Compte rendu TP1 Robotique Brunner Carrière.

Au cours des séances de TP portant sur le premier TP, l'objectif sera de prendre en main la programmation avec MPLAB-X et avec les différents éléments qui composent le robot. En effet, l'objectif final est de réussir à programmer un robot étant capable de se déplacer seul dans un environnement comportant des obstacles et ayant la capacité à les éviter en totale autonomie pendant plusieurs minutes. Pour ce faire, il faut donc se familiariser avec tous les éléments qui sont utilisés sur le robot et apprendre à les utiliser dans le but de mettre au point un programme permettant d'accomplir la tâche demandée.

I. MPLAB-X :

La première partie du TP se concentre donc sur la prise en main du logiciel MPLAB-X.
Cette première partie commence par expliquer le procédé permettant de créer un projet MPLAB et comment paramétrer ce dernier en fonction du robot sur lequel porte le travail. Il faut alors renseigner le type de microcontrôleur qui est installé sur le robot, dans le cas présent, il s'agit du dsPIC33EP512GM306. Ce microcontrôleur appartient à la famille des dsPIC33 et dispose de 16 bits.
Il faut ensuite renseigner l'outil de programmation utilisé et le compilateur, ici, c'est l'outil ICD 3 et le compilateur XC16 qui seront utilisés.
Afin de faciliter la sauvegarde et l'édition du projet sur plusieurs ordinateurs, l'entièreté du projet est archivée sur GitHub via l'application GitHub Desktop à chaque fin de séance.

Lors de la création d'un projet sur MPLAB-X, il faut créer un fichier intitulé main.c, ce fichier sera le programme principal du projet. En effet, sur MPLAB-X, il est possible de travailler sur plusieurs programmes permettant de faire fonctionner chacun un aspect précis du projet. L'avantage de cette méthode est qu'elle permet d'obtenir des programmes moins chargés et par conséquents plus lisibles. Cela permet donc de maintenir un programme principal avec seulement l'appel des fichiers contenant les fonctions permettant de contrôler le robot.

Dans le cas du projet robot, l'on distinguera deux types de fichiers. Il y a les fichiers en .c, et les fichiers en .h ou Header. Les fichiers en .c se trouvant à côté du main permettent de définir et de réaliser les différentes fonctions, ou alors d'effectuer des tâches ou des initialisations sur les entrées et sorties du robot. Ces fichiers sont liés à des fichiers homonymes en .h. Ce lien entre les programmes et les fichiers Header permet d'inclure les fonctions d'un programme dans un autre. Il faut pour cela inclure le Header associé au programme dans lequel se trouve la fonction que l'on souhaite rajouter au main.

Après avoir compris le fonctionnement du projet sur MPLAB-X, il faut maintenant s'intéresser à un point important de la programmation sur MPLAB-X, le débogueur. Le débogueur permet d'insérer des « points d'arrêts » dans les programmes que l'ont envoi sur le robot. Ces points d'arrêts permettent de mettre le projet en pause lorsque le programme passe sur les lignes de codes où ils sont insérés. Cela permet alors d'observer des processus que l'on ne pourra pas voir lorsque le programme tourne.
Le débogueur permet aussi d'observer l'état ou la valeur de certaines variables pendant l'exécution du programme. Combiner avec les points d'arrêts, il est alors possible de réaliser des tests pour vérifier que le programme fonctionne comme prévu et pour comprendre d'où peuvent potentiellement provenir certaines erreurs.

II. Microcontrôleur et périphériques :

1) Les timers :

La seconde partie du TP, vient s'intéresser aux fonctions internes du microcontrôleur et aux composants qui sont installés sur le robot. Le microcontrôleur dsPIC33EP512GM306 permet de mettre en place des timers.

Afin de pouvoir faire fonctionner ces timers, il faut rajouter un fichier timers.c contenant toutes les fonctions et définitions liées aux timers dans le projet. Il est possible d'initialiser deux types de timers, soit des timers sur 16 bits, pour un nombre maximal de 9, soit des timers sur 32 bits, ce qui donne donc un nombre maximal de 4. Pour obtenir des timers sur 32 bits, il faut mettre deux timers de 16 bits ensembles dans un seul timer de 32 bits. Le fichier timers.c permet de choisir le nombre de timers que l'on initialise et leurs tailles, mais aussi de les paramétrer la fréquence à laquelle ils fonctionnent. Il faut aussi ajouter le fichier timers.h contenant les entêtes des fonctions et permettant d'inclure le programme en .c dans les autres programmes pour qu'ils puissent en utiliser les fonctions.

Les timers sont paramétrés en fonction d'une valeur de période sur 16 bits, et 32 si deux timers sont mis ensembles, et d'une valeur de prescaler. L'incrémentation du timer se base sur la vitesse d'horloge fournie par le microcontrôleur. La valeur de période définie la fréquence à laquelle le timer effectuera ses interruptions, étant programmé sur 16 bits, la période maximale est donc 0xFFFF soit 65 535. En utilisant deux timers ensembles, on obtient alors une période maximale de 0xFFFF FFFF soit 4 294 967 295. Cette seconde option offre donc une période maximale bien plus importante pour les actions qui en ont le besoin. Le second paramètre est le prescaleur, il permet de diviser la fréquence de l'horloge interne au microcontrôleur. Cette division de la fréquence d'horloge offre donc la capacité de rallonger la période du timer en ralentissant son incrémentation. Afin de choisir la valeur du prescaler, il faut un mot binaire sur 2 bits, chaque mot est associé à une valeur pour le prescaler. Ces valeurs sont 1,8,64,256. Il s'agit du nombre par lequel sera divisée la vitesse d'horloge. Donc la période pour un prescaleur de 1 sera 256 fois plus petite que pour un prescaler de 256.

Une fois que les timers sont paramétrés, il faut ensuite pouvoir les utiliser dans le projet. Pour ce faire, il y a une fonction qui vient activer le flag associé au timer une fois que ce dernier a effectué une période. Ce flag d'interruption permet alors de savoir lorsque le timer atteint sa valeur maximale et d'effectuer des actions à ce moment. Il faut aussi penser à rabaisser le flag d'interruption du timer une fois qu'il s'est activé.

Dans le cas du projet robot, il est demandé de changer l'état d'une LED lorsque le flag du timer se lève afin de pouvoir témoigner de son bon fonctionnement.

Si l'on souhaite atteindre une fréquence précise pour un timer codé sur 16 bits et son prescaler, il faut utiliser la fonction suivante :

$$f = \frac{Fcy}{PS \times PR1}$$

PR1 correspond à la période renseignée pour le timer 1, PS à la valeur choisie pour le prescaler et Fcy à la fréquence d'horloge du microcontrôleur, dans notre cas cette dernière vaut 40 MHz.

Pour obtenir une fréquence de clignotement de la LED cible sur le timer 1 de 6 kHz, il faut alors utiliser cette fonction en prenant les paramètres de prescaler valant 1:1, et f = 6 kHz.
Le calcul est alors le suivant :

$$PR1 = \frac{Fcy}{PS \times f}$$

Il faut ensuite doubler cette période si l'on souhaite que la LED clignote bien à 6 kHz, auquel cas, ce sera le changement d'état de la LED qui se fera sur cette fréquence.
En vérifiant le résultat grâce à l'oscilloscope, on trouve une valeur de PR1 = 12500, soit 0x30D4.

Pour les timers de 32 bits, si l'on prend les timers 2 et 3 ensembles, il faut alors utiliser l'équation suivante :

$$f = \frac{Fcy}{PS \times (PR3 \times 2^{16} + PR2)}$$

Cependant, étant donné que les deux périodes n'en définissent qu'une, on peut en réalité utiliser la même équation que pour le timer 1, mais en prenant en compte que tant que le résultat est en dessous de 65 535, on peut se contenter de n'utiliser que le PR2, mais une fois cette valeur atteinte, il faudra aussi utiliser PR3. Car les deux valeurs étant concaténées, les bits de PR3 correspondent aux 16 bits de poids fort du mot.
Donc pour choisir PR2 et PR3, il suffit de calculer la valeur que l'on souhaite obtenir et renseigner les quatre premiers octets dans PR3 et les quatre suivants dans PR2.

2) Gestion des états du robot :

Dans cette partie, on s'intéresse à préparer le pilotage du robot et de ses capteurs.
Pour préparer le descripteur d'état, il faut ajouter un fichier Robot.c et le header Robot.h au projet.

Dans le fichier en .c, on vient définir les différentes commandes qui pourront être envoyées aux moteurs ou reçues depuis ces derniers.

En intégrant ce fichier dans les autres programmes, il sera alors possible de connaitre l'état dans lequel se trouve le robot et les commandes moteurs.

Il faut aussi ajouter un fichier permettant de réaliser des calculs et contenants des définitions mathématiques pour faire fonctionner les composants du robot. On appellera ce fichier Toolbox.c ainsi que son header Toolbox.h.

Ce programme contient plusieurs fonctions mathématiques qui seront utilisées dans les fonctions de commandes des moteurs. Comme une fonction permettant de retourner la valeur absolue d'un terme, une fonction permettant de déterminer si une valeur est plus grande qu'une autre, ainsi que la fonction permettant de vérifier l'inverse. Mais aussi une fonction permettant de s'assurer qu'une valeur ne dépasse pas une valeur maximum ou minimum, une fonction permettant de convertir des radians en degrés et vice versa. Ainsi que la définition de la valeur de π.

3) Moteurs :

Sur le robot utilisé dans le projet, il est possible d'utiliser simultanément six moteurs.
Afin de contrôler les moteurs, il est possible d'utiliser la méthode de la PWM ou, en français, de MLI. Cette méthode permet de choisir la vitesse à laquelle tourne un moteur à courant continu en jouant sur le rapport cyclique d'un signal.
Il faut donc ajouter un fichier PWM.c dans lequel sera effectué toutes les tâches permettant d'initialiser le mode de fonctionnement des moteurs et les broches auxquelles ils sont reliés.
L'ajout du fichier header PWM.h permet également d'inclure le programme dans le reste du projet et d'en utiliser les définitions.

Afin de démarrer la PWM, il faut utiliser la fonction initPWM() dans le programme principal, puis il faut renseigner la valeur que l'on souhaite lui assigner grâce à la fonction PWMSetSPeed(). Il suffit juste de renseigner le rapport cyclique souhaité dans les parenthèses.
À l'aide de l'oscilloscope, il est possible d'observer que le fonctionnement du contrôle moteur est basé sur un principe de différence de tensions. En effet, le moteur utilise une liaison comprenant deux canaux. Pour faire tourner le moteur dans un sens ou dans l'autre, il faut que l'un des deux canaux soit à l'état haut et que l'autre soit en mode PWM. Le canal à l'état permet de définir dans quel sens tourne le moteur, et le canal en mode PWM permet alors de donner l'instruction de vitesse au moteur.

Lorsque l'on observe la valeur du courant absorbé par le moteur sur l'alimentation, on constate que celui-ci augmente lors de phases de démarrage et d'accélération, mais aussi qu'il est plus important si le moteur est en contact avec le sol ou qu'il est retenu. Cela peut s'expliquer par le fait que le moteur ne dispose pas d'inertie lors du démarrage et qu'il doit donc forcer pour commencer à bouger, et de façon générale, le courant augmente lorsque le moteur doit forcer. En effet, on sait que la vitesse d'un moteur à courant continu dépend de la tension et que le couple dépend lui de l'intensité.

La fonction fournie PWMSetSpeed ne permet pas de faire tourner le moteur à l'envers. Il faut donc produire une fonction qui accepte les valeurs négatives et qui comprend comment les interpréter.
La nouvelle fonction regarde la valeur de PWM choisie par l'utilisateur, puis si cette valeur est positive, alors il définit les canaux de sorte que le moteur tourne à l'endroit. Cependant, si la valeur est négative, il définit les canaux à l'inverse et prend la valeur absolue de la PWM avant de l'envoyer au moteur.

Après avoir réussi à paramétrer et comprendre le fonctionnement du moteur droit, il faut alors implémenter le moteur gauche.
Pour ce faire, il faut encore une fois modifier le fonctionnement de la fonction PWMSetSpeed pour qu'elle prenne en compte le moteur dont on choisit de modifier la vitesse.
En utilisant le timer 23 définis auparavant, il est possible de tester le bon fonctionnement de nos deux moteurs et dans les deux sens. On observe alors que lorsque le timer se déclenche, les moteurs changent tous deux de sens et s'adaptent à la vitesse demandée.

Cependant, bien qu'il soit possible de choisir la vitesse et le sens des deux moteurs de façons indépendantes, le fonctionnement actuel des moteurs ne permet pas un contrôle précis dans le cas d'une utilisation en conditions réelles. En effet, si l'on tente contrôler le robot dans l'état actuel, il y aurait des à-coups entre chaque commande de vitesse. Et lors du démarrage, le robot pourrait avoir du mal à partir si l'accélération n'est pas progressive.

Afin de résoudre ce problème, la solution de la rampe de vitesse sera choisie. Cette solution consiste en le fait de faire croitre la valeur de la PWM par paliers dans le but d'éviter les changements trop brutaux.
Pour ce faire, il faut encore une fois modifier la fonction PWMSetSpeed afin qu'elle ne modifie plus directement la valeur envoyée au moteur, mais la valeur de la consigne que l'on veut que le moteur adopte. La valeur qui est-elle envoyée au moteur est comparée à la consigne et s'incrémente ou se décrémente par palier jusqu'à atteindre cette valeur. Par ce procédé, il n'y a plus d'à-coups brusques. Il y a donc l'ajout d'une fonction intermédiaire appelée PWMUpdateSpeed qui, couplée au timers, permet de comparer les valeurs des sorties vers les moteurs aux valeurs des consignes à un intervalle régulier.

Le fonctionnement de la fonction PWMUpdateSpeed est le suivant :

Lorsque le timer actionne son flag, la fonction est appelée.
Puis, elle vient comparer si la valeur courante du moteur est inférieure à la consigne, si c'est le cas, alors elle vient vérifier que la valeur courante auquel on ajoute l'accélération (donc notre palier de vitesse ici définit à 5) ne vient pas dépasser la consigne. Si elle est effectivement plus petite, la commande prend alors la valeur de commande + accélération, sinon, elle prend la valeur de consigne, car cela signifie qu'elle se trouve à moins d'un palier de la vitesse courante.
Si la vitesse courante est supérieure à celle de la consigne, alors elle vient vérifier si la valeur courante – l'accélération est supérieur à la consigne. Si c'est le cas, alors la commande prend la valeur de la vitesse courante – l'accélération, sinon elle prend la valeur de la consigne, car cela signifie qu'elle est à moins d'un palier d'écart.

Donc dans le cas où la consigne est fixée à 47 et que le robot est à l'arrêt, la fonction verra que la valeur courante + l'accélération est inférieure à la consigne neuf fois, puis elle affectera la valeur de la consigne à la commande, car il n'y aura plus que deux d'écart entre la consigne et la commande. Le robot sera donc arrivé à la vitesse ciblée en 10 tours de timer.

On n'appelle plus la fonction PWMSetSpeed, car elle ne correspond plus à la façon dont on veut contrôler les moteurs du robot. Elle est donc remplacée par la fonction PWMSetSpeedConsigne qui vient changer les valeurs de consignes plutôt que les commandes directes.

Si l'on change la valeur de l'accélération, on constate que le robot met plus ou moins de temps avant d'atteindre la vitesse demandée. On constate aussi que si l'on règle l'accélération à des valeurs extrêmes, le robot reprend son fonctionnement brutal ou qu'il ne réussit pas à démarrer immédiatement.

Il est maintenant possible de contrôler les deux moteurs dans le sens que l'on souhaite de manière indépendante et en ayant implémenté un système de rampe de vitesse permettant d'éviter qu'il n'y ait des à-coups lors des changements de consignes envoyés aux moteurs.

4) Convertisseurs analogiques numériques :

Afin de pouvoir utiliser les informations qui seront données pas les capteurs présents sur le robot, il faut d'abord être en capacité de les lires dans le programme principal. Les télémètres infrarouges présents sur le robot envoient des données analogiques donc sous forme de niveaux de tensions que le programme n'est pas directement capable d'interpréter. Pour qu'il puisse interpréter ces données analogiques, il faut d'abord qu'elles soient traitées dans un convertisseur analogique vers

numériques permettant de donner un chiffre en binaire, en décimal ou en hexadécimal correspondant à la tension d'entrée fournit par le capteur.

Cependant, les convertisseurs analogiques numériques présents sur le robot doivent avant tout être initialisés et configurés avant de pouvoir convertir quoique ce soit. Pour simplifier le programme, les convertisseurs ne seront pas utilisés en mode Direct Access Memory, mais en mode séquentiel.

Il faut donc créer un fichier ADC.c où se trouveront toutes les informations relatives aux convertisseurs analogiques numériques, ainsi que son fichier header ADC.h.

Le programme contenu dans ADC.c nous permet de comprendre le fonctionnement des convertisseurs.
Dans la partie configuration du programme, on peut observer les différents paramètres qui sont utilisés pour faire fonctionner le convertisseur.

Premièrement, il faut demander au convertisseur de se désactiver pendant la séquence de configuration. Puis, la commande AD1CON1bits.AD12B permet de choisir sur combien de bits fonctionnera l'ADC, le paramètre 0 correspondants à 10 bits, et le 1 à 12 bits.
Puis, il faut ensuite choisir le type de donnée en sortie, dans le cas présent, il s'agira d'un entier, correspondant au paramètre 0b00.
Il est ensuite indiqué à partir de quel moment doit commencer la conversion et à quel moment elle doit s'arrêter.
La seconde partie de la configuration permet de définir la tension par rapport à laquelle se basera le convertisseur pour traduire le signal analogique en données numériques. Puis, on sélectionne le channel que l'on souhaite convertir et le nombre de conversions à réaliser.
La troisième partie vient définir l'horloge qui doit être utilisée et ses paramètres, et la quatrième vient préciser si le DMA est utilisé.
Puis, il faut configurer les ports sur lesquels seront lues les valeurs à convertir. Ici, il s'agira des ports ANA6, ANA11 et ANA16, soit respectivement les ports C0, C11 et G9.
Enfin, une fois que les ports sont initialisés, il faut vérifier que le flag est bien abaissé, autorisé les interruptions par le convertisseur et finalement démarrer celui-ci.

En intégrant le programme du convertisseur dans celui des timers, il est possible de demander à effectuer des conversions à un certain intervalle.

Pour vérifier que l'ADC est fonctionnel, on peut regarder les valeurs renvoyées pour une tension de 3.3 V et pour une tension de 0 V. Si l'on connecte le 3.3 V avec la broche d'entrée de l'ADC, la valeur trouvée après la conversion est de 2048. Et si l'on branche cette broche avec le GND ou le 0 V, on trouve alors une valeur de 0. Ce fonctionnement est conforme à ce à quoi on pourrait s'attendre pour ce type de convertisseur.

Après avoir vérifié le bon fonctionnement de l'ADC, il est maintenant possible de stocker les valeurs des trois conversions dans trois différentes variables que nous pourrons donc réutiliser dans les autres programmes.

5) Télémètres infrarouges :

Maintenant que le convertisseur permet d'obtenir des données numériques à partir d'une valeur analogique, il est possible de rajouter les télémètres infrarouges au robot et de s'en servir.
Ils seront notamment utiles pour détecter la présence d'un obstacle et sa distance afin de pouvoir modifier les consignes transmises aux moteurs dans le but de corriger son déplacement.

Cependant, les télémètres utilisés sur le robot disposent d'une plage d'utilisation allant de 10 cm à 80 cm. Ce qui veut dire qu'en dessous de 10 cm, le robot devient aveugle et pareil pour les valeurs au-dessus de 80 cm.

Afin d'assurer une compatibilité avec d'autres capteurs, ces derniers sont branchés sur un pont diviseur dont le rapport est de 3.2. Ce qui signifie qu'il faudra donc penser à multiplier par 3.2 au moment où l'on utilisera ces valeurs.

En plaçant des obstacles devant les télémètres, il est possible de trouver les valeurs suivantes :

20 cm correspond à environ 500.
30 cm correspond à approximativement 320.
Et 40 cm correspond à approximativement 170.

Les données obtenues correspondent à peu près à ce qu'indique la courbe fournie par le constructeur, ce qui signifie que les capteurs fonctionnent correctement.

Afin de voir lorsque le robot se trouve à une distance inférieure à 30 cm d'un obstacle sur un de ses trois capteurs, on associe les LED présentent sur carte électronique à un des capteurs.

Maintenant que les télémètres fonctionnent et que nous sommes capables d'utiliser les valeurs qu'ils renvoient, il faut convertir ces valeurs grâce à l'ADC. Pour ce faire, on implémente un extrait du programme contenant les calculs pour la conversion des données.

Le problème du programme et que lorsque la distance devient trop petite, le capteur émet une valeur qui devient trop importante ou alors, il devient tout simplement aveugle, renvoyant donc une valeur correspondant à une distance bien plus élevée.

```
        if (ADCIsConversionFinished()) {
            result = ADCGetResult();
            droite = ((float) result [1])* 3.3 / 4096 * 3.2;
            robotState.distanceTelemetreDroit = 34 / droite - 5;
            if (robotState.distanceTelemetreDroit > 80) {
                robotState.distanceTelemetreDroit = 80;
            }
            centre = ((float) result [2])* 3.3 / 4096 * 3.2;
            robotState.distanceTelemetreCentre = 34 / centre - 5;
            if (robotState.distanceTelemetreCentre > 80) {
                robotState.distanceTelemetreCentre = 80;
            }
            gauche = ((float) result [4])* 3.3 / 4096 * 3.2;
```

```
            robotState.distanceTelemetreGauche = 34 / gauche - 5;
            if (robotState.distanceTelemetreGauche > 80) {
                robotState.distanceTelemetreGauche = 80;
            }
            gauche2 = ((float) result [3])* 3.3 / 4096 * 3.2;
            robotState.distanceTelemetreGauche2 = 34 / gauche2 - 5;
            if (robotState.distanceTelemetreGauche2 > 80) {
                robotState.distanceTelemetreGauche2 = 80;
            }
            droite2 = ((float) result [0])* 3.3 / 4096 * 3.2;
            robotState.distanceTelemetreDroit2 = 34 / droite2 - 5;
            if (robotState.distanceTelemetreDroit2 > 80) {
                robotState.distanceTelemetreDroit2 = 80;
            }
```

Le programme précédent est placé dans la boucle infinie du main, elle est responsable de la conversion des valeurs brutes récupérées par le convertisseur depuis les télémètres en cm. Il y a aussi l'ajout d'une sécurité pour éviter que les valeurs se trouvant en dehors de la plage de fonctionnement des capteurs ne puissent en altérer le fonctionnement. Donc si la valeur convertie dépasse les 80 cm, on indique alors que cette valeur est égale à 80 cm. Cela permet d'éviter un comportement aléatoire du robot, en effet, au-delà des 80 cm la courbe des valeurs n'est plus exploitable pour un comportement précis.

III. Robot autonome :

Maintenant que tous les composants installés sur le robot fonctionnent, il faut produire le code permettant de rendre le robot totalement autonome. Le robot fonctionnera selon le principe d'une machine à états, ce qui signifie que lorsqu'une certaine condition arrive, le robot change d'état et donc change de comportement. Cela permet donc de définir des phases ou le robot doit avancer en ligne droite, ou s'il doit tourner légèrement à droite ou à gauche, ou bien s'il doit tourner plus rapidement ou encore s'il doit faire marche arrière.

1) Réglage automatique des timers :

Si l'on souhaite pouvoir rendre le robot autonome, il faut alors procéder à plusieurs changements dans les différents programmes du projet.
Il faut d'abord commencer par régler les paramètres des timers afin que ces derniers puissent modifier leurs vitesses d'exécution lorsque cela est nécessaire. En effet, la modification effectuée au programme timers.c permet de mettre en place des conditions permettant de choisir PR1 et le prescaler les plus adaptés en fonction de la fréquence demandé est de la fréquence d'horloge du microcontrôleur. Il est donc maintenant possible de donner une fréquence cible que le timer doit atteindre et le programme se chargera de déterminer les paramètres optimums.
Après avoir implémenté la configuration automatique au timer1, il faut désormais faire de même pour le timer4.

2) Horodatage :

Le timer 4 est réglé à 1 kHz, ce qui signifie qu'il intervient 1000 fois sur une seconde, ou alors une fois toutes les millisecondes. Cela permet donc d'établir un système de temps au sein du programme.

Et, il sera donc possible d'établir une « continuité temporelle » entre les différentes actions des programmes afin d'obtenir un fonctionnement plus régulier.
Le temps écoulé depuis le début du programme sera donc contenu dans une variable appelée timestamp qui sera accessible dans tous les programmes du projet.

3) Machine à état

Le projet étant dorénavant capable de se repérer dans le temps, il est possible de passer à la dernière phase du projet de robot autonome. Cette dernière phase consiste en la programmation de la machine à état qui contrôlera le robot.

Pour ce faire, il faut commencer par attribuer une valeur à chacun des états que le robot sera amené à rencontrer. Il faut donc prendre en compte 16 états pour pouvoir effectuer toutes les actions nécessaires au bon fonctionnement du robot.

Les choix des actions à faire sont réalisés en fonction des distances lues par les télémètres, il faut donc indiquer au robot où se trouve le ou les obstacles pour qu'il puisse activer l'état lui permettant de l'éviter.

Pour faire fonctionner la machine à états, il faut alors créer une fonction qui sera appelée à un intervalle régulier permettant d'appliquer les actions prévues pour chacun des états. Ce code est le suivant :

```
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(25, MOTEUR_DROIT);
            PWMSetSpeedConsigne(25, MOTEUR_GAUCHE);
            stateRobot = STATE_AVANCE_EN_COURS;
            break;
        case STATE_AVANCE_EN_COURS:
            SetNextRobotStateInAutomaticMode();
            break;
        case STATE_TOURNE_GAUCHE:
            PWMSetSpeedConsigne(15, 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(15, 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;
        case RALENTISSEMENT :
            PWMSetSpeedConsigne(15, MOTEUR_DROIT);
            PWMSetSpeedConsigne(15, MOTEUR_GAUCHE);
            SetNextRobotStateInAutomaticMode();
            break;
    }
}
```

Cette fonction permet de régler les consignes des moteurs en fonction de la situation dans laquelle se trouve le robot. Il fait aussi appel à une seconde fonction qui permet de faire en sorte que le passe dans le bon état une fois que les changements de consignes aient été appliqués. Il s'agit d'un long switch case se référant à un des états qui a été défini au-dessus.

La seconde fonction appelée dans OperatingSystemLoop permet de choisir les états que doit adopter la machine en fonction des valeurs renvoyées par les télémètres, et en fonction de l'état actuel dans lequel elle se trouve. Le programme est le suivant :

```
void SetNextRobotStateInAutomaticMode(void) {
```

```
    unsigned char positionObstacle = PAS_D_OBSTACLE;

    //D?termination de la position des obstacles en fonction des t?l?m?tres
    if (distg2 > 30 && distg > 30 && distc > 20 && distd > 30 && distd2>30)
//pas d?obstacle
        positionObstacle = PAS_D_OBSTACLE;
    else if (distg2 > 30 && distg > 30 && distc > 20 && distd < 30 && distd2 >
30 ){
        positionObstacle = OBSTACLE_A_DROITE;
    }
    else if (distg2 >20 && distg > 30 && distc > 20 && distd > 30 && distd2 <
20){
        positionObstacle = OBSTACLE_A_DROITE;
    }
    else if (distg2 >30 && distg > 30 && distc > 20 && distd < 20 && distd2 <
30){
        positionObstacle = OBSTACLE_A_DROITE_TOUTE;
    }
    else if (distg2 >20 && distg < 30 && distc > 20 && distd > 30 && distd2
>20){
        positionObstacle = OBSTACLE_A_GAUCHE;
    }
    else if (distg2 < 30 && distg > 30 && distc > 20 && distd > 30 && distd2
>30){
        positionObstacle = OBSTACLE_A_GAUCHE;
    }
    else if (distg2 < 20 && distg < 30 && distc > 20 && distd > 30 && distd2
>20){
        positionObstacle = OBSTACLE_A_GAUCHE_TOUTE;
    }
    else if (distc < 30) //Obstacle en face
        positionObstacle = OBSTACLE_EN_FACE;
    else if (distg2 < 50 || distg <50 || distc < 50 || distd < 50 ||distd2 <
50)
        positionObstacle = RALENTIR;

    //D?termination de l??tat ? 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_A_GAUCHE_TOUTE)
        nextStateRobot = STATE_TOURNE_SUR_PLACE_DROITE;
    else if (positionObstacle == OBSTACLE_A_DROITE_TOUTE)
        nextStateRobot = STATE_TOURNE_SUR_PLACE_GAUCHE;
    else if (positionObstacle == OBSTACLE_EN_FACE)
        nextStateRobot = STATE_TOURNE_SUR_PLACE_GAUCHE;
```

```
    else if (positionObstacle == RALENTIR)
        nextStateRobot = RALENTISSEMENT;

    //Si l?on n?est pas dans la transition de l?étape en cours
    if (nextStateRobot != stateRobot);
    stateRobot = nextStateRobot;
}
```

Cette fonction prend en compte les valeurs des différents télémètres et les comparent avec des valeurs définies pour être capable de situer approximativement les obstacles aux alentours du système et donc d'indiquer où se trouve ce ou ces obstacles.
La seconde partie du programme permet de choisir l'action à réaliser en fonction de l'emplacement de l'obstacle qui a été précédemment défini.

En conclusion, le robot est maintenant capable de se repérer dans le temps et d'effectuer des actions à intervalles réguliers grâce à l'implémentation de timers. Il est aussi capable de se déplacer et de procéder à des changements de vitesses et de directions fluides en fonction des consignes qu'il reçoit. Et enfin, il est capable de repérer et donc d'éviter les obstacles qui se trouveront dans ses environs grâce à des télémètres infrarouges et à des convertisseurs analogiques numériques.
Ce robot est donc capable de se déplacer dans un environnement comportant des obstacles sans entrer en collision avec ces derniers en totale autonomie. Il permet donc de répondre à l'objectif principal du TP.