Projet
conception d'un robot mobile

Professeur : Valentin Gies

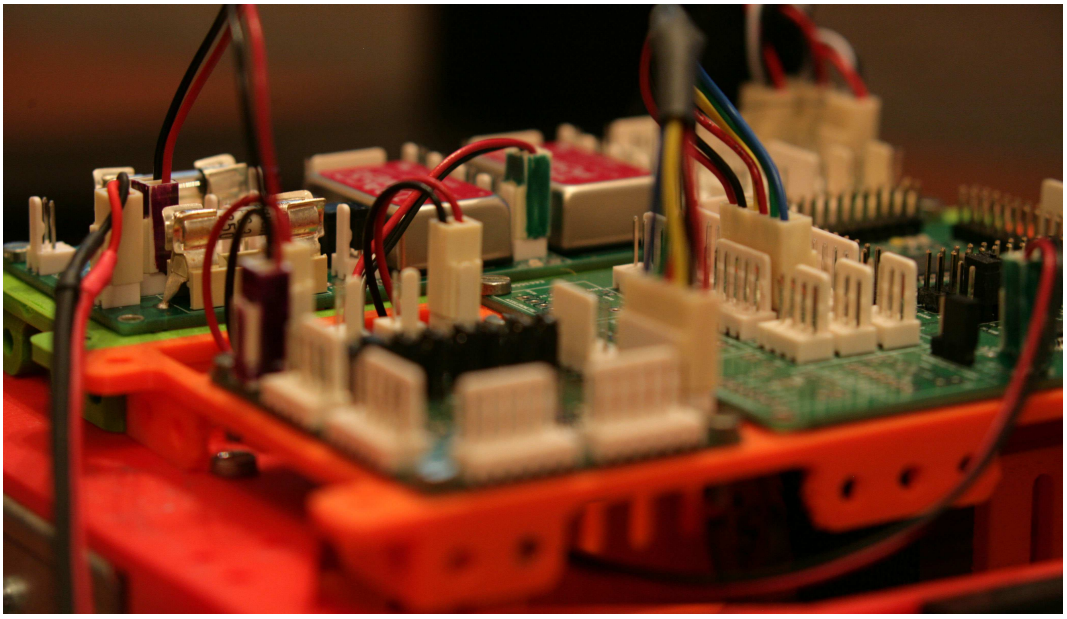


Table des matières

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 $C\#$.

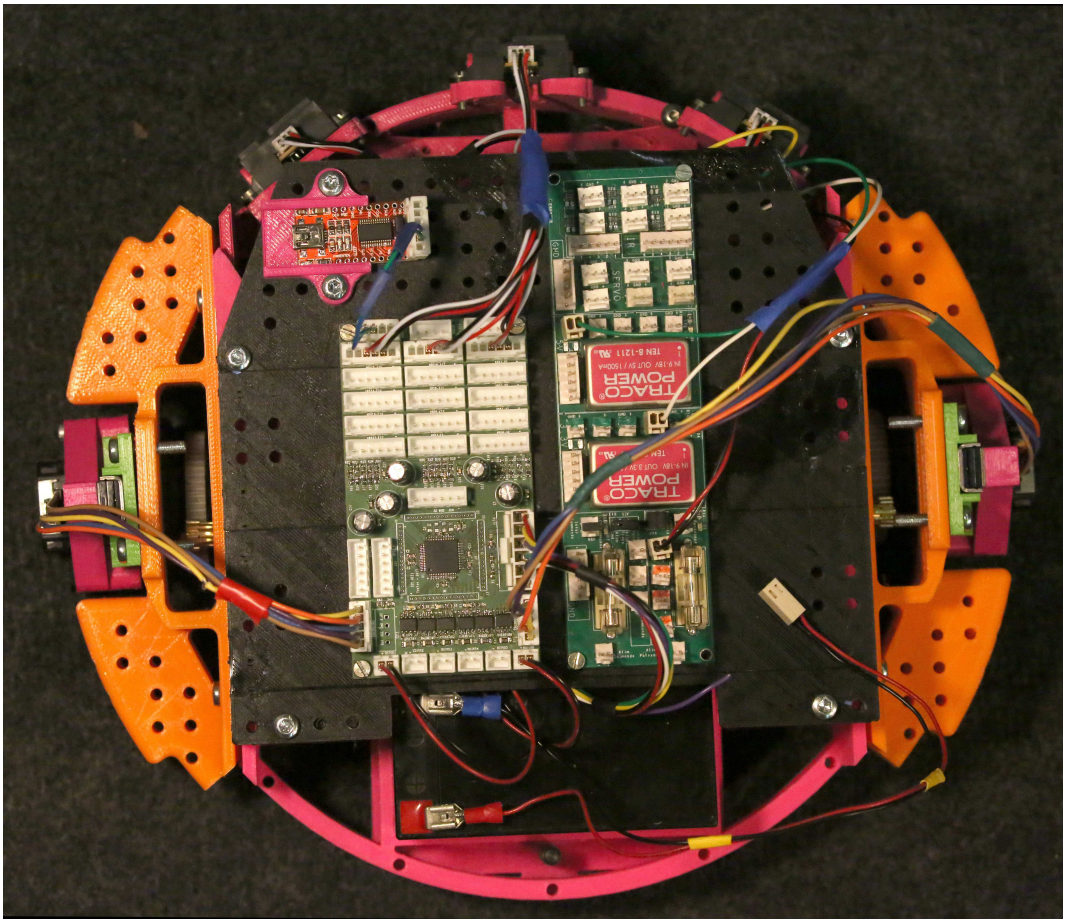


FIGURE 1 – Cartes électroniques du robot

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).

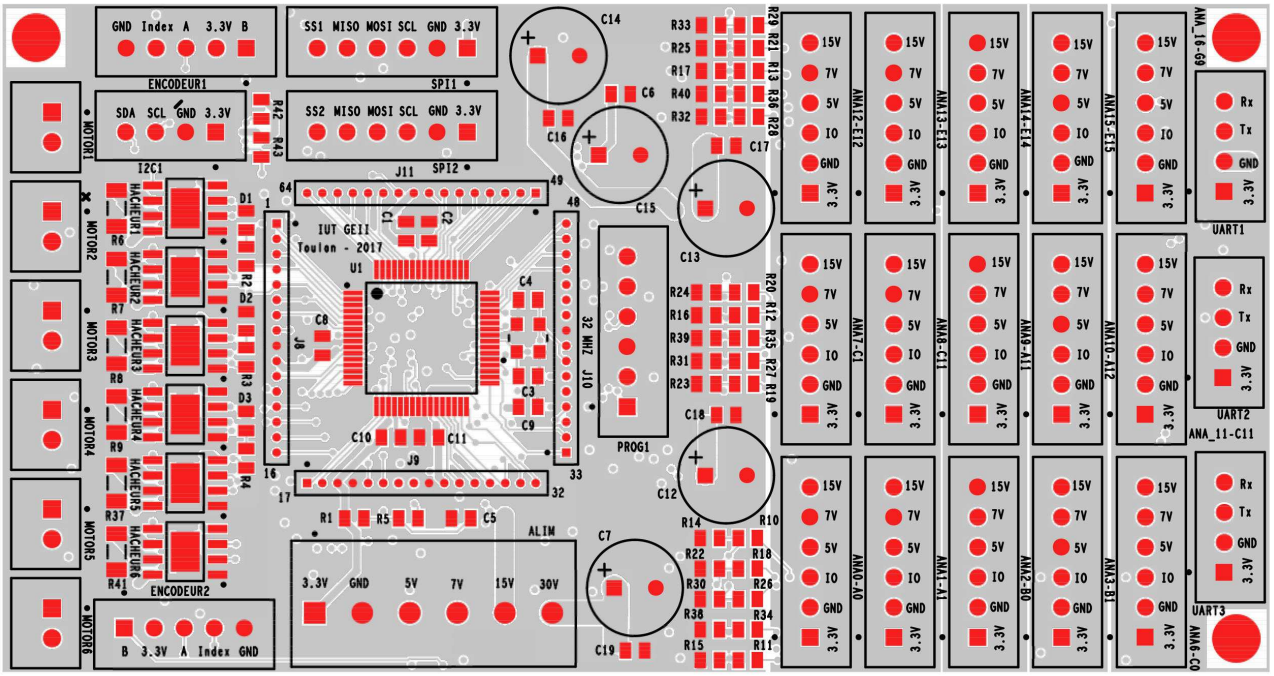


FIGURE 2 – Carte principale

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.

$\Longrightarrow$ 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)**

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

$\Longrightarrow$ 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.

$\Longrightarrow$ 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***.

$\Longrightarrow$ 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.

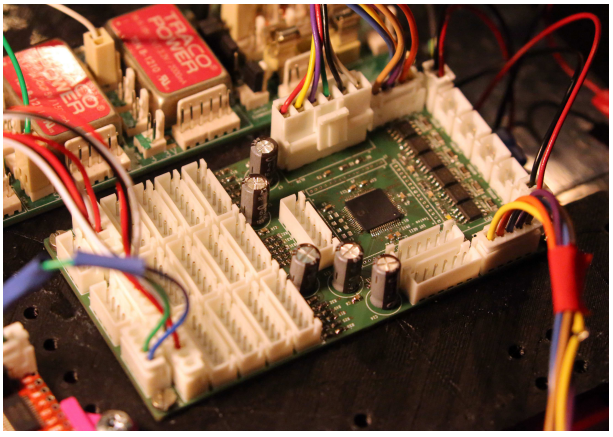


FIGURE 3 – Carte principale

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

$\Longrightarrow$ 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. Vous pouvez récupérer le code à l'adresse suivante :

```
#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.

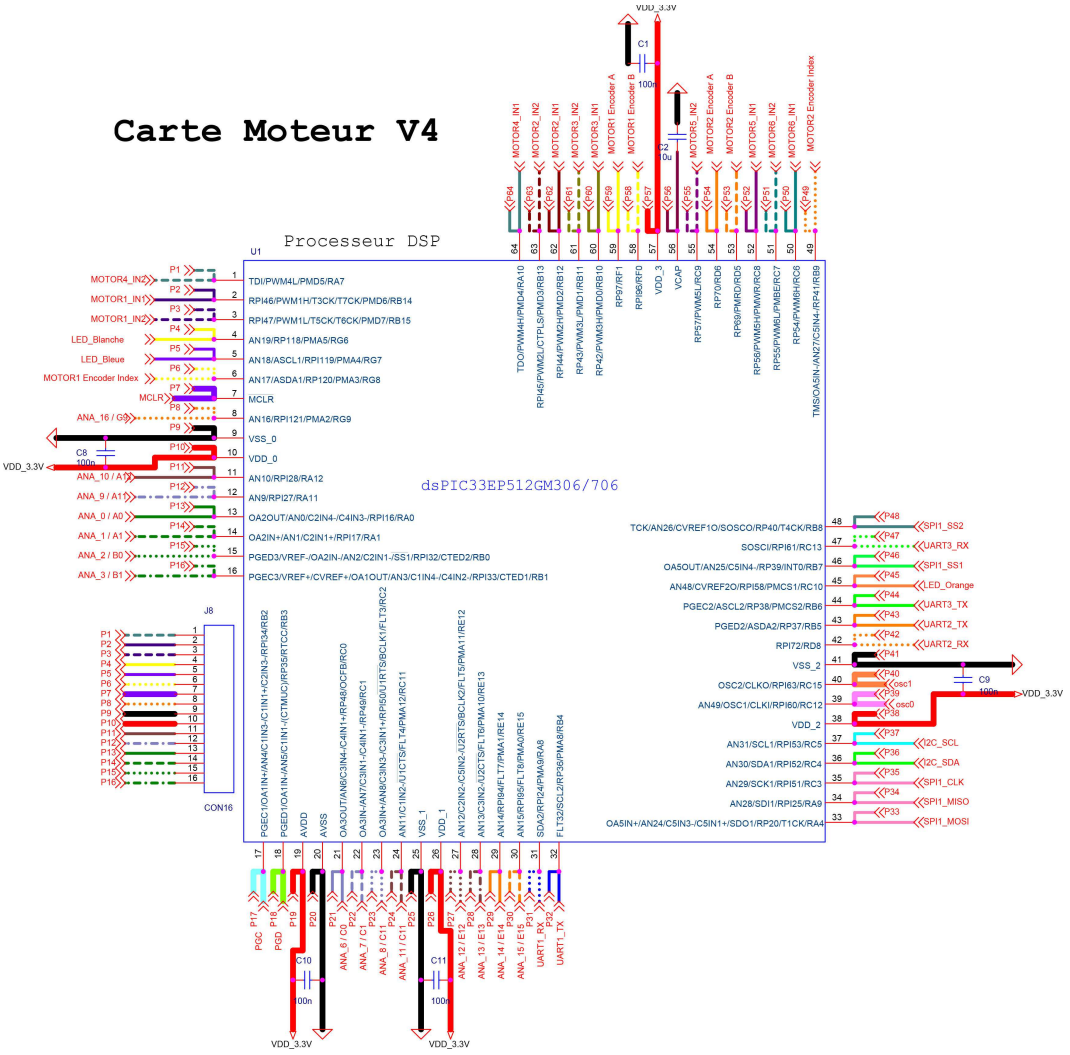


FIGURE 4 – Carte principale : Schéma électronique

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).

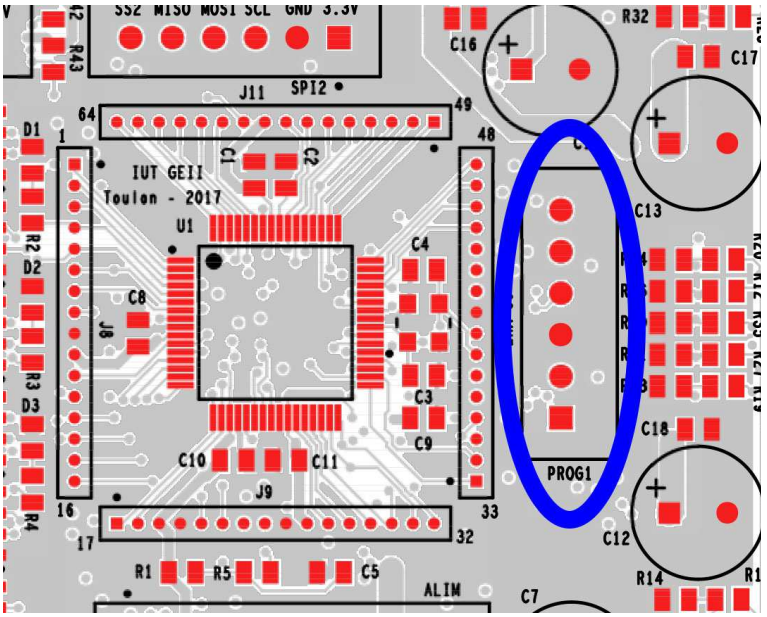


FIGURE 5 – Connecteur de programmation sur la carte principale

⟹ 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

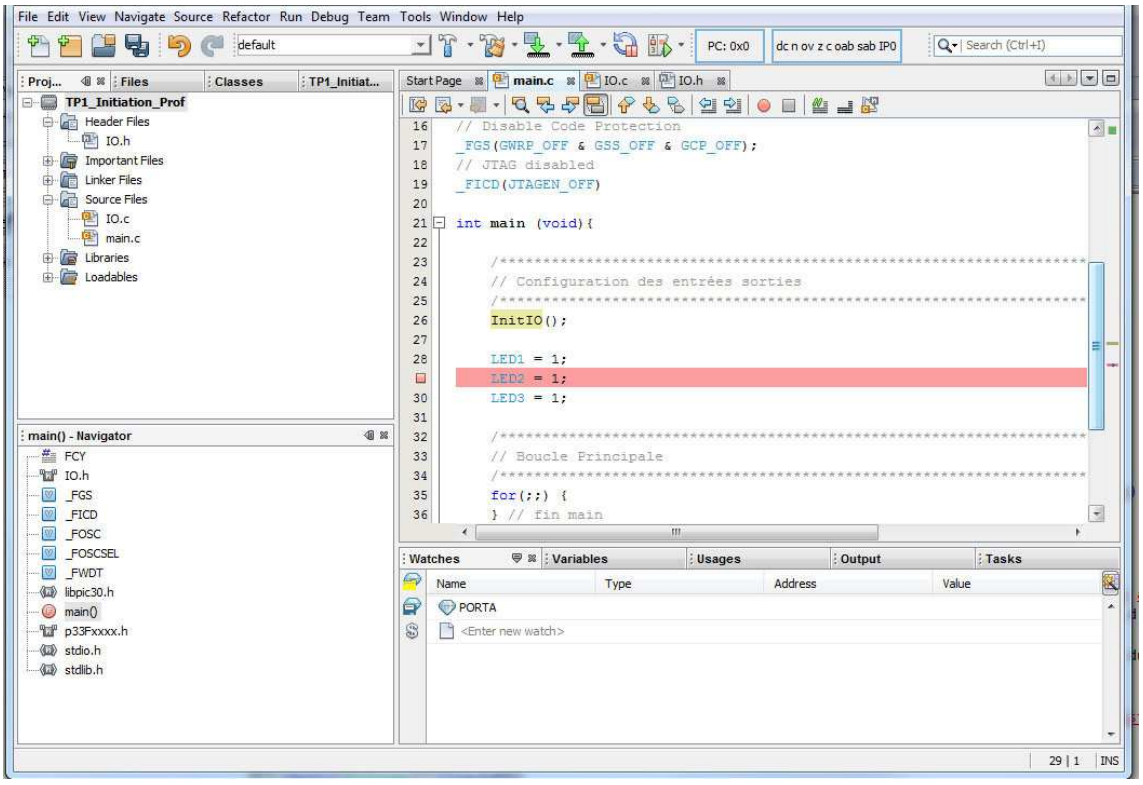
⟹ Il est possible d'insérer des points d'arrêt dans le programme. Leur nombre est limité à *6*.

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**.

$\Longrightarrow$ Créer un fichier source *timer.c* contenant le code suivant. Vous pouvez récupérer le code à l'adresse suivante :

```
#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.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.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;
}
```

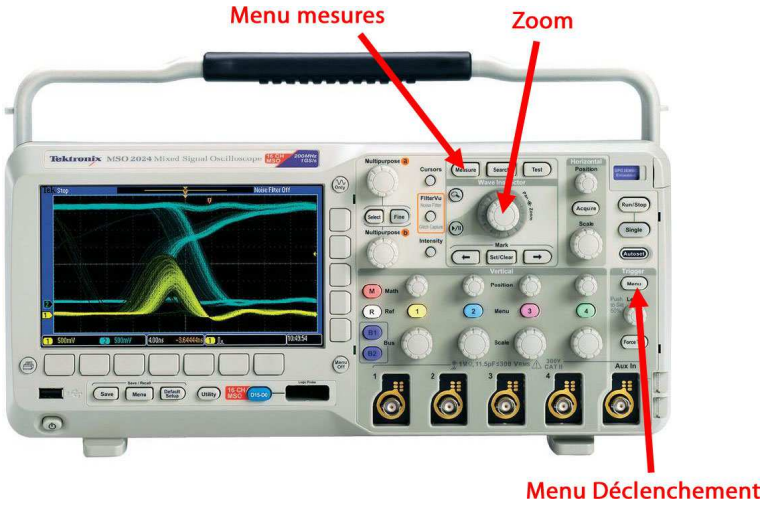
⟹ Créer un fichier *timer.h* contenant le code suivant :

```
#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.

⟹ 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 codes 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*. Vous pouvez récupérer le code à l'adresse suivante.

```
#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. Vous pouvez récupérer le code à l'adresse suivante :

```
#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.vitesseGaucheCommandeCourante = vitesseEnPourcents;
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
MOTEUR_GAUCHE_DUTY_CYCLE = Abs(robotState.vitesseGaucheCommandeCourante*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. Vous pouvez récupérer le code à l'adresse suivante :

```
//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

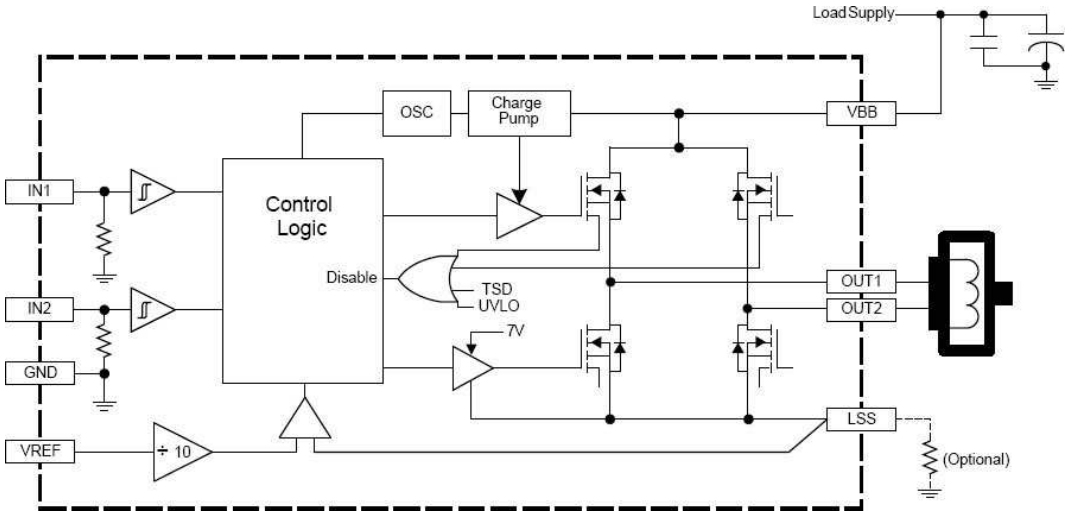


FIGURE 6 – Hacheur *A*4950 : diagramme fonctionnel

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 :

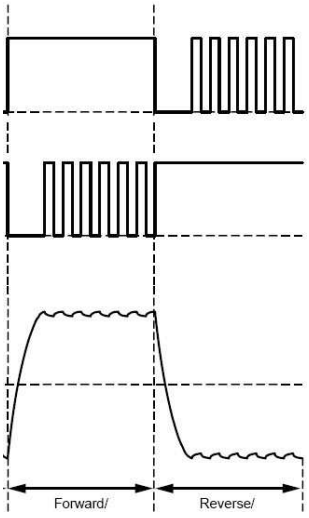


FIGURE 8 – Hacheur $A4950$: Chronogrammes de fonctionnement

$\Longrightarrow$ 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.

$\Longrightarrow$ 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*.

$\Longrightarrow$ Validez le fonctionnement de votre hacheur en présence du professeur.

$\Longrightarrow$ 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
```

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

$\Longrightarrow$ 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);
}
```

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

2.3.3 Commande en rampes de vitesse

$\Longrightarrow$ 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. Vous pouvez récupérer le code à l'adresse suivante.

```
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;
}
```

⟹ Implantez cette fonction en pensant à rajouter les variables nécessaires dans *Robot.h*.

⟹ 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.

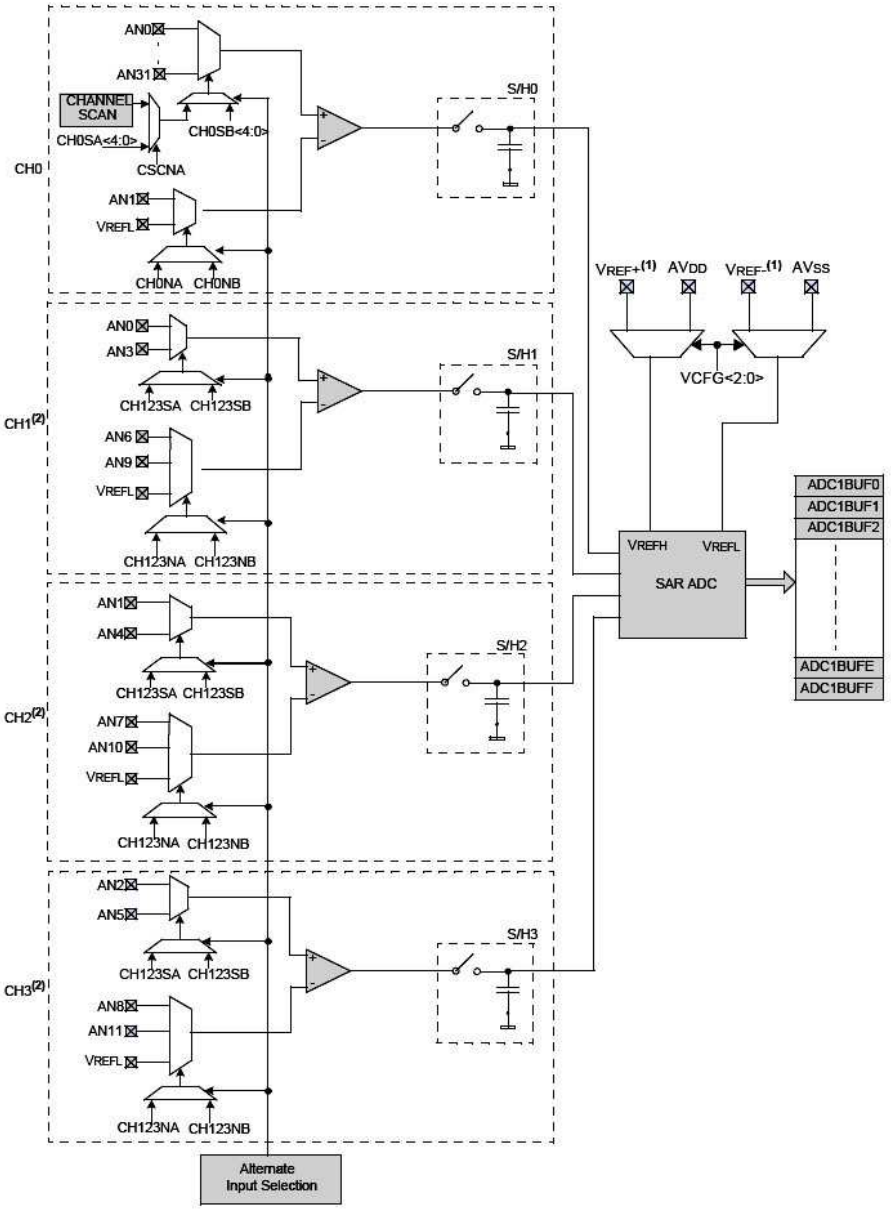
Vous disposez à présent d'une commande de moteur propre vous permettant de faire se déplacer un robot ayant deux roues motorisées séparément.

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. Vous pouvez trouver le code à l'adresse suivante.

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

```
#include "adc.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.

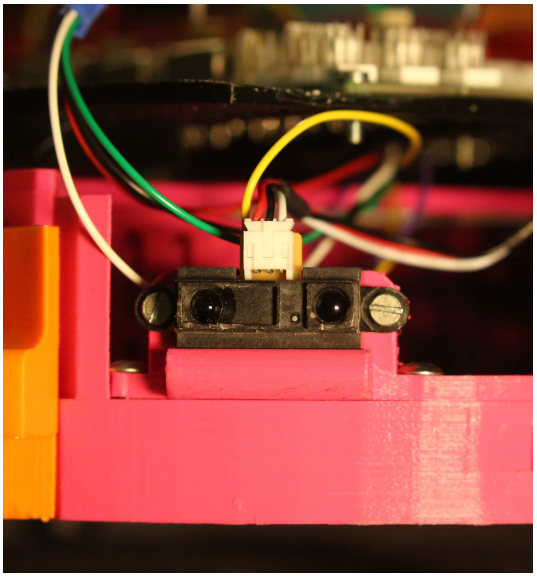


FIGURE 9 – Télémètre infrarouge installés sur le robot

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.

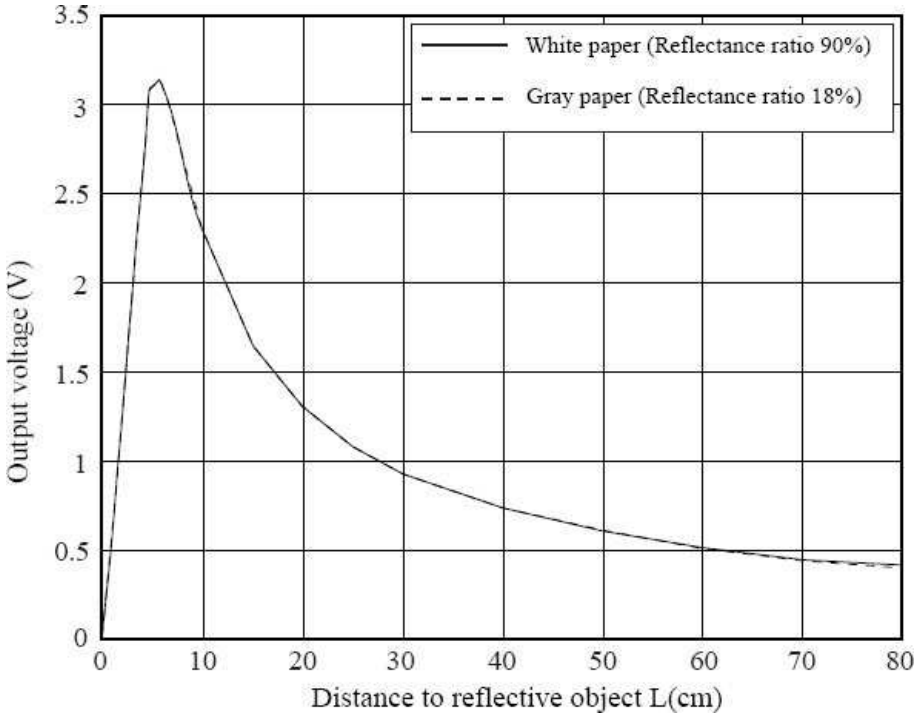


FIGURE 10 – Relation tension-distance des télémètres GP2Y0A21YK0F

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 = 34 / volts - 5;
volts = ((float) result [1])* 3.3 / 4096 * 3.2;
robotState.distanceTelemetreCentre = 34 / volts - 5;
volts = ((float) result [0])* 3.3 / 4096 * 3.2;
robotState.distanceTelemetreGauche = 34 / volts - 5;
}
```

$\Longrightarrow$ 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. Vous pouvez trouver le code à l'adresse suivante.

```
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*. Vous pouvez trouver le code à l'adresse suivante :

```
#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. Vous pouvez trouver le code à l'adresse suivante :

```
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.