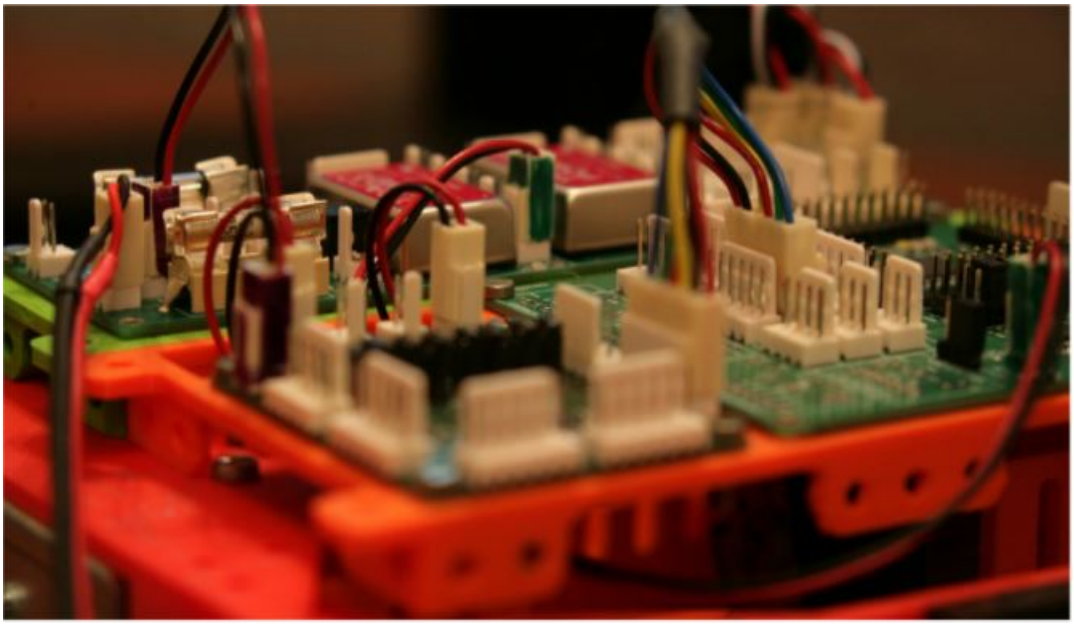
SeaTech
ÉCOLE D'INGÉNIEURS

ELECTRONIQUE



Conception d'un robot mobile autonome

PROVOST MATHIS
ROMENSKY VLADIMIR
Groupe A - Table 6

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

Table des matières

I Introduction **2**

II Environnement de travail et logiciel **3**

1 MPLAB X **3**

2 Premier programme en C **3**

3 Débogueur **5**

III Les micro-contrôleurs **6**

4 Timers **6**

5 Gestion orientée objet du robot en C **6**

6 Pilotage des moteurs **7**
6.1 Le module PWM 7
6.2 Le hacheur 9
6.3 La rampe de vitesse 10

7 Conversion analogique/numérique **11**

8 Capteurs IR et utilisation **12**

IV Programmation du robot mobile et autonome **14**

9 Initialisation du système **14**

10 Réglages des timers **14**

11 Création de la machine à états **15**

12 Optimisation du robot **17**

V Conclusion **19**

Partie I
Introduction

Ce TP se déroule sur toute l'année et se décompose en deux parties. Au travers de la première partie, étudiée lors du premier semestre, qui s'inscrit dans le cadre du cours d'électronique, notre projet est de concevoir un robot mobile capable de se déplacer de façon autonome dans un environnement inconnu de sorte à pouvoir éviter les obstacles qui peuvent se présenter.

Nous allons nous intéresser à la partie physique avec l'étude des processeurs bits de chez Microchip, leur structure et comment interagir avec leur périphériques intégrés tels que les timers, ADC pour interfacer les capteurs externes (télémètres-infrarouge) et PWM pour le contrôle des moteurs via des hacheurs de puissance. Nous allons également nous intéresser à la partie programmation en C afin de pouvoir mettre en action ces périphériques et commander le robot.

Partie II
Environnement de travail et logiciel

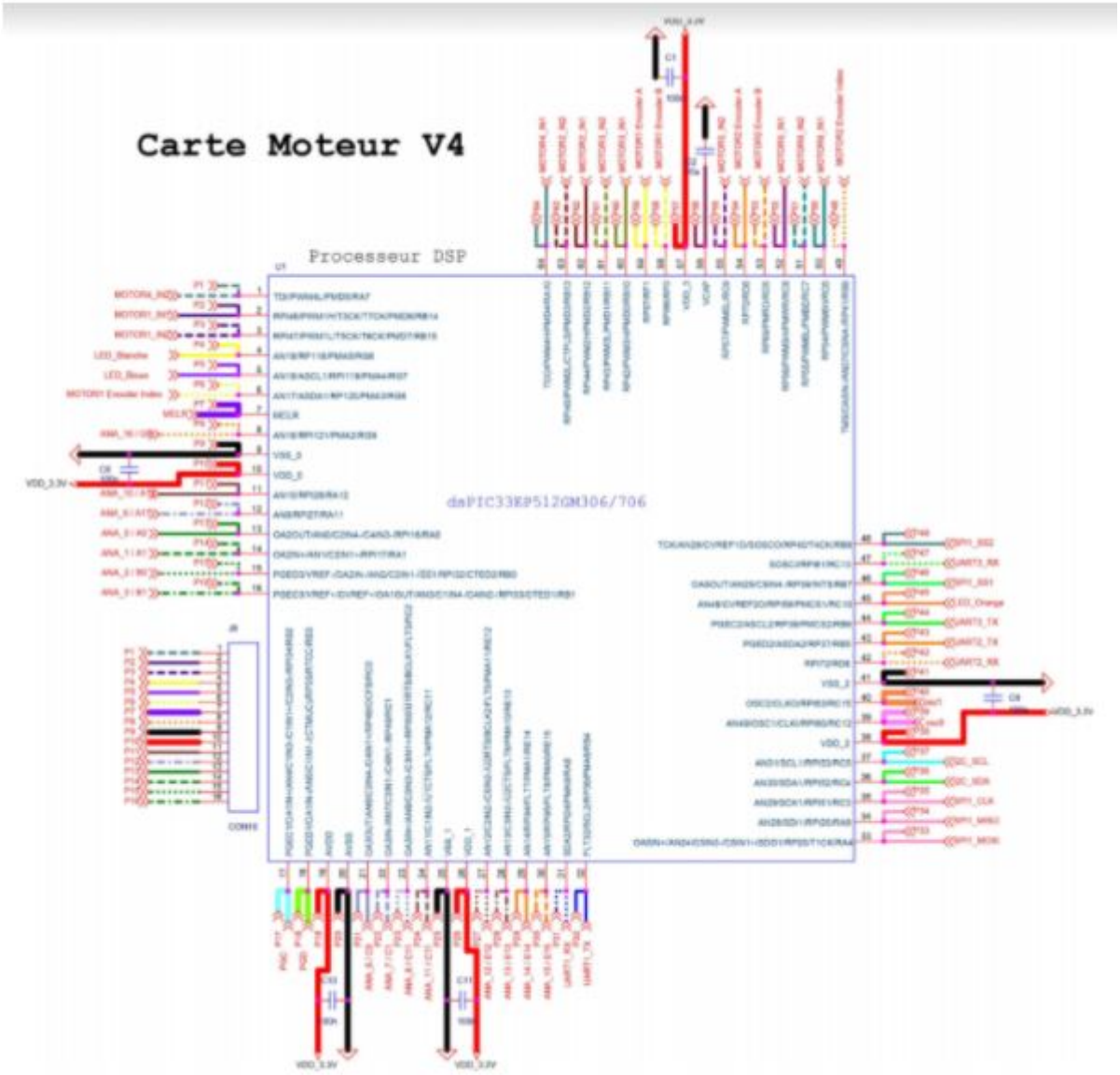
1 MPLAB X

MPLAB X Integrated Development Environment (IDE) est un programme logiciel gratuit extensible hautement configurable qui incorpore des outils puissants pour développer, déboguer et qualifier les conceptions embarquées pour la plupart des microcontrôleurs et contrôleurs de signaux numériques de Microchip. Il permet, entre autres, de convertir du code C/C++ en code machine, ce qui épargne à l'utilisateur cette conversion, qui nécessite une connaissance aiguisée du langage machine.

Plusieurs manipulations sont toutefois nécessaires afin d'envoyer le code sur la carte embarquée. En premier lieu, le branchement sans défaut du boîtier ICD3 qui permet d'effectuer le pont entre les deux entités ainsi que la conversion en code machine en aval de MPLAB X.

2 Premier programme en C

Pour commencer le projet nous branchons la carte du robot à l'ordinateur au travers du boîtier ICD3 pour pouvoir ainsi le programmer. On peut alors créer notre programme avec le main.c. Notre projet est composé de plusieurs scripts, par exemple, la configuration des pins du microcontrôleur se fait grâce au "IO.c" avec le fichier header associé "IO.h". Le microcontrôleur a un nombre de films en entrée/sortie limité. Ainsi un certain nombre de pins peuvent servir d'entrée ou de sortie selon la configuration choisie. Un schéma du circuit permet de savoir comment il a été câblé.



Ainsi dans le script “IO.c” les lignes “TrisC10”, “TrisG6” et “TrisG7” initialisées à 0 permettent de les utiliser en sorties pour faire allumer les LED correspondantes à l’exécution du programme par le microcontrôleur. A l’inverse, si l’on avait fixé ces valeurs à 1, les pins correspondantes auraient été traitées en entrée. Le premier programme nous servira simplement à allumer les 3 LEDs .

```
//********** Configuration des sorties : _TRISxx = 0
// LED
_TRISC10 = 0;  // LED Orange
_TRISG6 = 0; //LED Blanche
_TRISG7 = 0; // LED Bleue
```

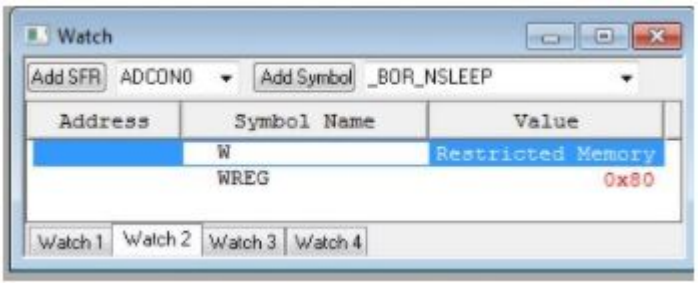
Dans le fichier “main.c” on vient ensuite indiquer au microcontrôleur que l'on souhaite allumer ces 3 LEDs en fixant leurs valeurs à 1 (ou 0 pour les éteindre). On n'oubliera pas d'appeler la fonction “InitIO( )” qui contient le code pour initialiser les entrées et sorties du microcontrôleur sans quoi on ne pourra pas contrôler les LEDS.

```
InitIO();

LED_BLANCHE = 1;
LED_BLEUE = 1;
LED_ORANGE = 1;
```

3 Débogueur

Dans cette partie nous allons apprendre à utiliser le point d'arrêt. Le débogueur nous permet de fixer des points d'arrêts, au nombre maximal de 6. Lorsque le programme les atteint, il se fige et attend qu'on lui donne l'ordre de continuer. De plus, l'outil Watch nous donne accès aux valeurs des variables à chaque fois que le programme se met en pause aux points d'arrêts. Cette technique est très utile pour vérifier le bon fonctionnement d'un programme et pour trouver une erreur dans le fonctionnement de ce dernier.



Partie III
Les micro-contrôleurs

4 Timers

Les timers vont nous être très utiles pour contrôler notre robot. Ils servent à exécuter une action au bout d'un certain temps réglable. Ce temps est la cadence et la durée maximale fixée par les caractéristiques du microcontrôleur. Mais l'utilisation des prescalers va permettre de débarrasser cette limite puisqu'ils vont multiplier la durée maximale du timer. En revanche, ils vont diminuer la précision entre deux durées de timer différentes que l'on peut choisir. Dans le code du timer, nous avons ces deux lignes :

```
PR3 = 0x0262; // Load 32-bit period value (msw)
PR2 = 0x5A00; // Load 32-bit period value (lsw)
```

Nous devrions avoir 40 000 000 coups d'horloge par seconde, car 0x5A00 en hexadécimal est égal à 40 000 000 en décimal. Donc la LED devrait s'allumer et s'éteindre toutes les 4 secondes. En pratique, nous observons à l'oscilloscope une période T=2,0s et une fréquence f= 0.5Hz. Le signal d'alimentation de la LED est alors un signal créneaux. Ce qui confirme la théorie.

En changeant le réglage du prescaler à 0b01 le clignotement est plus rapide et à l'oscilloscope on peut lire f= 500 Hz et T = 2ms. Avec un prescaler à 0b00 pour le timer 1, nous avons f=4 kHz et T = 250µs.

Nous souhaitons obtenir une fréquence de clignotement à 6 kHz. Grâce à la formule on peut obtenir PR1 en fixant PS. Si PS= 1/1, alors PR1 = 6666 (en decimal) = 1A0A (en hexadécimal). En paramétrant le timer avec un PS à 1 :1 , on obtient une fréquence d'interruption de 6 kHz. On observe alors à l'oscilloscope un signal créneaux avec une fréquence de 3kHz ce qui est normal, car la LED a besoin de deux interruptions pour un cycle de clignotement.

5 Gestion orientée objet du robot en C

La suite consiste à piloter le robot en lui-même. Pour ce faire, nous avons eu recours à une structure, qui nous permet de regrouper des objets au sein d'une entité repérée par un seul nom de variable. Cela nous permet également de remédier à l'absence du concept d'objet dans un langage de bas niveau comme le C. Dans notre cas, nous avons créé un fichier Robot.h et un fichier Robot.c et

avons défini une structure dont le nom est ROBOT_STATE_BITS dans le fichier header. Nous y avons joint les variables correspondantes aux vitesses gauche et droite de consigne et de commande courante. Cette structure nous a permis de pouvoir faire appel aux différentes caractéristiques du robot comme sa vitesse depuis n'importe quel autre fichier et de pouvoir ajouter de nouvelles variables si besoin. Pour poursuivre le TP, nous avons également joint un fichier ToolBox qui rassemble toutes les fonctions utiles à notre code telles que Max,Min, conversion degré/radian ou la valeur PI approximée.

```
typedef struct robotStateBITS {

    union {

        struct {
            unsigned char taskEnCours;
            float vitesseGaucheConsigne;
            float vitesseGaucheCommandeCourante;
            float vitesseDroiteConsigne;
            float vitesseDroiteCommandeCourante;
            float acceleration;
            float distanceTelemetreCentre;
            float distanceTelemetreGauche;
            float distanceTelemetreDroit;
            float distanceTelemetreExtremeDroit;
            float distanceTelemetreExtremeGauche;
        };
    };
} ROBOT_STATE_BITS;
```

6 Pilotage des moteurs

6.1 Le module PWM

Pour contrôler les signaux PMW envoyés, on crée des fichiers PMW.h et PMW.c. Dans le fichier PMW.c, on initialise le PMW en réglant les moteurs 1 et 2 respectivement sur le hacheur 1 et le hacheur 6 afin de pouvoir commander en puissance les moteurs en leur délivrant un courant modulable à partir d'un courant maximal.

```
//Réglage 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; //Désactive 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; //Désactive la gestion des faults

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

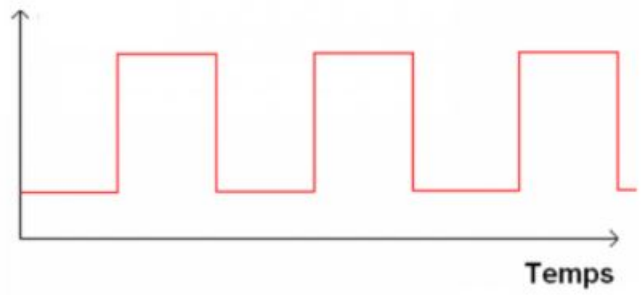
Un hacheur permet de commander en puissance un moteur électrique en délivrant un courant réglable à partir d'un courant maximal. Ainsi il va alimenter avec le courant maximal pendant un temps donné, puis ne plus alimenter. La valeur moyenne sera alors le courant d'alimentation souhaité.

Dans ce même fichier, une deuxième fonction permet de contrôler les pins de la carte de puissance. Ces dernières sélectionnent des canaux moteurs à démarrer, démarrent ou arrêtent les moteurs, etc.

Dans le fichier IO.c responsable de la définition des entrées/sorties, on définit alors les pins B14 et B15 c'est-à-dire les pins associées aux canaux moteurs, comme des sorties (valeur égale à 0).
Avec différents réglages de "PWMSetSpeed" nous pouvons observer à l'oscilloscope le courant fourni par un hacheur pour son moteur.

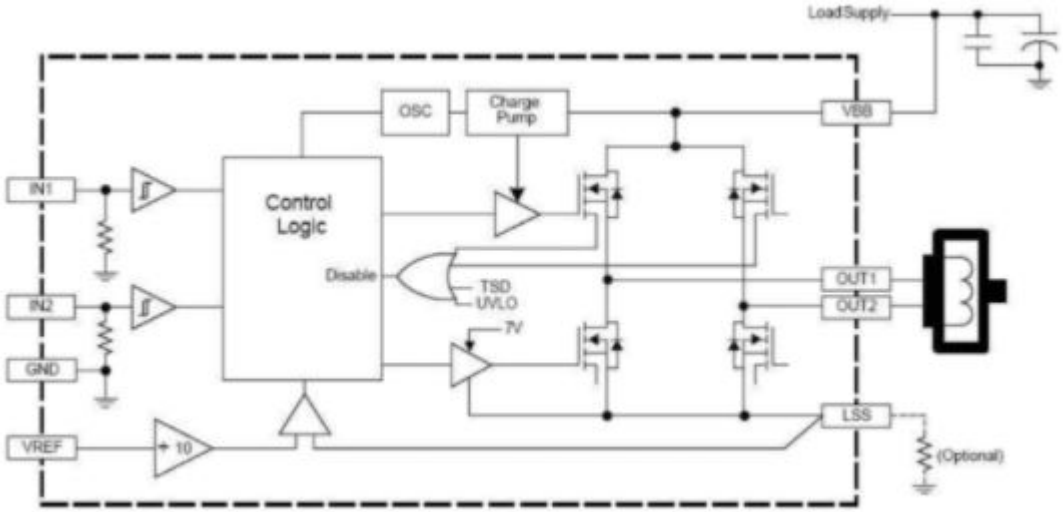
```
void PWMSetSpeed(float vitesseEnPourcents)
{
robotState.vitesseGaucheCommandeCourante = vitesseEnPourcents;
MOTEUR_GAUCHE_L_PWM_ENABLE = 0;  //Pilotage de la pin en mode IO
MOTEUR_GAUCHE_L_IO_OUTPUT = 1;  //Mise à 1 de la pin
MOTEUR_GAUCHE_H_PWM_ENABLE = 1;  //Pilotage de la pin en mode PWM
MOTEUR_GAUCHE_DUTY_CYCLE = Abs(robotState.vitesseGaucheCommandeCourante*PWMPER);
}
```

Nous obtenons des signaux créneaux de différentes périodes pour différents réglages :



6.2 Le hacheur

On s'intéresse à présent aux hacheurs de puissance dont le diagramme fonctionnel est le suivant :



On peut en retenir que la partie gauche correspond aux signaux et alimentations de commande tandis que la partie droite correspond à la partie puissance. Il s'agit d'un hacheur pouvant fonctionner jusqu'à 40V et 3.5A. Nous savons que Couple=Kc*I. Avec Kc la constante du MCC et I l'intensité qui parcourt ce moteur. Cette équation se vérifie quand nous bloquons la rotation de la roue, l'intensité

d'alimentation augmente, car le couple nécessaire pour annuler le blocage augmente lui aussi.

En reproduisant le code pour le deuxième moteur, nous pouvons piloter dorénavant deux moteurs. De plus, on intègre leur pilotage dans les interruptions des timers en utilisant le même principe que l'allumage des LED précédemment. L'alimentation des deux moteurs se fait ainsi en simultanéité et avec interruptions. Pour réaliser cette simultanéité, on définit dans le fichier timer.c une variable toggle qui gèrera l'alternance. Chaque moteur tourne et s'arrête alternativement suivant une période T.

6.3 La rampe de vitesse

Cependant un problème se pose, durant la phase la phase d'accélération du robot, les roues peuvent glisser sur le sol, ce qui n'est pas souhaitable. Pour éviter ce problème, nous allons implémenter une rampe de vitesse.
Nous disposions donc du code nous permettant d'implémenter une rampe de vitesse dans PMW.c qui remplace la fonction PWMSetSpeed. Désormais, en cas de changement de consigne, seule la variable correspondant à la vitesse de consigne change, tandis que la vitesse de commande est progressivement modifiée sur les interruptions du timer 1.
Le programme fonctionne en comparant la vitesse de consigne à la vitesse de commande. Si la première est plus grande que la deuxième, on ajoute à la vitesse de commande une accélération positive, afin de progressivement arriver à la vitesse de consigne. Si la vitesse de consigne est inférieure à la vitesse de commande, alors on décélère progressivement. De plus, étant donné que l'on incrémente toujours l'accélération dont la valeur est fixée à la vitesse, on utilise les fonctions Min et Max pour comparer la vitesse de consigne + ou – l'accélération à la vitesse de commande. Cela nous permet de s'arrêter d'incrémenter au bon moment pour finalement obtenir la vitesse souhaitée. Ce programme agit sur les deux moteurs et dans les deux sens de rotation.

```
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);
```

7 Conversion analogique/numérique

Dans ce projet, nous utiliserons le convertisseur analogique-numérique dans une version simplifiée : les conversions se feront uniquement sur le Channel 0 et séquentiellement. En lisant le code lié aux ADC et le schéma de câblage du microcontrôleur, il faut configurer les pins G9,C11 et C0 en sortie.

```
//Configuration en mode 12 bits mono canal ADC avec conversions successives sur 4 entrées
/************************************************************/
//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-convert)
```

La fonction ADCGetResult permet de récupérer les données transmises et de les stocker. La variable ADCConversionFinishedFlag et les fonctions ADCIsConversionFinished et ADCClearConversionFinishedFlag permettent de savoir quand la conversion est finie, et de pouvoir remettre le flag à zéro, afin de pouvoir lancer une nouvelle conversion et donc, une nouvelle acquisition de données.

```
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;
}
```

Pour tester le bon fonctionnement du convertisseur nous avons relié l'entrée à la sortie de tension continue 3.3V puis à la masse. Ainsi à l'aide d'un breakpoint et de watch nous avons pu observer la valeur convertie.

En étudiant la documentation technique et l'intégration de l'ADC, nous avons trouvé que le signal en tension est diminué par un pont diviseur de tension de coefficient 1/3.2. La valeur maximale convertissable est 3.3V ce qui donne une discrétisation de 4096 valeurs différentes. Connaissant le facteur pris en compte lors de la conversion, nous avons pu lier la tension renvoyée par le télémètre à la distance à l'obstacle détecté.

8 Capteurs IR et utilisation

Le convertisseur analogique numérique est à présent prêt à être utilisé pour la lecture de télémètres infrarouges. Nous utiliserons 5 télémètres branchés au microcontrôleur. L'utilisation de 5 télémètres permet d'avoir une vision de presque 180° sur l'avant du véhicule et permet au robot de prendre des décisions d'évitement plus complexes comme nous le verrons par la suite. Les télémètres sont également raccordés à des ponts diviseurs de tension 3.2V. Ces télémètres sont prévus pour fonctionner à une distance comprise entre 5 et 80 cm ce qui est adapté pour un robot mobile évoluant à vitesse modérée.En dessous de 5 cm, la valeur retournée par le télémètre est incorrecte alors qu'une valeur supérieure à 80 cm renvoie une valeur infinie. Chaque télémètre renvoie tour à tour une tension proportionnelle à la distance entre le capteur et l'objet détecté. Nous avons estimé qu'un objet à 20cm du télémètre retourne la valeur de 500 en décimal.

Ensuite, nous avons exploité cette information en ajoutant du code dans le main.c dans l'objectif de voir une LED s'éteindre dès lors qu'un objet se trouve à moins de 30 cm du robot et s'allumer sinon. Cela nous permet également de vérifier la détection d'objet par les LEDs. Nous avons alors obtenu le code suivant, qui permet de coupler respectivement les LED blanche, bleue et orange aux télémètres droit, centre et gauche.

```
if (robotState.distanceTelemetreDroit > 30) {
    LED_BLANCHE = 1;
} else
    LED_BLANCHE = 0;
if (robotState.distanceTelemetreCentre > 30) {
    LED_BLEUE = 1;
} else
    LED_BLEUE = 0;
if (robotState.distanceTelemetreGauche > 30) {
    LED_ORANGE = 1;
} else
    LED_ORANGE = 0;
```

Partie IV
Programmation du robot mobile et autonome

9 Initialisation du système

Dans le cadre de notre projet de robot autonome, nous étaient fournis : un robot à trois roues, dont deux motorisées ainsi qu'une roue folle. Ce robot dispose aussi d'un microcontrôleur, qui pourra commander les deux moteurs ainsi que d'emmagasiner un programme informatique transmis depuis l'ordinateur à l'aide d'un boîtier. L'objectif de ce projet est de rendre ce robot autonome dans son déplacement, afin qu'il puisse se déplacer sur une surface donnée en évitant les obstacles pouvant se présenter à lui. Nous avons donc procédé en plusieurs étapes, la création d'une machine à état pour le déplacement grossier du robot, la création d'un lien entre la machine à état et les informations renvoyées par les différents capteurs présents sur le robot (codeurs incrémentaux et infrarouges, essentiellement.). Enfin, nous avons procédé à une série de tests afin d'affiner les réglages et les paramètres de la machine à état afin que le robot soit rapide, tout en étant précis dans ses mouvements.

10 Réglages des timers

La fonction "SetFreqTimer1( )" utilise un système de conditions imbriquées permettant de choisir le plus petit prescaler possible, cela afin de paramétrer le timer1 (PS et PR1) à la fréquence désirée nous disposons de fonctions similaires pour régler les autres timers disponibles.

```
void SetFreqTimer1(float freq) {
    T1CONbits.TCKPS = 0b00; //00 = 1:1 prescalervalue
    if (FCY / freq > 65535) {
        T1CONbits.TCKPS = 0b01; // 01 = 1: 8 prescalervalue
        if (FCY / freq / 8 > 65535) {
            T1CONbits.TCKPS = 0b10; // 10 = 1: 6 4 p r e s c a l e r v a l u e
            if (FCY / freq / 64 > 65535) {
                T1CONbits .TCKPS = 0b11; // 11 = 1: 2 5 6 p r e s c a l e r v a l u e
                PR1 = (int) (FCY / freq / 256);
            } else
                PR1 = (int) (FCY / freq / 64);
        } else
            PR1 = (int) (FCY / freq / 8);
    } else
        PR1 = (int) (FCY / freq);
}
```

11 Création de la machine à états

Le fonctionnement d'un système peut être décrit par un GRAFCET implantable sous forme de machine à états. 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. On pourrait illustrer ce système d'état et leurs liaisons par une arborescence avec, dans des bulles, les états et des traits entre eux illustrant les liens possibles. Les deux dernières lignes de du code permettent de ne passer qu'une seule fois par un état avant le prochain, c'est-à-dire que l'exécution des actions à réaliser à cet état ne sont faites qu'une seule fois. Ainsi on évite la répétition de certains états. En téléchargeant le code dans le microcontrôleur et en alimentant le robot avec une batterie externe nous avons pu tester le robot dans un espace fermé. Ainsi le robot détecte les objets devant lui et change de direction de fonction de quel télémètre détecte l'obstacle et à quelle distance. Le robot évite bien les obstacles et ne rentre en collision que lors de cas rares.

```
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(0, MOTEUR_DROIT);    //20
            PWMSetSpeedConsigne(0, MOTEUR_GAUCHE);   //20
            stateRobot = STATE_AVANCE_EN_COURS;
            break;
        case STATE_AVANCE_EN_COURS:
            SetNextRobotStateInAutomaticMode();
            break;
```

12 Optimisation du robot

Lors du premier essai du robot sur une surface de 2x5m, sur laquelle étaient disposés un certain nombre d'obstacle de forme cubique (10x10cm en moyenne), le robot se montrait extrêmement nerveux dans tous ses mouvements, ce pourquoi, beaucoup de collisions étaients à déplorer.

Nous avons donc réglé les timers à une fréquence plus élevée, en effet, en augmentant la fréquence des timers, nous pouvons ainsi augmenter le nombre de conversion analogique-numériques qui s'effectuent par seconde et donc, rendre plus précise la connaissance de l'environnement du point de vue du robot. Le résultat est sans appel, le robot, malgré une grande vitesse, n'entre plus en collision avec les obstacles. Toutefois un nouveau problème survient, le robot ne prend pas le moindre risque.
Nous avions fait le choix de créer de grandes marges de détection, ce qui empêcha, par exemple, le robot de passer entre deux obstacles, dont l'espacement dépassait amplement la largeur du robot. C'est pourquoi nous avons placé le seuil de détection en forme de pointe. C'est à dire que le capteur IR placé à l'avant du robot avait un seuil très élevé tandis que les deux capteurs placés aux extrémités droite et gauche de ce dernier avaient un seuil beaucoup plus faible. Enfin, les capteurs intermédiaires avaient des seuils intermédiaires. Cette dernière modification a permis de rendre le robot plus agile et lui a permis de passer entre des obstacles distant de la taille du robot, à peu de choses près.

```
void SetNextRobotStateInAutomaticMode() {
    unsigned char positionObstacle = PAS_D_OBSTACLE;

    //Détermination de la position des obstacles en fonction des télémètres
    if (robotState.distanceTelemetreDroit < 20 &&
            robotState.distanceTelemetreCentre > 30 &&
            robotState.distanceTelemetreGauche > 30 &&
          robotState.distanceTelemetreExtremeGauche > 30
            ) //Obstacle à droite
            positionObstacle = OBSTACLE_A_DROITE;

 else if (robotState.distanceTelemetreCentre < 30 ||
          (robotState.distanceTelemetreDroit < 15 &&
          robotState.distanceTelemetreGauche < 15) ||
          (robotState.distanceTelemetreExtremeDroit < 10 &&
          robotState.distanceTelemetreExtremeGauche < 10)||
          (robotState.distanceTelemetreDroit < 20 &&
          robotState.distanceTelemetreExtremeGauche < 20 )||
          (robotState.distanceTelemetreExtremeDroit < 20 &&
           robotState.distanceTelemetreGauche < 20))
          //Obstacle en face
          positionObstacle = OBSTACLE_EN_FACE;
```

Partie V
Conclusion

Au travers de cette première partie du TP nous avons abordé et approfondi notre connaissances des microcontrôleurs et du langage C afin de paramétrer le robot pour qu'il soit mobile et autonome.

Nous avons également été amené à réfléchir sur le programme de comportement du robot, à son implémentation, la façon dont le code interagit avec les éléments physiques du robot (capteurs, moteurs) mais également sur une phase de tests et d'optimisation pour le rendre plus intelligent dans sa prise de décision. Le robot connaît néanmoins des limites qui l'empêche de faire face à tous les obstacles, il ne dispose que de 5 capteurs unidirectionnel positionnés sur l'avant du robot avec un angle non négligeable entre chacun d'entre eux, ce qui limite grandement son champ de vision et notamment des petits éléments qui peuvent ne pas être détectés entre deux capteurs. De notre côté, nous aurions pu améliorer la prise de décision du robot en prenant en compte la distance de chaque objet détecté par un capteur ainsi que l'angle de ce dernier, pour pouvoir déterminer plus efficacement, à l'aide de la trigonométrie, si le robot était apte à passer dans des endroits exigus par rapport à ses dimensions.

Atteignant la fin de ce projet, nous nous sommes rendus compte qu'afin de pouvoir aller plus loin dans l'étude du robot autonome, il était nécessaire d'obtenir des informations en temps réel sur l'état du robot. C'est pourquoi nous avons débuté la création d'une interface robot-utilisateur en C#. Ce qui permettrait de contrôler en temps réel le robot et de modifier son comportement *in vivo*, sans avoir à l'éteindre, le brancher à l'ordinateur et téléverser le code mis à jour dessus.