Projet
conception d'un robot mobile

Professeur : Valentin Gies

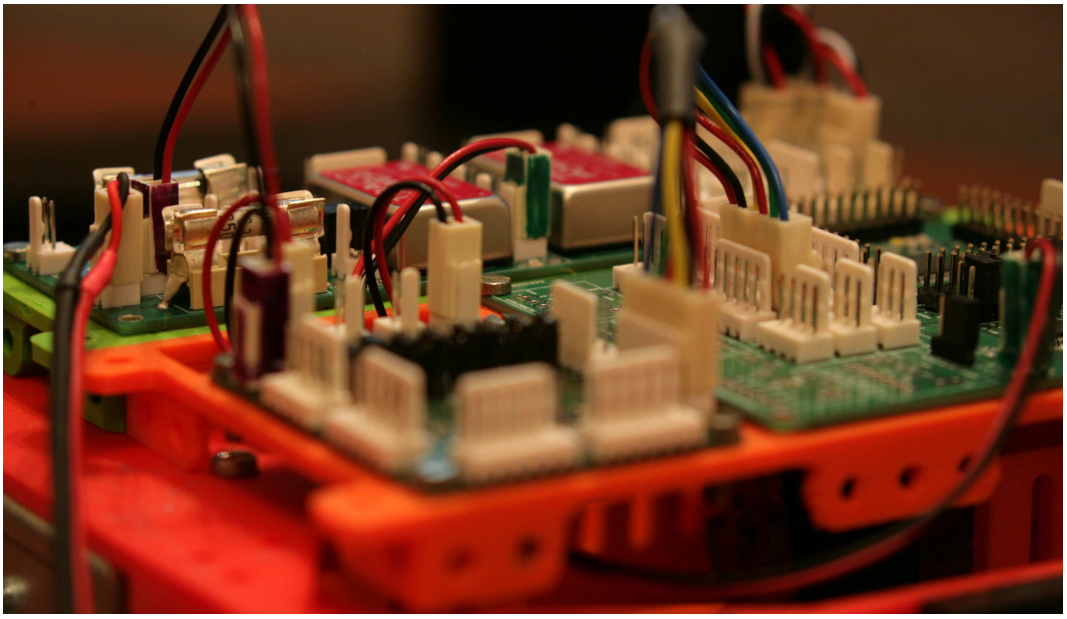


Table des matières

Ce projet est prévu pour se dérouler sur de plusieurs séances de 3 ou 4 heures. Il vous permettra de développer de A à Z les bases d'un robot mobile. Il vous permettra de vous familiariser avec les processeurs 16 bits de chez Microchip dans un premier temps, avant de mettre en oeuvre les périphériques qu'il intègre, en particulier : timers, ADC, PWM. L'étude des timers vous permettra de comprendre comment lancer périodiquement des évènements sur interruption. Celle de l'ADC vous permettra d'interfacer des capteurs externes tels que des télémètres infra-rouge. L'étude des PWM vous permettra de piloter des moteurs à courant continu via des hacheurs de puissance.

Vous découvrirez ensuite comment implanter intégralement plusieurs bus terrains (bus série UART, I2C, SPI), et vous vous familiariserez avec la programmation orientée objet en $C\#$, utilisée à des fins de contrôle distant de la plate-forme.

Le premier d'entre eux est le bus série, étudié ici dans une implantation proche d'une application industrielle et des niveaux 1 à 3 de la norme OSI, caractéristiques de bus terrains. En particulier ce bus sera implanté au niveau hardware avec un fonctionnement sur interruptions (couche 1 de la norme OSI), des buffers circulaires de stockage, et un protocole de messages (couches 2 et 3 de la norme OSI).

Cette liaison terrain série permettra d'échanger des données et des commandes avec un logiciel écrit en $C\#$. Cela permettra de piloter la carte électronique de manière distante et de faire remonter des données en provenance des capteurs situés sur la carte.

Le second bus terrain mis en oeuvre dans ce TP est le bus SPI, permettant ici de contrôler un gyroscope 3 axes, de récupérer les données et de les afficher en temps réel, toujours sur l'interface $C\#$.

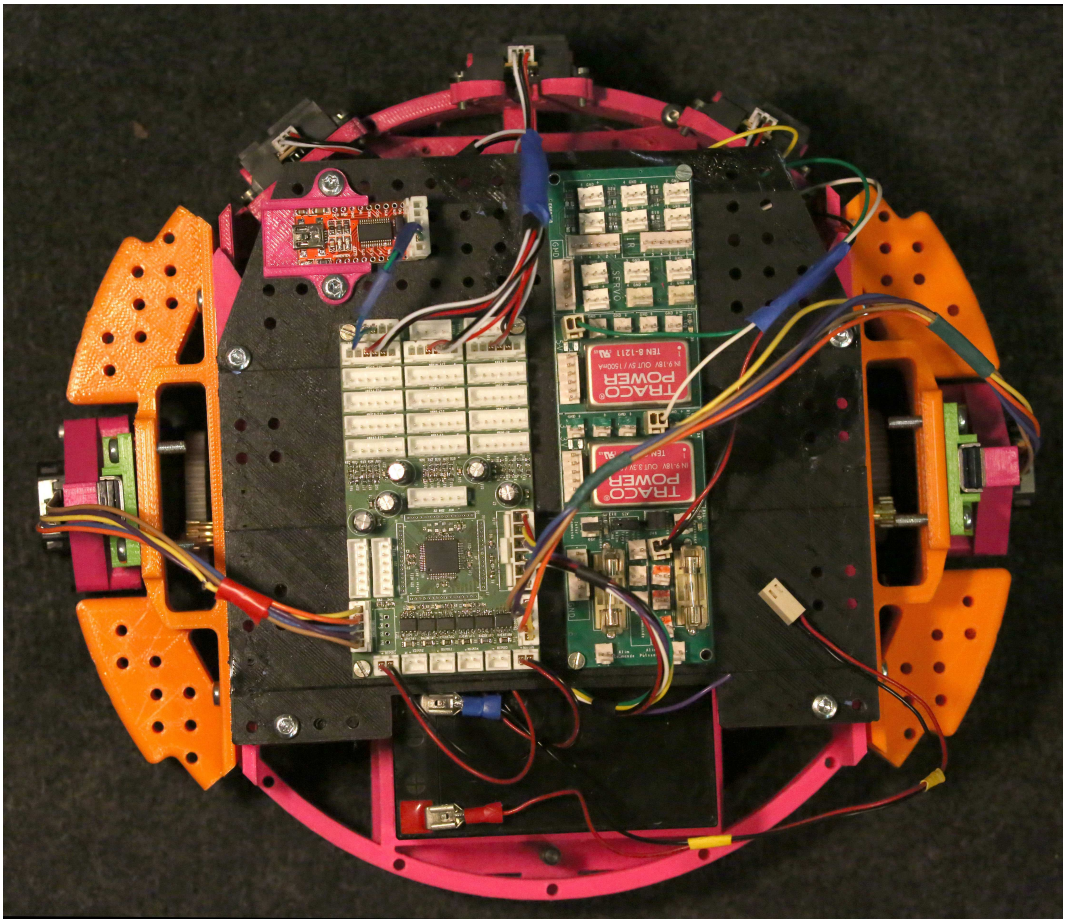


FIGURE 1 – Cartes électroniques du robot

L'électronique du robot est constituée de deux cartes dans sa version standard :

— La *carte d'alimentation* assure la génération des tensions stabilisées pour l'ensemble du robot.
— La *carte principale* permet de piloter 6 moteurs avec un contrôleur de type dsPIC dédié. Elle permet

également d'interfacer trois bus série (UART), deux bus *SPI* et un bus *I2C*.15 capteurs ou actionneurs peuvent être également connectés sur la carte moteur afin de piloter servomoteurs ou récupérer les données capteurs. Cette carte, ne possède pas de circuit de régulation de tension, elle doit donc être alimentée par l'extérieur, par exemple par la carte d'alimentation.

L'alimentation se fait uniquement par le connecteur de puissance. L'emplacement est indiqué sur la sérigraphie (fig. 2).

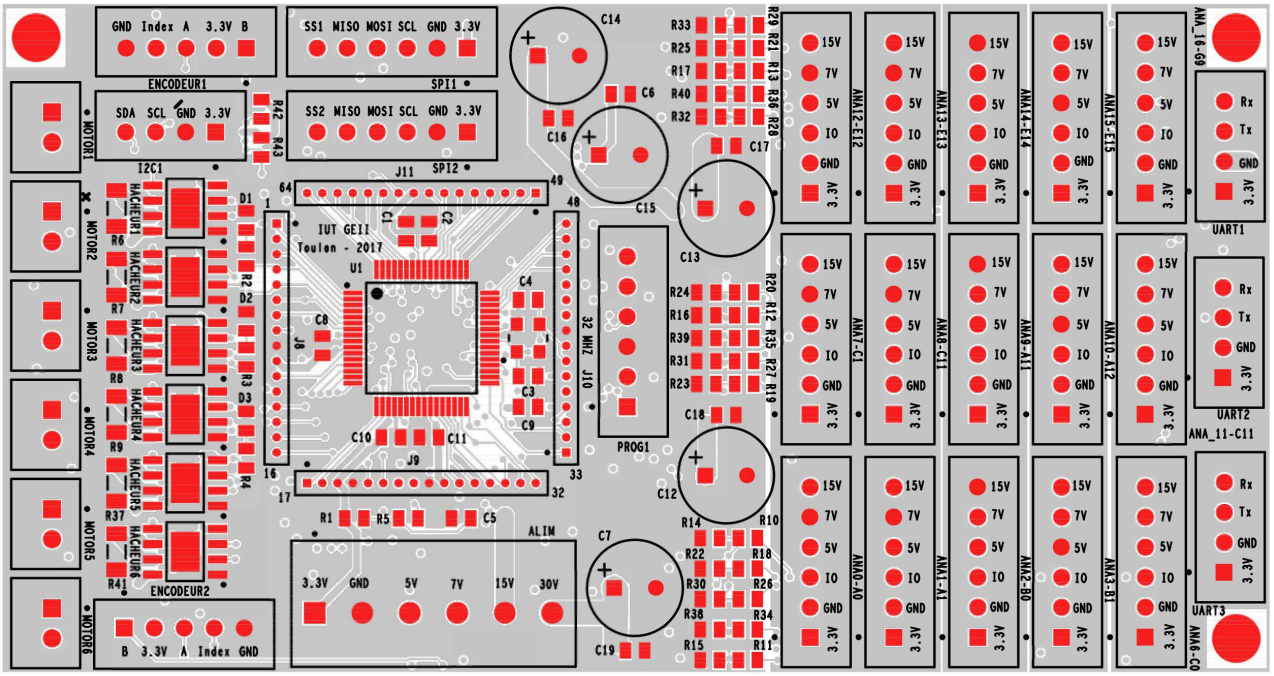


FIGURE 2 – Carte principale

1 Installation et prise en main de l'environnement de développement

L'environnement de développement se compose de 2 logiciels couplés :

— MPLAB X : MPLAB X permet de gérer des projets utilisant des microcontrôleurs PIC ou dsPIC. Il comprend un éditeur de code, un compilateur de code assembleur en code machine, et un module permettant d'envoyer le programme compilé au PIC via une interface (ici on utilisera l'interface MPLAB ICD 3).
— XC16 : permet de générer du code assembleur destiné au PIC 16 bits à partir d'un code C.

Ces deux logiciels permettent donc de programmer un PIC à l'aide d'un code en C.

1.1 Installation de MPLAB

Cette partie peut être optionnelle si les logiciels sont déjà installés sur votre machine. Elle est toutefois présentée afin que vous puissiez réaliser l'installation sur vos ordinateurs (les logiciels sont gratuits) pour travailler sur vos projets.

Attention : Ne pas brancher le programmateur ICD 3 avant d'avoir terminer l'installation du soft.

⟹ Vous pouvez télécharger les fichiers d'installation des 2 logiciels (versions Windows) dans leurs dernières versions à l'aide des liens suivants :
- **MPLAB X**
- **XC16 (free version)**

⟹ Installer si ça n'est pas déjà fait MPLAB X, en laissant les options par défaut. Redémarrer si besoin.

⟹ Installer ensuite XC16 si ça n'est pas déjà fait, en laissant également les options par défaut. XC16 sera intégré automatiquement à MPLAB X, sans action de votre part.

⟹ Au premier lancement de MPLAB X, un message peut vous demander si vous voulez importer les paramétrages d'une version antérieure. **Attention : il est important de répondre *NON***.

⟹ Branchez à présent le boîtier de programmation *ICD 3* sur le PC. Il doit être reconnu par Windows. En cas de problème, demandez au professeur.

1.2 Premier programme

Cette partie utilise la carte principale du robot. C'est donc cette carte qu'il faudra programmer !

Pour les tests, on utilisera une carte principale conçue pour ce projet. Elle intègre un *dsPIC33EP512GM306*, programmable à l'aide du programmateur *ICD 3*. Cette carte vous permettra d'effectuer des tests sans avoir à réaliser de montage électronique, et au final de piloter votre robot quasiment sans câblage électronique.

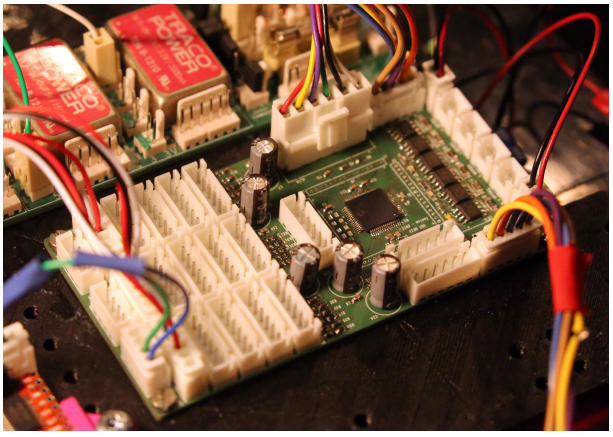


FIGURE 3 – Carte principale

Le microcontrôleur dsPIC33EP512GM306 utilisé est décrit en détail sur le site internet de chez Microchip :
Lien vers le dsPIC33EP512GM306.
Vous trouverez sur le site du constructeur toutes les informations nécessaires au fonctionnement de chacun des périphériques du microcontrôleur avec des exemples de code permettant de les utiliser.

1.2.1 Création du projet

⟹ Nous allons créer un premier projet. Pour cela, lancer MPLAB X, puis allez dans

$$File \rightarrow NewProject$$

Le projet à créer est de type :

$$MicrochipEmbedded \rightarrow StandaloneProject$$

L'assistant de création de projet vous demande de spécifier :
- La famille de composant : *16-bit DSCs (dsPIC33)*
- Le composant utilisé (device) : *dsPIC33EP512GM306*
- L'outil de programmation : *ICD 3* qui doit s'afficher avec deux points verts
- Le compilateur : *XC16* dans sa version la plus récente.
- Le nom et le chemin de votre projet : créez **sur le bureau** un répertoire *Projet_votre_nom**DATE_Robot_votre_nom*\, où DATE est la date du jour au format *année-mois-jour* (par exemple *20180101* pour le 1er Janvier *2018*). Utiliser ce répertoire comme chemin pour le projet, et nommez votre projet *carte_moteur_votre_nom*.

Attention :
Vous penserez archiverez du répertoire *Projet_votre_nomDATE_Carte_moteur_votre_nom*\ sur clé *USB* à la fin de chaque séance de projet. En début de séance suivante, vous copierez ce répertoire sur le bureau de l'ordinateur utilisé, et vous dupliquerez le répertoire *DATE_Robot_votre_nom*\ en ajustant la date à la date du jour, ce qui vous permettra de travailler sur une version courante, tout en gardant des archives de votre travail passé. Les ordinateurs étant nettoyés régulièrement, vous n'avez aucune garantie que vos fichiers ne seront pas supprimés d'une séance sur l'autre.**

Votre projet est à présent créé, avec 6 répertoires visibles dans le navigateur de projet, et en particulier les répertoires *Source Files* et *Header Files* dans lesquels vous allez travailler. Au cas où le navigateur (*Projects*) ne serait pas visible, il faut l'ouvrir en allant sur :

$$Window \rightarrow Projects$$

Attention :
Si d'autres projets sont ouverts, fermez les en cliquant-droit sur le projet puis *Close*. Ne travaillez jamais avec plusieurs projets ouverts en même temps !

1.2.2 Écriture du programme

Il n'y a pour l'instant pas de fichier de programme en C. Nous allons le créer et l'ajouter au projet.

$\Longrightarrow$ Pour cela, cliquer-droit sur le dossier *Source Files*, et sélectionner $New \rightarrow CMainFile...$. Appelez-le *main.c* et terminez la création.

$\Longrightarrow$ Nommez le fichier *main.c* et entrez le code suivant :

```
#include <stdio.h>
#include <stdlib.h>
#include <xc.h>
#include "ChipConfig.h"
#include "IO.h"

int main (void){
/**************************************************************************************************
//Initialisation de l'oscillateur
/**************************************************************************************************
InitOscillator();

/**************************************************************************************************
// Configuration des éentres sorties
/**************************************************************************************************
InitIO();

LED_BLANCHE = 1;
```

```
LED_BLEUE = 1;
LED_ORANGE = 1;

/**************************************************************************************************
// Boucle Principale
/**************************************************************************************************
while(1){
} // fin main
}
```

La coloration syntaxique automatique doit vous informer que plusieurs erreurs existent encore dans le code en raison de l'absence des fichiers *ChipConfig.h*, *ChipConfig.c*, *"IO.h"* et *"IO.h"*,ainsi que des fonctions *InitOscillator* et *InitIO* déclarées dans ces fichiers.

La fonction *InitOscillator* présente dans le fichier *ChipConfig.c* permet d'initialiser l'oscillateur de votre microcontrôleur de manière à le faire tourner à la vitesse de *40 MIPS* en utilisant l'oscillateur interne. Cette configuration avancée permet d'éviter l'utilisation d'un quartz externe, ce qui simplifie le montage des cartes. Cette configuration est trop difficile pour débuter, pour cette raison vous allez télécharger et intégrer les fichiers *ChipConfig.c* et *ChipConfig.h* depuis la page :

Lien vers les ressources nécessaires au projet robot

Cela revient à utiliser une librairie toute faite comme vous l'avez peut être déjà fait en *Arduino*, mais en ayant à disposition de code source de cette librairie pour le comprendre.

$\Longrightarrow$ Commençons par intégrer *"ChipConfig.h"* et *"ChipConfig.c"* au projet.

Pour ajouter le fichier *"ChipConfig.c"* au projet, cliquer-droit sur le dossier *Source Files* et sélectionner *New* $\rightarrow$ *C Source File...* si il est disponible dans le menu contextuel. Si il n'est pas disponible, sélectionner *New* $\rightarrow$ *other*, puis le répertoire *C*, puis *C source File*. Appeler le fichier *ChipConfig.c* et terminez la création.
Faire la même chose avec le fichier *ChipConfig.h*, mais depuis le dossier *Header Files* et en ajoutant un fichier de type *Header File*.

$\Longrightarrow$ Remplacer la totalité du code de chacun de ces deux fichiers *ChipConfig.c* et *ChipConfig.h* par les codes téléchargés depuis la page **Lien vers les ressources nécessaires au projet robot**.

$\Longrightarrow$ Faire de même avec les fichiers *"IO.c"* et *"IO.h"* . Commencer par les créer dans les répertoires *Source Files* pour *"IO.c"* et *Header Files* pour *"IO.h"* , puis remplacer leur contenu par le contenu téléchargé depuis la page **Lien vers les ressources nécessaires au projet robot**.

Voici à présent quelques explications sur le fonctionnement du code que vous avez importé dans votre projet. Le fichier *main.c* appelle le header *"IO.h"* qui contient les prototypes des fonctions (entre *""* car c'est une librairie créée par l'utilisateur et donc locale), et le header $< xc.h >$(entre <> car faisant partie des librairies Microchip), qui contient les définitions des pins et périphériques du modèle de *dsPIC* utilisé.

La fonction *InitIO()* du fichier *IO.c* contient le code pour initialiser les entrées et sorties du microcontrôleur, en l'occurrence pour l'instant uniquement les sorties correspondant aux 3 LED blanche, orange et bleue, après avoir désactivé les entrées analogiques sur toutes les pins du microcontrôleur.

La ligne *_TRISC10 = 0; // LED Orange* permet de configurer la pin C10 en sortie. Le *0* signifie *Output*, si on avait eu un *1*, cela aurait signifié *Input*. Il en va de même pour les autres sorties pilotant les LED.

$\Longrightarrow$ Vérifier que la configuration correspond bien au schéma électronique de connexion du *dsPIC* (fig. 4) pour chacune des LED.

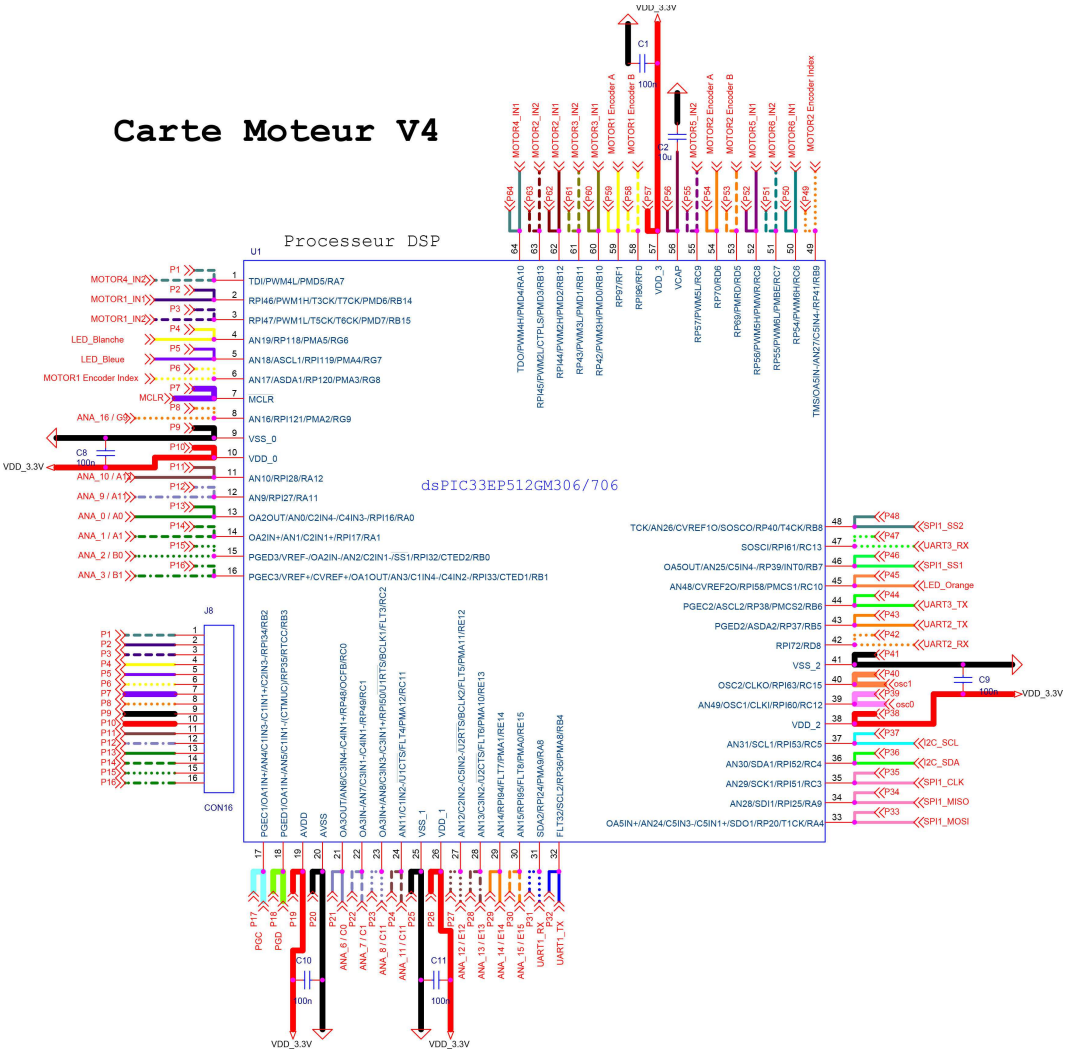


FIGURE 4 – Carte principale : Schéma électronique

Il est à présent temps detester le code que vous venez d'importer.

$\Longrightarrow$ Mettez la carte d'alimentation sous tension en l'alimentant en $12V$ à l'aide d'une alimentation stabilisée que vous aurez au préalable limitée à $0.5A$. Les LED rouge ($12V$), verte ($3.3V$) et orange ($5V$) de la carte d'alimentation doivent être allumées.

$\Longrightarrow$ Brancher le programmateur sur la carte de test. Le connecteur de test est indiqué PROG1 sur la sérigraphie (fig. 5).

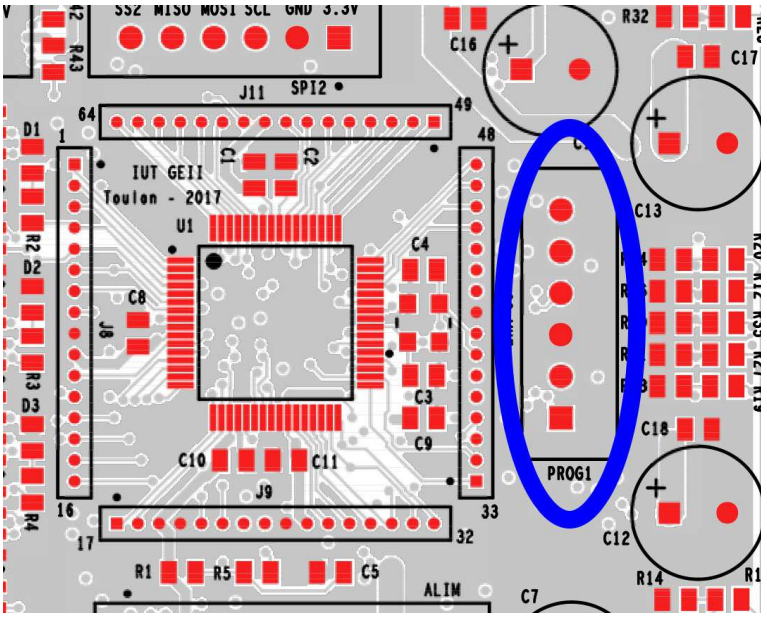


FIGURE 5 – Connecteur de programmation sur la carte principale

⟹ Compiler et lancer le projet en mode debug (icône avec une flèche verte orientée vers la droite). Normalement, le projet devrait compiler sans le moindre warning.

1.3 Debogueur

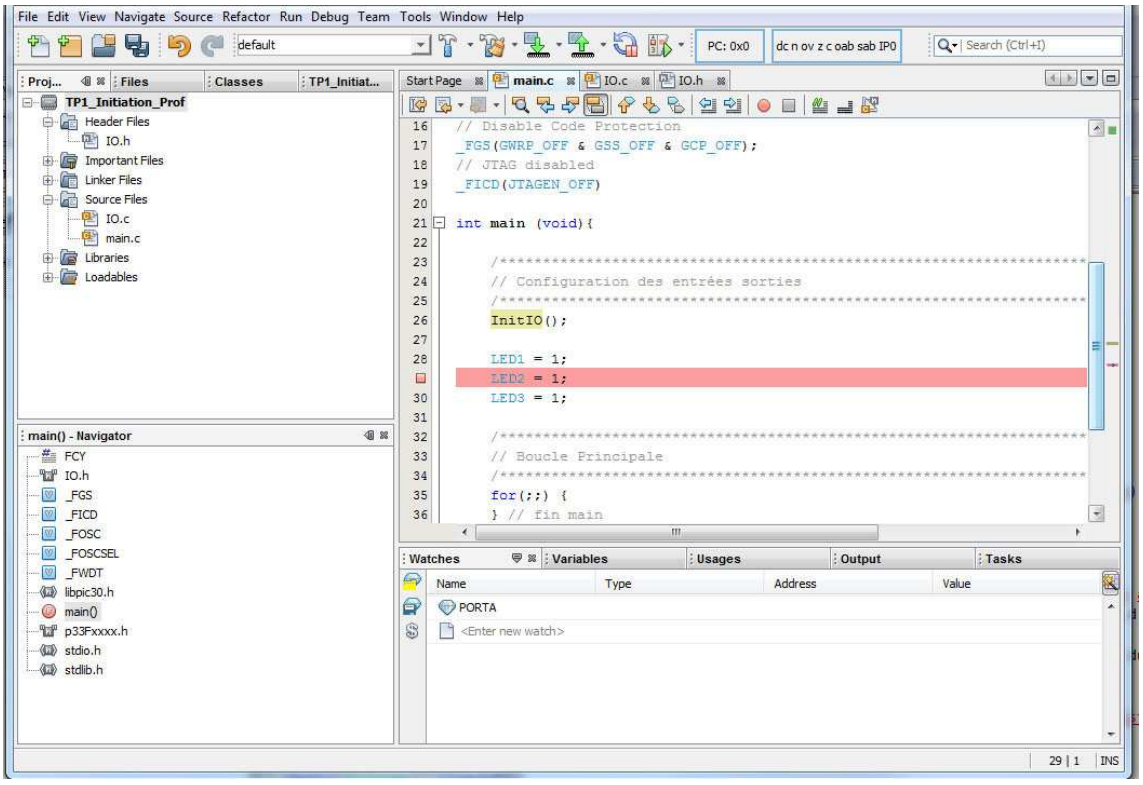
⟹ Il est possible d'insérer des points d'arrêt dans le programme. Leur nombre est limité à *6*.

Lorsque le programme les atteint, il se fige et attend qu'on lui donne l'ordre de continuer. Insérer des points d'arrêt dans le programme en double-cliquant sur le numéro de la ligne à laquelle vous voulez insérer le point d'arrêt.

⟹ Quand le programme est figé, il est alors possible de visualiser l'état de certaines variables du programme, ce qui est très utile pour voir si tout fonctionne comme prévu. Pour cela ouvrir la fenêtre *watch* :

$$Window \rightarrow Debugging \rightarrow Watches$$

Dans la fenêtre *watches*, double-clicker sur <*Enter new watch*> et ajouter à la liste des variables visualisée le port C (*PORTC*).



$\Longrightarrow$ Lancer le programme, il s'arrête sur le premier point d'arrêt. Regarder la valeur du *Port C*. Continuez l'exécution du programme.

$\Longrightarrow$ Rajouter une ligne dans la boucle infinie permettant d'inverser l'état de la *LED blanche* à chaque tour de boucle. Mettre un *breakpoint* sur cette ligne, exécutez le code et regarder ce qu'il se passe avec la *watch*

On utilisera dans la suite ce debugger chaque fois que l'on en aura besoin.

2 A la découverte du microcontrôleur : les périphériques classiques

Cette partie utilise la carte principale du robot. C'est donc cette carte qu'il faudra programmer !

2.1 Les timers

Qu'est-ce qu'un timer ? C'est compteur qui compte à une fréquence donnée. Le *dsPIC33EP512GM306* utilisé possède *9* timers *16 bits* pouvant former jusqu'à 4 timers *32 bits*
(voir la **datasheet du dsPIC33EP512GM306**) qui peuvent être ajustés de manière différente. L'intérêt d'un Timer est de permettre la génération d'événements périodiques tels que le clignotement d'une diode ou la génération d'un signal PWM.

Dans un premier temps, nous utiliserons le timer *2* couplé au timer *3* pour former un timer *32 bits* et le timer *1* en mode *16 bits*. Les initialisations se font par configuration directe des registres, ce qui offre le maximum de contrôle sur le composant. Le fonctionnement des timers est décrit en détail dans le *Reference Manual* du module *Timer*, celui-ci est disponible en téléchargement depuis la page depuis la page **Lien vers les ressources nécessaires au projet robot**.

⟹ Créer un fichier source *timer.c* contenant le code suivant :

```
#include <xc.h>
#include "timer.h"
#include "IO.h"

//Initialisation d'un timer 32 bits
void InitTimer23(void) {
T3CONbits.TON = 0; // Stop any 16-bit Timer3 operation
T2CONbits.TON = 0; // Stop any 16/32-bit Timer3 operation
T2CONbits.T32 = 1; // Enable 32-bit Timer mode
T2CONbits.TCS = 0; // Select internal instruction cycle clock
T2CONbits.TGATE = 0; // Disable Gated Timer mode
T2CONbits.TCKPS = 0b00; // Select 1:1 Prescaler
TMR3 = 0x00; // Clear 32-bit Timer (msw)
TMR2 = 0x00; // Clear 32-bit Timer (lsw)
PR3 = 0x0262; // Load 32-bit period value (msw)
PR2 = 0x5A00; // Load 32-bit period value (lsw)
IPC2bits.T3IP = 0x01; // Set Timer3 Interrupt Priority Level
IFS0bits.T3IF = 0; // Clear Timer3 Interrupt Flag
IEC0bits.T3IE = 1; // Enable Timer3 interrupt
T2CONbits.TON = 1; // Start 32-bit Timer
/* Example code for Timer3 ISR */
}

//Interruption du timer 32 bits sur 2-3
void __attribute__((interrupt, no_auto_psv)) _T3Interrupt(void) {
IFS0bits.T3IF = 0; // Clear Timer3 Interrupt Flag
LED_ORANGE = !LED_ORANGE;
}

//Initialisation d'un timer 16 bits
void InitTimer1(void)
{
//Timer1 pour horodater les mesures (1ms)
T1CONbits.TON = 0; // Disable Timer
T1CONbits.TSIDL = 0 ; //continue in idle mode
T1CONbits.TGATE = 0 ; //Accumulation disabled
T1CONbits.TCKPS = 0b01; //Prescaler
//11 = 1:256 prescale value
//10 = 1:64 prescale value
```

```
//01 = 1:8 prescale value
//00 = 1:1 prescale value
T1CONbits.TCS = 0; //clock source = internal clock
PR1 = 0x1388;

IFS0bits.T1IF = 0; // Clear Timer Interrupt Flag
IEC0bits.T1IE = 1; // Enable Timer interrupt
T1CONbits.TON = 1; // Enable Timer
}

//Interruption du timer 1
void __attribute__((interrupt, no_auto_psv)) _T1Interrupt(void)
{
IFS0bits.T1IF = 0;
LED_BLANCHE = !LED_BLANCHE;
}
```

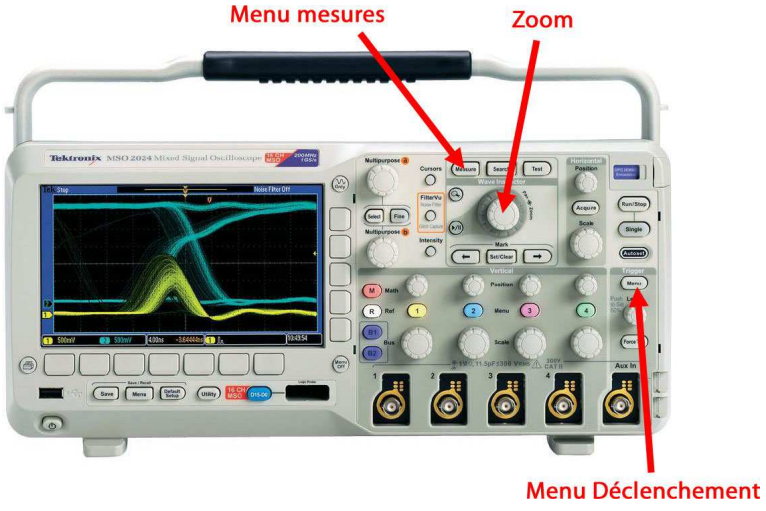
⟹ Créer un fichier *timer.h* contenant le code suivant :

```
#ifndef TIMER_H
#define TIMER_H

void InitTimer23(void);
void InitTimer1(void);

#endif  /* TIMER_H */
```

⟹ A partir de l'analyse des commentaires placés dans le code, que pensez-vous qu'il fasse ? Quelles sont en particulier les fréquences que vous devriez obtenir ?

⟹ Ajouter les appels aux fonctions d'initialisation dans le *main*, ainsi que l'include de *"timer.h"*.

⟹ Observer l'état des LED blanche et orange visuellement et à l'oscilloscope.



⟹ L'oscilloscope doit être synchronisé sur le signal le plus lent à l'aide du menu *trigger* ou *déclenchement*. Vous disposez d'un zoom sur l'oscilloscope vous permettant de regarder en détail une partie du signal. Vous pouvez également ajoutez des mesures (menu *mesure*) telles que la valeur de la fréquence d'un signal. Testez ces fonctionnalités, si vous n'y arrivez pas, appelez-le professeur.

$\Longrightarrow$ Les chronogrammes que vous observez sont-ils cohérents avec vos prévisions ?

$\Longrightarrow$ Changez à présent la valeur du prescaler et des périodes des timers, qu'observe-t-on ?

$\Longrightarrow$ Vous devez à présent régler la fréquence du timer le plus rapide à $6kHz$. Ajustez dans la fonction d'initialisation du *Timer 1* la valeur du registre $PR1$ pour obtenir cette fréquence en sachant que la fréquence du timer est égale à :

$$f = \frac{F_{CY}}{PS * PR1}$$

Dans cette formule, PS est la valeur du *prescaler* et F_{CY} la *fréquence d'horloge interne* égale à $40MHz$.

$\Longrightarrow$ Exprimez la valeur de $PR1$ choisie en decimal ($PR1 = valeur$), puis en binaire($PR1 = 0bvaleur\_binaire$), puis en hexadecimal ($PR1 = 0xvaleur\_hexadecimale$), et vérifier que dans chaque cas, la fréquence obtenue est bien $6kHz$ (soit un signal créneau à $3kHz$ puisque l'état de la LED doit être inversé deux fois pour avoir une période de signal créneau).

$\Longrightarrow$ Choisir à présent des valeurs de $PR1$ et du prescaler permettant d'obtenir une fréquence de sortie de $50Hz$ sur le $Timer1$.

$\Longrightarrow$ Réglez les valeurs de $PR2$ et $PR3$ en hexadecimal de manière à obtenir une fréquence de $0.5Hz$ en sortie du timer 32 bits. Pour le *Timer23*, la formule de la fréquence est la suivante :

$$f = \frac{F_{CY}}{PS * (PR3 * 2^{16} + PR2)}$$

$\Longrightarrow$ Validez les résultats obtenus avec le professeur avant de passer à la suite.

2.2 Vers une gestion orientée objet du robot en *C*

Dans la suite du projet, nous serons amenés à piloter et superviser le robot. Afin d'apporter de la rigueur à ce fonctionnement, il est souhaitable de disposer d'un descripteur de l'état courant de robot, auquel on pourra se référer à tout moment depuis l'importe lequel des fichiers du code embarqué. Un tel descripteur serait dans un langage de haut niveau un *objet*, et nous aurons l'occasion d'y revenir dans la partie commande-supervision en *C#*, mais le *C* ne dispose pas du concept d'objet. Nous aurons donc recours à une structure qui sera définie dans un fichier *Robot.h* et initialisée dans un fichier *Robot.c*. Les ocdes à implanter pour l'instant sont les suivants :

Dans *Robot.h*, placez le code suivant :

```
#ifndef ROBOT_H
#define ROBOT_H

typedef struct robotStateBITS {

union {
struct {
unsigned char taskEnCours;

float vitesseGaucheConsigne;
float vitesseGaucheCommandeCourante;
float vitesseDroiteConsigne;
float vitesseDroiteCommandeCourante;
};
};
} ROBOT_STATE_BITS;
extern volatile ROBOT_STATE_BITS robotState;
#endif  /* ROBOT_H */
```

Dans *Robot.c*, placez le code suivant :

```
#include "robot.h"
volatile ROBOT_STATE_BITS robotState;
```

Désormais, lorsque l'on aura besoin de décrire une variable caractéristique du robot, on l'ajoutera à la structure définie précédemment. On pourra également utiliser cette variable, par exemple *vitesseGaucheConsigne* en l'appelant sous la forme *robotState.vitesseGaucheConsigne* depuis n'importe quel *fichier.c*, dans lequel on aura rajouté *#include Robot.h*.

Vous aurez également besoin d'une boite à outil contenant des fonctions utiles, dont voici le code, à placer dans *ToolBox.c*. Pensez à ajouter les prototypes des fonctions dans le fichier *header* correspondant, ainsi que la définition de π sous la forme *#define PI 3.141592653589793*.

```
#include "Toolbox.h"
float Abs(float value)
{
if (value >= 0)
return value;
else return -value;
}

float Max(float value, float value2)
{
if (value > value2)
return value;
else
return value2;
}

float Min(float value, float value2)
{
if (value < value2)
```

```
return value;
else
return value2;
}

float LimitToInterval(float value, float lowLimit, float highLimit)
{
if (value > highLimit)
value = highLimit;
else if (value < lowLimit)
value = lowLimit;

return value;
}

float RadianToDegree(float value)
{
return value / PI * 180.0;
}

float DegreeToRadian(float value)
{
return value * PI / 180.0;
}
```

2.3 Le pilotage des moteurs

Le pilotage des moteurs est assuré par la carte principale conçue autour d'un *dsPIC33EP512GM306* à l'IUT de Toulon. Elle dispose de 12 sorties *PWM* permettant de piloter 6 moteurs à courant continu simultanément.

2.3.1 Le module PWM du microcontrôleur

⟹ Ajoutez au projet un fichier *"PWM.c"* dans lequel vous mettrez le code suivant :

```
#include <xc.h> // library xc.h inclut tous les uC
#include "IO.h"
#include "PWM.h"
#include "Robot.h"
#include "ToolBox.h"

#define PWMPER 40.0
unsigned char acceleration = 20;

void InitPWM(void)
{
PTCON2bits.PCLKDIV = 0b000;      //Divide by 1
PTPER = 100*PWMPER;              //éPriode en pourcentage

//éRglage PWM moteur 1 sur hacheur 1
IOCON1bits.POLH = 1;             //High = 1 and active on low =0
IOCON1bits.POLL = 1;             //High = 1
IOCON1bits.PMOD = 0b01;          //Set PWM Mode to Redundant
FCLCON1 = 0x0003;                //éDsactive la gestion des faults

//Reglage PWM moteur 2 sur hacheur 6
IOCON6bits.POLH = 1;             //High = 1
IOCON6bits.POLL = 1;             //High = 1
IOCON6bits.PMOD = 0b01;          //Set PWM Mode to Redundant
FCLCON6 = 0x0003;                //éDsactive la gestion des faults

/* Enable PWM Module */
PTCONbits.PTEN = 1;
}

void PWMSetSpeed(float vitesseEnPourcents)
{
robotState.vitesseDroiteCommandeCourante = vitesseEnPourcents;
MOTEUR_DROIT_ENL = 0;  //Pilotage de la pin en mode IO
MOTEUR_DROIT_INL = 1;  //Mise à 1 de la pin
MOTEUR_DROIT_ENH = 1;  //Pilotage de la pin en mode PWM
MOTEUR_DROIT_DUTY_CYCLE = Abs(robotState.vitesseDroiteCommandeCourante*PWMPER);
}
```

⟹ Ne pas compiler pour l'instant. L'analyse du contenu de ce fichier sera faite ultérieurement lorsqu'il sera nécessaire d'implanter la commande du moteur gauche.

⟹ Ajouter au projet le fichier *header* correspondant en veillant à ne pas oublier les prototypes des fonctions.

⟹ Définissez dans le fichier *IO.c* les pins *B14* et *B15* comme étant des sorties.

⟹ Rajoutez au fichier *IO.h* les *define* suivants :

```
//éDfinitions des pins pour les hacheurs moteurs
```

```
#define MOTEUR1_IN1 _LATB14
#define MOTEUR1_IN2 _LATB15

//Configuration éspcifique du moteur gauche
#define MOTEUR_GAUCHE_INH MOTEUR1_IN1
#define MOTEUR_GAUCHE_INL MOTEUR1_IN2
#define MOTEUR_GAUCHE_ENL IOCON1bits.PENL
#define MOTEUR_GAUCHE_ENH IOCON1bits.PENH
#define MOTEUR_GAUCHE_DUTY_CYCLE PDC1
```

$\Longrightarrow$ Ajoutez le *#include "PWM.h"* dans le *main*, et ajoutez dans le code l'appel des fonctions *InitPWM* et *PWMSetSpeed*. L'argument de la seconde étant 50.

$\Longrightarrow$ Exécutez à présent le code et observez sur la carte les signaux issus des pin $MOTOR1\_IN1$ (pin 2 sur la carte) et *MOTOR1_IN2*. Pour ce faire, vous pouvez utiliser du fil à wrapper pour relier la sonde de l'oscilloscope et le connecteur de test (percé de trous très fins).

$\Longrightarrow$ Effectuez différents tests en modifiant la valeur de l'argument de la fonction $PWMSetSpeed$ entre 0 et 100, et en regardant les allures des signaux *MOTEUR1_IN1* et *MOTEUR1_IN2*. Que constatez-vous ? Relier cela à votre cours d'électrotechnique, à quoi cela peut-il servir ?

2.3.2 Mise en oeuvre du hacheur

On s'intéresse à présent au hacheur de puissance. Il s'agit d'un hacheur pouvant fonctionner jusqu'à 40V et $\pm 3.5A$, ce qui peut paraître surprenant au vu de sa petite taille (boîtier *SOIC*). Fabriqué par *Allegro*, sa référence est *A*4950. Son diagramme fonctionnel est donné à la figure 6

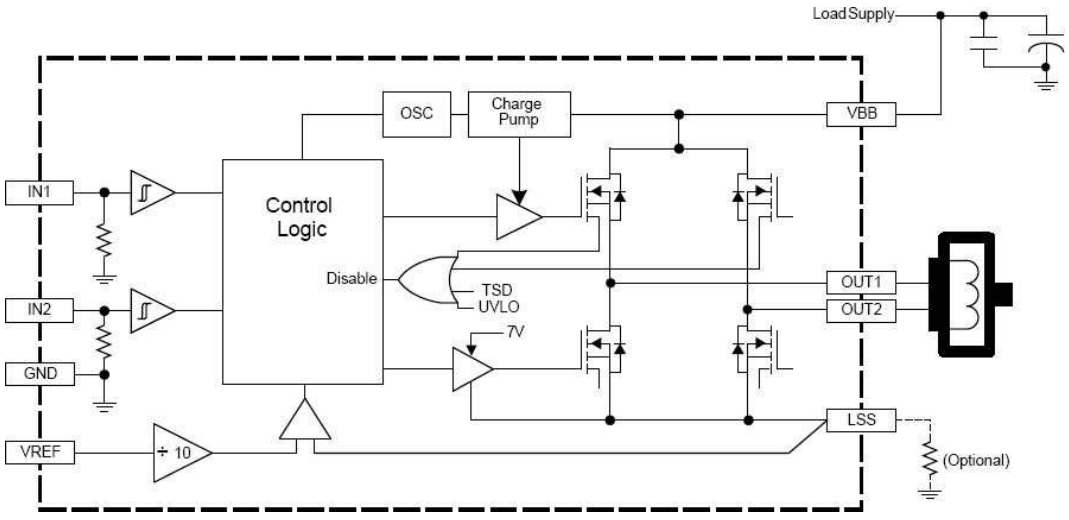


FIGURE 6 – Hacheur *A*4950 : diagramme fonctionnel

La partie puissance correspondant au hacheur 1 pilotant le moteur gauche est présentée à la figure 7.

La partie gauche du hacheur correspond aux signaux et alimentations de commande, la partie droite correspond à la partie puissance, V_{BB} étant la tension d'alimentation. Cette tension d'alimentation est fournie par la carte d'alimentation et n'est pas régulée.

FIGURE 7 – Partie puissance (hacheur du moteur 1) de la carte principale

Le moteur 1 se branche sur le connecteur $MOTOR1$.

Les chronogrammes de fonctionnement de ce hacheur en rotation directe et inverse sont les suivants :

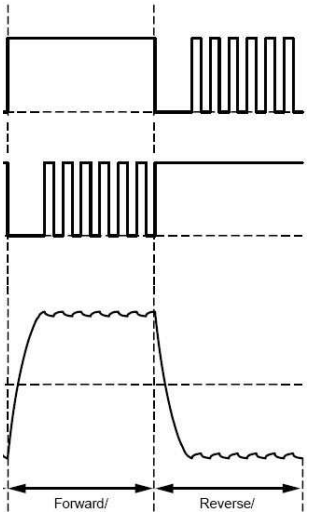


FIGURE 8 – Hacheur $A4950$: Chronogrammes de fonctionnement

$\Longrightarrow$ Les signaux *MOTEUR1_IN1* et *MOTEUR1_IN2* obtenus précédemment sont-ils conformes à l'utilisation qui doit en être faite par le hacheur ?

$\Longrightarrow$ Effectuez les branchements du moteur gauche du robot à la carte principale et tester le fonctionnement pour différentes valeurs de consigne moteur.

$\Longrightarrow$ Observez le courant absorbé par le moteur en lisant la valeur directement sur l'alimentation stabilisée. Comment pouvez-vous l'expliquer ?

$\Longrightarrow$ Observez le courant absorbé par le moteur lorsqu'il est freiné à la main. Relier cela aux équations de la MCC vues en cours.

$\Longrightarrow$ La résistance $R6$ de 0.1Ω (fig. 7) permet une visualisation instantanée du courant consommé par le moteur par la lecture de la tension aux bornes de la résistance (loi d'Ohm). En piquant une des bornes de cette résistance (si il n'y a pas de signal changez de borne !) à l'aide d'une sonde d'oscilloscope dont on aura enlevé le grip-fil (pensez à le remettre après !), observer l'allure de ce courant lorsque le moteur fonctionne et qu'il est freiné à la main par votre binôme. Relier cela aux équations de la MCC et du *hacheur* vues en cours.

A ce stade du projet, le moteur peut tourner à vitesse réglable dans un seul sens.

⟹ Implantez le code dans le fichier *PWM.c* permettant de le faire tourner dans le sens opposé si l'on passe en argument de la fonction *PWMSetSpeed* une consigne négative. Attention à bien avoir étudié le code fourni avant et en particulier l'usage des bits de choix du mode de pilotage des pins (*MOTEUR_GAUCHE_ENL* et *MOTEUR_GAUCHE_ENH*). Pour cela, référez-vous à la *datasheet* du DSPIC ou au *Reference Manual* traitant des *PWM*.

⟹ Validez le fonctionnement de votre hacheur en présence du professeur.

⟹ Implantez à présent une seconde commande moteur utilisant le hacheur n°6 pour piloter le moteur gauche. Pour cela, utiliser le schematic complet de la carte disponible en téléchargement ici : **Lien vers les ressources nécessaires au projet robot**

Il faut pour cela modifier la fonction *PWMSetSpeed* afin qu'elle prenne en arguments le pourcentage de vitesse maximale (comme c'est déjà le cas) mais également le numéro du moteur. Celui-ci sera défini sous la forme *MOTEUR_DROIT* ou *MOTEUR_GAUCHE*, définie dans *PWM.h* à l'aide de fonctions define du type :

```
#define MOTEUR_DROIT 0
```

⟹ Attention, pensez à dupliquer le code correspondant à chaque PWM dans la fonction d'initialisation des PWM.

⟹ Faites à présent changer à chaque seconde la vitesse d'un des moteurs en utilisant le Timer23 vu précédemment dans le projet. Les vitesse choisies correspondront à 0% et à 50% de la vitesse maximale. Pour cela, définissez une variable *consignePWM* de type *float* dans *timer.c*, et implanter le code ci-dessous permettant de changer la vitesse des deux moteurs dans l'interruption timer :

```
float consignePWM = 0;
//Interruption du timer 32 bits sur 2-3
void __attribute__((interrupt, no_auto_psv)) _T3Interrupt(void) {
IFS0bits.T3IF = 0; // Clear Timer3 Interrupt Flag

if(consignePWM == 0)
consignePWM = 50;
else
consignePWM = 0;

PWMSetSpeed(consignePWM, MOTEUR_DROIT);
PWMSetSpeed(consignePWM, MOTEUR_GAUCHE);
}
```

⟹ Validez le fonctionnement de votre code en présence du professeur. Que constatez-vous ?

2.3.3 Commande en rampes de vitesse

⟹ Le fonctionnement précédent est-il acceptable pour faire fonctionner un robot sans glissement au démarrage ?

Pour éviter le problème, vous allez implanter une rampe de vitesse. Le principe est le suivant : la PWM et donc la vitesse du moteur ne doivent plus changer brusquement, mais elles doivent changer en suivant une rampe. La pente de cette rampe (l'accélération) est définie par une variable *acceleration* de type *float* initialisée à 5.

Lors d'un changement de consigne de vitesse, seule la valeur de (*PWMSpeedConsigne*) change. La valeur de la

commande moteur de la PWM (*PWMSpeedCurrentCommand*), quant à elle, ne peut changer que par incréments successifs sur les interruptions du *Timer1* à la fréquence de *100 Hz*. La valeur des incréments successifs est *acceleration*.

Le code ci-dessous détaille la fonction *PWMUpdateSpeed* appelée par le *Timer1* et assurant la génération des rampes.

```
void PWMUpdateSpeed()
{
// Cette fonction est éappele sur timer et permet de suivre des rampes d'ééacclration
if (robotState.vitesseDroiteCommandeCourante < robotState.vitesseDroiteConsigne)
robotState.vitesseDroiteCommandeCourante = Min(
robotState.vitesseDroiteCommandeCourante + acceleration,
robotState.vitesseDroiteConsigne);
if (robotState.vitesseDroiteCommandeCourante > robotState.vitesseDroiteConsigne)
robotState.vitesseDroiteCommandeCourante = Max(
robotState.vitesseDroiteCommandeCourante - acceleration,
robotState.vitesseDroiteConsigne);

if (robotState.vitesseDroiteCommandeCourante > 0)
{
MOTEUR_DROIT_ENL = 0;  //pilotage de la pin en mode IO
MOTEUR_DROIT_INL = 1;  //Mise à 1 de la pin
MOTEUR_DROIT_ENH = 1;  //Pilotage de la pin en mode PWM
}
else
{
MOTEUR_DROIT_ENH = 0;  //pilotage de la pin en mode IO
MOTEUR_DROIT_INH = 1;  //Mise à 1 de la pin
MOTEUR_DROIT_ENL = 1;  //Pilotage de la pin en mode PWM
}
MOTEUR_DROIT_DUTY_CYCLE = Abs(robotState.vitesseDroiteCommandeCourante)*PWMPER;

if (robotState.vitesseGaucheCommandeCourante < robotState.vitesseGaucheConsigne)
robotState.vitesseGaucheCommandeCourante = Min(
robotState.vitesseGaucheCommandeCourante + acceleration,
robotState.vitesseGaucheConsigne);
if (robotState.vitesseGaucheCommandeCourante > robotState.vitesseGaucheConsigne)
robotState.vitesseGaucheCommandeCourante = Max(
robotState.vitesseGaucheCommandeCourante - acceleration,
robotState.vitesseGaucheConsigne);

if (robotState.vitesseGaucheCommandeCourante > 0)
{
MOTEUR_GAUCHE_ENL = 0; //pilotage de la pin en mode IO
MOTEUR_GAUCHE_INL = 1; //Mise à 1 de la pin
MOTEUR_GAUCHE_ENH = 1; //Pilotage de la pin en mode PWM
}
else
{
MOTEUR_GAUCHE_ENH = 0; //pilotage de la pin en mode IO
MOTEUR_GAUCHE_INH = 1; //Mise à 1 de la pin
MOTEUR_GAUCHE_ENL = 1; //Pilotage de la pin en mode PWM
}
MOTEUR_GAUCHE_DUTY_CYCLE = Abs(robotState.vitesseGaucheCommandeCourante) * PWMPER;
}
```

$\Longrightarrow$ Implantez cette fonction en pensant à rajouter les variables nécessaires dans *Robot.h*.

$\Longrightarrow$ Expliquez son fonctionnement de manière détaillée. En particulier, le rôle des *min* ou *max* sera expliqué dans le cas où par exemple on passerait d'une consigne de vitesse égale à 0 à une consigne de vitesse égale à 37

par incréments de 5.

$\Longrightarrow$ A présent, la fonction *PWMSetSpeed* ne doit plus être utilisée car elle change la valeur des consignes moteurs de manière directe et sans rampe. Supprimez-la ou commentez-la dans *PWM.c* et dans *PWM.h*.

$\Longrightarrow$ Créez une fonction PWMSetSpeedConsigne, prenant en arguments *(float vitesseEnPourcents, char moteur)*, permettant de définir les consignes de vitesse pour chacun des moteurs. Le générateur de rampe, appelé sur les interruptions du *Timer1* est chargé d'atteindre les consignes automatiquement.

$\Longrightarrow$ Changez la valeur de l'accélération et testez à nouveau. Que constatez-vous ?

$\Longrightarrow$ Validez le fonctionnement à l'oscilloscope et avec le professeur.

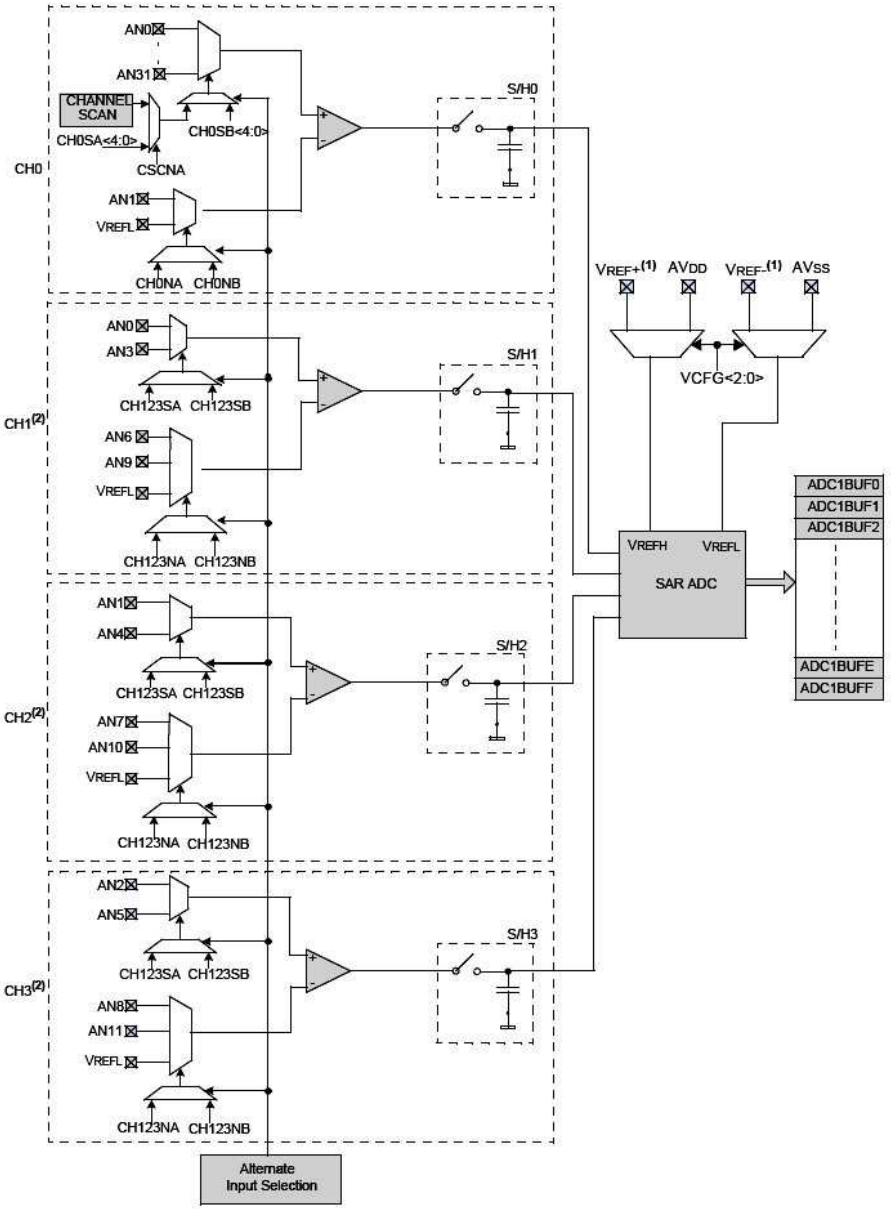
Vous disposez à présent d'une commande de moteur propre vous permettant de faire se déplacer un robot ayant deux roues motorisées séparément.

2.4 Les conversions analogique-numérique

Cette partie utilise la carte principale du robot.

Dans, cette partie, nous allons utiliser le convertisseur analogique numérique du *dsPIC*. Il sera utilisé pour récupérer les distances vues par des télémètres infrarouges situés sur le robot.

Le convertisseur analogique-numérique du *dsPIC* est un convertisseur à approximation successive (*SAR ADC*). Les entrées sorties utilisables ont un certain nombre de restrictions, indiquées sur le schéma suivant :



2.5 Mise en oeuvre du convertisseur analogique numérique

Dans ce projet, nous utiliserons le convertisseur analogique numérique dans une version simplifiée : les conversion se feront uniquement sur le *channel 0* et séquentiellement. Le mode *DMA (Direct Memory Access* permettant de faire des acquisitions simultanées sur plusieurs canaux ne sera pas utilisé car trop complexe.

⟹ Créer un fichier *"ADC.c"* contenant le code suivant.

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

```
#include "adc.h"
#include "main.h"

unsigned char ADCResultIndex = 0;
static unsigned int ADCResult[4];
unsigned char ADCConversionFinishedFlag;

/*****************************************************************************************************
// Configuration ADC
/*****************************************************************************************************
void InitADC1(void)
{
//cf. ADC Reference Manual page 47

//Configuration en mode 12 bits mono canal ADC avec conversions successives sur 4 éentres
/*********************************************************/
//AD1CON1
/*********************************************************/
AD1CON1bits.ADON = 0;        // ADC module OFF - pendant la config
AD1CON1bits.AD12B = 1;       // 0 : 10bits - 1 : 12bits
AD1CON1bits.FORM = 0b00;     // 00 = Integer (DOUT = 0000 dddd dddd dddd)
AD1CON1bits.ASAM = 0;        // 0 = Sampling begins when SAMP bit is set
AD1CON1bits.SSRC = 0b111;    // 111 = Internal counter ends sampling and starts conversion (auto-co

/*********************************************************/
//AD1CON2
/*********************************************************/
AD1CON2bits.VCFG = 0b000;    // 000 : Voltage Reference = AVDD AVss
AD1CON2bits.CSCNA = 1;       // 1 : Enable Channel Scanning
AD1CON2bits.CHPS = 0b00;     // Converts CH0 only
AD1CON2bits.SMPI = 2;        // 2+1 conversions successives avant interrupt
AD1CON2bits.ALTS = 0;
AD1CON2bits.BUFM = 0;


/*********************************************************/
//AD1CON3
/*********************************************************/
AD1CON3bits.ADRC = 0;        // ADC Clock is derived from Systems Clock
AD1CON3bits.ADCS = 15;       // ADC Conversion Clock TAD = TCY * (ADCS + 1)
AD1CON3bits.SAMC = 15;       // Auto Sample Time

/*********************************************************/
//AD1CON4
/*********************************************************/
AD1CON4bits.ADDMAEN = 0;         // DMA is not used

/*********************************************************/
//Configuration des ports
/*********************************************************/
//ADC éutiliss : 16(G9)-11(C11)-6(C0)
ANSELCbits.ANSC0 = 1;
ANSELCbits.ANSC11 = 1;
ANSELGbits.ANSG9 = 1;

AD1CSSLbits.CSS6=1; // Enable AN6 for scan
AD1CSSLbits.CSS11=1; // Enable AN11 for scan
AD1CSSHbits.CSS16=1; // Enable AN16 for scan

/* Assign MUXA inputs */
AD1CHS0bits.CH0SA = 0;// CH0SA bits ignored for CH0 +ve input selection
AD1CHS0bits.CH0NA = 0;// Select VREF- for CH0 -ve inpu
```

```
IFS0bits.AD1IF = 0; // Clear the A/D interrupt flag bit
IEC0bits.AD1IE = 1; // Enable A/D interrupt
AD1CON1bits.ADON = 1; // Turn on the A/D converter
}

/* This is ADC interrupt routine */
void __attribute__((interrupt, no_auto_psv)) _AD1Interrupt(void)
{
IFS0bits.AD1IF = 0;
ADCResult[0] = ADC1BUF0;// Read the AN-scan input 1 conversion result
ADCResult[1] = ADC1BUF1;// Read the AN3 conversion result
ADCResult[2] = ADC1BUF2;// Read the AN5 conversion result
ADCConversionFinishedFlag = 1;
}

void ADC1StartConversionSequence()
{
AD1CON1bits.SAMP = 1 ; //Lance une acquisition ADC
}

unsigned int * ADCGetResult(void)
{
return ADCResult;
}

unsigned char ADCIsConversionFinished(void)
{
return ADCConversionFinishedFlag;
}

void ADCClearConversionFinishedFlag(void)
{
ADCConversionFinishedFlag = 0;
}
```

$\Longrightarrow$ Ajoutez au projet le fichier *ADC.h* correspondant.

$\Longrightarrow$ Examinez la partie configuration de l'ADC dans le code et commentez le dans votre rapport. En particulier déterminez quelles sont les voies utilisées comme entrées analogiques. Regardez sur le schéma de la carte principale fourni en pièce jointe sur quelles pins vous devrez câbler les entrées analogiques.

$\Longrightarrow$ Dans le code de l'interruption du Timer 1, appelez la fonction permettant de lancer la conversion analogique numérique sur les 4 voies séquentiellement. Il vous faut pour cela inclure *"ADC.h"* dans *"timer.c"*.

$\Longrightarrow$ En insérant un point d'arrêt dans l'interruption au niveau de la ligne $ADCConversionFinishedFlag = 1;$, vérifiez que votre ADC fonctionne bien. Pour cela, vous pouvez raccordez deux des trois entrées au potentiel $+3.3V$ et une autre à $0V$ en utilisant des fils souple placés sur les connecteurs des entrées analogiques. Pensez à respecter le code de couleur : rouge pour le $3.3V$ et noir pour le $0V$.

$\Longrightarrow$ D'apprès la documentation technique et l'usage que vous faites de l'ADC, quelles valeurs numériques devraient correspondre aux tensions $3.3V$ et $0V$ en entrée de l'ADC ? Justifier en regardant en particulier le rôle des champs de bit *AD12B* et *FORM* du registre de configuration *AD1CON1* de l'ADC. Pour cela vous pouvez consulter le *Reference Manual* de l'ADC :
Lien vers les ressources nécessaires au projet robot.

Vous voulez à présent utiliser le résultat des conversions lancées sur les interruptions du *Timer1* dans le main,

afin de pouvoir ultérieurement piloter votre robot.

$\Longrightarrow$ Insérez dans la boucle infinie du main un code permettant de tester si la conversion ADC est terminée (voir le fichier *"ADC.c"*), et si c'est le cas, nettoyez le flag de fin de conversion et récupérez le résultat de la conversion dans des variables $ADCValue0$, $ADCValue1$, $ADCValue2$. Pour cela vous devrez au préalable récupérez le contenu du tableau des résultat de la conversion à l'aide de la ligne de code suivante :

```
unsigned int * result = ADCGetResult();
```

$\Longrightarrow$ Montrez au professeur le bon fonctionnement de votre programme, deux entrées étant toujours branchées sur $V_{dd} = 3.3V$ et l'autre sur $V_{SS} = 0V$.

2.6 Télémètres infrarouge branchés sur l'ADC

Le convertisseur analogique numérique est à présent prêt à être utilisé pour la lecture de télémètres infrarouges de type $GP2D12$ ou $GP2Y0A21YK0F$. Nous utiliserons 3 télémètres branchés sur les entrées préalablement utilisées du microcontrôleur.

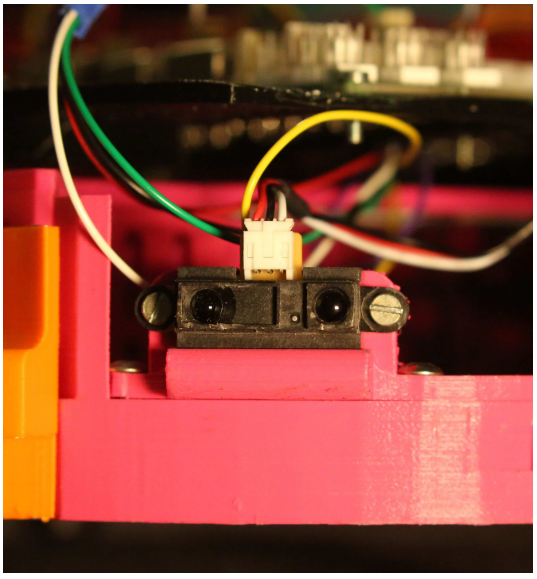


FIGURE 9 – Télémètre infrarouge installés sur le robot

Ces télémètres, placés à l'avant du robot (fig. 9), sont prévus pour fonctionner dans un intervalle de distances compris entre $10cm$ et $80cm$ (fig. 10), ce qui est adapté pour un robot mobile évoluant à vitesse modérée.

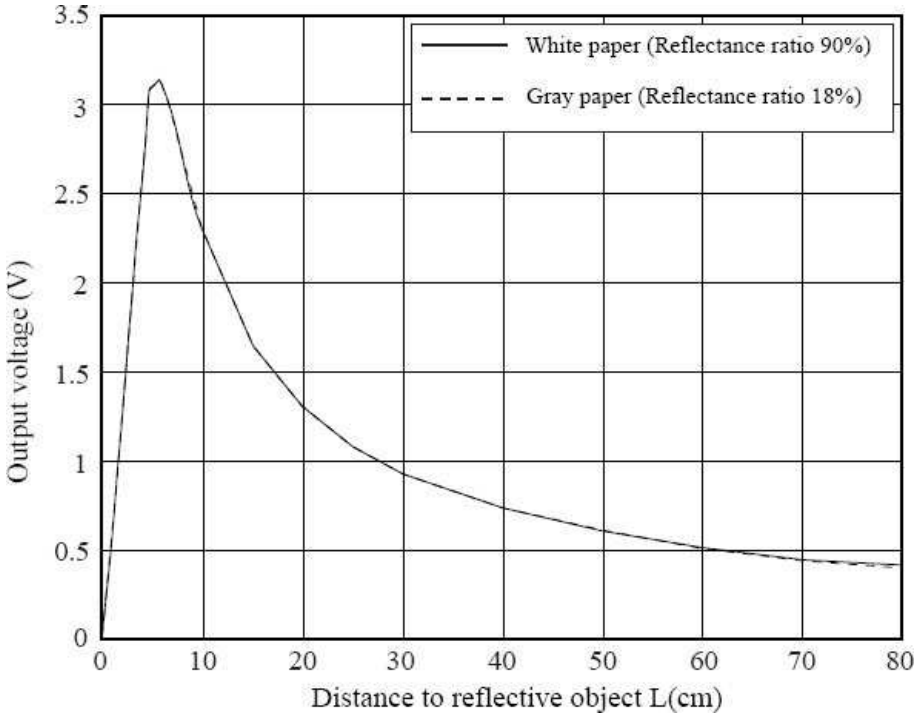


FIGURE 10 – Relation tension-distance des télémètres GP2Y0A21YK0F

Attention : ces télémètres doivent être alimentés en $5V$, la tension caractéristique de la distance ayant des valeurs possibles entre $0V$ et $3.1V$. Il serait donc possible de brancher directement ces télémètres en entrée du microcontrôleur, mais par sécurité au cas ou des capteurs sortant des tensions de 10V soient utilisés (capteur industriels), les entrées analogiques sont raccordées à un pont diviseur par *3.2*.

Chacun des connecteur d'entrée analogique dispose d'une tension d'alimentation en $5V$. Attention toutefois à bien fabriquer votre connecteur si il n'est pas déjà fait. Pour information, la consommation des télémètres infrarouge est relativement importante en pointe (ils absorbent un courant variant périodiquement à environ 1kHz, avec des pics à plus de 100mA).

$\Longrightarrow$ Brancher les connecteurs de chacun des 3 télémètres infrarouge sur les connecteurs correspondants de la carte d'alimentation.

$\Longrightarrow$ Placez des obstacles à une distance fixe devant les capteurs et vérifiez en insérant un point d'arrêt dans la fonction de récupération des données *ADC* dans le *main* que les valeurs analogiques récupérées correspondent bien aux distances mesurées. La conversion des valeurs lues en distance mesurée doit se faire d'après la documentation technique des capteurs (fig.10) en prenant en compte la présence d'un pont diviseur par *3.2* !.

$\Longrightarrow$ Relevez les valeurs correspondant à une distance de détection de $20cm$ et à une distance de détection de $40cm$.

$\Longrightarrow$ Dans la boucle infinie du main, rajouter du code permettant de comparer les valeurs lues par les télémètres à la valeur correspondant à une distance de $30cm$. Pour chacun des 3 télémètres du robot, si cette valeur est supérieure, allumez une LED, sinon éteignez là. Couplez de préférence le télémètre droit avec la LED orange, le télémètre central avec la LED bleue et le télémètre de gauche avec la LED blanche.

$\Longrightarrow$ Afin de rendre efficace l'usage des télémètres, il est souhaitable de convertir les valeurs lues par le convertisseur *ADC* en distance. Pour cela déclarez en *float* les variables *distanceTelemetreDroit*, *distanceTelemetreCentre* et *distanceTelemetreGauche* dans *Robot.h*, et remplacez le code de récupération des valeurs analogiques situé

dans le main par le code suivant :

```
if (ADCIsConversionFinished() == 1)
{
ADCClearConversionFinishedFlag();
unsigned int * result = ADCGetResult();
float volts = ((float) result [2])* 3.3 / 4096 * 3.2;
robotState.distanceTelemetreDroit = 30 / volts;
volts = ((float) result [1])* 3.3 / 4096 * 3.2;
robotState.distanceTelemetreCentre = 30 / volts;
volts = ((float) result [0])* 3.3 / 4096 * 3.2;
robotState.distanceTelemetreGauche = 30 / volts;
}
```

$\Longrightarrow$ Expliquez le fonctionnement du code précédent. Expliquez en quoi il est limité pour les très faibles distances de détection.

$\Longrightarrow$ Ajustez dans le code précédent l'ordre des télémètres en plaçant un point d'arrêt à l'avant dernière ligne du code précédent et en plaçant des obstacles devant les télémètres IR tour à tour.

$\Longrightarrow$ Valider le résultat avec le professeur. Si tout est correct, vous disposez d'un système efficace de détection d'obstacles qui vous permettra d'éviter les collisions lors de la phase de pilotage de votre système.

3 Un premier robot mobile autonome

Dans cette partie, nous allons mettre en oeuvre simultanément les fonctions vues dans les partie précédentes afin de réaliser un robot capable de se déplacer de manière autonome en évitant des obstacles et les autres robots.

Un robot mobile est un système pouvant être décrit par une machine à états. Le robot pourra ainsi passer d'un état à un autre sous certaines conditions. Par exemple, supposons que le robot roule en marche avant, et qu'un obstacle passe devant lui ou qu'il arrive à proximité d'un obstacle. Un ou plusieurs télémètres indiqueront alors une distance réduite entre le robot et l'objet. Si cette distance est inférieure à un certain seuil (*condition de changement d'état*), le robot pourra changer d'état et se mettre à tourner afin d'éviter l'obstacle.

Une machine à état est implantée en *C* par la structure *switch-case*. Cette machine à état peut être appelée dans la boucle infinie du programme principal et donc à une fréquence maximale, mais il est préférable de l'appeler à une fréquence périodique, depuis un timer. Ce fonctionnement, à la base des *OS* (*Operating System*) temps réel, permet d'attribuer une priorité à l'interruption du timer appelant la machine à état, et donc de hiérarchiser les processus fonctionnant en parallèle dans le système.

3.1 Réglage automatique des timers

Pour régler plus facilement la fréquence de fonctionnement des timers, vous allez modifier légèrement la structure du code déjà réalisé.

$\Longrightarrow$ Créer un fichier *main.h* qui contiendra les paramètres essentiels de la carte tels que la fréquence de fonctionnement ou les *#define* des évènements système (voir après). Ce fichier sera appelé par les autres fichiers *.c* qui auront ainsi accès facilement aux paramètres du système. Déplacer dans ce fichier les lignes de codes suivantes depuis le *main.c* :

```
// Configuration des èparamtres du chip
#define FCY 40000000
```

$\Longrightarrow$ Dans le fichier *timer.c*, ajoutez l'include de *main.h*, puis ajoutez une fonction *SetFreqTimer1* permettant de régler automatiquement PR1 et le prescaler *TCKPS* du *Timer 1* en fonction de la fréquence demandée. Le code de cette fonction est proposé ci-dessous, expliquez en quoi le réglage de *TCKPS* et de *PR1* est optimal.

```
void SetFreqTimer1(float freq)
{
T1CONbits.TCKPS = 0b00;                 //00 = 1:1 prescaler value
if(FCY /freq > 65535)
{
T1CONbits.TCKPS = 0b01;              //01 = 1:8 prescaler value
if(FCY /freq / 8 > 65535)
{
T1CONbits.TCKPS = 0b10;          //10 = 1:64 prescaler value
if(FCY /freq / 64 > 65535)
{
T1CONbits.TCKPS = 0b11;    //11 = 1:256 prescaler value
PR1 = (int)(FCY / freq / 256);
}
else
PR1 = (int)(FCY / freq / 64);
}
else
PR1 = (int)(FCY / freq / 8);
}
else
PR1 = (int)(FCY / freq);
}
```

$\Longrightarrow$ Appelez cette fonction depuis la fonction *InitTimer1()*, en veillant à supprimer au préalable la configuration manuelle de *PR1* et *TCKPS*.

$\Longrightarrow$ Testez cette fonction en faisant clignoter la *LED bleue* dans l'interruption timer, pour différentes fréquences, et en particulier pour des fréquences non entières (par exemple *2.5 Hz*).

$\Longrightarrow$ Ajoutez au fichier *timer.c* des fonctions permettant de configurer le *Timer 4* et de régler sa fréquence automatiquement (comme pour le *Timer 1*). Appelez la fonction d'initialisation de ce timer depuis le *main*, la fréquence du timer sera réglée à 1kHz. Il est à noter que le flag *T4IF* se trouve dans le registre IFS1bits et que l'enable des interruptions timer 4 (*T4IE*) se trouve dans le registre *IEC1bits*.

3.2 Horodatage : génération du temps courant

Le *Timer 4* que vous venez de configurer va servir à générer un horodatage à la milliseconde qui pourra être utilisé dans tout le reste du code.

$\Longrightarrow$ Déclarez un *unsigned long* appelé *timestamp* dans le fichier *timer.c*. Cette variable contiendra le temps courant en millisecondes, elle doit être accessible de n'importe où dans le code. Pour la rendre appelable de n'importe quel fichier incluant *timer.h*, déclarez également la variable *timestamp* dans *timer.h* sous la forme suivante :

```
extern unsigned long timestamp;
```

$\Longrightarrow$ Ajoutez du code permettant d'incrémenter de 1 la variable *timestamp* à chaque milliseconde (dans l'interruption du *timer 4*). Vous avez à présent la possibilité de connaître le temps courant depuis le lancement du programme (ou un reset de la variable *timestamp*, si vous ajoutez une fonction permettant de le faire).

3.3 Machine à état de gestion du robot

Le fonctionnement d'un système peut être décrit par un *GRAFCET* implantable sous forme de machine à états (*state machine*). Une machine à états s'implante en *C* sous forme d'une structure *switch case*, l'argument du switch étant l'état courant du système.

$\Longrightarrow$ Afin de rendre le code lisible, définissez quelques états de la *state machine* dans *main.h* :

```
#define STATE_ATTENTE 0
#define STATE_ATTENTE_EN_COURS 1
#define STATE_AVANCE 2
#define STATE_AVANCE_EN_COURS 3
#define STATE_TOURNE_GAUCHE 4
#define STATE_TOURNE_GAUCHE_EN_COURS 5
#define STATE_TOURNE_DROITE 6
#define STATE_TOURNE_DROITE_EN_COURS 7
#define STATE_TOURNE_SUR_PLACE_GAUCHE 8
#define STATE_TOURNE_SUR_PLACE_GAUCHE_EN_COURS 9
#define STATE_TOURNE_SUR_PLACE_DROITE 10
#define STATE_TOURNE_SUR_PLACE_DROITE_EN_COURS 11
#define STATE_ARRET 12
#define STATE_ARRET_EN_COURS 13
#define STATE_RECULE 14
#define STATE_RECULE_EN_COURS 15

#define PAS_D_OBSTACLE 0
#define OBSTACLE_A_GAUCHE 1
#define OBSTACLE_A_DROITE 2
#define OBSTACLE_EN_FACE 3
```

$\Longrightarrow$ Ajouter dans le main (en prenant soin de ne pas oublier de rajouter le prototype dans le fichier *main.h*) une fonction *OperatingSystemLoop* implantée comme suit :

```
unsigned char stateRobot;

void OperatingSystemLoop(void)
{
switch (stateRobot)
{
case STATE_ATTENTE:
timestamp = 0;
PWMSetSpeedConsigne(0, MOTEUR_DROIT);
PWMSetSpeedConsigne(0, MOTEUR_GAUCHE);
stateRobot = STATE_ATTENTE_EN_COURS;

case STATE_ATTENTE_EN_COURS:
if (timestamp > 1000)
stateRobot = STATE_AVANCE;
break;

case STATE_AVANCE:
PWMSetSpeedConsigne(30, MOTEUR_DROIT);
PWMSetSpeedConsigne(30, MOTEUR_GAUCHE);
stateRobot = STATE_AVANCE_EN_COURS;
break;
case STATE_AVANCE_EN_COURS:
SetNextRobotStateInAutomaticMode();
break;

case STATE_TOURNE_GAUCHE:
PWMSetSpeedConsigne(30, MOTEUR_DROIT);
PWMSetSpeedConsigne(0, MOTEUR_GAUCHE);
stateRobot = STATE_TOURNE_GAUCHE_EN_COURS;
break;
case STATE_TOURNE_GAUCHE_EN_COURS:
SetNextRobotStateInAutomaticMode();
break;

case STATE_TOURNE_DROITE:
PWMSetSpeedConsigne(0, MOTEUR_DROIT);
PWMSetSpeedConsigne(30, MOTEUR_GAUCHE);
stateRobot = STATE_TOURNE_DROITE_EN_COURS;
break;
case STATE_TOURNE_DROITE_EN_COURS:
SetNextRobotStateInAutomaticMode();
break;

case STATE_TOURNE_SUR_PLACE_GAUCHE:
PWMSetSpeedConsigne(15, MOTEUR_DROIT);
PWMSetSpeedConsigne(-15, MOTEUR_GAUCHE);
stateRobot = STATE_TOURNE_SUR_PLACE_GAUCHE_EN_COURS;
break;
case STATE_TOURNE_SUR_PLACE_GAUCHE_EN_COURS:
SetNextRobotStateInAutomaticMode();
break;

case STATE_TOURNE_SUR_PLACE_DROITE:
PWMSetSpeedConsigne(-15, MOTEUR_DROIT);
PWMSetSpeedConsigne(15, MOTEUR_GAUCHE);
stateRobot = STATE_TOURNE_SUR_PLACE_DROITE_EN_COURS;
```

```
break;
case STATE_TOURNE_SUR_PLACE_DROITE_EN_COURS:
SetNextRobotStateInAutomaticMode();
break;

default :
stateRobot = STATE_ATTENTE;
break;
}
}

unsigned char nextStateRobot=0;

void SetNextRobotStateInAutomaticMode()
{
unsigned char positionObstacle = PAS_D_OBSTACLE;

//éDtermination de la position des obstacles en fonction des éèètlmtres
if (    robotState.distanceTelemetreDroit < 30 &&
robotState.distanceTelemetreCentre > 20 &&
robotState.distanceTelemetreGauche > 30) //Obstacle à droite
positionObstacle = OBSTACLE_A_DROITE;
else if(robotState.distanceTelemetreDroit > 30 &&
robotState.distanceTelemetreCentre > 20 &&
robotState.distanceTelemetreGauche < 30) //Obstacle à gauche
positionObstacle = OBSTACLE_A_GAUCHE;
else if(robotState.distanceTelemetreCentre < 20) //Obstacle en face
positionObstacle = OBSTACLE_EN_FACE;
else if(robotState.distanceTelemetreDroit > 30 &&
robotState.distanceTelemetreCentre > 20 &&
robotState.distanceTelemetreGauche > 30) //pas d'obstacle
positionObstacle = PAS_D_OBSTACLE;

//éDtermination de lé'tat à venir du robot
if (positionObstacle == PAS_D_OBSTACLE)
nextStateRobot = STATE_AVANCE;
else if (positionObstacle == OBSTACLE_A_DROITE)
nextStateRobot = STATE_TOURNE_GAUCHE;
else if (positionObstacle == OBSTACLE_A_GAUCHE)
nextStateRobot = STATE_TOURNE_DROITE;
else if (positionObstacle == OBSTACLE_EN_FACE)
nextStateRobot = STATE_TOURNE_SUR_PLACE_GAUCHE;

//Si l'on n'est pas dans la transition de lé'tape en cours
if (nextStateRobot != stateRobot - 1)
stateRobot = nextStateRobot;
}
```

Le code proposé ci dessus permet de décrire un *GRAFCET*. En effet, les étapes (par exemple *STATE_AVANCE*) sont suivies d'une attente de transition (par exemple *STATE_AVANCE_EN_COURS*) comme dans un *GRAFCET*.
Ce fonctionnement permet de ne pas remettre constamment à jour les consignes moteurs si il n'y a pas lieu de les changer. Il permettra également de renvoyer ultérieurement un message à chaque passage dans une étape.

$\Longrightarrow$ Expliquer le fonctionnement des deux dernières lignes du code en détail, et appelez le professeur si vous n'avez pas compris.

$\Longrightarrow$ Appelez la fonction *OperatingSystemLoop* précédente depuis l'interruption du *timer 4*. Valider que le robot change de comportement quand on passe la main devant les différents télémètres infrarouge.

$\Longrightarrow$ Si ça n'est pas déjà le cas, passez la vitesse du *timer 1* à 50 Hz de manière à avoir un rafraîchissement des télémètres IR à fréquence élevée, et des rampes de vitesse plus rapides (on conservera une accélération à la valeur de 5).

$\Longrightarrow$ Tester le code en branche le robot sur batteries et en le laissant évoluer seul. Lorsque vous avez terminé la mise au point et que tout se passe bien, valider avec le professeur.

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

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

4.1 Simulateur de messagerie instantanée

$\Longrightarrow$ Créez un projet C# dans Visual Studio. Pour cela lancer Visual Studio puis *Fichier* $\rightarrow$ *Nouveau* $\rightarrow$ *Projet*. Dans l'onglet *Modèles* $\rightarrow$ *Visual C#*, choisir *Application Windows Form*.
Avant de créer le projet, donnez lui un nom explicite tel que *Robot Interface* et spécifiez un chemin d'accès sur le disque dur tel que *C :/Visual Studio Projects/*. Assurez vous que la case, *Créer un répertoire pour la solution* est cochée. Créez le projet, vous devriez avoir un écran similaire à celui de la figure 11.

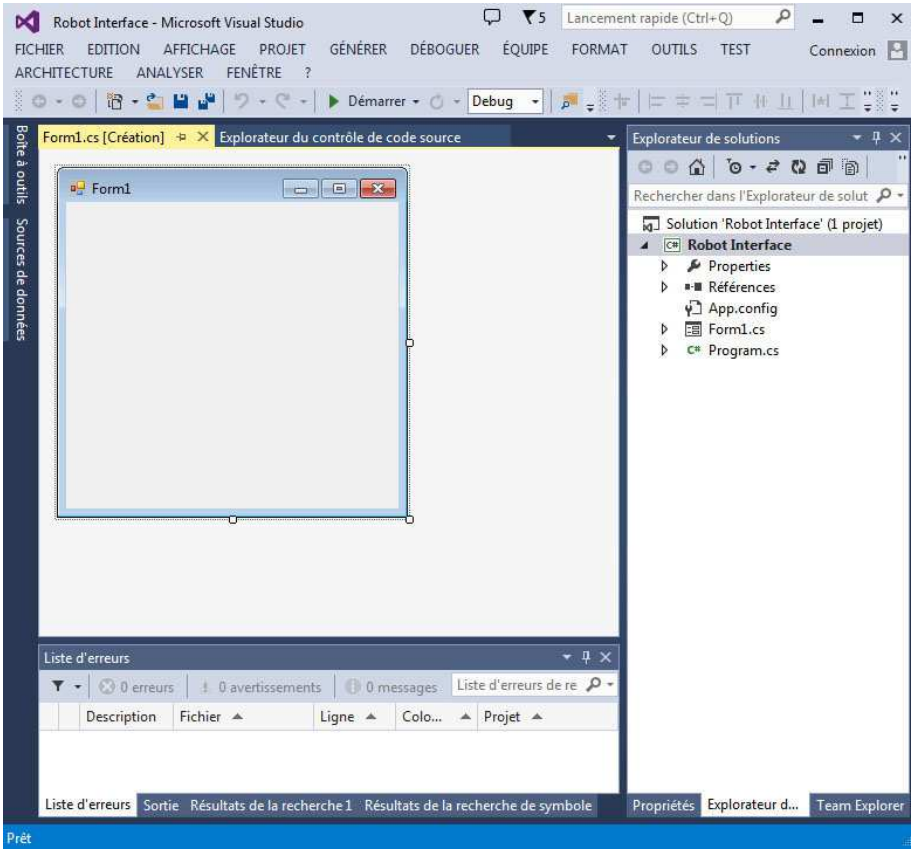


FIGURE 11 – Projet Visual Studio nouvellement créé

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

$\Longrightarrow$ Ajoutez à présent à *Form1* deux objets de type *GroupBox* en les faisant glisser depuis la *Boite à outils* $\rightarrow$ *Conteneurs*. Positionnez les comme indiqué à la figure 12. Sélectionnez une de ces *GroupBox* et cliquez ensuite sur l'onglets *Propriétés* dans la partie droite de l'écran. Vous avez accès à une liste importante d'attributs de

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

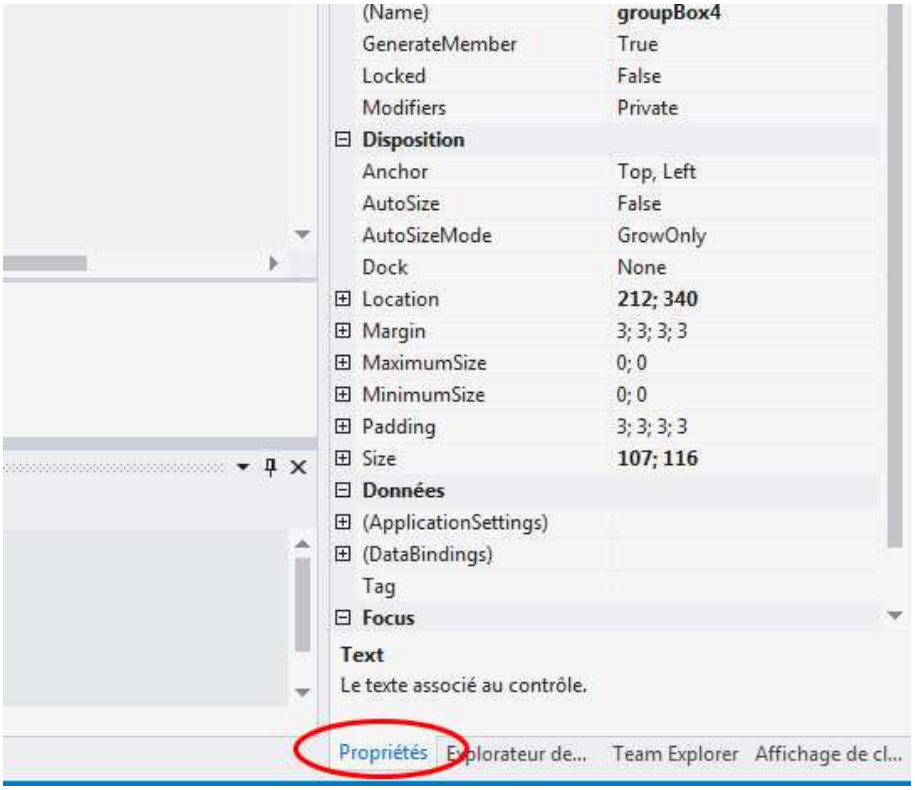


FIGURE 12 – Propriétés d'un objet de type GroupBox

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

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

⟹ Modifiez ensuite l'attribut *Name* des *RichTextBox* en *rtbEmission* et *rtbReception*. Vous devriez avoir un projet ressemblant à celui de la figure 13.

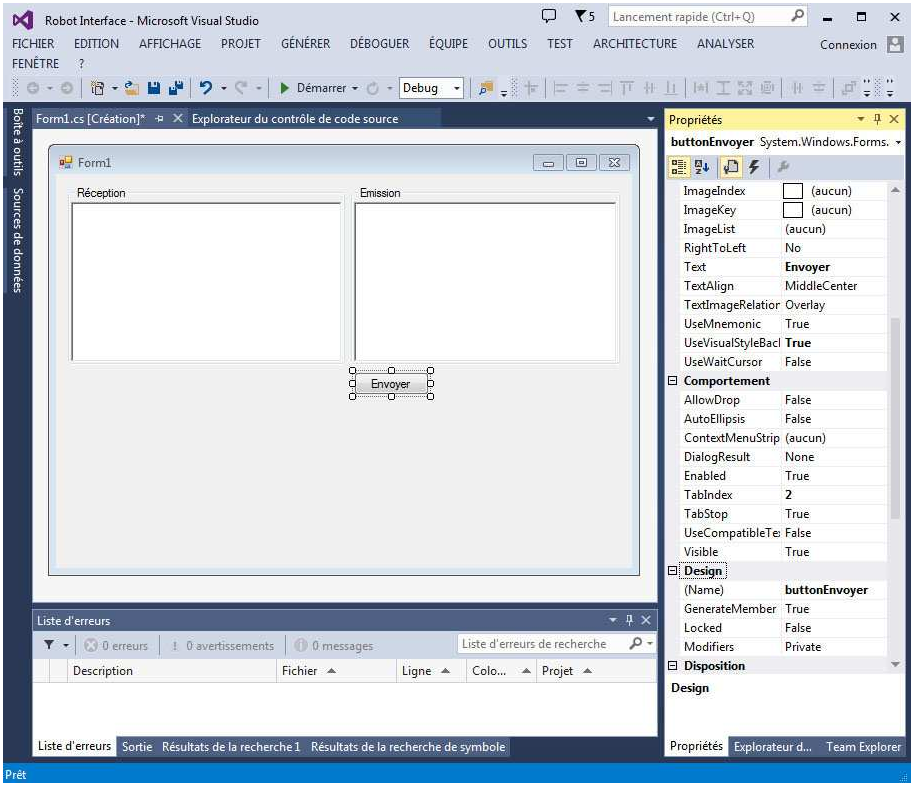


FIGURE 13 – État du projet après insertion du bouton d'envoi

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

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

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

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

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

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

sur un appui sur le bouton *Envoyer*, le contenu de la *RichTextBox* d'envoi doit être ajouté à celui de la *RichTextBox* de réception, précédé d'un retour à la ligne et de la mention : "Recu : ". La *RichTextBox* d'envoi doit également être vidée.
Le comportement doit être proche de celui de la figure 14 où 4 messages ont été envoyés successivement. Valider avec le professeur.

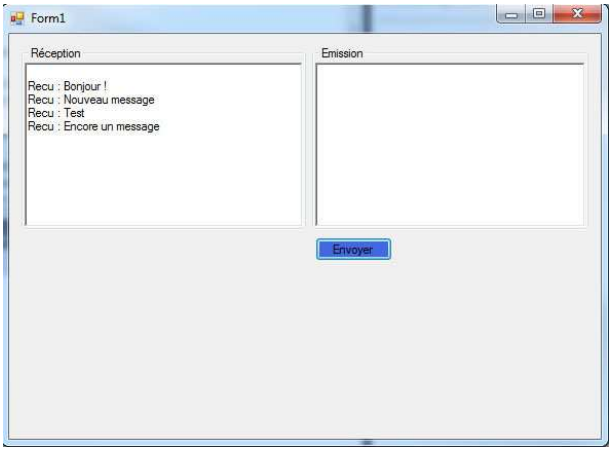


FIGURE 14 – Exemple d'exécution du simulateur d'envois

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

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

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

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

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

4.2 Messagerie instantanée entre deux PC

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

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

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

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

Les données envoyées sur le port série sont visualisables à l'aide d'un oscilloscope. Après avoir correctement configuré l'oscilloscope pour qu'il déclenche sur les arrivées de données série, et après avoir activé l'affichage en mode bus de la liaison série, vérifiez que les données envoyées correspondent aux données reçues par l'oscilloscope. Pour cela, vous brancherez un petit fil sur la pin *Transmit Data* du connecteur série comme indiqué à la figure 15 pour sortir vers l'oscilloscope.

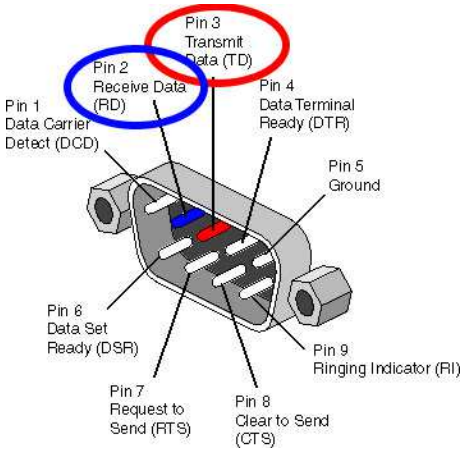


FIGURE 15 – Pinout du port RS232

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.3 Liaison série hexadécimale

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

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

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

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

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

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

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

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

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

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

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

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

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

5.1 La liaison série en embarqué

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

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

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

void InitUART(void) {
    U1MODEbits.STSEL = 0; // 1-stop bit
    U1MODEbits.PDSEL = 0; // No Parity, 8-data bits
    U1MODEbits.ABAUD = 0; // Auto-Baud Disabled
    U1MODEbits.BRGH = 1; // Low Speed mode
    U1BRG = BRGVAL; // BAUD Rate Setting

    U1STAbits.UTXISEL0 = 0; // Interrupt after one Tx character is transmitted
    U1STAbits.UTXISEL1 = 0;
    IFS0bits.U1TXIF = 0; // clear TX interrupt flag
    IEC0bits.U1TXIE = 0; // Disable UART Tx interrupt

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

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

⟹ Créer également un fichier *UART.h* contenant le code suivant :

```
#ifndef UART_H
#define UART_H

void InitUART(void);

#endif  /* UART_H */
```

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

⟹ Le programme en l'état peut-il fonctionner ? Pourquoi ?

5.2 Échange de données entre le microcontrôleur et le PC

5.2.1 Validation du bon fonctionnement du convertisseur USB/série

⟹ Ouvrir le projet *Visual Studio* que vous avez réalisé précédemment.

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

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

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

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

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

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

5.2.2 Émission UART depuis le microcontrôleur

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

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

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

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

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

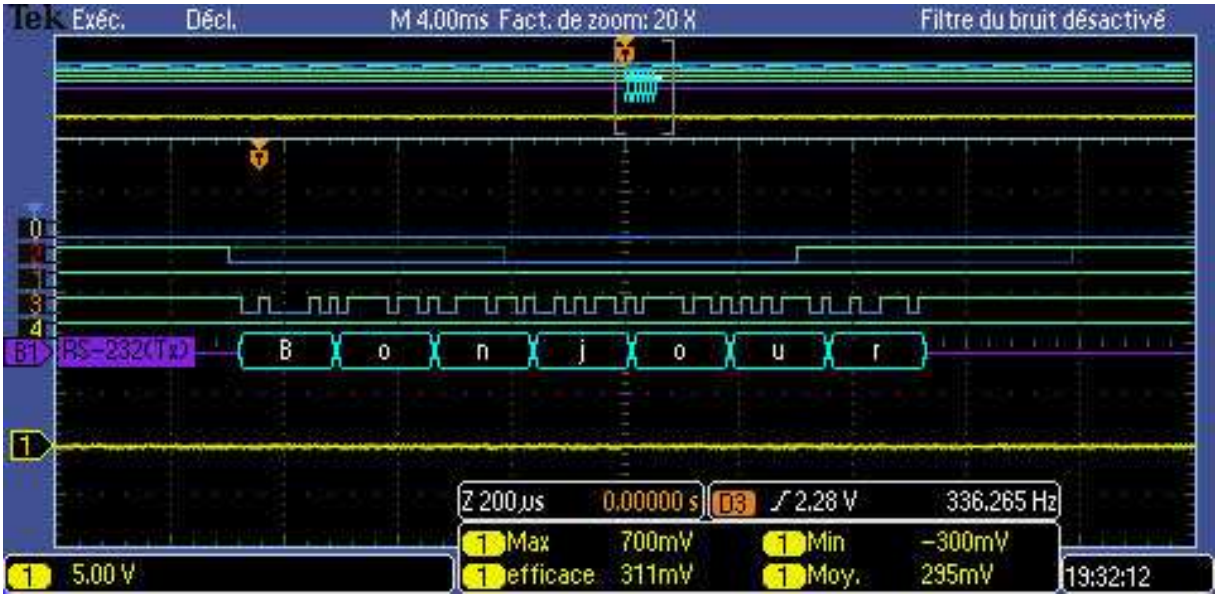
```
void SendMessageDirect(unsigned char* message, int length)
{
    unsigned char i=0;
    for(i=0; i<length; i++)
    {
        while ( U1STAbits.UTXBF); // wait while Tx buffer full
        U1TXREG = *(message)++; // Transmit one character
    }
}
```

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

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

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

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



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

5.2.3 Réception

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

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

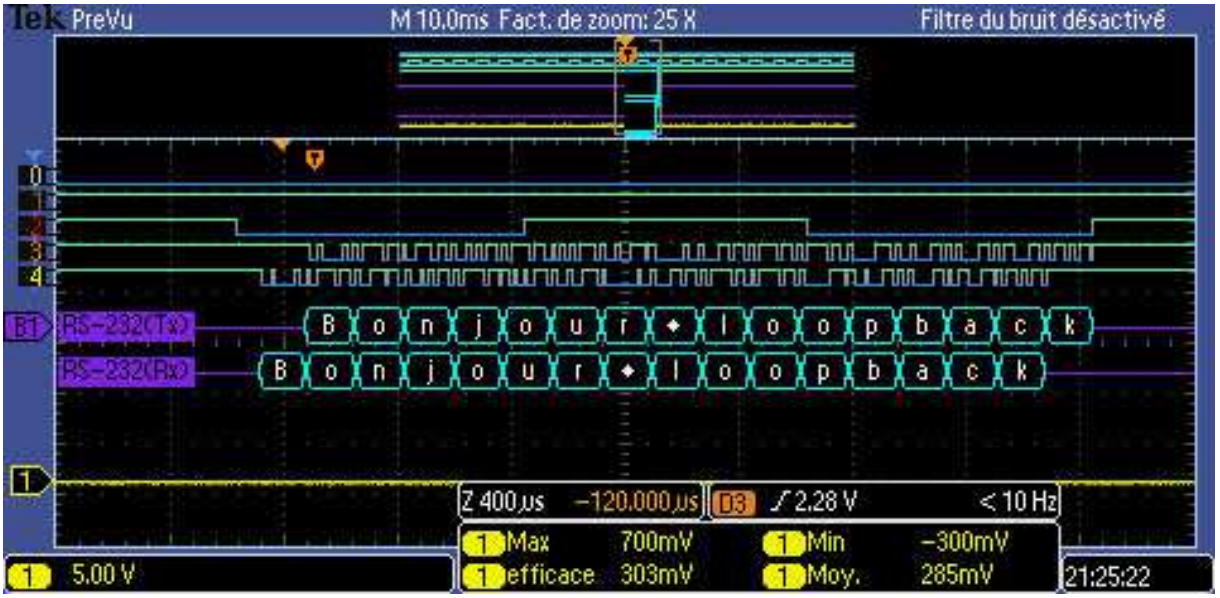
⟹ Autoriser les interruptions en réception sur l'UART en modifiant la fonction d'initialisation du port série.

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

```
//Interruption en mode loopback
void __attribute__((interrupt, no_auto_psv)) _U1RXInterrupt(void) {
    IFS0bits.U1RXIF = 0; // clear RX interrupt flag
    /* check for receive errors */
    if (U1STAbits.FERR == 1) {
        U1STAbits.FERR = 0;
    }
    /* must clear the overrun error to keep uart receiving */
    if (U1STAbits.OERR == 1) {
        U1STAbits.OERR = 0;
    }
    /* get the data */
    while (U1STAbits.URXDA == 1) {
        U1TXREG = U1RXREG;
    }
}
```

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

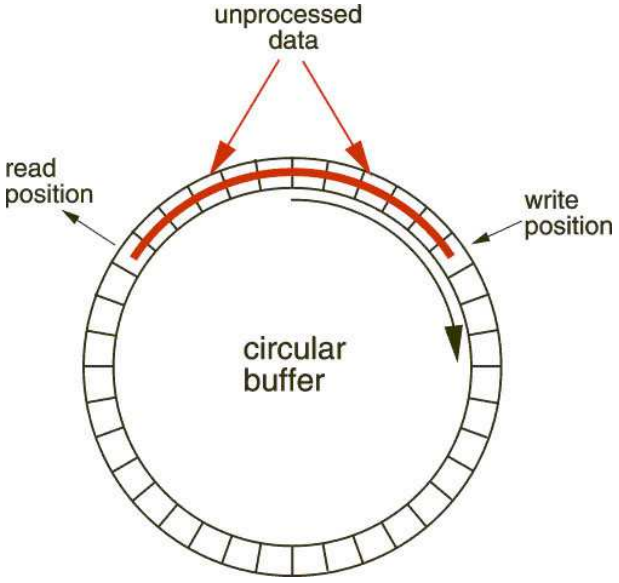
$\Longrightarrow$ Depuis l'interface *C#*, envoyez un message, celui-ci doit revenir sur la console de réception. Visualisez l'envoi et le retour de la trame sur l'oscilloscope. Vous devez obtenir un fonctionnement semblable à la capture d'écran ci-dessous.



5.3 Liaison série avec FIFO intégré

5.3.1 Le buffer circulaire de l'UART en émission

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



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

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

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

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

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

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

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

unsigned char CB_TX1_Get(void)
{
    ...
}


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

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

unsigned char CB_TX1_IsTranmitting(void)
```

```
{
    ...
}

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

⟹ Créez le header correspondant.

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

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

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

5.3.2 Le buffer circulaire de l'UART en réception

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

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

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

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

#define CBRX1_BUFFER_SIZE 128


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

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

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

```
    return value;
}

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

void __attribute__((interrupt, no_auto_psv)) _U1RXInterrupt(void) {
    IFS0bits.U1RXIF = 0; // clear RX interrupt flag
    /* check for receive errors */
    if (U1STAbits.FERR == 1) {
        U1STAbits.FERR = 0;
    }
    /* must clear the overrun error to keep uart receiving */
    if (U1STAbits.OERR == 1) {
        U1STAbits.OERR = 0;
    }
    /* get the data */
    while(U1STAbits.URXDA == 1) {
        CB_RX1_Add(U1RXREG);
    }
}

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

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

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

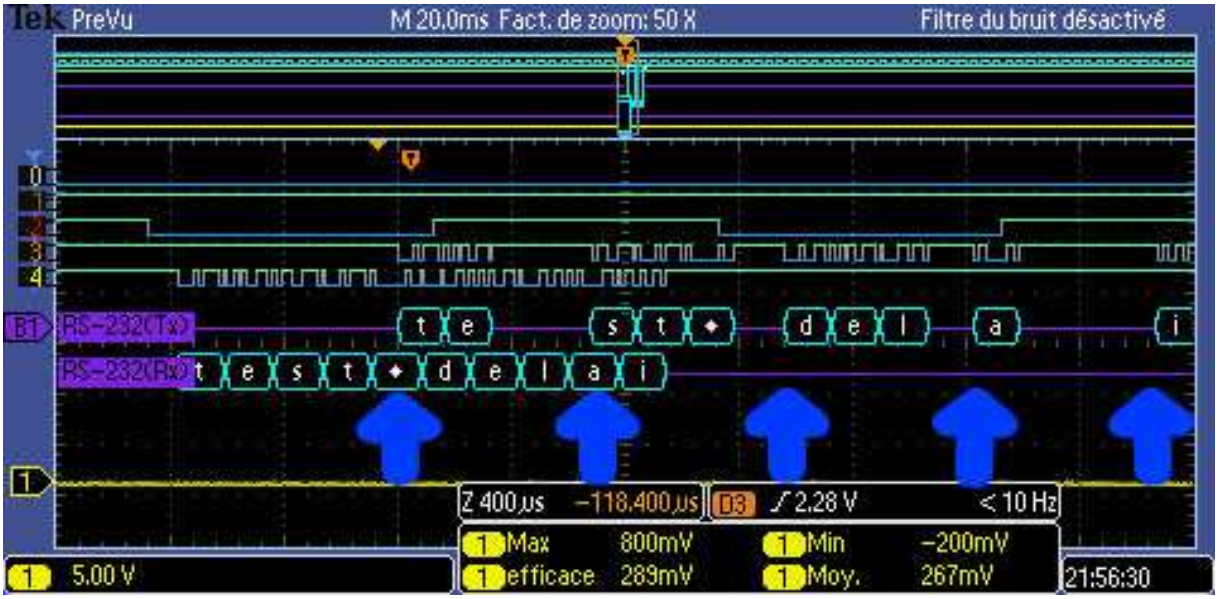
⟹ Insérez la boucle infinie du main par le code suivant, en n'oubliant pas de commenter au préalable le *SendMessage* précédent et son délai d'attente. Ce code regarde dans le buffer de réception si des caractères sont présents dans le buffer circulaire tRx, et les récupère avant de les envoyer dans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

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

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

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

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



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

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

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

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

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

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

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

6.1.1 Encodage des messages

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

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

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

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

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

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

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

6.1.2 Décodage des messages

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

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

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

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

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

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

```
    }
```

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

6.1.3 Pilotage et supervision du robot

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

Le pilotage et la supervision sont basés sur un ensemble de message définis à l'avance et qui forment une bibliothèque devant être connue du robot et de la plate-forme de supervision. Chacun de ces messages a un numéro de fonction unique, et une *payload* de taille définie à l'avance. Les premières fonction que nous allons implanter sont définies dans le tableau ci-dessous (fig. 16. Celui-ci sera complété au fur et à mesure.

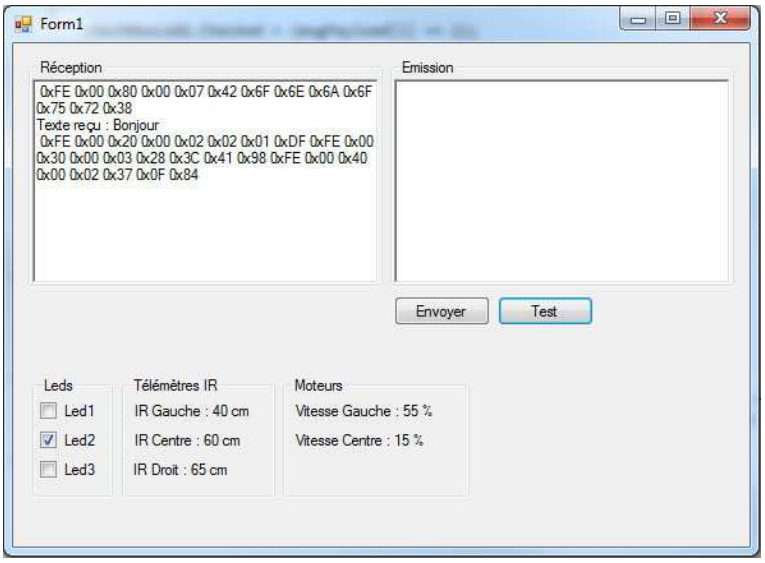
Command ID (2 bytes)	Description	Payload Length (2 bytes)	Description de la payload
0x0080	Transmission de texte	taille variable	texte envoyé
0x0020	Réglage LED	2 bytes	numéro de la LED - état de la LED (0 : éteinte - 1 : allumée)
0x0030	Distances télémètre IR	3 bytes	Distance télémètre gauche - centre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

FIGURE 16 – Fonctions de supervision

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

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

⟹ Vous pouvez vous inspirer de la figure 6.1.3 pour l'interface graphique. Valider le résultat avec le professeur.



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

6.2 Implantation en électronique embarquée

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

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

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

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

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

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

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

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

void UartProcessDecodedMessage(unsigned char function,
```

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


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

6.2.1 Supervision

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

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

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

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

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

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

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

⟹ A présent, désactivez l'appel de la fonction précédente et la temporisation bloquante qui a été ajoutée. A la fin de la fonction de conversion des valeurs issues des télémètres infrarouge, envoyez un message permettant de visualiser les valeurs sur l'interface graphique en *C#*. Le format de la trame envoyée doit être en accord avec le tableau de la figure 16.

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

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

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

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

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

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

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

6.2.2 Pilotage

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

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

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

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

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

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

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

```
}
```

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

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

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

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

6.2.3 Pilotage à l'aide d'un clavier

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

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

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

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

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

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

```
    private readonly KeyboardHookListener m_KeyboardHookManager;
```

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

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

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

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

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

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

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

7 A la découverte de la navigation par odométrie

Dans cette partie, vous allez apprendre à utiliser les modules *QEI* de gestion des encodeurs optiques à quadrature de phase. Ce encodeurs optiques sont constitués de roues codeuses perforées de trous.

Le but de cette partie est de :
- Mettre en oeuvre le module *QEI* dans un premier temps.
- Déterminer les déplacements du robot et les afficher sur une carte en temps réel.
- Mettre en oeuvre un asservissement de type *P*, *PI* ou *PID*.

7.1 Mise en oeuvre du module QEI

Le microcontrôleur utilisé possède deux modules QEI capables de gérer les signaux des encodeurs optiques à quadrature de phase placés sur des roues indépendantes. Leur role est de compter ou de décompter les impulsions du codeurs selon que la roue avance ou recule. Cette fonction basique pourrait être réalisée en mode interruption sur le microcontrôleur mais le flux étant très important, le taux d'occupation du processeur serait élevé. L'utilisation d'un module hardware permet donc de décharger le processeur de cette tache. Pour cela il faut configurer le module et les pins remappables correspondantes.

⟹ Ajouter au code de paramétrage des pins remappables situé dans *"IO.c"* le code suivant :

```
    //******************** QEI *****************
    _QEA2R = 97;     //assign QEI A to pin RP97
    _QEB2R = 96;     //assign QEI B to pin RP96

    _QEA1R = 70;     //assign QEI A to pin RP70
    _QEB1R = 69;     //assign QEI B to pin RP69
```

⟹ Ajouter dans un fichier *"QEI.c"* les fonctions suivantes permettant d'initialiser les deux modules QEI. Ajouter le header correspondant et appeler ces fonctions depuis le main avant la boucle infinie.

```
void InitQEI1()
{
    QEI1IOCbits.SWPAB = 1; //QEAx and QEBx are swapped
    QEI1GECL = 0xFFFF;
    QEI1GECH = 0xFFFF;
    QEI1CONbits.QEIEN = 1; // Enable QEI Module
}

void InitQEI2(){
    QEI2IOCbits.SWPAB = 1; //QEAx and QEBx are not swapped
    QEI2GECL = 0xFFFF;
    QEI2GECH = 0xFFFF;
    QEI2CONbits.QEIEN = 1; // Enable QEI Module
}
```

7.2 Détermination des déplacements du robot et supervision

A ce stade, les modules QEI sont actifs en tache de fond, mais les valeurs du QEI ne sont pas encore récupérées par le robot.

⟹ Ajouter à *"QEI.c"* une fonction *QEIUpdateData* permettant de récupérer les données issues du QEI. Cette fonction sera appelée à intervalle régulier, par exemple à 250 Hz sur l'interruption du timer 1. Elle contiendra le code suivant :

```
#define DISTROUES 281.2
void QEIUpdateData()
{
    //On sauvegarde les anciennes valeurs
```

```
    QeiDroitPosition_T_1 = QeiDroitPosition;
    QeiGauchePosition_T_1 = QeiGauchePosition;

    //On éractualise les valeurs des positions
    long QEI1RawValue = POS1CNTL;
    QEI1RawValue += ((long)POS1HLD<<16);

    long QEI2RawValue = POS2CNTL;
    QEI2RawValue += ((long)POS2HLD<<16);

    //Conversion en mm (éérgl pour la taille des roues codeuses)
    QeiDroitPosition = 0.01620*QEI1RawValue;
    QeiGauchePosition = -0.01620*QEI2RawValue;

    //Calcul des deltas de position
    delta_d = QeiDroitPosition - QeiDroitPosition_T_1;
    delta_g = QeiGauchePosition - QeiGauchePosition_T_1;
    //delta_theta = atan((delta_d - delta_g) / DISTROUES);
    delta_theta = (delta_d - delta_g) / DISTROUES;
    dx = (delta_d + delta_g) / 2;

    //Calcul des vitesses
    //attention à remultiplier par la éfrquence dé'chantillonnage
    robotState.vitesseDroitFromOdometry = delta_d*FREQ_ECH_QEI;
    robotState.vitesseGaucheFromOdometry = delta_g*FREQ_ECH_QEI;
    robotState.vitesseLineaireFromOdometry =
        (robotState.vitesseDroitFromOdometry + robotState.vitesseGaucheFromOdometry)/2;
    robotState.vitesseAngulaireFromOdometry = delta_theta*FREQ_ECH_QEI;

    //Mise à jour du positionnement terrain à t-1
    robotState.xPosFromOdometry_1 = robotState.xPosFromOdometry;
    robotState.yPosFromOdometry_1 = robotState.yPosFromOdometry;
    robotState.angleRadianFromOdometry_1 = robotState.angleRadianFromOdometry;

    //Calcul des positions dans le referentiel du terrain
    robotState.xPosFromOdometry = ...
    robotState.yPosFromOdometry = ...
    robotState.angleRadianFromOdometry = ...
    if(robotState.angleRadianFromOdometry > PI)
        robotState.angleRadianFromOdometry -= 2*PI;
    if(robotState.angleRadianFromOdometry < -PI)
        robotState.angleRadianFromOdometry += 2*PI;
}
```

$\Longrightarrow$ Expliquer dans la détail ce que fait le code précédent et complétez les parties manquantes signalées par des ... Pensez à justifier la présence des multiplications par la fréquence d'échantillonnage dans le calcul des vitesses de déplacement. Il vous faudra également ajouter à la structure robotState les grandeurs utilisées dans le code, elle sont de type *double*.
A présent, votre code est capable de calculer en temps réel la position du robot relative à sa position de départ.

Il reste à transmettre ces informations à l'interface de supervision.

$\Longrightarrow$ Implanter dans le fichier "QEI.c" une fonction permettant de réaliser la transmission des données de positionnement avec leur horodatage. Elle ressemblera au code suivant, en remplaçant les ... par le code qui convient. Pour la transmission de la vitesse linéaire du robot et de la vitesse angulaire, veiller à les multiplier au préalable par 1000 de manière à permettre une transmission des décimales. La valeur reçue sera divisée par 1000 dans l'interface en C# qui vous est fournie.

```
#define POSITION_DATA 0x0061
void SendPositionData() {
    unsigned char positionPayload[24];
```

```
    unsigned int i;
    for (i = 0; i < 4; i++) {
        positionPayload[3 - i] = (unsigned char) (timestamp>>(8 * i));
        positionPayload[7 - i] = (unsigned char) ... ; //Transmission de xPosFromOdometry
        positionPayload[11 - i] = (unsigned char) ... ; //Transmission de yPosFromOdometry
        positionPayload[15 - i] = (unsigned char) (((long) (RadianToDegree(robotState.angleRadianF
        positionPayload[19 - i] = (unsigned char) ... ; //Transmission de vitesseLineaireFromOdome
        positionPayload[23 - i] = (unsigned char) ... ; //Transmission de vitesseAngulaireFromOdon
    }
    Uart1EncodeAndSendMessage(POSITION_DATA, 24, positionPayload);
}
```

⟹ Appeler la fonction *SendPositionData* depuis l'interruption du Timer 1, en veillant à **sous-échantillonner les envois** de manière à ne pas envoyer plus de *10* messages par seconde pour éviter de saturer la liaison UART. Si tout se passe bien, vous devriez voir apparaitre les informations de positionnement sur l'interface du robot. Valider avec le professeur le bon fonctionnement de l'ensemble.

7.3 Asservissement polaire en vitesse du robot

Dans cette partie vous allez asservir votre robot en vitesse. Cet asservissement peut être réalisé roue par roue avec un correcteur de type PI ou PID, de manière à assurer une erreur statique nulle en vitesse comme présenté à la figure 17.

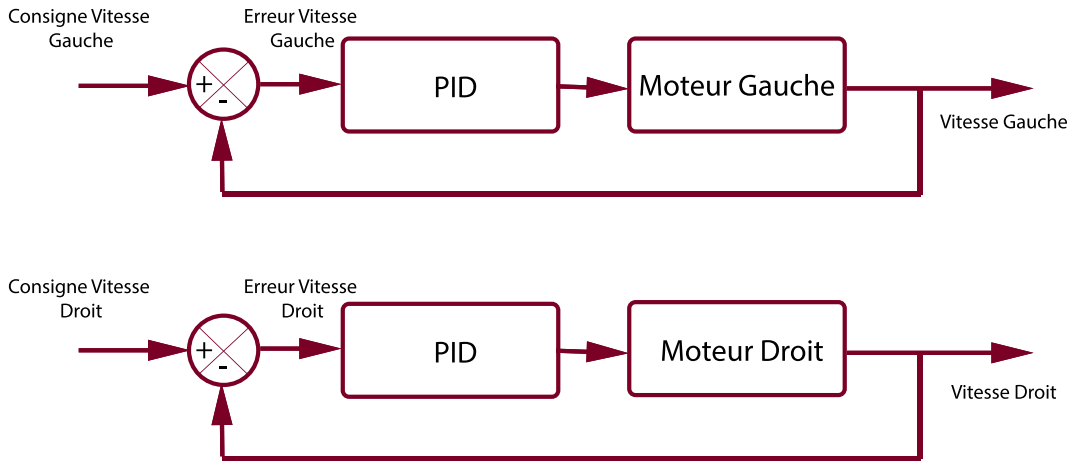


FIGURE 17 – Asservissement PID classique

Ce type d'asservissement est performant car il permet d'assurer que les deux roues tourneront à la même vitesse si on leur donne une consigne identique, le robot roulera donc en ligne droite en régime permanent, ce qui n'est pas le cas sans asservissement. Toutefois, l'inconvénient de cette méthode est d'asservir les moteurs indépendamment l'un de l'autre : si un moteur démarre plus lentement que l'autre en raison de frottements ou d'une charge mal répartie, alors il va prendre du retard avant d'arriver à sa vitesse de consigne. Ce retard se traduira par une erreur sur le cap du robot.
Cette erreur de cap devra être compensée par une contre-réaction sur les vitesses de chacun des moteurs. Ce modèle n'est pas optimal : il serait préférable de contrôler la vitesse angulaire qui donne le cap par intégration directe, ainsi que la vitesse linéaire du robot plutôt que de contrôler directement les vitesses de ses deux moteurs. C'est l'objet de l'**asservissement polaire** que vous allez mettre en oeuvre.
Les équations de couplage entre les vitesses des moteurs et les vitesses angulaires et linéaires du robot sont les

suivantes :

$$V_{lineaire} = \frac{V_{Droit} + V_{Gauche}}{2} \quad (mm.s^{-1})$$
$$V_{angulaire} = \frac{V_{Droit} - V_{Gauche}}{D_{Roues}} \quad (rad.s^{-1})$$

Si on inverse ces équations, on obtient :

$$V_{Gauche} = V_{lineaire} - V_{angulaire}\frac{D_{Roues}}{2} \quad (mm.s^{-1})$$
$$V_{Droit} = V_{lineaire} + V_{angulaire}\frac{D_{Roues}}{2} \quad (mm.s^{-1})$$

On peut donc en déduire le schéma synoptique de l'asservissement polaire de la figure 18.

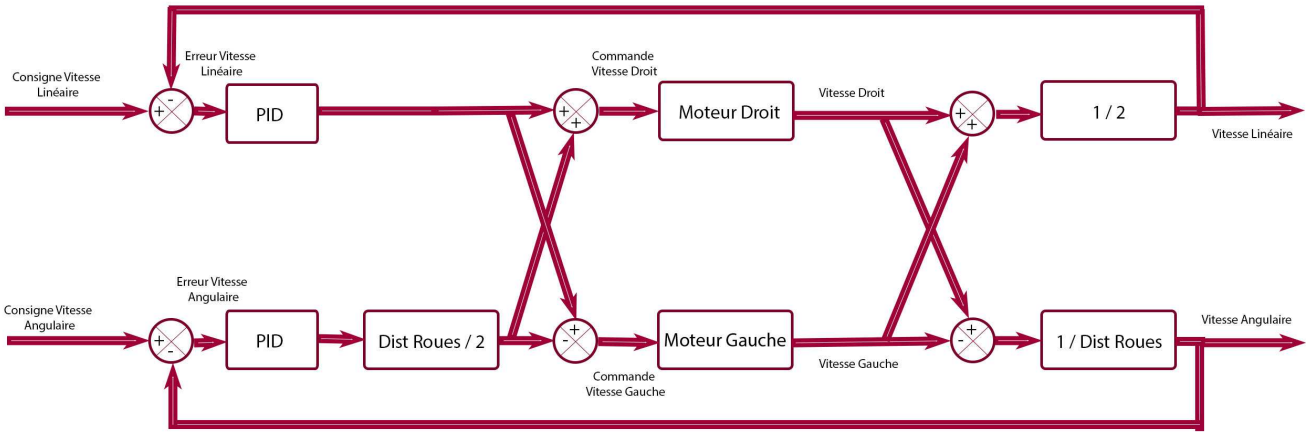


FIGURE 18 – Asservissement polaire en vitesse

⟹ Ecrire une fonction *PWMSetSpeedConsignePolaire* permettant de générer les commandes des moteurs droits et gauche dans cet asservissement polaire en complétant le code à trou ci-dessous. Dans un premier temps, remplacer les deux correcteurs PID par deux correcteurs P, ce qui revient à multiplier l'erreur par une constante de proportionnalité.

```
#define COEFF_VITESSE_LINEAIRE_PERCENT 1/25.
#define COEFF_VITESSE_ANGULAIRE_PERCENT 1/50.
void PWMSetSpeedConsignePolaire(){
    //Correction Angulaire
    double erreurVitesseAngulaire = ...
    double sortieCorrecteurAngulaire = ...
    double correctionVitesseAngulaire = ...
    double correctionVitesseAngulairePourcent =
            correctionVitesseAngulaire * COEFF\_VITESSE\_ANGULAIRE\_PERCENT;

    //Correction éLinaire
    double erreurVitesseLineaire = ...
    double sortieCorrecteurLineaire = ...
    double correctionVitesseLineaire = ...
```

```
    double correctionVitesseLineairePourcent =
                correctionVitesseLineaire * COEFF\_VITESSE\_LINEAIRE\_PERCENT;

    //ééGnration des consignes droite et gauche
    robotState.vitesseDroiteConsigne = correctionVitesseLineairePourcent
                + correctionVitesseAngulairePourcent ;
    robotState.vitesseDroiteConsigne = LimitToInterval(
                robotState.vitesseDroiteConsigne, -100, 100);
    robotState.vitesseGaucheConsigne = correctionVitesseLineairePourcent
                - correctionVitesseAngulairePourcent;
    robotState.vitesseGaucheConsigne = LimitToInterval(
                robotState.vitesseGaucheConsigne, -100, 100);
}
```

$\Longrightarrow$ Appeler cette fonction à la place de PWMSetSpeedConsigne() dans l'interruption du *Timer 1.*

Il est à présent temps de régler l'asservissement polaire du robot. Dans un premier temps, nous allons régler l'asservissement en vitesse angulaire avant de régler l'asservissement de vitesse linéaire. Pour cela, durant la phase de réglage de l'asservissement angulaire, il vous faudra mettre la correction de vitesse linéaire à 0 à la main dans la fonction PWMSetSpeedConsignePolaire, donner une consigne de vitesse de rotation (par exemple 3 $rad.s^{-1}$), puis augmenter progressivement le gain du correcteur angulaire jusqu'à atteindre l'oscillation du moteur. Celle-ci se traduit par un bruit de vibration et une forte augmentation du courant absorbé sur l'alimentation stabilisée.

$\Longrightarrow$ Réaliser la manipulation présentée précédemment et déterminer le gain proportionnel limite d'oscillation. Valider avec le professeur.

Il reste à présent à remplacer le gain proportionnel par un correcteur PID bien réglé. Les formules d'implantation du PID dans le domaine de Laplace et en temps discret sont rappelées ci-dessous :

$$H(s) = K_p + K_i/s + K_d s = K_p(1 + \frac{1}{T_i s} + T_d s)$$

Ce qui donne avec l'approximation d'Euler :

$$H(z) = \frac{K_p(1 - z^{-1}) + K_i T_e + \frac{K_d}{T_e}(1 - z^{-1})^2}{1 - z - 1}$$

Soit :

$$S(z)(1 - z^{-1}) = E(z)\left((K_p + K_i T_e + \frac{K_d}{T_e}) + (-K_p - 2\frac{K_d}{T_e})z^{-1} + \frac{K_d}{T_e}z^{-2}\right)$$

$$S_n = S_{n-1} + E_n(K_p + K_i T_e + \frac{K_d}{T_e}) + E_{n-1}(-K_p - 2\frac{K_d}{T_e}) + E_{n-2}(\frac{K_d}{T_e})$$

Le réglage du PID sera effectué par la méthode de Ziegler-Nichols en boucle fermée. C'est une méthode empirique qui consiste à rechercher dans un premier temps le pompage limite, c'est à dire le gain proportionnel limite K_u conduisant à l'oscillation, ce que vous avez fait précédemment. Leur période est T_u et doit être mesurée, mais on prendra 0.05s en première approximation. Une fois K_u et T_u obtenues, il est possible de mettre en oeuvre un correcteur, P, PI ou PID en réglant les gains de la manière suivante :

	P	PI	PID
K_p	$0.5K_u$	$0.45K_u$	$0.6K_u$
T_i	-	$0.83T_u$	$0.5T_u$
T_d	-	-	$0.125T_u$

$\Longrightarrow$ Ajouter un fichier *Asservissement.c* (pensez à créer le header correspondant !) à votre projet dans lequel vous mettrez le code à trous suivant en le complétant avec les équations précédentes :

```
#include "Asservissement.h"
#include "timer.h"

double Te = 1 / FREQ_ECH_QEI;

//************************ CORRECTEUR VITESSE ANGULAIRE ****************************
double eAng0, eAng1, eAng2; //valeurs de l'éentre du correcteur gauche à t, t-1, t-2
double sAng0, sAng1, sAng2; //valeurs de la sortie du correcteur gauche à t, t-1, t-2

void SetUpPiAsservissementVitesseAngulaire(double Ku, double Tu)
{
    //éRglage de Ziegler Nichols sans édpassements : un tout petit peu mou
    double Kp = ...
    double Ti = ...
    double Td = 0;
    double Ki = Kp / Ti ;
    double Kd = Kp * Td;

    alphaAng = ...
    betaAng = ...
    deltaAng = ...
}

double CorrecteurVitesseAngulaire(double e)
{
    eAng2 = eAng1;
    eAng1 = eAng0;
    eAng0 = e;
    sAng1 = sAng0;
    sAng0 = sAng1 + eAng0 * alphaAng + eAng1 * betaAng + eAng2*deltaAng;
    return sAng0;
}
```

$\Longrightarrow$ Appelez la fonction *SetUpPiAsservissementVitesseAngulaire* à l'initialisation du *main* en expliquant son rôle. Pensez à inclure les header.

$\Longrightarrow$ Dans le code de la fonction *PWMSetSpeedConsignePolaire*, remplacer dans le calcul de la correction de vitesse angulaire la proportionnalité à l'erreur par la sortie de la fonction correcteur *CorrecteurVitesseAngulaire*.

$\Longrightarrow$ Tester votre code. A ce stade, votre robot doit être asservi en vitesse de rotation. Vérifier que c'est bien le cas, en freinant un des rouleaux du support robot. Si c'est bien le cas, vous avez terminé la partie asservissement en vitesse angulaire.

$\Longrightarrow$ Il vous faut à présent réaliser l'asservissement en vitesse linéaire de votre robot. Pour cela, régler à 0 la consigne de vitesse angulaire asservie précédemment. Régler également la consigne de vitesse liénaire à 300 $mm.s^{-1}$. Réintégrer une correction de type proportionnel sur la vitesse linéaire dans *PWM.c*, trouver le gain limite et implanter comme précédemment un asservissement de type PI sur la vitesse angulaire.

$\Longrightarrow$ Faites varier la consigne de vitesse linéaire et tester. Votre moteur doit notament pouvoir fonctionner à des très aibles vitesses de rotations si l'asservissement fonctionne bien. Valider l'ensemble avec le professeur.

$\Longrightarrow$ Pour terminer cette partie sur l'asservissement de vitesse polaire. Reimplantez la machine à état de fonctionnement du robot en mode autonome pour l'évitement d'obstacles en pilotant votre robot grâce à l'asservissement polaire en vitesse.

8 Gestion des tâches - pilotage avancé

Un robot doit pouvoir effectuer des tâches très variables. Nous avons vu qu'il pouvait être amener à se déplacer de manière autonome en évitant des obstacles, mais il peut aussi être amener à effectuer une succession d'actions prédéterminées formant des tâches. Il est également important que celles-ci puissent être séquentialisées, interrompues ou parallèlisées.

Le but de cette partie est de découvrir les rudiments de la gestion de tâches, qui s'apparente à un *Operating System* temps réel (RTOS) très simplifié, en particulier au niveau des méthodes de synchronisation.

Afin d'explorer ces concepts, nous allons implanter un *Task Manager* qui aura pour rôle de synchroniser les tâches du robot. Ces tâches seront les suivantes :

- Attente pendant 5 secondes.
- Déplacement en mode autonome comme déjà vu.
- Déplacement vers un point donné(asservissement de position)
- Déplacement successif sur les quatres coins d'un carré dans le mode précédent.

8.1 Un exemple de tâche : l'asservissement en position du robot

$\Longrightarrow$ Vous avez à présent terminé la mise en oeuvre de l'odométrie, de la reconstruction de trajectoire et de l'asservissement de votre robot. Vous pouvez à présent imaginer un asservissement en position de votre robot, qui vous permettra d'effectuer des déplacements extrêmement précis.

9 A la découverte des bus terrains dans les microcontrôleurs

9.1 Le bus SPI

La carte de contrôle des robots permet de raccorder des capteurs à un bus SPI de manière simple, via deux connecteurs dénommés SPI1 et SPI2 (fig.2).

9.1.1 Interfaçage de capteurs SPI

La mesure des paramètres d'un système embarqué est une fonctionnalité qui se retrouve dans la grande majorité des systèmes actuels. En particulier les mesures de position, vitesse et accélération sont très communes.

Les capteurs permettant ces mesures sont le plus souvent interfacés avec un microcontrôleur via un bus terrain de type I2C ou SPI. Nous proposons dans cette partie de réaliser cet interfaçage sur une centrale inertielle comprenant un accéléromètre 3 axes, un gyroscope 3 axes et un magnétomètre 3 axes.
Le bus SPI du DSPIC permet des communications série à haute vitesse avec des composants externes. La communication est de type *Master-Slave*, le *Master* étant le DSPIC.

Le $dsPIC33FJ512GM306$ utilisé possède deux bus SPI. Nous utiliserons le premier. **Attention, une erreur s'est glissée dans la sérigraphie de la carte. SPI1 est en SPI2 et vice-versa.** Les pinsdu module *SPI1* ne sont pas remappables, et il n'est donc pas nécessaire de faire d'autre déclaration que leur état entrée ou sortie. Ces pins sont les suivantes :
— $SCLK$: Sortie *SPI Clock* : pin $C3$ (sortie)
— $MOSI$: Sortie *Master Out Slave In* : pin $A4$ (sortie)
— $MISO$: Entrée *Master In Slave Out* : pin $A9$ (entrée)
Le *Chip Select* (CS) est une pin simple configurée en sortie. On le placera sur la pin $B7$ correspondant sur la carte à *SPI SS1*.

La centrale inertielle utilisée est une *MPU9250* de chez InventSense, que l'on retrouve dans de nombreux produits grand public.

$\Longrightarrow$ Afin de comprendre comment fonctionne cette centrale inertielle et surtout d'avoir une référence complète sur le sujet, vous pouvez télécharger sa documentation technique à l'aide du lien suivant : Page de présentation de la $MPU9250$.

$\Longrightarrow$ Configurez les entrées-sorties du $dsPIC$ pour que le SPI puisse fonctionner, ainsi qu'une macro permettant d'accéder facilement au *Chip Select* à l'aide du code suivant, placé dans *IO.h* :

```
#define SS_MPU9250 _LATB7
```

$\Longrightarrow$ Configurez la pin remappable $RP18$ ($B5$), afin qu'elle gère l'interruption *INT1*. Cette interruption permettra de savoir quand les données du gyroscope sont disponibles.

$\Longrightarrow$ Insérez dans le projet les fichiers SPI.c, SPI.h, SPI_MPU9250.c et SPI_MPU9250.h.

$\Longrightarrow$ Examinez le code des fichiers fournis précédemment. Expliquer dans le détail dans votre rapport le fonctionnement du SPI, codé dans la fonction *WriteMultipleCommandMultipleReadSpi1* située dans le fichier *"SPI.c"*. Cette fonction est le coeur de la liaison SPI.

$\Longrightarrow$ Insérez dans le code du fichier *"main.c"* les initialisations du SPI, du driver du gyroscope et de l'interruption nécessaire au fonctionnement du gyroscope

$\Longrightarrow$ Insérez dans la boucle infinie du "main.c" le code à compléter suivant, qui permet la récupération des données du gyroscope :

```
if(MPU9250IsAccelDataReady())
{
    ...
}
```

⟹ Testez votre code. En inclinant successivement la carte sur les 3 axes, vous devriez avoir un résultat semblable à celui-ci dans l'interface de visualisation.

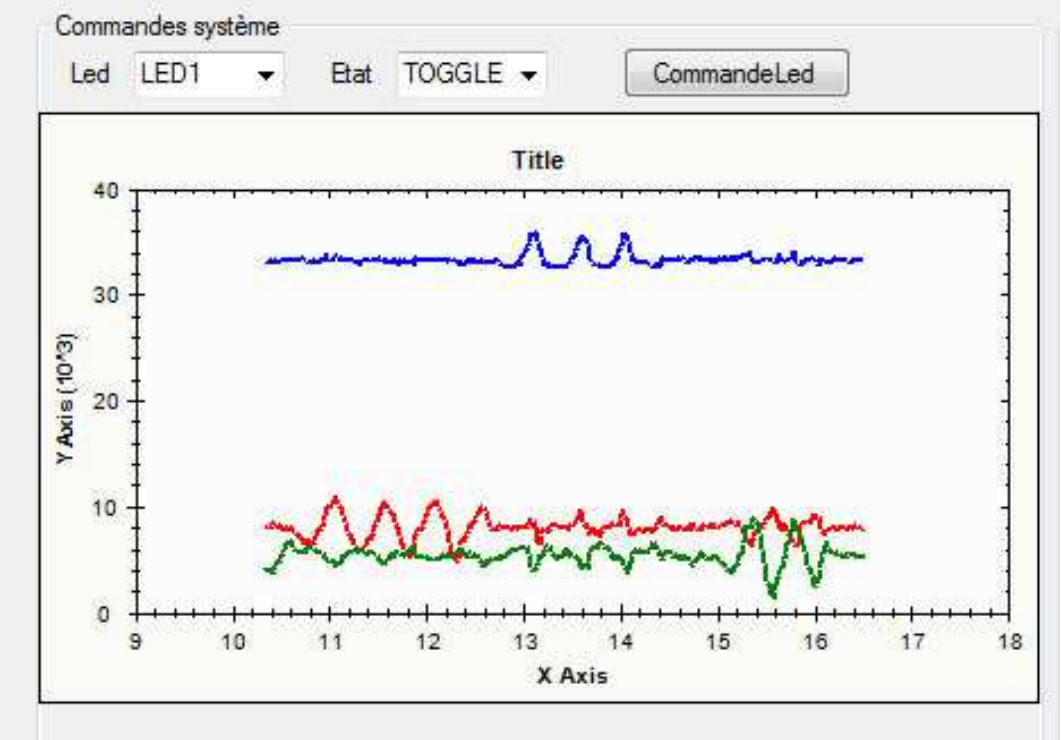


FIGURE 19 – Signaux du gyroscope 3 axes

⟹ Visualisez à l'aide de l'oscilloscope les signaux SPI du gyroscope. Vous devriez obtenir en cycle normal des signaux de la forme suivante :

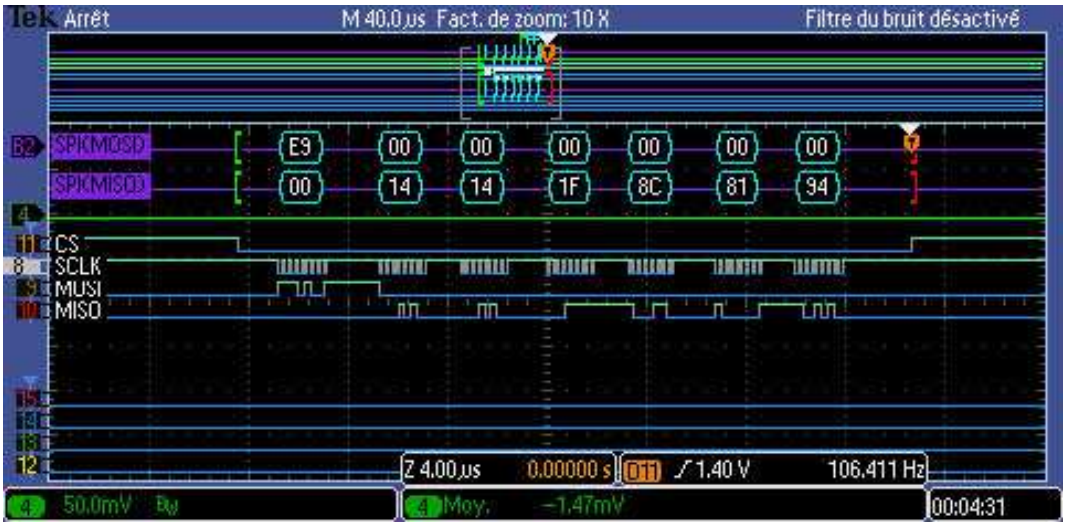


FIGURE 20 – Chronogrammes d'une récupération de données sur le gyroscope via la liaison SPI

⟹ En zoomant sur le signal *SCLK*, mesurez la fréquence d'horloge SPI. Est-ce conforme à ce que vous pouvez voir dans le code (regardez les pre et port scaler de la liaison SPI déclarés dans la fonction *L3G4200SetSPIFreq*) ?

⟹ Expliquez le fonctionnement de l'interruption du gyroscope en vous aidant de la datasheet et du code fourni. En particulier, indiquez à quelle condition l'interruption est levée et à quelle condition elle revient à 0. Décrivez le mécanisme de récupération des données associé à cette interruption.

⟹ Visualisez simultanément les signaux SPI et UART. Vous devriez avoir des signaux de la forme suivante :

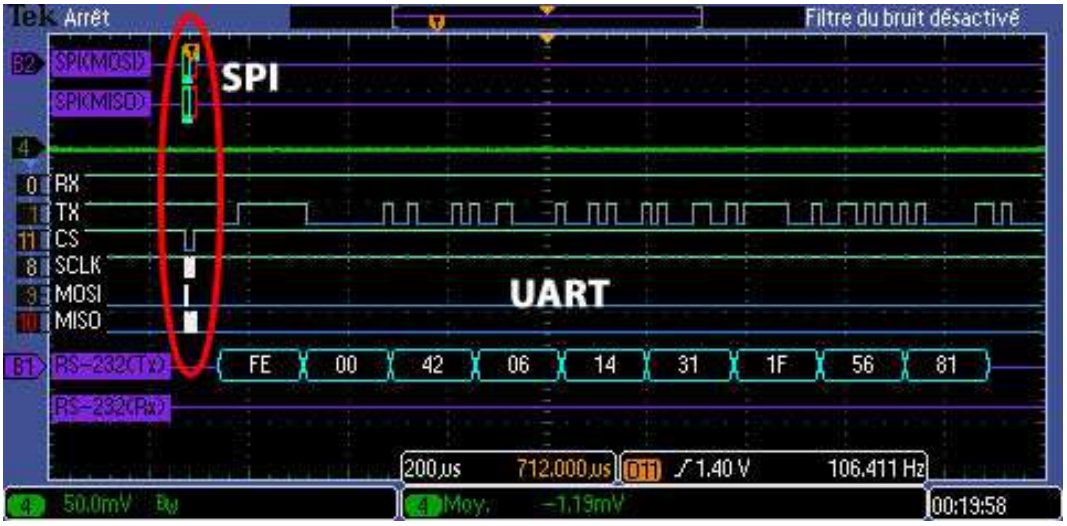


FIGURE 21 – Chronogrammes d'une récupération de données sur le gyroscope via la liaison SPI suivie de l'envoi UART

⟹ Que pouvez-vous dire des vitesses relatives de deux types de bus terrain étudiés ?
A présent que vous avez fait fonctionner l'accéléromètre de la $MPU9250$, faites de même avec le gyroscope.

⟹ En vous inspirant du driver de l'accéléromètre de la *MPU9250*, créez un driver pour le gyroscope de la

MPU9250, et les fonctions permettant de visualiser dans l'interface *C#* les données issues du capteur.

9.2 Le bus I2C

Le bus I2C est un bus terrain utilisé dans nombre de capteurs fonctionnant avec une fréquence d'acquisition modérée. Nous proposons dans cette partie de réaliser un interfaçage sur bus I2C pour un capteur de type télémètre à ultra-sons assez perfectionné.

$\Longrightarrow$ Sa référence est le *SFR08*. Afin de comprendre son fonctionnement, vous pouvez télécharger sa documentation technique à l'aide du lien suivant : Lien vers la page du SFR08.

$\Longrightarrow$ Afin de vous gagner un peu de temps un driver générique pour l'I2C vous est fourni. Il est téléchargeable ici : I2C.c et I2C.h. Attention : l'I2C est un bus typiquement non fiable, qui se fige de manière aléatoire. Dans ce cas, des procédures doivent être mise en oeuvre une fois le blocage détecté. Cette détection est prévue dans le driver ci-dessus, elle nécessite pour fonctionner l'appel de la fonction IncrementI2CAntiBlockCounter, depuis un timer à une fréquence de 1kHz.

$\Longrightarrow$ Le code suivant est appelé périodiquement sur une interruption timer qui lance un event lu dans le main. Expliquez sommairement ce qu'il fait à partir de la datasheet (la dernière ligne est très importante), et déterminer la fréquence maximale à laquelle doit fonctionner le timer. En quoi est-ce intéressant d'utiliser des télémètre à ultrasons ayant un mode broadcast ?

```
if(IsEventActive(EVENT_ULTRASONIC_SENSOR))
{
    ClearEvent(EVENT_ULTRASONIC_SENSOR);
    unsigned char trame[8];
    unsigned char data[10];
    unsigned int telemetreValue;

    //Cas Lecture et écriture synchrone
    I2C1ReadN(TELEMETRE_GAUCHE, 0x00, data, 10);
    telemetreValue = ((unsigned int)data[2])*256 + ((unsigned int)data[3]);
    if(telemetreValue > 0)
    {
        SetDistanceTelemetreFaceGauche(telemetreValue);
        telemetreValue = GetDistanceTelemetreFaceGauche();
        trame[0] = (unsigned char)(telemetreValue);
        trame[1] = (unsigned char)(telemetreValue>>8);
    }
    I2C1ReadN(TELEMETRE_DROIT, 0x00, data, 10);
    telemetreValue = ((unsigned int)data[2])*256 + ((unsigned int)data[3]);
    if(telemetreValue > 0)
    {
        SetDistanceTelemetreFaceDroit(telemetreValue);
        telemetreValue = GetDistanceTelemetreFaceDroit();
        trame[2] = (unsigned char)(telemetreValue);
        trame[3] = (unsigned char)(telemetreValue>>8);
    }
    I2C1ReadN(TELEMETRE_CENTRE, 0x00, data, 10);
    telemetreValue = ((unsigned int)data[2])*256 + ((unsigned int)data[3]);
    if(telemetreValue > 0)
    {
        SetDistanceTelemetreFaceCentre(telemetreValue);
        telemetreValue = GetDistanceTelemetreFaceCentre();
        trame[4] = (unsigned char)(telemetreValue);
        trame[5] = (unsigned char)(telemetreValue>>8);
    }
    #ifdef DEBUG_MODE_TELEMETRE
        UartEncodeAndSendMessage(0xAD, trame, 8);
    #endif

    I2C1Write1(ALL_I2C, 0x00, 0x51);  //Mesure en broadcast sur tous les SFR08 en êmme temps
```

```
}
```

$\Longrightarrow$ A partir de l'analyse de la documentation technique, déterminer un appel de fonction *I2C* d'initialisation possible pour les télémètres dans le cas où ils sont placés horizontalement et qu'il sont donc susceptibles de recevoir un premier écho en provenance du sol.

10 Interfaçage d'un télémètre laser à balayage RPLIDAR en C#

Le télémètre laser à balayage RPLIDAR permet d'effectuer de construire des cartes de distances à 360° autour d'un robot. Son usage est à la base des applications de cartographie et localisation simultanées, connues en anglais sous le nom de SLAM (Simultaneous Localization And Mapping) ou CML (Concurrent Mapping and Localization).
Le télémètre est présenté sur le site internet du constructeur :

http://www.slamtec.com/en/lidar.

Vous trouverez en particulier le SDK avec toutes les informations sur le protocole utilisé en fin de page.
Vous allez mettre en oeuvre ce télémètre en C# en analysant le protocole proposé par le constructeur et en proposant une machine à état permettant de décoder les trames en provenance du télémètre. Pour cela, un code mettant en oeuvre une liaison série, les fonctions permettant de lancer les acquisitions sur le télémètre, ainsi qu'un outil graphique de visualisation vous sont fournies afin de vous gagner du temps.
Ces outils sont disponibles à la page suivante sous forme d'un projet C# fonctionnel :

http://www.vgies.com/ressources-pour-le-telemetre-laser-a-balayage-rp-lidar/

Téléchargez dans un premier temps la 1^{ere} partie du projet, intitulée : RpLidar_Base_Project. Ce projet vous fourni une interface pour vous connecter rapidement (après réglage du port COM utilisé) sur le télémètre RP-LIDAR.

$\Longrightarrow$ Analysez du code fourni de manière à comprendre comment marche la bibliothèque d'affichage intégrée à l'application. Vérifiez à l'oscilloscope que les trames sont bien envoyée par le RP-LIDAR vers le PC.

$\Longrightarrow$ A partir de l'analyse de la documentation technique du RP-LIDAR, implantez la machine à état permettant de décoder les trames reçues. Cette étape peut vous prendre du temps, n'hésitez pas à confirmer vos analyses en demandant l'avis du professeur. Cette partie sera considérée comme validée lorsque vous serez capable de placer les données reçues dans deux tableaux contenant pour l'un la liste des angles récupéré et pour l'autre la liste des distances récupérées, et lorsque que vous serez capable de détecter un nouveau tour du télémètre.

Téléchargez à présent la seconde partie du projet intitulée : RpLidar_Grabbing_Project. Cette partie inclut le corrigé de la partie précédente, merci donc de jouer le jeu et de ne pas la télécharger avant d'avoir terminé la partie précédente !

$\Longrightarrow$ Implantez l'affichage en temps réel la carte en 2D de l'environnement du robot dans l'interface en C#.

$\Longrightarrow$ En regardant les données issues du télémètre attentivement, vous constaterez que celles-ci sont incomplètes, certaines mesures manquant pour certains angles. Afin de traiter efficacement ces données, proposez un premier algorithme permettant de générer deux listes, une pour les angles et une pour les distances, de même taille, sans trous et allant de 10° à 179° d'angle.

$\Longrightarrow$ Affichez également ces données sur l'interface graphique et vérifiez quelles se superposent bien aux précédentes pour la plupart.

$\Longrightarrow$ Afin de piloter notre robot efficacement, il est souhaitable de retourner une carte des positions accessibles par le robot au lieu de la carte de l'environnement. Proposez à présent un algorithme permettant de le faire, en prenant en compte le fait que votre robot a un diamètre donné et qu'il ne pourra donc pas passer entre deux obstacles distants de moins de cette valeur.

$\Longrightarrow$ Validez avec le professeur votre travail. Un corrigé est fourni, il est téléchargeable en tant que 3e partie du projet. Il pourra servir de base pour des applications de *SLAM* ultérieures.