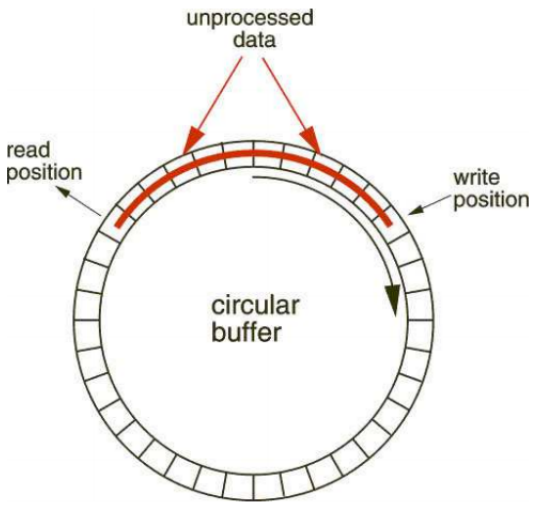
Pour confirmer la bonne réception des messages, nous allons utiliser le logiciel en mode LoopBack. Ainsi, dès qu’un caractère arrive en Rx, il est réémis en Tx. Afin de détecter les messages qui sont émis de manière asynchrone, nous allons utiliser les interruptions en réception du programme UART. Il nous est donc nécessaire de changer la variable du “IEC0bits.U1RXIE” qui était initialement à 0, nous la passons donc désormais à 1 afin de rendre possible ces interruptions. Pour vérifier le bon fonctionnement, on envoie un message depuis Visual Studio et on observe sur l’oscilloscope à la fois l’émission et la réception du message.



6 Liaison série avec FIFO intégré

6.1 Le buffer circulaire de l'UART en émission

Un buffer circulaire est une structure de données qui permet un stockage temporaire de données dont la tête et la queue sont considérées comme connectées. Les données sont stockées une par une dans les cases. La position de la tête indique le lieu de stockage du prochain byte, puis la position de la tête s'incrémente de 1 afin que le buffer soit prêt à accueillir le prochain byte, et la lecture des données se fait par la queue et remonte jusqu'à la tête. Les buffers circulaires ont un comportement type FIFO.



Le buffer circulaire permet d'augmenter la mémoire embarquée. Néanmoins, il faut faire attention à ce que celui-ci ne soit pas plein (c'est-à-dire que toutes les cases de la tête à la queue soient remplies) car cela aboutirait à la perte de données en enregistrant des nouvelles. Dans notre cas, nous avons choisi un buffer de 128 bytes et nous supposons qu'aucun de nos messages envoyé ne dépassera la taille du buffer. Nous devons donc programmer ce buffer circulaire sur un nouveau fichier source appelé "BCT X1.c". A chaque envoi de message, la fonction SendMessage fait appel à la fonction "SendOne( )" à condition qu'une transmission ne soit pas déjà en cours. Les caractères émis sont alors stockés dans le buffer circulaire jusqu'à ce que le pointeur de la queue soit dans la même position que le pointeur de la tête, auquel cas le buffer est plein, et il en résulterait une perte de données.

De manière analogue, comme le code fonctionne avec les interruptions Tx, on initialise dans le programme de l'UART "IEC0bits.U1TXIE" à 1 au lieu de 0 initialement. Les interruptions sont désormais possibles. Après inclusion de tous les fichiers utiles dans le main, nous compilons notre programme et remarquons que celui-ci fonctionne de manière analogue au précédent mais sans être bloquant cette fois.

6.2 Le buffer circulaire de l'UART en réception

Pour réceptionner les données venant de l'ordinateur, nous avons implémenté un second buffer circulaire qui permet de stocker les données avant de venir les lire. Ainsi le stockage se fera sur interruption. Nous pouvons facilement tester le fonctionnement de notre transmission avec l'oscilloscope en envoyant un message.

On constate que les caractères ne sont pas tous transmis correctement. Cela est dû au "delay32(1000)" . Cette commande simule une charge du processeur pendant un certain temps où les informations ne sont pas transmises au robot. Il va stocker les données dans le buffer de réception en attendant la prochaine interruption pour les lire. En augmentant la valeur de "delay32", il y aura plus de temps entre deux lectures des données sur le buffer et donc plus de données stockées dans le buffer entre deux lectures.

Partie IV
A la découverte de la supervision d'un système embarqué

7 Implantation d'un protocole de communication

Dans cette partie, nous cherchons à implémenter un protocole de communication plus développé que celui dont on dispose pour l'instant. En effet, il était précédemment possible de faire passer des suites d'octets entre 0x00 et 0xFE. Cependant ces suites d'octet ne sont pas identifiables en type de message et il est donc impossible de savoir s'il y a eu des erreurs de transmission ou non. Nous avons donc constitué une trame pour chaque message envoyé, afin de normaliser la communication et donc, la rendre compréhensible.

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

Les trames envoyées sont constituées de :

- Un octet symbolisant le début de la trame
- Deux octets pour catégoriser le message
- Deux octets pour la longueur du message
- N octets pour le message
- Un octet servant à vérifier la bonne transmission du message

Grâce à ce type de protocole, la transmission devient alors plus robuste et permet de détecter si le message reçu est conforme au message émis. En effet, le CheckSum est un nombre que l'on ajoute à un message à transmettre afin de permettre au récepteur de vérifier si le message qu'il a reçu est bien celui qui avait été envoyé. Cela permet la détection mais pas la correction des erreurs. Concrètement, si le CheckSum reçu n'est pas valide, les données ne sont pas récupérées pour ne pas encombrer l'espace avec des données fausses.

7.1 Encodage des messages

Pour encoder un message, la première des fonctions à coder est le calcul du CheckSum qui sera utilisé dans la génération de la trame. Le calcul du checksum se fait par l'opération d'un "ou exclusif" bit à bit des données de la trame.

L'opération du "ou exclusif" est réalisée par la commande "^=". La commande ">> " , quant à elle, permet de décaler le pointeur sur le message du nombre de bit indiqué.

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

Ce code permet de vérifier qu'il n'y ait pas eu d'erreur lors de la transmission et de son décodage à la réception, c'est une sécurité supplémentaire permettant d'assurer la bonne communication des appareils. Si la comparaison entre les deux checksums n'est pas validée, nous ne récupérerons pas les données. La fonction suivante permet de former et d'envoyer une trame de données sur la liaison série, elle utilise la fonction checksum à la fin afin de permettre au récepteur de savoir si le message reçu est le bon. Lors du test avec une trame correspondant à « Bonjour » nous obtenons à l'oscilloscope le bon checksum 0x38.

```
148         void UartEncodeAndSendMessage(int msgFunction, int msgPayloadLength, byte [] msgPayload)
149         {
150             byte [] message = new byte [msgPayloadLength+6];
151             int pos = 0;
152             message[pos++] = 0xFE;
153             message[pos++] = (byte)(msgFunction >> 8);
154             message[pos++] = (byte)(msgFunction >> 0);
155             message[pos++] = (byte)(msgPayloadLength >> 8);
156             message[pos++] = (byte)(msgPayloadLength >> 0);
157             for (int i = 0; i < msgPayloadLength; i++)
158                 message[pos++]= msgPayload[i];
159             message[pos++] = CalculateChecksum(msgFunction, msgPayloadLength, msgPayload);
160             serialPort1.Write(message,0,message.Length);
161         }
162
```

7.2 Décodage des messages

Les messages sont désormais encodés et envoyés correctement. Il faut maintenant programmer la fonction de décodage de la trame à sa réception. La programmation de la fonction de décodage se rapproche sensiblement de celle d'encodage mais effectuée à l'envers. Nous allons utiliser une machine à état avec une structure en « Switch case » avec la variable "rcvState" qui va indiquer l'état actuel du système. Cette structure permet d'analyser chaque byte réceptionné, et de déduire à quelle partie de la trame ce byte correspond.

```
180         private void DecodeMessage(byte c)
181         {
182             switch(rcvState)
183             {
184                 case StateReception.Waiting:
185                     if (c == 0xFE)
186                     {
187                         rcvState = StateReception.FunctionMSB;
188                     }
189                         break;
190                 case StateReception.FunctionMSB:
191                     msgDecodedFunction = (int)(c << 8);
192                     rcvState = StateReception.FunctionLSB;
193                     break;
194                 case StateReception.FunctionLSB:
195                     msgDecodedFunction += (int)(c<<0 );
196                     rcvState = StateReception.PayLoadLengthMSB;
197                     break;
198                 case StateReception.PayLoadLengthMSB:
199                     msgDecodedPayloadLength= (int)(c << 8);
200                     rcvState = StateReception.PayLoadLengthLSB;
201                     break;
202                 case StateReception.PayLoadLengthLSB:
203                     msgDecodedPayloadLength += (int)(c<<0);
204                     if(msgDecodedPayloadLength==0)
205                         rcvState = StateReception.Checksum;
206                     else if(msgDecodedPayloadLength > 512)
207                         rcvState = StateReception.Waiting;
208                     else
209                     {
210                         rcvState = StateReception.Payload;
211                         msgDecodedPayloadIndex = 0;
212                         msgDecodedPayload = new byte[msgDecodedPayloadLength];
213                     }
214                     break;
```

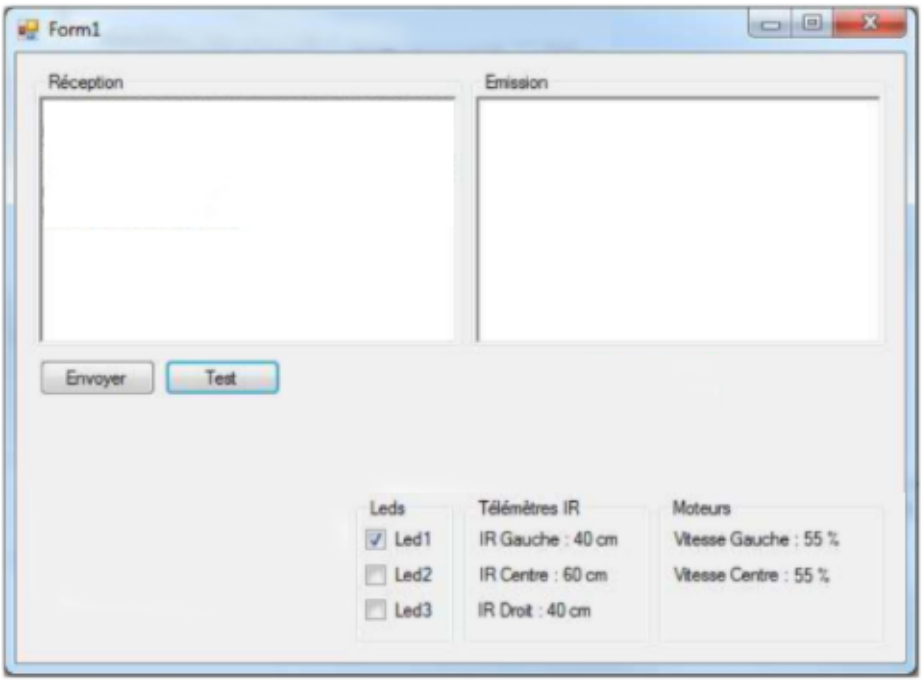
Lorsqu'un message arrive, on transmet un octet à la fois. Désormais, tous les messages auront la même forme : ils seront constitués d'un "StartofFrame", suivi d'une commande, suivie d'arguments (Payload) et enfin terminant par un CheckSum. Le CheckSum est un nombre que l'on ajoute à un message à transmettre afin de permettre au récepteur de vérifier si le message qu'il a reçu est bien celui qui avait été envoyé. Cela permet la détection mais pas la correction des erreurs. Concrètement, si le CheckSum reçu n'est pas valide, les données ne sont pas récupérées pour ne pas encombrer l'espace de données fausses.

7.3 Pilotage et supervision du robot

Maintenant que les messages peuvent être envoyés, reçus, décodés puis interprétés, on va pouvoir piloter et superviser le robot. La supervision permet de rendre observables certaines des variables internes au robot. Le pilotage et la supervision sont basés sur un ensemble de messages définis à l'avance et qui forment une bibliothèque devant être connue du robot et de la plate-forme de supervision. Chacun de ces messages a un numéro de fonction unique, et une payload de taille définie à l'avance. Grâce au travail fait précédemment, nous pouvons mettre en place une liste d'ID de commande, avec la description de leur payload

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)

On peut alors obtenir l'interface graphique suivante :



Nous avons donc maintenant une interface permettant de visualiser certains paramètres du robot ainsi que son état de fonctionnement actuel. Ce dialogue est aussi possible dans l'autre sens, nous pouvons envoyer des informations au robot, comme des consignes de vitesse de moteur. Après les fonctions de supervision, nous allons nous préoccuper des fonctions de pilotage du robot. Ici, il faut que le microcontrôleur puisse à son tour décoder les messages envoyés sur l'interface graphique. Nous reprenons donc les fonctions codées précédemment pour les écrire en embarqué via MPLAB X.

8 Implantation en électronique embarquée

8.1 Supervision

Nous allons alors implanter un protocole de communication par-dessus les précédents. Pour cela, on va réécrire les fonctions permettant de calculer le checksum et d'encoder un message en C cette fois-ci. On ajoute également une temporisation (__delay32(40000000)) qui correspond au DispatcherTimer que l'on avait initialisé en C. Cela permet de regarder régulièrement si un message a été reçu et répond au problème posé par le fait que les processeurs ne peuvent faire plusieurs tâches en même temps. Grâce à cette temporisation les tâches vont alors êtres mises à la file indienne et fractionnées pour qu'elles soient toutes avancées à peu près en parallèle Ensuite on peut tester les fonctions mises en place précédemment dans le main car on est désormais en C. Les caractères du message sont reçus en premier, puis le message s'affiche. On a alors considéré la fonction permettant d'obtenir les distances obstacle-robot grâce aux télémètres pour y ajouter une ligne de code permettant d'envoyer un message au niveau de l'interface graphique C pour avoir les valeurs. On implémente alors une nouvelle fonction de supervision permettant de savoir quand de nouvelles consignes de vitesse sont envoyées au moteur et donc quand le robot va changer de mode de déplacement Pour afficher dans l'interface graphique le mode de déplacement en cours et son déclenchement, on ajoute les lignes de code ci-dessous à la fonction ProcessDecodedMessage().

```
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;
```

8.2 Pilotage

Après les fonctions de supervision, nous allons nous préoccuper des fonctions de pilotage du robot. Ici, il faut que le microcontrôleur puisse décoder les messages envoyés sur l'interface graphique. Nous reprenons donc les fonctions codées précédemment pour les écrire en embarqué.

```
void UartProcessDecodedMessage(unsigned char function,
                               unsigned char payloadLength, unsigned char payload[])
{

    switch (msgFunction)
    {
        case SET_ROBOT_STATE:
            SetRobotState(msgPayload[0]);
            break;
        case SET_ROBOT_MANUAL_CONTROL:
            SetRobotAutoControlState(msgPayload[0]);
            break;
        default:
            break;
    }

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

8.3 Pilotage au clavier

Afin de pouvoir piloter le robot à l'aide du clavier nous allons utiliser une bibliothèque externe afin d'implanter des événements aux touches du clavier. On référence cette bibliothèque dans l'onglet "Référence" du projet, puis on dit au code que l'on va utiliser cette bibliothèque :

```
using MouseKeyboardActivityMonitor.WinApi;
using MouseKeyboardActivityMonitor;

private readonly KeyboardHookListener m_KeyboardHookManager;
```

On vient ensuite implanter une “SwitchCase” qui va détecter quel bouton du clavier est pressé et quelle action doit être réalisée (avancer, reculer, tourner, s’arrêter,....) :

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

Partie V
Conclusion

La seconde partie du projet a été dédiée à l'étude de procédés d'échanges de données entre les machines. Cette partie nous a permis de comprendre le fonctionnement de l'échange de donnée entre deux entités. De plus, elle nous a permis de comprendre l'intérêt des fonctions de codage et de décodage ainsi que leur structure. Cette partie avait pour but de pouvoir communiquer en temps réel avec le robot, afin d'obtenir des informations sur son état ou de lui envoyer des ordres directement, sans avoir à le manipuler. Nous avons découvert aussi, à travers cette partie, la programmation de la mémoire d'un microcontrôleur, absente jusque-là. Finalement, ce projet nous aura permis de mieux comprendre le fonctionnement et la programmation d'un microcontrôleur pour le contrôle d'un robot. L'autonomie donnée lors du projet nous aura permis de développer au mieux notre capacité de réflexion autour de problèmes de programmation, de communication entre ordinateurs et robots à des fins de supervision et de pilotage en temps réel.