

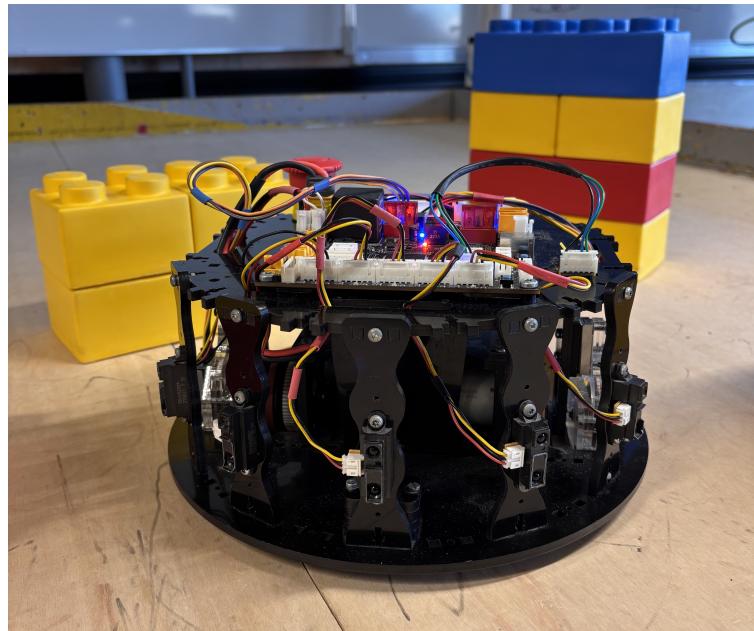
RAPPORT D'ÉLECTRONIQUE



PROGRAMMATION D'UN ROBOT AUTONOME

CLAVERIE Timm
DUBUC Julien

2A Ecole d'Ingénieur
Parcours SYSMER
2025-2026



Résumé

Ce rapport illustre la programmation d'un robot mobile autonome. Notre objectif d'évitement d'obstacles s'est fait sur plusieurs axes : une amélioration de la réactivité via l'optimisation de la conversion analogique, la création d'une machine à état codé en binaire et deux modes correspondant aux types de terrain que le robot peut rencontrer. Ce module nous a permis de nous familiariser avec les microcontrôleurs 16 bits Microchip, les systèmes embarqués et la programmation C.

Mots clés : Autonome, conversion analogique, machine à état, binaire, Microchip, systèmes embarqués, programmation C.

Table des matières

1	Introduction	2
2	Prise en main du robot	3
2.1	Les périphériques classiques	4
2.2	Les LEDs	5
2.3	Timers	5
2.4	Pilotage Moteurs	6
2.4.1	Module PWM et Hacheur	6
2.4.2	Comment expliquer les profils de signaux de PWML et PWMH?	7
2.4.3	Commande en Rampes de Vitesse	7
2.5	Convertisseurs Analogiques-Numériques (ADC)	9
2.6	Télémètres Infrarouges	10
3	Notre Stratégie	13
3.1	Machine à États	13
3.1.1	Logique de Transition	13
3.2	Boutons et Modes de Fonctionnement	14
3.3	L'Horodatage	15
3.4	Optimisation des Convertisseurs Analogiques (ADC)	15
3.5	Télémétries	16
3.6	Stratégie d'évitement par Masque Binaire	16
3.6.1	Le Masque Binaire	17
3.6.2	Hiérarchie des Réactions	17
3.7	Robustesse et Gestion des Blocages	17
3.7.1	Hystérésis de Rotation	17
3.7.2	Déblocage Automatique (Watchdog)	18
4	Conclusion	19
5	Annexe	20

1 Introduction

Ce projet avait pour but de programmer les bases d'un robot autonome. Nous avons travaillé avec une carte électronique (voir figure 2.1) qui utilise un microcontrôleur 16 bits dsPIC33EP512MU814 (dsPIC33EP512GM306 pour la carte) de Microchip (voir figure 2.3).

Nous avons utilisé MPLAB X pour la programmation (voir figure 2.2) et GitHub pour stocker nos codes.

L'objectif était de comprendre comment faire fonctionner les différents périphériques de ce microcontrôleur. Nous avons étudié :

1. Les timers pour gérer le temps et les actions périodiques.
2. Le module PWM pour contrôler la vitesse et le sens de rotation des moteurs à courant continu.
3. Le convertisseur ADC (Analogique-Numérique) pour lire les capteurs, comme les télémètres infrarouges (voir figure 2.7) qui mesurent les distances.

La partie la plus importante du projet était de créer un programme capable de rendre le robot autonome. Pour cela, nous avons utilisé une machine à états qui permet au robot de se déplacer tout seul, d'analyser les informations de ses capteurs de distance, et de changer son comportement pour éviter les obstacles dans son environnement.

Le but du système embarqué est d'éviter les obstacles. Nous avons remanié le code afin d'arriver à cette objectif. Pour cela, nous avons travaillé sur 4 axes d'amélioration :

1. L'optimisation de la réactivité avec les convertisseurs analogiques.
2. L'implémentation d'une machine à état codé en binaire.
3. L'usage des boutons disponibles sur le Microchip afin de pouvoir choisir les configurations de terrain.
4. L'adaptation des seuils de détection.

Ce rapport explique le fonctionnement de notre robot, les détails de notre code et présente la stratégie que nous avons mise en place pour que le robot puisse naviguer et éviter des cubes.

2 Prise en main du robot

Cette partie nous a permis de nous familiariser avec l'environnement de développement et la carte électronique du robot.

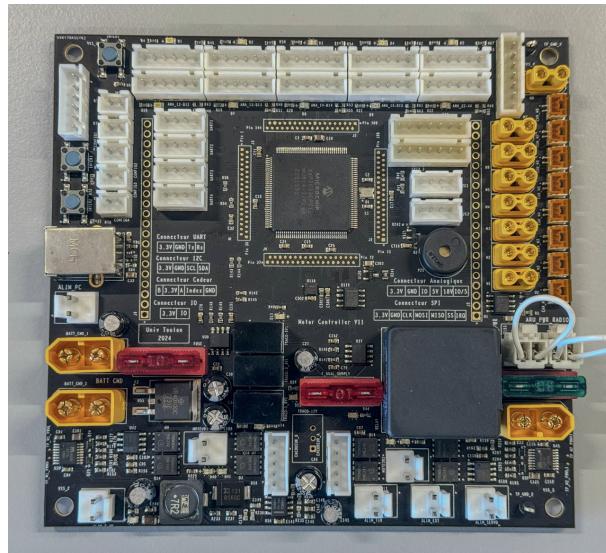


FIGURE 2.1 – Carte électronique du robot

Installation de l'Environnement de Développement

Pour programmer le microcontrôleur dsPIC33EP512MU814 intégré à la carte principale (voir figure 2.1), nous avons installé et configuré un environnement logiciel spécifique :

1. MPLAB X : Logiciel principal pour gérer le projet, écrire le code, le compiler, et envoyer le programme au microcontrôleur.
2. XC16 : Compilateur qui traduit notre code en C en code machine que le dsPIC peut exécuter.
3. GitHub : Stockage de nos programmes.

Une fois l'environnement prêt, nous avons créé un premier projet pour s'assurer que la carte et le processus de compilation/téléversement fonctionnaient correctement.

Organisation du code

Notre code s'organise en différents types de fichiers :

1. Les fichiers d'en-têtes (ex : main.h) qui permettent de déclarer les fonctions et les constantes.
2. Les fichiers sources (ex : main.c) qui contiennent le code concret des fonctions.

Utilisation du Débogueur (Debug)

Le débogueur de MPLAB X s'est avéré essentiel pour comprendre et vérifier l'exécution du code en temps réel. Il permet notamment d'ajouter des points d'arrêts (breakpoints) sur des lignes (voir figure 2.2) pour arrêter le programme à ces endroits et vérifier leurs fonctionnements.

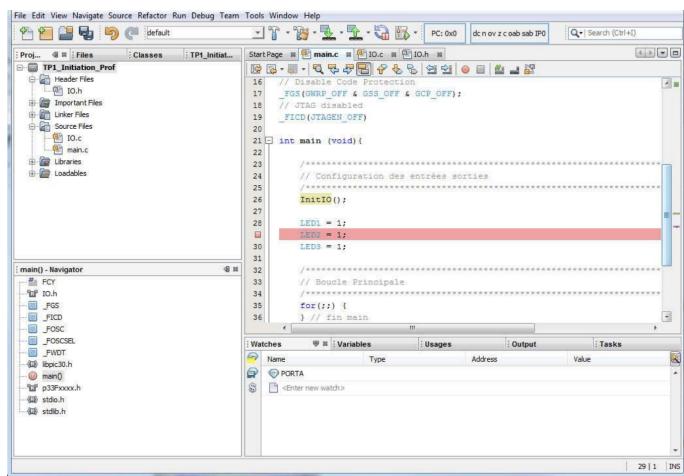


FIGURE 2.2 – Interface de programmation MPLAB X

Il est possible d'ouvrir une fenêtre "Watches" pour observer l'état des variables et des registres du microcontrôleur. Cela est indispensable pour s'assurer que les instructions du code ont bien l'effet attendu sur le matériel.

Exemple

Les méthodes d'ajout des points d'arrêts ont été utile pour vérifier la conversion au niveau de l'ADC, le bon fonctionnement des boutons et l'usage des interruptions de timer.

2.1 Les périphériques classiques

Cette partie détaille comment nous avons utilisé les composants internes du microcontrôleur dsPIC pour donner du temps au robot et contrôler ses mouvements et ses capteurs.

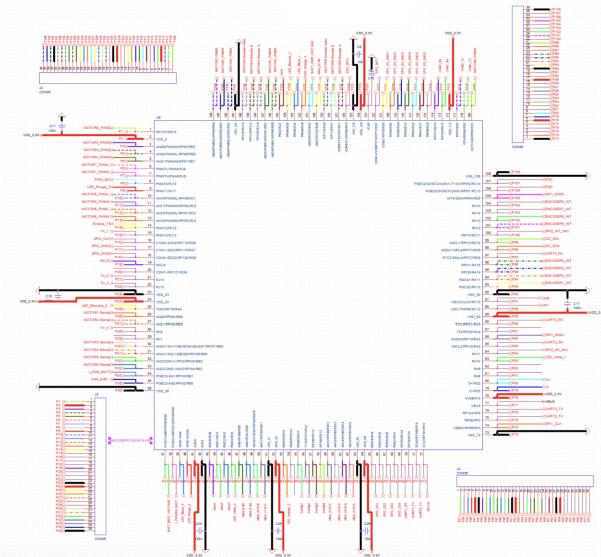


FIGURE 2.3 – Schéma du câblage microcontrôleur de la carte électronique

Ce schéma détaille les Pins en orange et les canaux analogiques en bleus.

2.2 Les LEDs

Cette sous-section fait figure d'exemple sur l'utilisation du schéma de câblage. Afin de pouvoir contrôler l'état des LEDs avant, nous avons modifiés les scripts *IO.c* et *IO.h* qui gèrent les "Input" et "Output" analogiques.

```
1 void InitIO(){
2     _TRISH3 = 0; // LED Verte }
```

Listing 2.1 – "IO.c"

```
1 #define LED_Verte_2 _LATH3
```

Listing 2.2 – "IO.h"

Le code du *IO.c* configure la broche 3 du port H en entrée (1) ou sortie (0). Dans notre cas, les bits de configurations du *TRISH* sont configurés en sortie. Il faut donc définir les **LATH**, dans le *IO.h*, pour contenir les valeur de sortie du port H.

2.3 Timers

Un Timer est un compteur intégré au microcontrôleur qui compte à une fréquence précise. C'est un outil essentiel dans tout système embarqué, car il permet de lancer des actions de manière périodique, comme faire clignoter une LED ou générer un signal de contrôle.

Le réglage d'un timer se fait en modifiant des registres. La fréquence (f) de sortie du timer dépend de la fréquence d'horloge du microcontrôleur (F_{CY}), du Prescaler (PS), et de la période ($PR1$) selon la formule :

$$f = \frac{F_{CY}}{PS * PR1} \quad (2.1)$$

avec $F_{CY} = 60MHz$, $PS = 16, 32, 64, 256, \dots$ bits, $PR1 < 2^{16}$.

En cas d'atteinte de la période $PR1$, le timer déclenche une interruption qui exécute une fonction spécifique du code (comme `_T1Interrupt` ou `_T4Interrupt`), permettant des actions précises dans le temps.

Nous vérifions ensuite nos prédictions aux chronogrammes observés à partir de l'oscilloscope.

Pour un timer en $PS = 32$ bits, il faut 2 prescalers ($PR2$ et $PR3$), la formule de la fréquence devient :

$$f = \frac{F_{CY}}{PS * (PR3 * 2^{16} + PR2)} \quad (2.2)$$

Ainsi pour obtenir une fréquence de 0.5Hz, on vérifie que $\frac{F_{CY}}{f} < 2^{32}$ car pour que le timer fonctionne correctement il faut que $PR2$ et $PR3$ puissent tenir sur du 32 bits. On effectue ensuite la division euclidienne de $\frac{F_{CY}}{f}$ par 2^{16} , le quotient et le reste correspondent respectivement à $PR3$ et $PR2$.

2.4 Pilotage Moteurs

Le mouvement du robot est contrôlé en ajustant la vitesse des deux moteurs à courant continu via des hacheurs de puissance.

2.4.1 Module PWM et Hacheur

La vitesse est réglée par le module PWM (Pulse Width Modulation) du dsPIC. Le PWM génère des signaux carrés dont le rapport cyclique (le temps passé à l'état haut) détermine la tension moyenne appliquée au moteur.

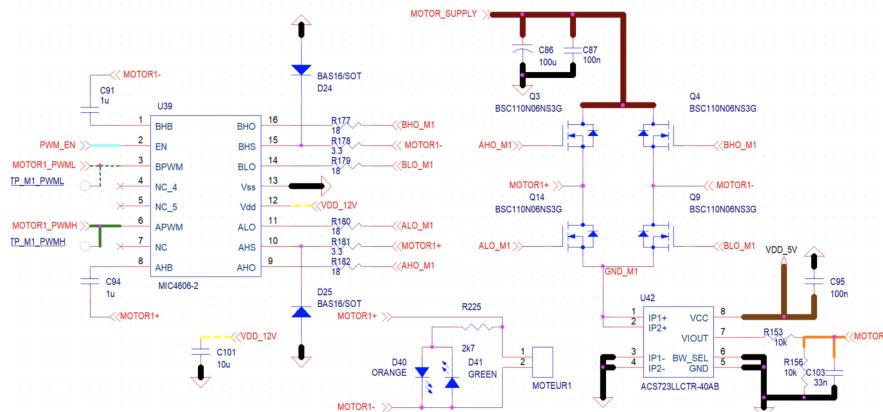


FIGURE 2.4 – Schéma du hacheur 1 du robot

Ces signaux PWM pilotent un Hacheur de puissance (un Pont en H, utilisant le driver MIC4606-2 et des transistors MOS). Ce hacheur agit comme un interrupteur électronique qui permet d'ajuster la tension aux bornes du moteur de -18V à +18V (la tension d'alimentation, MOTORSUPPLY), ce qui permet de faire tourner le moteur dans les deux sens (marche avant ou marche arrière).

Par ailleurs, lors des phases de tests, nous avons remarqué que MOTOR1_PWMH et MOTOR1_PWML sont proportionnels en fréquence. On remarque aussi qu'augmenter le talon diminue la vitesse des moteurs car cela revient à réduire la partie du cycle où le signal du PWM est "haut" (actif). De plus, on observe que faire tourner le moteur plus lentement en le freinant à la main, augmente le courant consommé. Cela s'explique par une diminution de sa force contre-électromotrice (noté E, proportionnel à la vitesse de rotation) : $I = \frac{V_{alim}-E}{R}$.

2.4.2 Comment expliquer les profils de signaux de PWML et PWMH ?

Le moteur est relié au V_{dd} et au V_{ss} par 4 drivers. On cherche toujours à avoir une synchronisation des drivers afin d'obtenir un signal. $PDC1$ et $SDC1$ correspondent respectivement à la voie filaire gauche et droite.

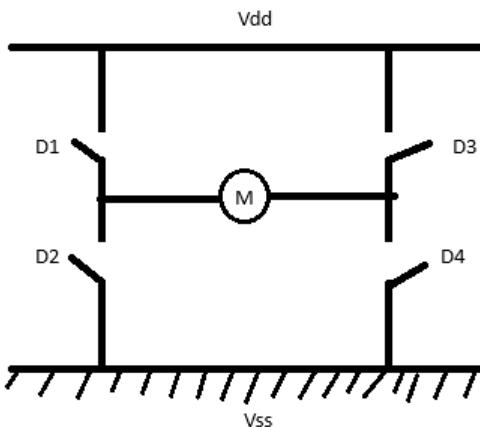


FIGURE 2.5 – Schéma synchronisation des drivers du moteur

Dans le sens de rotation positif, le $D1$ se ferme et les drivers $D2$ et $D4$ alternent périodiquement entre ouverture pour l'un et fermeture pour l'autre. On obtient ainsi un signal PWMH avec des créneaux correspondant l'oscillation entre V_{dd} (période active) et $0V$. Remarque : en modifiant l'accélération sur l'oscilloscope les créneaux de signal ne sont plus instantanément égale à V_{dd} (dépend de la fréquence de l'accélération) mais prennent une valeur progressive. Pour le signal PWML, on a des pics proportionnels aux fréquences d'activation de PWMH.

Dans le sens négatif, V_{dd} sera orienté vers la gauche, ainsi $D2$ sera fermé et, $D3$ et $D4$ alterneront successivement. On obtient donc l'inverse des signaux précédents.

2.4.3 Commande en Rampes de Vitesse

Pour éviter que le robot ne glisse au démarrage ou ne soit brusque lors des changements de direction, nous avons mis en place une commande en rampes de vitesse avec la fonction PWMUpdateSpeed(). Cette fonction est essentielle pour le pilotage des moteurs car elle assure le suivi de

la rampe d'accélération. Elle est appelée périodiquement par une interruption du Timer 1.

Explication du Code PWMUpdateSpeed()

Le code gère séparément les deux moteurs (Gauche et Droit) et les deux cas : accélération (commande inférieure à consigne) et décélération (commande supérieure à consigne).

1. Gestion de l'Accélération

Cette partie est exécutée si le robot doit augmenter sa vitesse (l'accélération).

```

1 if (robotState.vitesseGaucheCommandeCourante < robotState.vitesseGaucheConsigne
     )
2 robotState.vitesseGaucheCommandeCourante = Min(
3 robotState.vitesseGaucheCommandeCourante + acceleration,
4 robotState.vitesseGaucheConsigne);

```

Listing 2.3 – "PWM.c"

Rôle de Min() : Elle choisit la plus petite des deux valeurs entre la nouvelle commande et la vitesse de consigne.

Exemple $0 \rightarrow 37\%$ par incréments de 5 :

1. Initialement, Commande = 0, Consigne = 37.
2. Au 8^e cycle, la Commande est à 35. L'opération donne : $\min(35 + 5, 37) = \min(40, 37) = 37$.
3. Sans Min(), la commande passerait de 35 à 40, dépassant la consigne. Min() garantit que l'on atteint la consigne précisément et sans la dépasser.

2. Gestion de la Décélération

Cette partie est exécutée si le robot doit diminuer sa vitesse (la décélération ou le freinage).

```

1 if (robotState.vitesseGaucheCommandeCourante > robotState.vitesseGaucheConsigne
     )
2 robotState.vitesseGaucheCommandeCourante = Max(
3 robotState.vitesseGaucheCommandeCourante - acceleration,
4 robotState.vitesseGaucheConsigne);

```

Listing 2.4 – "PWM.c"

Rôle de Max() : Elle choisit la plus grande des deux valeurs entre la nouvelle commande et la vitesse de consigne.

Exemple $50\% \rightarrow 42\%$ par incréments de 5 :

1. Initialement, Commande = 50, Consigne = 42.
2. Au cycle où la commande doit passer de 40 à 42, l'opération donne : $\max(40 - 5, 42) = \max(35, 42) = 42$.
3. Max() garantit que la commande s'arrête exactement sur la consigne de 42% sans descendre en dessous.

Visualisation des Rampes

Nous avons visualisé ces rampes en utilisant un oscilloscope (voir figure 2.6) afin de confirmer le fonctionnement de nos commandes en rampes de vitesse.

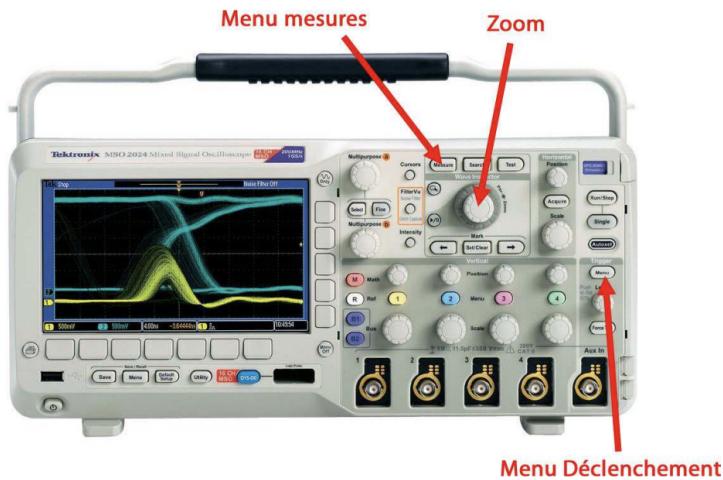


FIGURE 2.6 – Oscilloscope

2.5 Convertisseurs Analogiques-Numériques (ADC)

Les capteurs externes donnent une tension électrique (signal analogique) qui varie. L'ADC (Analog-to-Digital Converter) est utilisé pour transformer cette tension en une valeur numérique que le dsPIC peut traiter.

1. Le dsPIC utilise un ADC à approximation successive (SAR).
2. Nous l'avons configuré en mode 12 bits (`AD1CON1bits.AD12B = 1`) pour avoir une bonne précision (soit $2^{12} = 4096$ possibilités).

Explication : configuration de l'ADC

Le registre AD1CON1 permet la configuration de base de l'ADC, il définit le mode de fonctionnement général.

Le registre AD1CON2 permet la configuration du multiplexeur et du balayage. Son rôle est d'activer le processus de Scan, c'est-à-dire l'ADC va automatiquement convertir plusieurs canaux les uns après les autres au lieu d'un seul. Attention, le démarrage de cette séquence de scan n'est pas automatique mais doit être lancé manuellement par la fonction `ADC1StartConversionSequence()`. Le registre AD1CON3 gère l'horloge. Il compte le temps et a un impact sur la vitesse de conversion.

On s'intéresse maintenant aux ports. Le registre de sélection manuelle AD1CHS0 est ignoré par le processus de Scan. On a alors 3 voies utilisées comme entrées analogiques qui sont activées par le San sur le CH0 :

1. pins RB8 → canal AN8
2. pins RB9 → canal AN9

3. pins RB10 → canal AN10

Valeurs Numériques Correspondantes

Les valeurs numériques attendues en sortie de l'ADC sont les suivantes :

1. Tension maximale : 3.3V, Valeur numérique : 4095
2. Tension minimale : 0V, Valeur numérique : 0

Justification

1. Résolution (**AD12B**) : **AD12B** est mis à 1, configurant l'ADC en mode **12 bits**. $2^{12} = 4096$ niveaux de mesure, allant de 0 à **4095**.
2. Plage de Référence (**VCFG**) : Le registre **AD1CON2** est configuré pour utiliser l'alimentation du microcontrôleur : **AVDD (3.3V)** comme tension de référence positive (V_{REF+}) et **AVSS (0V)** comme tension de référence négative (V_{REF-}).
3. Format (**FORM**) : Le champ **FORM** est configuré à **0b003**, garantissant que le résultat est un **entier non sign**.

Ainsi, la tension minimale (0V) correspond au niveau numérique 0, et la tension maximale (3.3V) correspond au niveau numérique maximum, **4095**.

2.6 Télémètres Infrarouges

Les capteurs choisis pour la détection d'obstacles sont 5 télémètres infrarouges de type GP2Y0A21YK0F (voir figure 2.7).

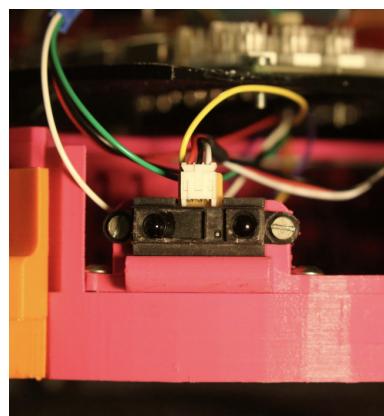


FIGURE 2.7 – Télémètre GP2Y0A21YK0F

1. Ces capteurs sont placés à l'avant du robot et sont efficaces de 10 cm à 80 cm.
2. La tension de sortie du capteur est lue par l'ADC.
3. Nous l'avons converti en distance réelle (cm) en utilisant une formule dérivée de la courbe caractéristique du capteur (voir figure 2.8).

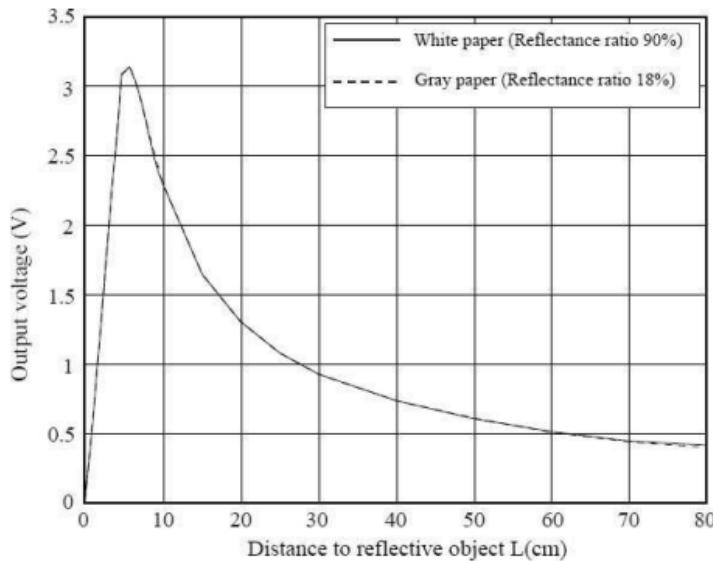


FIGURE 2.8 – Relation tension-distance des télémètres GP2Y0A21YK0F

La conversion implique d'abord de retrouver la tension V lue, puis d'appliquer la relation suivante :

$$V = \frac{\text{Valeur } ADC * 3.3}{4096} \quad (2.3)$$

$$\text{Distance}(L) = \frac{34}{V} - 5 \quad (2.4)$$

Voici l'équivalent des formules (2.3 et 2.4) dans notre code :

```

1 float volts;
2 volts = ((float) result[0])*3.3/4096; robotState.distanceTelemetreGaucheGauche
   = 34/volts - 5;
3 volts = ((float) result[1])*3.3/4096; robotState.distanceTelemetreGauche = 34/
   volts - 5;
4 volts = ((float) result[2])*3.3/4096; robotState.distanceTelemetreCentre = 34/
   volts - 5;
5 volts = ((float) result[3])*3.3/4096; robotState.distanceTelemetreDroit = 34/
   volts - 5;
6 volts = ((float) result[4])*3.3/4096; robotState.distanceTelemetreDroiteDroite
   = 34/volts - 5;
```

Listing 2.5 – "main.c"

Cette étape nous a permis de disposer de cinq variables d'état (GaucheGauche, Gauche, Centre, Droite, DroiteDroite) contenant la distance (cm), prêtes à être utilisées pour la navigation.

Limitation pour les Très Faibles Distances de Détection

Le code et la formule sont limités pour les très faibles distances de détection (en dessous de 10 cm) en raison de la forme de la courbe caractéristique (voir figure 2.8).

1. Le capteur peut produire la même tension pour deux distances différentes : dans la zone de pente positive (3 cm) et dans la zone de pente négative (30 cm).

2. Une tension de 1.5V pourrait correspondre à 16 cm ou à 4 cm.

Comme la formule de conversion (2.4) est basée sur la partie utile (pente négative, 10 cm à 80 cm) où la distance et la tension sont inversement proportionnelles, elle ne peut pas distinguer les deux distances.

Pour le robot, toute distance en dessous de 10 cm sera mal interprétée comme une distance beaucoup plus grande. Le robot pourrait donc entrer en collision avec des objets très proches sans les "voir" correctement (Les tests ont confirmé ce problème).

3 Notre Stratégie

3.1 Machine à États

L'autonomie du robot est gérée par une Machine à États Finis, implantée dans la fonction `OperatingSystemLoop()`. Chaque comportement est décomposé en deux états distincts :

1. **État d'Action (Pair)** : C'est l'état d'initialisation (ex : `STATE_AVANCE`). On y définit les consignes de vitesse (`PWMSetSpeedConsigne`) et on réinitialise le `timestamp`. Le robot passe immédiatement à l'état suivant (Transition) sans attendre.
2. **État de Transition (Impair)** : C'est l'état d'exécution (ex : `STATE_AVANCE_EN_COURS`). Le robot maintient l'action et appelle la fonction d'intelligence (`SetNextRobotState`) pour surveiller les capteurs et décider de la suite.

```

1 // Principaux
2 #define STATE_ATTENTE 0
3 #define STATE_ATTENTE_EN_COURS 1
4 #define STATE_AVANCE 2
5 #define STATE_AVANCE_EN_COURS 3
6 #define STATE_AVANCE_PEU 4
7 #define STATE_AVANCE_PEU_COURS 5
8 // Rotation gauche
9 #define STATE_TOURNE_LEGER_GAUCHE 6
10 #define STATE_TOURNE_LEGER_GAUCHE_EN_COURS 7
11 #define STATE_TOURNE_GAUCHE 8
12 #define STATE_TOURNE_GAUCHE_EN_COURS 9
13 // Rotation droite
14 ...
15 // Rotation sur place
16 #define STATE_TOURNE_SUR_PLACE_GAUCHE 14
17 #define STATE_TOURNE_SUR_PLACE_GAUCHE_EN_COURS 15
18 #define STATE_TOURNE_SUR_PLACE_DROITE 16
19 #define STATE_TOURNE_SUR_PLACE_DROITE_EN_COURS 17
20 // Arret ou recul
21 #define STATE_ARRET 18
22 #define STATE_ARRET_EN_COURS 19
23 #define STATE_RECULE 20
24 #define STATE_RECULE_EN_COURS 21
25 #define STATE_DEMI_TOUR 22
26 #define STATE_DEMI_TOUR_EN_COURS 23

```

Listing 3.1 – "Validation de la transition (main.c)"

Cette structure permet de ne configurer les registres moteurs qu'une seule fois au début de l'action, puis de simplement surveiller l'environnement.

3.1.1 Logique de Transition

Le changement d'état n'est pas systématique à chaque cycle d'horloge. Il est contrôlé par une condition de stabilité située à la fin de la fonction de décision :

```

1 // 5. Application de l'état (Gestion des transitions)
2 if (nextStateRobot != stateRobot && nextStateRobot != (stateRobot - 1)) {
3     stateRobot = nextStateRobot;
4 }

```

Listing 3.2 – "Validation de la transition (main.c)"

Si la condition est **VRAIE** (une nouvelle action différente est requise), on change la variable `stateRobot`. Sinon, le robot reste dans son état "En Cours" et continue son mouvement actuel sans réinitialiser ses timers.

3.2 Boutons et Modes de Fonctionnement

L'interaction homme-machine est assurée par deux boutons poussoirs (`RH1` et `RH2`) configurés en entrées numériques. Le robot initialise d'abord ses périphériques et entre dans une boucle d'attente (`STATE_ATTENTE`) où les moteurs sont désactivés. Le démarrage est déclenché par l'appui sur l'un des boutons, lançant un timer de 60 secondes.

Nous avons programmé deux stratégies distinctes selon le bouton pressé, permettant d'adapter le comportement du robot à son environnement sans reprogrammation :

- **Mode "Vitesse" (Bouton RH1)** : Adapté aux espaces ouverts. La vitesse est élevée (35%) et les distances de détection sont longues (40 cm) pour anticiper les obstacles tôt.
- **Mode "Précision" (Bouton RH2)** : Adapté aux labyrinthes. La vitesse est réduite (30%) et les seuils de détection sont raccourcis (30 cm, 20 cm) pour éviter de détecter les murs latéraux des couloirs étroits comme des obstacles frontaux.

```

1 extern unsigned long t1;
2 const unsigned long T_60_SECONDS_TICKS = 60000;
3 unsigned long start_time_ticks = 0;
4
5 while(!robot_is_running){
6     // MODE 1 : SPRINT (RH1)
7     if (_RH1 == 1) {
8         start_time_ticks = t1;
9         robot_is_running = 1;
10        EN_PWM = 1;
11        stateRobot = STATE_AVANCE;
12        vitesse_avance=35; // Vitesse rapide
13        pfn_SetNextRobotState = &SetNextRobotStateInAutomaticMode;
14        // Détection lointaine
15        DIST_OBSTACLE_DETECTE=40.0;
16        DIST_OBSTACLE_DETECTE1=30.0;
17    }
18    // MODE 2 : LABYRINTHE (RH2)
19    else if (_RH2 == 1) {
20        // ... (Initialisation identique)
21        vitesse_avance = 30; // Vitesse réduite
22        // Détection courte pour couloirs
23        DIST_OBSTACLE_DETECTE=30.0;
24        DIST_OBSTACLE_DETECTE1=20.0;
25    }

```

26 }

Listing 3.3 – "Sélection du mode dans main.c"

3.3 L'Horodatage

L'horodatage est essentiel pour que le robot puisse gérer le temps et les délais de manière précise. Il permet de connaître le temps courant écoulé depuis le démarrage. Il est réalisé grâce au Timer 4 qui s'incrémente toutes les millisecondes.

```

1 void __attribute__((interrupt, no_auto_psv)) _T4Interrupt(void) {
2     IFS1bits.T4IF = 0;
3     timestamp+=1;
4     t1+=1;
5     ADC1StartConversionSequence(); // Lancement synchronisé des mesures
6 }
```

Listing 3.4 – "Interruption Timer 4 (timer.c)"

La variable `timestamp` est réinitialisée à 0 au début de chaque action (virage, recul) pour mesurer sa durée, tandis que `t1` compte le temps global pour l'arrêt au bout de 60 secondes.

3.4 Optimisation des Convertisseurs Analogiques (ADC)

Dans la sous-section [2.1.4], on a expliqué comment fonctionne les convertisseurs analogiques-numériques. Pour optimiser la réactivité du robot, nous avons ajusté les paramètres temporels de conversion.

On cherche à réduire le temps de conversion T_{AD} donné par la formule :

$$TAD = TCY \times (ADCS + 1)$$

, où TCY correspond à la vitesse du processeur. Celui-ci est de $70MIPS$ (5), ce qui signifie que son cycle d'instruction le plus rapide est de $\frac{1}{70} \approx 14.28ns$. De plus, la vitesse minimale de conversion est aussi fournie, elle est de $117.6ns$ (5). Ainsi :

$$\begin{aligned} T_P \times (ADCS + 1) &\geq T_{AD(min)} \\ 14.28ns \times (ADCS + 1) &\geq 117.6ns \\ ADCS &\geq 7.235 \end{aligned}$$

On fixe donc la ligne correspondante :

```
1 AD1CON3bits.ADCS = 8;
```

Listing 3.5 – "ADC.c"

Il faut aussi réduire le temps d'échantillonnage correspondant au temps pendant lequel l'ADC "capture" la tension d'entrée pour charger son condensateur. On a $T_{samp} == SAMC \times T_{AD}$. En remplaçant la valeur de $SAMC = 15$ par :

```
1 AD1CON3bits.SAMC = 2;
```

Listing 3.6 – "ADC.c"

La durée du temps de "capture" de la tension est presque divisé par 7.

Cependant la valeur minimale de SAMC dépend de l'impédance de la source analogique. Plus celle-ci est faible et plus la charge du condensateur sera rapide, et inversement si elle est forte. Ici, nous n'arrivons pas à trouver/mesurer l'impédance du condensateur, nous avons quand même pris le risque de baisser *SAMC* quitte à faussée nos valeurs d'entrées.

3.5 Télémétries

Optimisation de la réactivité

Initialement, on utilisait des flottants (*float*) pour les distances. Cependant, nous avons observé que cela consommait trop de temps processeur risquant de retarder la boucle principale.

Pour corriger cela, nous avons intégré à *_AD1Interrupt* de l'ADC.C le calcul de distance des télémètres. Nous avons utilisé la même formule : $42200/ADC - 5$. Cela a réduit le temps de calcul, rendant le robot plus réactif aux changements rapides d'environnement.

```
1 ADCResult[0] = val_gauche_gauche;
2 ADCResult[1] = val_gauche;
3 ADCResult[2] = val_centre;
4 ADCResult[3] = val_droit;
5 ADCResult[4] = val_droit_droit;
6 // Formule : Dist = 42200 / ADC - 5
7 if(val_centre > 100) robotState.distanceTelemetreCentre = (42200 /
val_centre) - 5;
8 else robotState.distanceTelemetreCentre = 80.0; // Trop loin
9
10 if(val_gauche > 100) robotState.distanceTelemetreGauche = (42200 /
val_gauche) - 5;
11 else robotState.distanceTelemetreGauche = 80.0;
12 ...
```

Listing 3.7 – "AD1Interrupt (ADC.c)"

Pour permettre les calculs directement dans l'ADC.C, on a inclus les librairies *Robot.h*, *main.h* et *PWM.h*.

3.6 Stratégie d'évitement par Masque Binaire

L'autonomie du robot est gérée par une Machine à États Finis (FSM) exécutée périodiquement dans *OperatingSystemLoop()*. Contrairement à une logique séquentielle simple, nous utilisons une approche par *fusion de capteurs*.

3.6.1 Le Masque Binaire

La fonction `SetNextRobotStateInAutomaticMode` lit les 5 télémètres et construit un "Masque Binaire" (un octet). Chaque bit correspond à une zone :

- `MASK_TC` (0x04 : 00100) : Centre
- `MASK_TG` (01000) / `MASK_TD` (00010) : Gauche / Droite
- `MASK_TGG` / `MASK_TDD` : Extrêmes Gauche / Droite

```

1  unsigned char obstacleMask = 0;
2  if (d_TGG < DIST_OBSTACLE_DETECTE2) obstacleMask |= MASK_TGG;
3  if (d_TG  < DIST_OBSTACLE_DETECTE1) obstacleMask |= MASK_TG;
4  if (d_TC  < DIST_OBSTACLE_DETECTE)   obstacleMask |= MASK_TC;
5  // ... idem pour Droite

```

Listing 3.8 – "Construction du masque (main.c)"

Cela permet de gérer les 32 combinaisons d'obstacles possibles via un `switch case`. Par exemple, le cas 0b11100 (Obstacle à Gauche + Centre) déclenche une fuite vers la Droite.

Retour Visuel (LEDs) : Pour faciliter le débogage, nous avons associé les LEDs de la carte aux capteurs détectés. Ainsi, si le capteur central détecte un obstacle, la `LED_ORANGE_1` centrale s'allume instantanément, permettant de visualiser le masque en temps réel.

3.6.2 Hiérarchie des Réactions

La machine à état suit une logique de priorité stricte :

Arrêt d'Urgence (Priorité 1) : Si un capteur détecte un objet à moins de la distance critique (`DIST_CRITIQUE`, 33cm), le robot ignore le masque et effectue une rotation sur place (`STATE_TOURNE_SUR_PLACE`) pour éviter la collision imminente.

Évitement Standard (Priorité 2) : Si la voie est libre en critique, le masque décide :

- *Obstacle Latéral* : Virage sur une roue (`STATE_TOURNE_GAUCHE/DROITE`) pour corriger la trajectoire en douceur.
- *Cul-de-sac ou obstacles multiples* : Rotation sur place.
- *Voie Libre* : Le robot avance (`STATE_AVANCE`).

3.7 Robustesse et Gestion des Blocages

Pour éviter que le robot ne reste coincé (par exemple dans un coin), nous avons implémenté des sécurités dans la boucle principale `OperatingSystemLoop`.

3.7.1 Hystérésis de Rotation

Lors d'un virage, l'arrêt de la rotation ne se fait pas aveuglément. Nous utilisons la fonction dédiée `IsPathClear()` qui vérifie si les capteurs Centre, Gauche et Droit indiquent une voie libre (distance supérieure à `DIST_VOIE_LIBRE`).

Si `IsPathClear()` est validé, le robot ne s'arrête pas immédiatement : il continue de tourner pendant une durée supplémentaire (`MARGE_SECURITE_ROTATION`) pour s'assurer que l'arrière du robot a bien dégagé l'obstacle.

3.7.2 Déblocage Automatique (Watchdog)

Si le robot reste dans un état de rotation pendant plus de 2 secondes (`TIMEOUT_BLOCAGE`), on considère qu'il est bloqué physiquement. Une séquence de dégagement est alors forcée :

1. **Recul** : Le robot passe en `STATE_RECULE` pendant 15 ticks pour s'éloigner du mur.
2. **Demi-Tour** : Il enchaîne avec un `STATE_DEMI_TOURN` pour faire face à une nouvelle direction.
3. **Reprise** : Le mode automatique reprend la main.

```
1 case STATE_TOURNE_GAUCHE_EN_COURS:
2     // Securite Timeout : Si on tourne depuis trop longtemps
3     if (timestamp > TIMEOUT_BLOCAGE) {
4         stateRobot = STATE_RECULE; // On recule
5         return;
6     }
7     // ... Logique de fin de virage
```

Listing 3.9 – "Gestion du timeout (main.c)"

Limites persistantes

Malgré cette adaptation, l'absence de cartographie (mémoire) reste une limite. Si le robot entre dans un cul-de-sac complexe (forme en U), il peut mettre du temps à en sortir via la séquence de "Déblocage Automatique" (Recul/Demi-tour), là où un robot cartographie ne s'y engagerait pas une seconde fois.

Perspectives

Pour fonctionner efficacement dans un environnement de type labyrinthe, notre robot aurait besoin d'une stratégie plus avancée, intégrant : Un algorithme de suivi de mur (maintenir une distance constante avec le mur latéral). Une forme de mémoire (ne pas revenir immédiatement sur une position où il vient d'être bloqué).

4 Conclusion

L'objectif du projet est de permettre au robot d'éviter les obstacles en nous familiarisant avec les enjeux de l'embarqué. Nous avons pu élaborer une stratégie pour exploiter les périphériques du Microchip et les optimiser. Nous avons aussi appris 2 nouvelles méthodologies très importantes : la gestion d'une machine à états codée en Bits et l'utilisation de GitHub pour gérer un projet.

Acquis Techniques

Nous avons réussi à maîtriser :

1. L'implantation des **rampes de vitesse** via le **PWM** pour éviter les glissements brutaux et assurant la stabilité mécanique.
2. Les **Timers** qui assurent la gestion de l'**Horodatage** pour temporiser nos actions et maintenir la fréquence de la **Machine tats**.
3. La machine à états est notre solution pour l'évitement. Elle est réactive pour éviter efficacement les obstacles dans un espace ouvert et fermé.

Les Limites

Deux problèmes persistent et définissent les limites de notre système :

1. Les **télémètres IR** lus par l'**ADC** sont imprécis pour les distances **inférieures à 10 cm**. Le robot peut confondre un objet très proche avec un objet lointain. C'est une faille critique.
2. Notre robot manque de réactivité. Bien qu'il soit bon en "champ" et en "labyrinthe", lancé à pleine vitesse, notre robot peine à détecté rapidement les obstacles pour freiner.
3. Nous avons aussi remarqué que le robot avait du mal à avancer en ligne droite, il serait intéressant d'introduire un système d'asservissement (PID par exemple) pour corriger les différences entre les moteurs.

En conclusion, nous disposons d'une base solide. Pour améliorer le robot, le prochain défi serait d'intégrer un capteur d'orientation, d'asservissement des moteurs et d'ajouter une logique de mémoire pour pouvoir explorer les environnements confinés.

5 Annexes

Code

Cliquez [ici](#) pour voir le code du robot.

Vitesse du processeur

Sur la page 1 du datasheet, on voit que la vitesse du processeur est de 70 MIPS

Operating Conditions

- 3.0V to 3.6V, -40°C to +125°C, DC to 60 MIPS
- 3.0V to 3.6V, -40°C to +85°C, DC to 70 MIPS

Vitesse maximale de conversion

Sur la page 560 du datasheet, on voit que la vitesse maximale de conversion est de 117.6ns.

Param.	Symbol	Characteristic	Min.	Typ. ⁽²⁾	Max.	Units
Clock Parameters						
AD50	TAD	ADC Clock Period	117.6	—	—	ns