

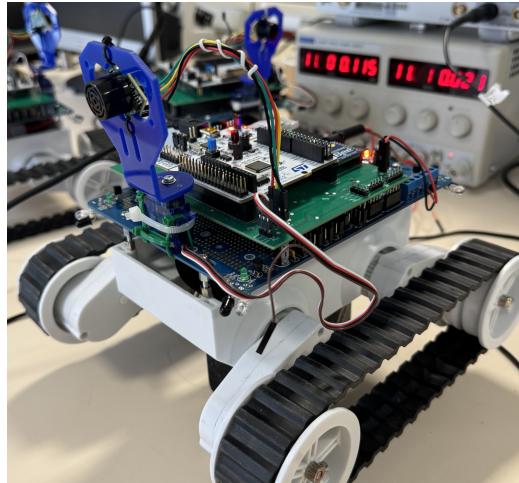


Projet robot

Contrat 7

Rose CYMBLER
École des Mines de Saint-Étienne

24 septembre 2024



Supervisé par : Monsieur Marques, Monsieur Yacoubi et Monsieur Marietti

Table des matières

1	Introduction	4
1.1	Contexte du projet	4
1.2	Cahier des charges et spécifications	4
2	Conception	5
2.1	Organigramme	5
2.1.1	Vue d'ensemble du système	5
2.1.2	Boucle principale, machine d'état et surveillance batterie	6
2.1.3	Fonctions pour le sonar	8
2.1.4	Fonctions de déplacement	11
2.1.5	Fonctions pour éviter l'obstacle	15
2.2	Fonctionnement des capteurs et actionneurs utilisés et périphériques associés . .	16
2.2.1	Le sonar	16
2.2.2	Le servomoteur	17
2.2.3	Le module Bluetooth	17
2.2.4	Le moteur des roues et les encodeurs	17
2.2.5	Les boutons poussoirs	18
2.3	Choix des timers	19
2.3.1	TIMER 1 - Servomoteur et Mesure de Distance	19
2.3.2	TIMER 2 - Contrôle des Moteurs (PWM)	19
2.3.3	TIMERS 3 & 4 - Encodeurs	21
2.3.4	TIMER 6 - Interruption Périodique	21
2.3.5	TIMER 7 - Servomoteur et Sonar	22
3	Réalisation et tests	22
3.1	Comparaison valeurs théoriques et pratiques	22

3.1.1	PWM des roues	22
3.1.2	Vérification de la distance sonar	24
3.1.3	Encodeurs	25
3.2	Améliorations apportées	25
3.2.1	Anti-rebond	25
3.2.2	Encodeurs	27
3.2.3	Filtrage	27
4	Conclusion	29
5	Annexes	30

1 Introduction

1.1 Contexte du projet

Pour ce projet, nous avons à disposition un robot constitué de plusieurs éléments qui permet d'inclure des notions à la fois mécaniques, électroniques et de logiciel typique des systèmes embarqués.

Les capteurs embarqués sur le robot sont des LEDS infrarouges, un sonar, un servo-moteur permettant de faire tourner le sonar, deux moteurs à courant continu avec encodeur à effet de Hall permettant de faire tourner les roues du robot, un module Bluetooth 2.0 (communication avec le téléphone) et un module Zigbee (communication entre machines).

En ce qui concerne l'alimentation du robot, il est équipé d'un pack de batterie (Ni-Mh) 7.2 V, 500 mAh qui est rechargeable. En termes d'électronique, le robot est équipé d'une carte électronique permettant de contrôler les deux moteurs à courant continu. Elle sert également à alimenter le servo moteur en 6V et la carte NUCLEO. Cette dernière est une carte NUCLEO-L476RG utilisée pour la programmation de systèmes embarqués. Elle est basée sur un microcontrôleur STM32L476RG de type ARM Cortex-M4.

A l'aide de cette carte et du logiciel STM32CubeIDE, nous allons pouvoir réaliser et tester notre programme afin de contrôler le robot de sorte à ce qu'il valide les critères du contrat n°7.

1.2 Cahier des charges et spécifications

Le principal objectif de ce contrat était d'intégrer et d'utiliser un sonar pour permettre au robot de détecter les obstacles et d'adapter son parcours en conséquence. L'utilisation du sonar est cruciale pour garantir que le robot puisse naviguer de manière autonome dans son environnement en évitant les collisions. Le sonar mesure la distance entre le robot et les obstacles, permettant ainsi au robot de réagir de manière appropriée en fonction des données recueillies.

Le cahier des charges du contrat n°7 est le suivant :

Fonctions	Critères	Niveaux
Avancer le robot	Vitesse	20 cm/s (+/- 2cm)
Contrôler la vitesse du robot	Fréquence du signal PWM	1 kHz
Contrôler le déplacement du robot	Module Bluetooth	Avancer, Tourner à droite, Tourner à gauche
Esquiver un obstacle	Sens de dégagement intelligent	40 cm : distance d'arrêt face à l'obstacle
	Utilisation du Sonar	
Arrêter d'urgence le robot	Arrêt à tout moment	Bouton poussoir

TABLE 1 – Cahier des charges du contrat n°7

On code en C embarqué, sur le logiciel STM32CubeIDE. Nous devons contrôler notre robot en

Bluetooth grâce à une application. Puis, dès que le robot détecte un mur à moins de 40 cm de lui, il s'arrête. Selon l'angle qu'il fait avec le mur, il tourne à droite ou à gauche en longeant le mur. Enfin, l'utilisateur reprend les commandes du robot.

2 Conception

2.1 Organigramme

2.1.1 Vue d'ensemble du système

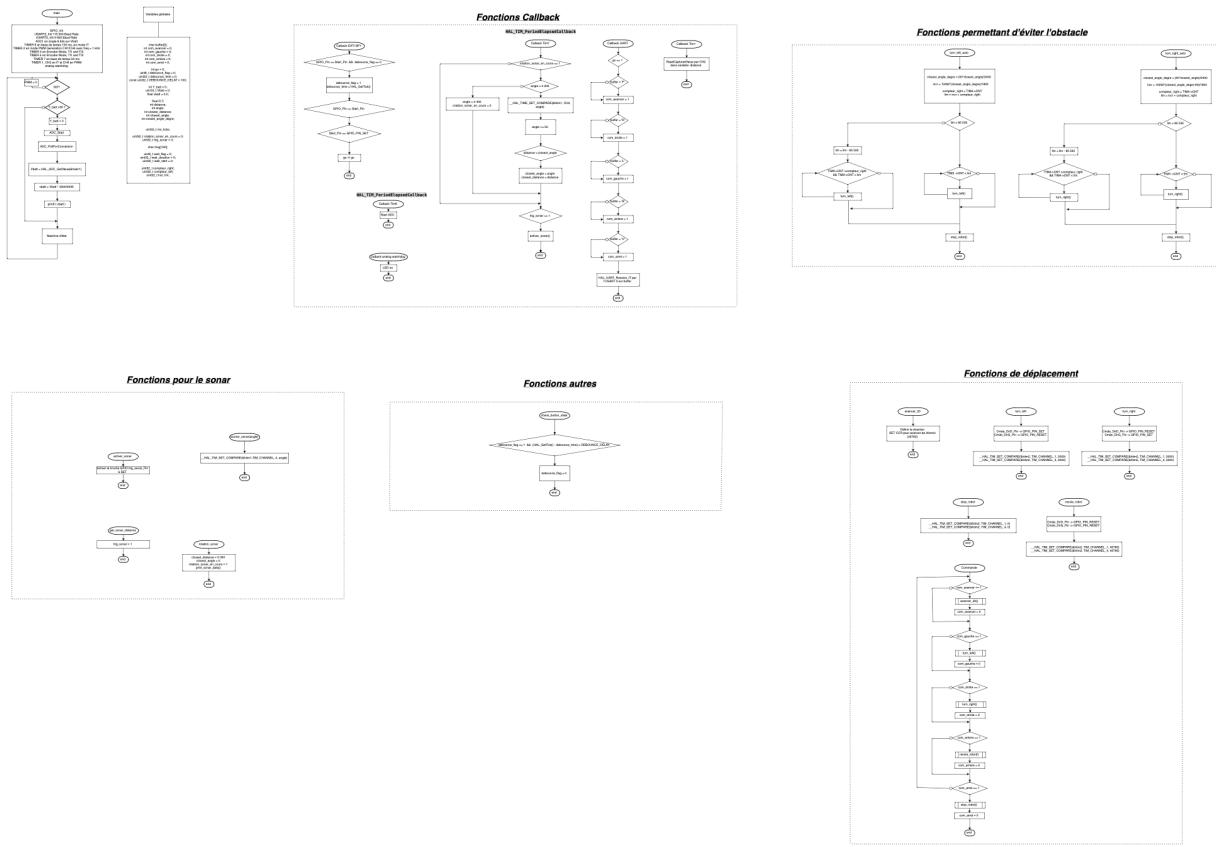


FIGURE 1 – Organigramme complet

L'organigramme complet sera aussi en annexe.

2.1.2 Boucle principale, machine d'état et surveillance batterie

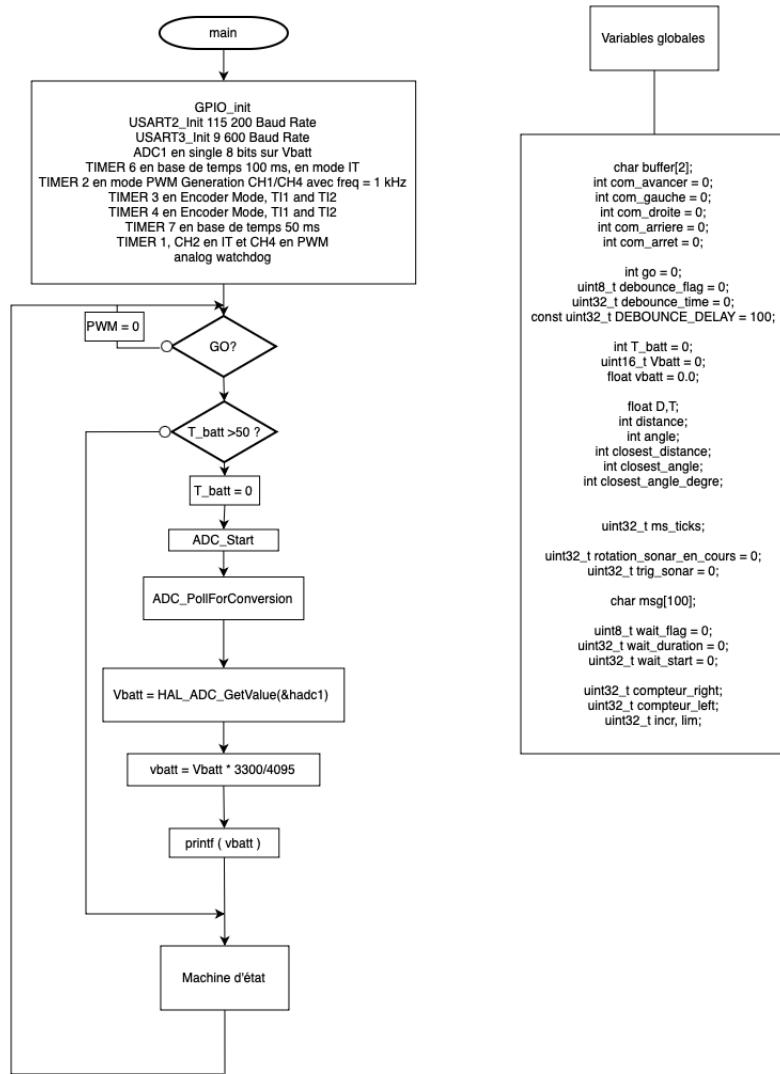


FIGURE 2 – Organigramme de la boucle principale

La boucle principale du programme effectue les opérations suivantes :

- **Initialisation du microcontrôleur et des périphériques** : Cela inclut l'initialisation des ports GPIO, des UART, des ADC, et des timers.
- **Démarrage des périphériques** : Les timers pour les PWM des moteurs, les interruptions de base, les réceptions UART, etc., sont démarrés.
- **Détection de l'état du bouton “Start”** : La variable `go` est utilisée pour déterminer si le robot doit être en marche (`go=1`) ou en pause (`go=0`).
- **Surveillance de la tension de la batterie** : L'ADC mesure la tension de la batterie à chaque itération de la boucle lorsque le robot est en marche (`go=1`). La valeur est convertie et affichée via UART.
- **Gestion de la machine d'état** : En fonction de l'état courant (`currentState`), différentes actions sont effectuées pour contrôler le comportement du robot.

La tension de la batterie est surveillée via l'ADC :

- **Conversion ADC** : À chaque itération de la boucle principale, si le robot est en marche (`go=1`), une conversion ADC est démarrée et le résultat est lu.
- **Conversion en millivolts** : La valeur brute de l'ADC est convertie en millivolts.
- **Affichage** : La tension de la batterie est affichée via UART.
- **Détection de seuil** : Si la tension dépasse un certain seuil (4095), un message indiquant une tension élevée est envoyé. Un seuil inférieur peut déclencher une alerte sur une LED.

La machine d'état utilise l'énumération `State_t` pour gérer différents comportements du robot :

- **STATE_IDLE** :
 - Vérifie l'état du bouton pour démarrer le robot.
 - Tourne le sonar pour détecter les obstacles.
 - Passe à l'état `STATE_FORWARD`.
- **STATE_FORWARD** :
 - Exécute les commandes reçues via UART.
 - Vérifie la distance au mur avec le sonar.
 - Si un obstacle est détecté, passe à `STATE_EVITE_MUR`.
- **STATE_EVITE_MUR** :
 - Arrête le robot.
 - Lance la rotation du sonar pour détecter le meilleur chemin.
 - Passe à l'état `STATE_ROTATE SONAR`.
- **STATE_TURN_LEFT** :
 - Tourne le robot à gauche.
 - Passe à `STATE_FORWARD`.
- **STATE_TURN_RIGHT** :
 - Tourne le robot à droite.
 - Passe à `STATE_FORWARD`.
- **STATE_ROTATE SONAR** :
 - Analyse les données du sonar pour déterminer la meilleure direction à prendre.
 - En fonction des résultats, décide de tourner à gauche, ou à droite.

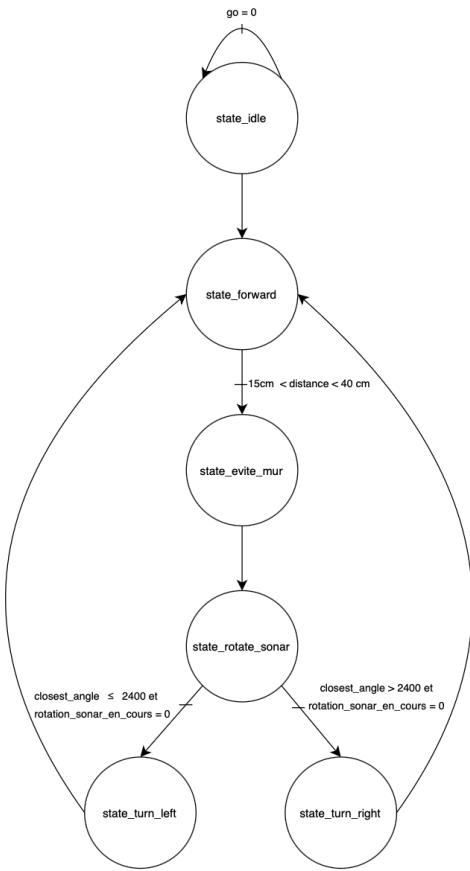
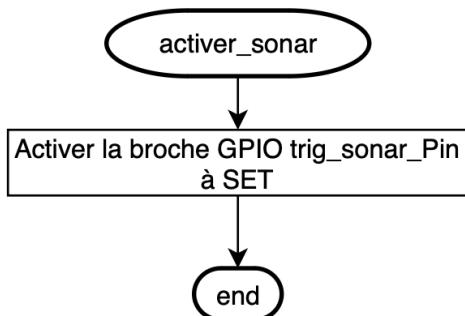


FIGURE 3 – Organigramme de la machine d'état

2.1.3 Fonctions pour le sonar

activer_sonar

FIGURE 4 – Organigramme de la fonction `activer_sonar`

Cette fonction active le pin de déclenchement du sonar (`trig_sonar_Pin`). Elle envoie un signal pour déclencher une mesure de distance.

tourner _ sonar

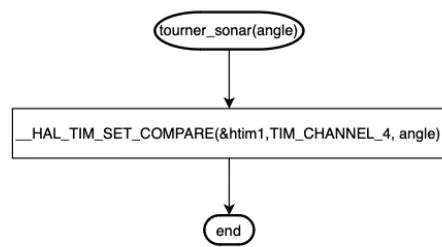


FIGURE 5 – Organigramme de la fonction tourner_sonar

Cette fonction positionne le servomoteur à un angle donné pour balayer l'environnement. L'angle que l'on utilise n'est pas un angle en degré. Ce sont les angles CCR. Ils sont utilisés pour contrôler la position du servomoteur qui oriente le sonar. Ainsi, cela permet au robot de balayer son environnement et de détecter les obstacles.

Le servomoteur est contrôlé par un signal PWM généré par un timer. Le rapport cyclique de ce signal PWM détermine l'angle du servomoteur. On utilise le Timer 1 (TIM1) et le canal 4 (TIM_CHANNEL_4) pour générer ce signal PWM.

Le registre de capture/comparaison (CCR) du timer est utilisé pour définir le rapport cyclique du signal PWM. En changeant la valeur du CCR, on change la durée du signal PWM, ce qui déplace le servomoteur à un angle différent.

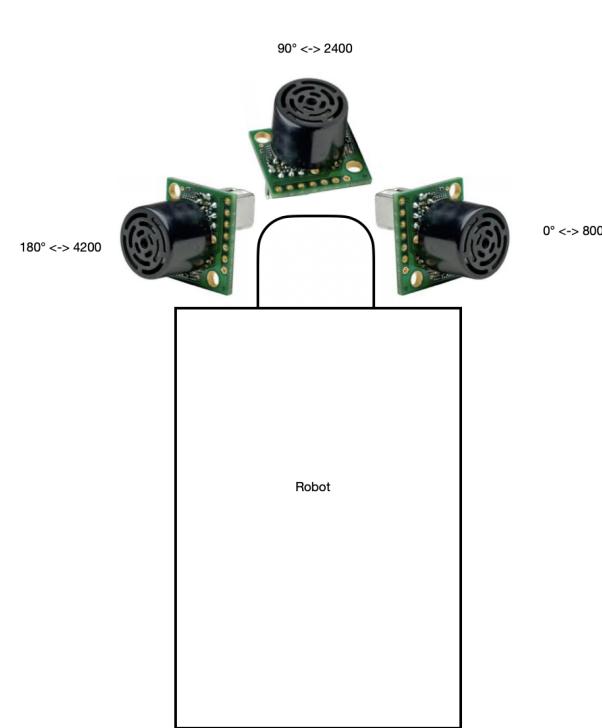


FIGURE 6 – Angles du sonar

Par exemple, si on écrit tourner_sonar(2400), alors, le sonar se remettra droit.

get_sonar_distance

Cette fonction active le sonar en faisant appel à `get_sonar_distance`. Puis, elle passe la variable flag `trig_sonar` à 1.

Le fait que `trig_sonar` soit à 1 permet que dans la fonction de callback

`HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)` soit appelée. Elle permet de calculer la distance en fonction du temps mesuré entre l'envoi et la réception du signal.

Avoir `trig_sonar` à 1, permet aussi d'assurer dans la fonction de callback

`HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)` que le sonar reste bien activé.

Ainsi, la fonction `get_sonar_distance` récupère en permanence des distances et renvoie la distance mesurée par le sonar.

Cependant, cette distance n'est pas en centimètres. Pour avoir cette distance en cm, il faut la diviser par 101.

Nous allons démontrer d'où vient ce facteur 101.

Nous utilisons le timer 1 pour le sonar et le servomoteur. On a un prescaler qui vaut 45.

Nous souhaitons convertir le temps mesuré par le sonar (T) en une distance D en centimètres.

On sait que

- La fréquence d'horloge du microcontrôleur `f_clock` vaut 80MHz.
- Le sonar détecte des objets à 645cm au maximum et 37,5ms est le temps d'écho nécessaire au sonar pour atteindre un obstacle qui est à 645cm.

On a alors

$$D = T \times \frac{46}{80MHz} \times \frac{645}{37,5ms}$$

$$D = T \times 0,00989$$

Donc, on a bien

$$D = \frac{T}{101}$$

rotation_sonar

Cette fonction initialise la rotation du sonar sur un arc de cercle pour balayer l'environnement. Elle met à zéro les variables utilisées pour stocker les mesures et démarre le balayage.

En passant la variable flag `rotation_sonar_en_cours` à 1, la fonction `rotation_sonar` garantit que la rotation du servomoteur du sonar commence.

En effet, quand `rotation_sonar_en_cours` est à 1, la fonction de callback

`HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)` est appelée. Ainsi, la rotation du servomoteur est gérée.

La fonction de callback `HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)` est aussi appelée. Ainsi, les distances calculées par le sonar sont additionnées. Cela va nous permettre de réaliser un filtrage sur les distances pendant la rotation (cf la partie *Améliorations* pour plus de détails).

2.1.4 Fonctions de déplacement

avancer_20

Cette fonction fait avancer le robot en réglant les pins de direction des moteurs et en configurant les PWM pour les moteurs.

Le but est que le robot avance à 20cm/s comme il est demandé dans le contrat.

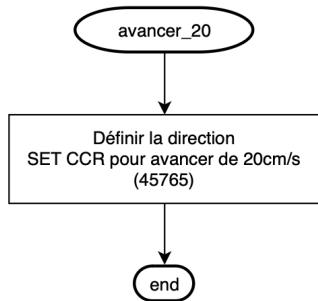


FIGURE 7 – Organigramme de la fonction avancer_20

turn_left

Cette fonction fait tourner le robot vers la gauche. Par cette fonction, on règle les moteurs de manière à ce qu'ils tournent dans des directions opposées, ce qui entraîne une rotation sur place vers la gauche. On contrôle aussi la vitesse en ajustant le rapport cyclique.

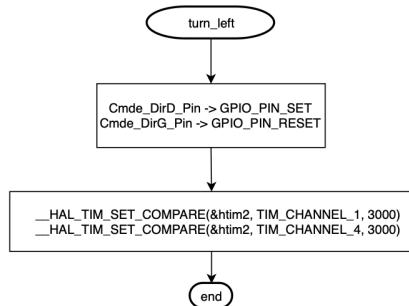


FIGURE 8 – Organigramme de la fonction turn_left

turn_right

Elle fonctionne comme turn_left mais pour tourner à droite au lieu d'à gauche.

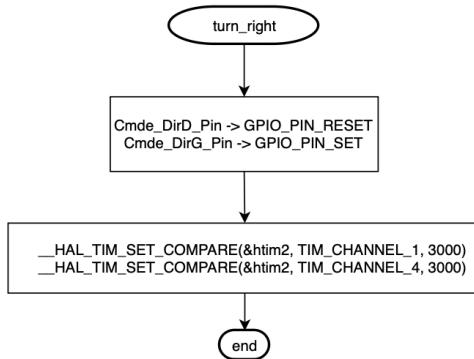


FIGURE 9 – Organigramme de la fonction turn_right

recule_robot

Cette fonction fait reculer le robot. On règle les deux moteurs pour qu'ils tournent en sens inverse par rapport à l'avance.

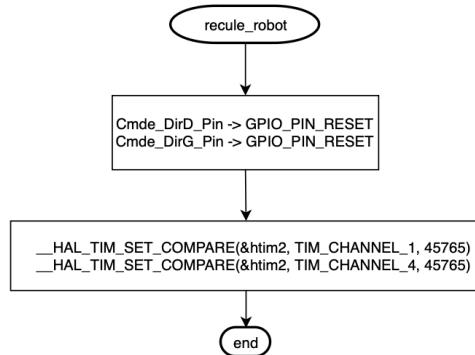


FIGURE 10 – Organigramme de la fonction recule_robot

stop_robot

Cette fonction arrête le robot. On règle les rapports cycliques des moteurs à 0.

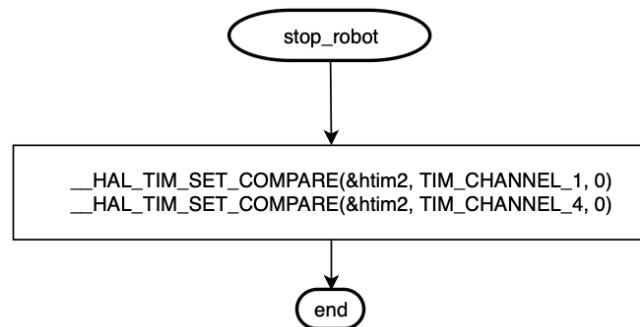


FIGURE 11 – Organigramme de la fonction stop_robot

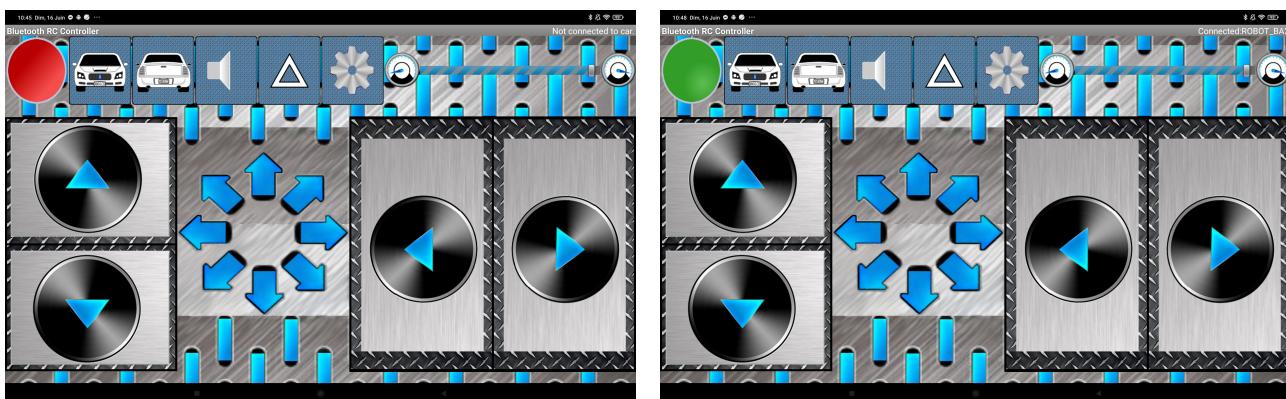
commande

FIGURE 12 – Application non connectée à un robot

FIGURE 13 – Application connectée avec un robot

La fonction *commande* est conçue pour gérer les commandes de mouvement reçues via une communication UART. On utilise une application Bluetooth.



FIGURE 14 – Organigrammes des fonctions *commande* et de callback pour l'UART

L'application sur un smartphone ou une tablette envoie des commandes de mouvement (par exemple, "F" pour avancer, "L" pour tourner à gauche, "R" pour tourner à droite, "B" pour reculer, "V" pour arrêter) via une connexion UART. Chaque bouton de commande a donc un caractère ASCII qui lui est associé.

Notre robot est équipé d'un module UART (ici, huart3) qui reçoit les commandes envoyées par l'application.

Lorsqu'une commande est reçue, une interruption UART est générée, déclenchant la fonction *HAL_UART_RxCpltCallback*.

Dans cette fonction, le code vérifie la commande reçue et met à jour les variables de commande (*com_avancer*, *com_gauche*, *com_droite*, *com_arriere*, *com_arret*) en fonction de la commande reçue.

Puis, la fonction remet également en place la réception UART en appelant *HAL_UART_Receive_IT*

pour continuer à recevoir les commandes suivantes.

La fonction commande est appelée régulièrement dans la boucle principale du programme lorsque go est activé. Cette fonction vérifie les variables de commande et exécute les actions appropriées en appelant les fonctions correspondantes (comme *avancer_20*, *turn_left*, *turn_right*, etc.)

2.1.5 Fonctions pour éviter l'obstacle

turn_left_auto La fonction *turn_left_auto()* permet de longer un obstacle en lançant une rotation automatique vers la gauche suite à un balayage du sonar ayant permis de déterminer la distance la plus courte ainsi que l'angle associé.

Pour longer l'obstacle, il est nécessaire de déterminer l'angle de rotation. Cet angle (*closest_angle*), correspondant à la rotation du servomoteur du sonar lors de la mesure de la plus petite distance (en rouge sur le schéma) avec l'obstacle, doit ensuite être converti en degrés, sachant que 2400 (angle CCR) correspondent à 90 degrés.

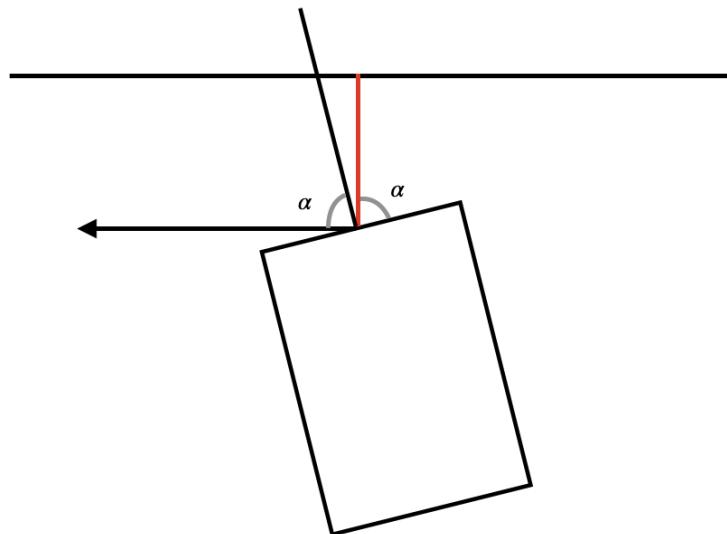


FIGURE 15 – Schéma du robot incliné à gauche

Ensuite, il faut calculer le nombre d'incrémentations nécessaires à l'encodeur droit pour réaliser la rotation désirée, en tenant compte que 1800 degrés correspondent à 10292 incrémentations.

Il existe deux cas de figure, car la valeur du compteur associé à l'encodeur droit s'incrémentera de 0 à 65535 et se réinitialise à 0 une fois cette limite atteinte :

Premier cas : Si la valeur initiale du compteur droit, ajoutée aux incrémentations nécessaires, est inférieure ou égale à 65535, il n'y a pas de problème. La rotation vers la gauche se poursuit tant que le compteur auquel on accède avec TIM3->CNT est inférieur à sa valeur initiale plus le nombre d'incrémentations requis pour réaliser la rotation.

Deuxième cas : Si la valeur initiale du compteur droit, ajoutée aux incrémentations nécessaires, dépasse 65535, le compteur va donc se réinitialiser à 0 au cours de la rotation. Dans ce cas, la condition change. Il faut continuer de tourner à gauche tant que le compteur droit est inférieur à 65535 ou tant qu'il est inférieur à la valeur limite (initialement `compteur_right + incr`) que l'on s'est fixée mais ramenée entre 0 et 65535 (`lim = lim - 65535`).

turn_right_auto

La fonction `turn_right_auto()` permet aussi de longer un obstacle mais avec une rotation automatique vers la droite. Elle fonctionne de la même manière que `turn_left_auto()` si ce n'est que l'on se sert de l'encodeur gauche. Le compteur de cet encodeur est accessible avec `TIM4->CNT`. En outre, la détermination de l'angle est différente et on obtient l'angle de rotation avec `closest_angle_degre - 90°`.

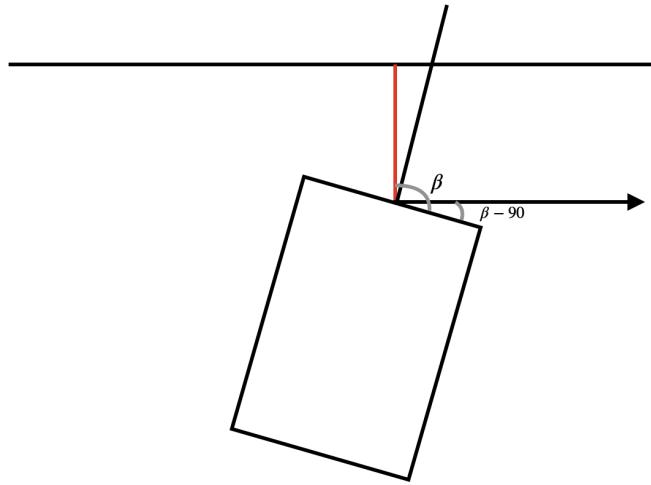


FIGURE 16 – Schéma du robot incliné à droite

2.2 Fonctionnement des capteurs et actionneurs utilisés et périphériques associés

2.2.1 Le sonar

Le capteur sonar LV-MaxSonar-EZ24 est utilisé pour mesurer des distances en envoyant une onde ultrasonore et en mesurant le temps de retour de l'écho.

Le sonar est déclenché toutes les 50 ms, et l'écho est capturé avec un délai maximal de 37,5 ms. Ce délai correspond à une distance de 645cm.

Le sonar fonctionne comme suit :

Un signal de déclenchement est envoyé au capteur.

Le capteur envoie une onde ultrasonore et attend l'écho.

Le temps entre l'envoi et la réception de l'écho est mesuré pour calculer la distance.

Les périphériques associés sont :

- Un timer est utilisé pour générer le signal de déclenchement et mesurer le temps d'écho.
On utilise le timer 1.
- Entrées/Sorties GPIO pour le signal de déclenchement et la lecture de l'écho. On utilise PB10

2.2.2 Le servomoteur

Le servomoteur est utilisé pour la rotation précise du sonar.

Le servomoteur reçoit un signal PWM pour définir sa position angulaire et la largeur de l'impulsion détermine l'angle de rotation.

Comme périphérique, on utilise le timer 1 pour générer le signal PWM de contrôle.

2.2.3 Le module Bluetooth

Ce module est utilisé pour la communication sans fil entre les appareils.

BLE est utilisé pour des communications à courte distance avec faible consommation d'énergie.

Grâce à ce module Bluetooth, on peut utiliser une application mobile pour diriger notre robot.

Pour la communication série, c'est le périphérique UART qui est utilisé (UART3).

2.2.4 Le moteur des roues et les encodeurs

Les moteurs des roues sont des moteurs à courant continu (DC) équipés d'encodeurs à effet Hall pour permettre un contrôle précis de la vitesse et de la position des roues. Chaque moteur est contrôlé par un signal PWM (modulation de largeur d'impulsion) généré par le microcontrôleur.

- **Contrôle de vitesse** : Le rapport cyclique du signal PWM détermine la vitesse de rotation des moteurs. En ajustant ce rapport, nous pouvons contrôler la vitesse du robot avec précision.
- **Direction** : La direction des moteurs est contrôlée par des GPIO configurées pour déterminer le sens de rotation.

Les périphériques associés sont :

- TIM2 : Utilisé pour générer les signaux PWM pour les moteurs des roues.
- GPIO : Utilisés pour configurer la direction des moteurs.
- TIM3 et TIM4 : Configurés en mode encodeur pour lire les signaux des encodeurs des moteurs. TIM3 est utilisé pour l'encodeur de la roue gauche et TIM4 pour l'encodeur

de la roue droite.

2.2.5 Les boutons poussoirs

Les boutons poussoirs sont utilisés pour les entrées utilisateur.
La pression du bouton change l'état d'un signal GPIO.

On utilise les GPIO pour lire l'état du bouton.

Voici le tableau qui recense les pins utilisés.

Nom de la broche	Fonction	Explication
PC11	USART_RX	Réception de données série UART pour la communication.
PA2	USART_TX	Utilisé pour la communication série avec le module Bluetooth.
PA3	USART_RX	Utilisé pour la communication série avec le module Bluetooth.
PA5	Alert_batt	Broche utilisée pour indiquer une alerte de batterie faible. Contrôle de la LED verte.
PA8	TIM1_CH1	Pour contrôler le servomoteur du sonar
PA11	TIM1_CH4	Utilisé pour générer le signal PWM pour le servomoteur.
PA15	TIM2_CH1	Canal 1 du timer 2 pour le contrôle PWM du moteur de la roue droite.
PB2	Cmde_DirG	Commande de direction pour le moteur de la roue gauche.
PC8	Cmde_DirD	Commande de direction pour le moteur de la roue droite.
PB4	ENC1A (TIM3_CH1)	Entrée pour l'encodeur du moteur de la roue gauche.
PB5	ENC1B (TIM3_CH2)	Entrée pour l'encodeur du moteur de la roue gauche.
PB6	ENC2B (TIM3_CH1)	Entrée pour l'encodeur du moteur de la roue droite.
PB7	ENC2A (TIM3_CH1)	Entrée pour l'encodeur du moteur de la roue droite.
PB10	trig_sonar	Broche de déclenchement pour envoyer des impulsions au sonar.
PC13	Start	Broche utilisée pour le bouton de démarrage du robot et l'arrêt d'urgence.

TABLE 2 – Tableau des broches utilisées, leurs fonctions et explications pour le projet robot

2.3 Choix des timers

Pour assurer le bon fonctionnement du système, il est crucial de configurer correctement les registres PSC (Prescaler) et ARR (Auto-Reload Register) des timers. Une valeur de ARR la plus grande possible est souhaitable car elle permet d'augmenter la résolution, offrant ainsi une précision accrue.

2.3.1 TIMER 1 - Servomoteur et Mesure de Distance

Un timer est un registre à l'intérieur du microcontrôleur qui s'incrémente (ou se décrémente) chaque fois qu'il reçoit une impulsion d'un signal d'horloge.

TIM1 est utilisé pour générer le signal PWM qui contrôle le servomoteur du sonar. En modulant le signal PWM, nous pouvons faire tourner le sonar à différents angles pour scanner l'environnement.

TIM1 est également utilisé en mode capture pour mesurer le temps entre l'envoi du signal ultrason et la réception de son écho. Cela permet de calculer la distance entre le robot et un obstacle.

Le sonar détecte des objets jusqu'à 6,45 mètres et fournit des informations de distance entre 15 cm et 645 cm. Le temps d'écho maximum est de 37,5 ms et la distance aller-retour correspondant est de 645cm.

On calcule le prescaler.

$$(PSC + 1) \times \frac{ARR_{max} + 1}{f_{clock}} > 37,5 \times 10^{-3}$$

$$(PSC + 1) > \frac{37,5 \times 10^{-3} \times 80 \times 10^6}{2^{16}}$$

$$(PSC + 1) > 45,77$$

D'où, $PSC + 1 = 46$

Pour l'ARR, on prend $2^{16} - 1 = 65535$

2.3.2 TIMER 2 - Contrôle des Moteurs (PWM)

Dans notre projet, nous utilisons la PWM pour contrôler les moteurs des roues du robot. La PWM permet de moduler la puissance délivrée aux moteurs en faisant varier le rapport cyclique du signal PWM.

La PWM est utilisée pour contrôler la vitesse des moteurs des roues du robot. Le signal PWM est généré par le microcontrôleur et appliqué aux drivers de moteurs qui alimentent les moteurs en courant continu. En modifiant le rapport cyclique du signal PWM, nous pouvons ajuster la vitesse de rotation des moteurs.

Nous utilisons les timers du microcontrôleur STM32L476RG pour générer les signaux PWM. Deux canaux PWM sont configurés, un pour chaque moteur (roue gauche et roue droite).

L'utilisation de la PWM présente plusieurs avantages pour le contrôle des moteurs. La PWM permet de contrôler la vitesse des moteurs sans dissiper une grande quantité d'énergie sous forme de chaleur, contrairement à d'autres méthodes de contrôle analogique. La PWM offre un contrôle précis de la vitesse des moteurs en ajustant finement le rapport cyclique.

TIM2 est configuré pour générer deux signaux PWM indépendants utilisés pour contrôler les moteurs des roues du robot. Le rapport cyclique de ces signaux PWM détermine la vitesse de chaque moteur, permettant de diriger le robot. TIM2 est démarré en mode PWM pour initialiser les moteurs à une vitesse contrôlée dès le début.

Selon les spécifications du contrat n°7, la fréquence de la PWM doit être de 1 kHz. Cela signifie que chaque période du signal PWM dure 1 milliseconde.

On calcule dans un premier temps le prescaler (PSC) :

$$(PSC + 1) \times \frac{2^{32}}{80 \times 10^6} > 1 \times 10^{-3}$$

D'où,

$$(PSC + 1) > \frac{1 \times 10^{-3} \times 80 \times 10^6}{2^{32}} = 0,000019$$

On a alors $PSC + 1 = 1$

On calcule l'ARR,

$$(ARR + 1) \times (PSC + 1) \times \frac{1}{80 \times 10^6} = 1 \times 10^{-3}$$

Alors,

$$(ARR + 1) = \frac{1 \times 10^{-3} \times 80 \times 10^6}{1} = 80000$$

2.3.3 TIMERS 3 & 4 - Encodeurs

TIM3 et TIM4 sont configurés en mode encodeur pour lire les signaux des encodeurs à effet Hall des moteurs. Cela permet de mesurer la rotation des roues et de calculer la distance parcourue et la vitesse du robot.

Pour les timers 3 et 4, les prescalers sont à 0 et les ARR à 65535.

La période de ces deux timers est donc de

$$T = \frac{(PSC + 1) \times (ARR + 1)}{f_{clock}}$$

$$T = \frac{1 \times 65536}{80000000} = 0.0008192s = 0.001s$$

2.3.4 TIMER 6 - Interruption Périodique

TIM6 est configuré pour générer des interruptions périodiques. Ces interruptions sont utilisées pour démarrer les conversions ADC pour la surveillance de la tension de la batterie.

L'interruption périodique permet de vérifier régulièrement la tension de la batterie pour s'assurer qu'elle est suffisante pour le fonctionnement du robot.

La période T du timer est donnée par :

$$T = \frac{(PSC + 1) \times (ARR + 1)}{f_{clock}}$$

où :

- T est la période désirée (1s),
- PSC est le prescaler,
- ARR est la valeur de l'auto-reload register,
- f_{clock} est la fréquence de l'horloge du microcontrôleur (80 MHz).

Pour le prescaler, on a :

$$(PSC + 1) \times \frac{2^{16}}{80 \times 10^6} > 1$$

D'où,

$$(PSC + 1) > \frac{1 \times 80 \times 10^6}{65536} = 1220,7$$

On a alors $PSC + 1 = 1221$

On calcule l'ARR,

$$(ARR + 1) \times (PSC + 1) \times \frac{1}{80 \times 10^6} = 1$$

Alors,

$$(ARR + 1) = \frac{1 \times 80 \times 10^6}{1221} = 65520,1$$

Donc, $ARR = 65520$

2.3.5 TIMER 7 - Servomoteur et Sonar

TIM7 est utilisé pour gérer la rotation du servomoteur du sonar en créant des interruptions à intervalles réguliers. Cela permet de changer l'angle du sonar et de balayer l'environnement pour détecter des obstacles. TIM7 s'assure aussi que le sonar est bien activé.

Nous voulons configurer le Timer 7 pour générer une interruption toutes les 500 ms.
La période du timer est donnée par :

$$T = \frac{(PSC + 1) \times (ARR + 1)}{f_{clock}}$$

où :

- T est la période désirée (500 ms),
- PSC est le prescaler,
- ARR est la valeur de l'auto-reload register,
- f_{clock} est la fréquence de l'horloge du microcontrôleur (80 MHz).

Nous voulons une période de 500 ms.

Nous avons comme prescaler 7999 et 4999 en ARR.

D'où,

$$T = \frac{8000 \times 5000}{80000000} = 0,5s$$

Ces réglages permettent bien d'avoir une période de 500ms.

3 Réalisation et tests

3.1 Comparaison valeurs théoriques et pratiques

3.1.1 PWM des roues

D'après les spécifications du contrat 7, la PWM des roues doit être égale à 1 kHz.
On souhaite le vérifier à l'oscilloscope.

On connecte la sonde à la masse et au pin associé à la PWM du moteur de la roue droite et du moteur de la roue gauche.

On observe bien à l'oscilloscope une fréquence de 1 kHz.

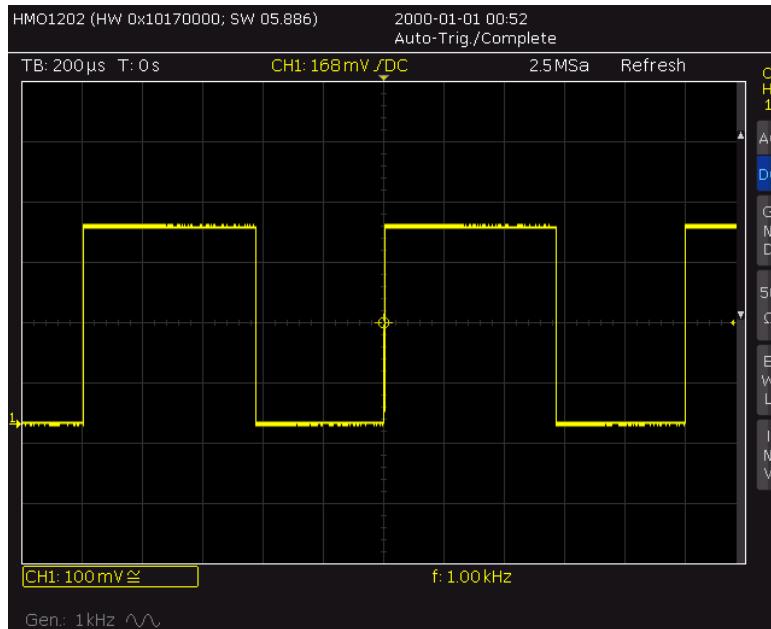


FIGURE 17 – Mesure sur l'oscilloscope de la PWM des moteurs des roues

Le signal présente une forme d'onde carrée caractéristique des signaux PWM. La transition entre les états haut et bas du signal est nette, indiquant une bonne qualité du signal PWM sans distorsion significative.

Le contrat impose aussi que le robot avance à 20cm/s. Pour les moteurs des roues, le registre ARR est défini à 80000. Le rapport cyclique de la sortie PWM sur les deux chenilles est fixé à 57,2 % ($= \frac{45765}{80000}$) de sorte à ce que le robot ait une vitesse de 20 cm/s. On vérifie cela sur l'oscilloscope.

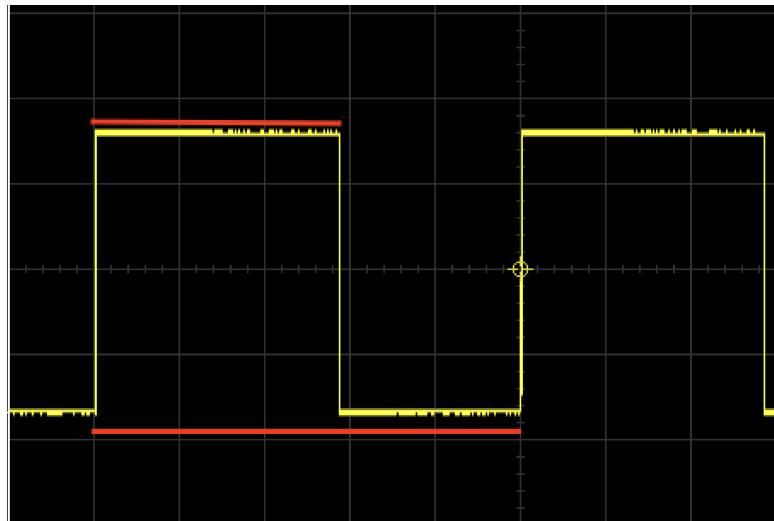


FIGURE 18 – Détermination du rapport cyclique (ou dutycycle) sur l'oscilloscope

On mesure $\frac{2.85}{5} = 57\%$.

Les spécifications de l'énoncé sont alors bien respectées.

3.1.2 Vérification de la distance sonar

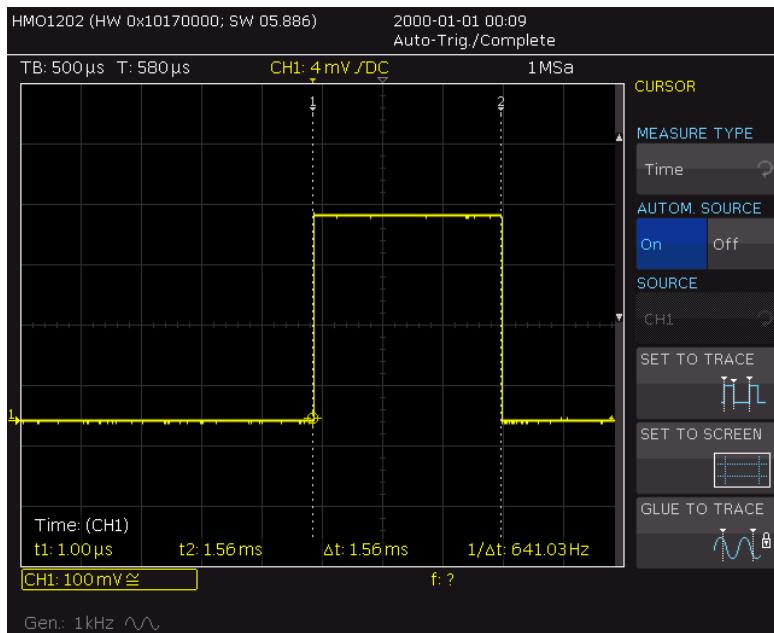


FIGURE 19 – Mesure sur l'oscilloscope de la distance sonar

Nous plaçons une feuille à 26 cm du sonar.

Le temps mesuré (Δt) est de 1,56 ms, tel que montré par les curseurs de l'oscilloscope. Ce temps représente l'aller-retour du signal ultrasonique du sonar à l'obstacle et retour.

$$distance = \frac{645 \times 1,56}{37,5} = 26,8cm.$$

La mesure est donc bien cohérente.

3.1.3 Encodeurs

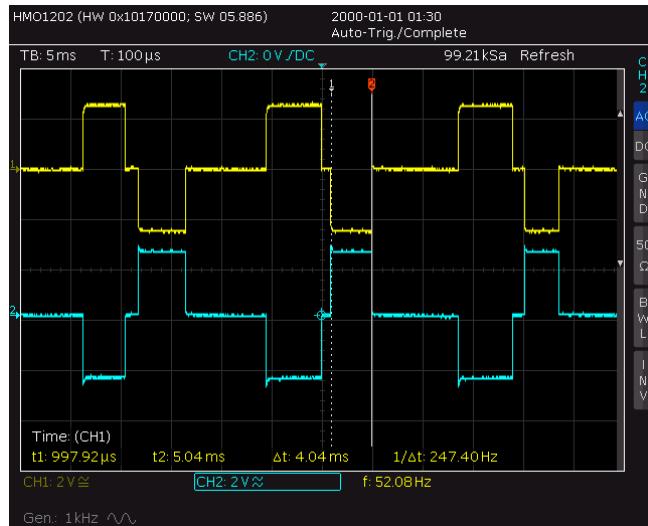


FIGURE 20 – Mesure sur l'oscilloscope des encodeurs

On remarque bien un déphasage de 90° entre le signal jaune et le signal bleu. Ceci est bien typique des encodeurs.

3.2 Améliorations apportées

3.2.1 Anti-rebond

Pour débuter le programme du robot, il faut appuyer sur le bouton start (le bouton bleu). Cependant, nous sommes souvent confronté au phénomène du rebond.

Lorsqu'on enfonce un bouton ou bien qu'on le relâche, il oscille généralement pendant quelques millisecondes entre l'état ouvert et l'état fermé. En conséquence, pendant ces quelques fractions de seconde, le bouton peut basculer de façon aléatoire entre l'état ouvert et l'état fermé avant de se stabiliser dans l'état attendu

Le phénomène de rebond est un effet indésirable qui se produit lors de l'activation ou de la désactivation du bouton. Lorsqu'on appuie sur le bouton, les contacts métalliques à l'intérieur se touchent et se séparent rapidement plusieurs fois avant de se stabiliser dans l'état fermé (contact établi) ou ouvert (contact interrompu). Cette oscillation rapide entre les états ouvert et fermé peut durer quelques millisecondes. Alors, sans gestion du rebond, le microcontrôleur peut interpréter un seul appui sur un bouton comme étant plusieurs appuis successifs. Ainsi, on peut appuyer sur le bouton pour démarrer le programme mais à cause de l'effet du rebond, le robot ne démarre pas.

On décide alors de faire un filtre anti-rebond.

Lorsque le bouton est pressé, une interruption est générée. Cette interruption est gérée par la fonction *HAL_GPIO_EXTI_Callback*.

Dans la fonction *HAL_GPIO_EXTI_Callback*, lorsqu'une interruption est détectée sur la broche du bouton (Start_Pin), la variable flag debounce_flag est mis à 1 et debounce_time est mis à jour avec le temps actuel (*HAL_GetTick()*). La variable go est basculée pour indiquer le changement d'état du système, et l'état de debounce_flag empêche de traiter de nouvelles interruptions pendant une période de temps définie par DEBOUNCE_DELAY (100ms).

Dans la boucle principale, la fonction *check_button_state()* est appelée pour vérifier si le délai anti-rebond est écoulé.

Cette fonction compare le temps actuel (*HAL_GetTick()*) avec le temps enregistré lors de l'interruption (debounce_time). Si le temps écoulé dépasse DEBOUNCE_DELAY (100 ms dans ce cas), debounce_flag est réinitialisé à 0, permettant ainsi de traiter de nouvelles interruptions.

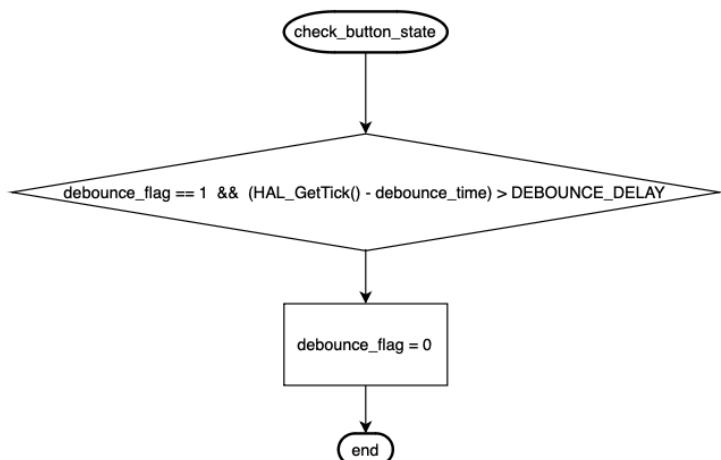


FIGURE 21 – Organigramme de la fonction anti-rebond

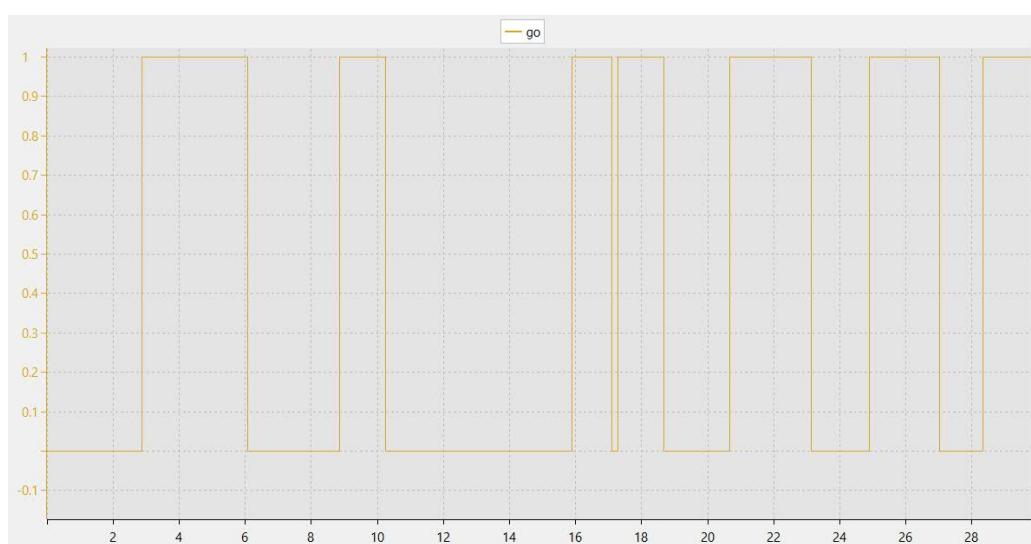


FIGURE 22 – Variation du bouton go

On remarque sur cette figure que les variations du bouton go sont bien claires et définies. Le filtre anti-rebond fonctionne.

3.2.2 Encodeurs

Les encodeurs à effet Hall sont utilisés pour mesurer la rotation des roues du robot et ainsi contrôler avec précision la vitesse et la direction du déplacement.

Ainsi, cela nous a permis d'avoir une mesure plus précise pour la rotation des roues et pour que le robot se retrouve bien parallèle à l'obstacle.

Les encodeurs exploitent l'effet Hall pour détecter la variation du champ magnétique produit par les aimants fixés sur un disque rotatif.

Ces encodeurs génèrent une série d'impulsions en fonction de la rotation du disque. Le nombre d'impulsions par révolution (PPR) est une caractéristique clé qui détermine la résolution de l'encodeur (333,33 top/tour).

3.2.3 Filtrage

Lors de la rotation du sonar pour balayer l'environnement, nous avons observé des valeurs incohérentes et anormales dans les mesures de distance. Ces anomalies peuvent provenir de diverses sources telles que des interférences environnementales, des réflexions multiples, ou des limitations intrinsèques du capteur.

Nous avons alors décidé de faire une fonction de filtrage des distances.

Sur les résultats obtenus grâce à SWV (en mode debug), on affiche en fonction du temps les variables :

- rotation_sonar_en_cours (en vert)
- distance (en jaune)
- average_distance (en bleu)
- tourne_gauche ou tourne_droite (en rouge)

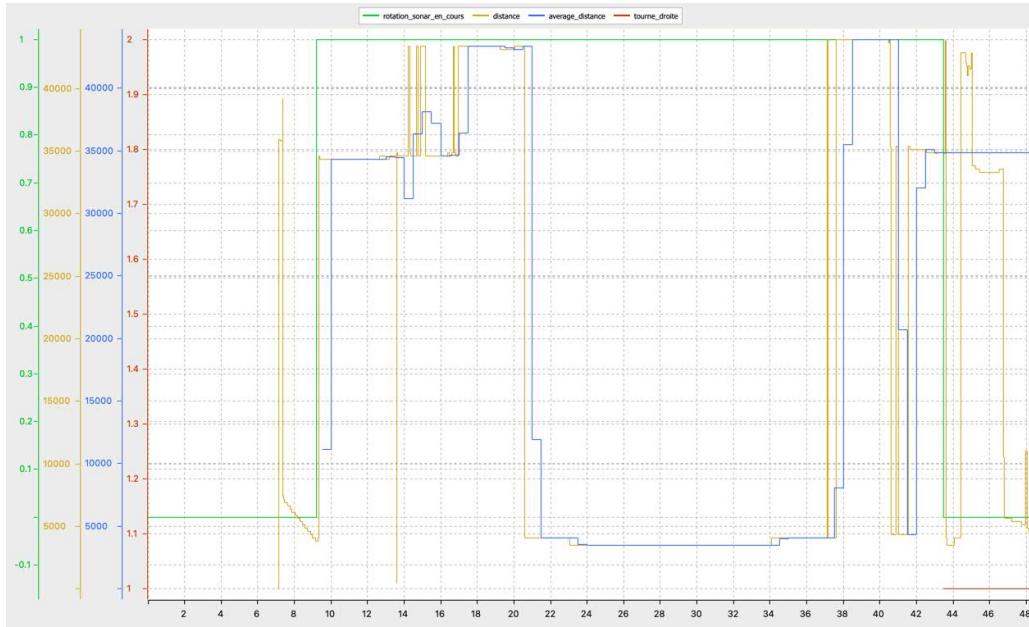


FIGURE 23 – Mesure via SWV de la rotation du sonar (cas du robot qui tourne à droite)

La courbe jaune (distance) montre les distances récupérées par le sonar lors de la rotation du sonar (lorsque la variable `rotation_sonar_en_cours` est à 1). On observe qu'il arrive que le sonar renvoie des valeurs quelque peu incompréhensibles. Pourtant, lors de la rotation du sonar, il y a seulement le mur comme obstacle.

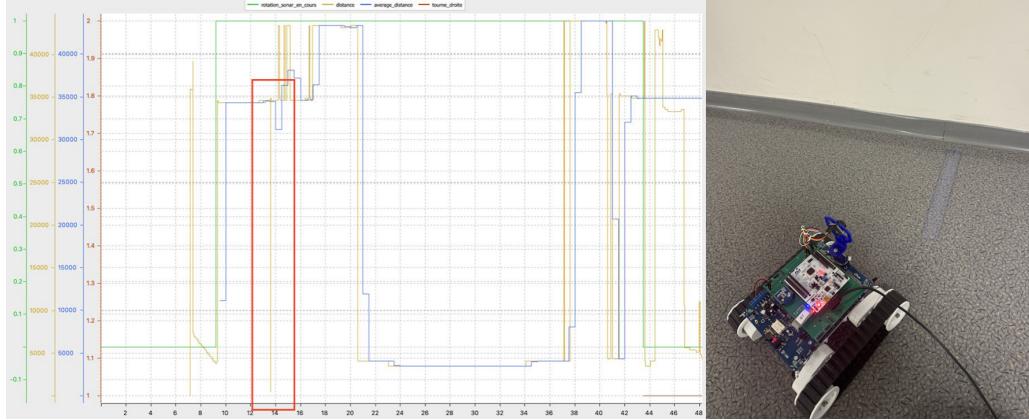


FIGURE 24 – Erreur de la valeur renvoyée par le sonar alors que le robot n'a aucun obstacle devant lui

La valeur encadrée en rouge sur la figure est la plus basse. Cette valeur est plutôt anormale et elle fausse l'ensemble de l'algorithme. En effet, elle va être enregistrée comme `closest_distance` et l'angle associé va être retenu comme `closest_angle`. On voit, de plus, que cette valeur se situe dans la moitié gauche de la zone délimitée par le signal carré de `rotation_sonar_en_cours`. Alors, le robot tourne à gauche, ce qui n'est pas logique.

C'est pour cela que l'on a décidé de faire une fonction de filtrage. Ainsi, on ne s'intéresse plus à la courbe jaune mais à la courbe bleue. On remarque que la courbe bleue n'a pas de variations aussi brutales que la courbe jaune.

Cela s'explique par le fait que pour réaliser ce filtrage, on a fait une moyenne glissante. On a augmenté la période du timer qui gère la rotation du servomoteur. Elle est maintenant de 500 ms. Ainsi, le servomoteur reste à un angle α pendant 500 ms. Durant ce temps, le sonar prend 10 mesures et on prend la moyenne de ces 10 distances "récupérées". Puis, on incrémente l'angle α de 50 et on réitère le processus. Ces étapes sont répétées pour des angles allant de 800 (droite) à 4200 (gauche) avec un pas de 50.

D'où, lorsqu'on suit la courbe bleue, on voit bien que la plus petite distance est dans la moitié droite et le robot va bien tourner à droite. En effet, la courbe rouge (variable *tourne_droite*) passe bien à 1, lorsque *rotation_sonar_en_cours* repasse à 0.

De même, on fait le test dans le cas où le robot est incliné de telle façon qu'il doit tourner à gauche.

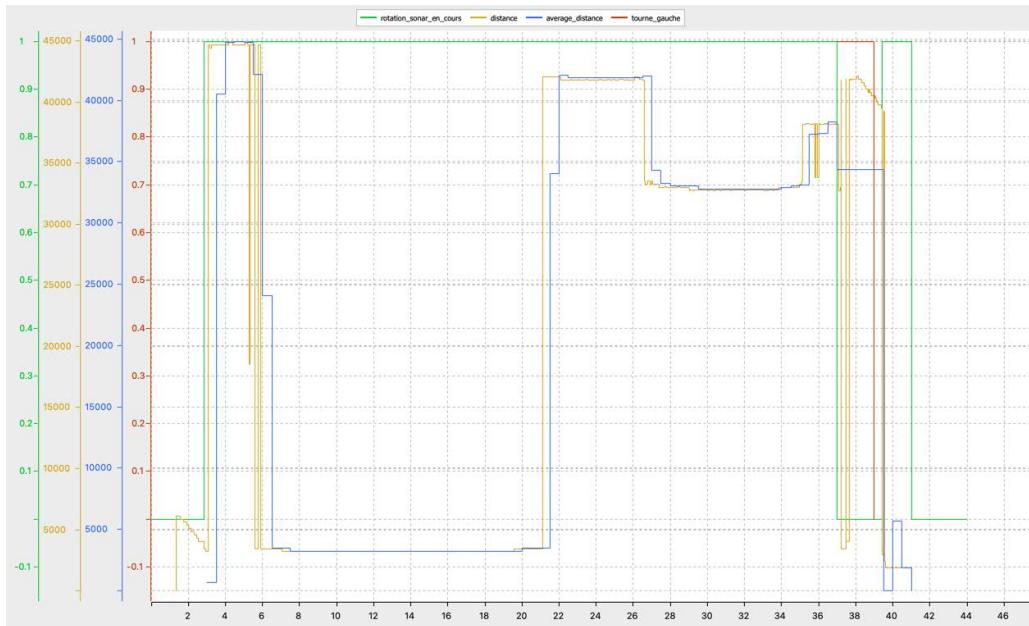


FIGURE 25 – Mesure via SWV de la rotation du sonar (cas du robot qui tourne à gauche)

La plus petite distance est bien dans la moitié gauche. Par conséquent, le robot tournera bien à gauche.

4 Conclusion

Ce projet robot nous a permis de concevoir et de tester un robot capable de naviguer de manière autonome grâce à l'intégration d'un sonar. Le robot peut détecter des obstacles et ajuster son parcours en conséquence, en respectant les spécifications du contrat n°7.

Nous avons rencontré quelques difficultés lors de ce projet. En effet, nous avions été très guidés lors du cours de SAM mais pour ce projet, nous étions beaucoup plus en autonomie. Cela rend donc ce projet gratifiant et stimulant. Les résultats obtenus montrent un fonctionnement

global satisfaisant du robot. Les améliorations apportées, comme l'anti-rebond et le filtrage des mesures du sonar, ont permis d'affiner les performances du robot.

Pour améliorer notre contrat, nous pourrions imposer un asservissement en vitesse sur le robot, dont la vitesse varierait en fonction de la charge de la batterie. De plus, on pourrait exiger que le robot ne s'arrête pas pour faire son balayage de mesure avec le sonar avant de longer un obstacle. Cela impliquerait une amélioration de l'algorithme de navigation et une optimisation des temps de réponse du robot.

Dans un contexte logistique, un tel robot pourrait naviguer dans un entrepôt en longeant les murs et les allées pour scanner et vérifier les stocks, tout en détectant et évitant les obstacles comme les palettes et les chariots. Il pourrait également transporter des pièces ou des produits d'un point à un autre, contournant les zones encombrées, pour assurer une logistique fluide et ininterrompue. D'autres applications industrielles de ce robot pourraient être le nettoyage et l'inspection, toujours par sa capacité d'esquiver des obstacles et de les longer, pouvant permettre de balayer des surfaces encombrées, telles que des usines ou des entrepôts. On pourrait même envisager l'utilisation d'un tel robot pour l'inspection de réacteurs nucléaires.

Nous tenons à remercier M. Marques, M. Yacoubi et M. Marietti pour leur aide précieuse et leurs cours, sans lesquels ce projet n'aurait pas pu aboutir.

5 Annexes

```
/* USER CODE BEGIN Header */
/**
 ****
 * @file          : main.c
 * @brief         : Main program body
 ****
 * @attention
 *
 * Copyright (c) 2024 STMicroelectronics.
 * All rights reserved.
 *
 * This software is licensed under terms that can be found in the LICENSE file
 * in the root directory of this software component.
 * If no LICENSE file comes with this software, it is provided AS-IS.
 *
 ****
 */
/* USER CODE END Header */
/* Includes ----- */
```

```
#include "main.h"

/* Private includes -----
/* USER CODE BEGIN Includes */
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <string.h>
/* USER CODE END Includes */

/* Private typedef -----
/* USER CODE BEGIN PTD */
typedef enum {
    STATE_IDLE,
    STATE_FORWARD,
    STATE_TURN_LEFT,
    STATE_TURN_RIGHT,
    STATE_ROTATE SONAR,
    STATE_EVITE_MUR
} State_t;
/* USER CODE END PTD */

/* Private define -----
/* USER CODE BEGIN PD */
/* USER CODE END PD */

/* USER CODE END PD */

/* Private macro -----
/* USER CODE BEGIN PM */
/* USER CODE END PM */

/* Private variables -----
ADC_HandleTypeDef hadc1;

TIM_HandleTypeDef htim1;
TIM_HandleTypeDef htim2;
TIM_HandleTypeDef htim3;
TIM_HandleTypeDef htim4;
TIM_HandleTypeDef htim6;
TIM_HandleTypeDef htim7;

UART_HandleTypeDef huart2;
UART_HandleTypeDef huart3;

/* USER CODE BEGIN PV */

State_t currentState = STATE_IDLE;
```

```
void check_button_state(void);

void avancer_20(void);
void turn_left(void);
void turn_left_auto(void);
void turn_right(void);
void turn_right_auto(void);
void stop_robot(void);
void recule_robot(void);
void commande();

int get_sonar_distance(void);
void tourner_sonar(int angle);
void rotation_sonar(void);
void rotation_sonar_recursive(int angle);

void activer_sonar(void);

volatile char buffer[2];
volatile int com_avancer = 0;
volatile int com_gauche = 0;
volatile int com_droite = 0;
volatile int com_arriere = 0;
volatile int com_arret = 0;

volatile uint32_t go = 0;
volatile uint8_t debounce_flag = 0;
volatile uint32_t debounce_time = 0;
const uint32_t DEBOUNCE_DELAY = 100; // 100 ms debounce delay

//volatile int T_batt = 0;
volatile int mesure_batt = 0;
int Vbatt = 0;
int vbatt = 0;

float D,T;
volatile int distance;
volatile int angle;
volatile int closest_distance = 5050;
volatile int closest_angle = 0;
int closest_angle_degre;

#define NUM_SAMPLES 10
int sonar_readings[NUM_SAMPLES]; // Tableau pour stocker les lectures du sonar
int sonar_index = 0; // Indice pour le tableau des lectures
int sonar_sum = 0; // Somme des lectures pour la moyenne
```

```
int sonar_count = 0; // Nombre de lectures effectuées

volatile uint32_t rotation_sonar_en_cours = 0;
volatile uint32_t trig_sonar = 0;
volatile uint32_t tourne_droite = 0;
volatile uint32_t tourne_gauche = 0;
volatile uint32_t average_distance = 0;

char msg[100];

volatile uint32_t compteur_right;
volatile uint32_t compteur_left;
volatile uint32_t incr, lim;

/* USER CODE END PV */

/* Private function prototypes -----
void SystemClock_Config(void);
static void MX_GPIO_Init(void);
static void MX_USART2_UART_Init(void);
static void MX_ADC1_Init(void);
static void MX_TIM6_Init(void);
static void MX_TIM2_Init(void);
static void MX_USART3_UART_Init(void);
static void MX_TIM1_Init(void);
static void MX_TIM7_Init(void);
static void MX_TIM3_Init(void);
static void MX_TIM4_Init(void);
/* USER CODE BEGIN PFP */

/* USER CODE END PFP */

/* Private user code -----
/* USER CODE BEGIN 0 */
#define SEUIL_3V 3723 //la plus grande valeur de l'adc étant 4096 = 2^12 pour 3.3V

/* USER CODE END 0 */

/**
 * @brief The application entry point.
 * @retval int
 */
int main(void)
{
    /* USER CODE BEGIN 1 */
```

```
/* USER CODE END 1 */

/* MCU Configuration-----*/
/* Reset of all peripherals, Initializes the Flash interface and the Systick. */
HAL_Init();

/* USER CODE BEGIN Init */

/* USER CODE END Init */

/* Configure the system clock */
SystemClock_Config();

/* USER CODE BEGIN SysInit */

/* USER CODE END SysInit */

/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_USART2_UART_Init();
MX_ADC1_Init();
MX_TIM6_Init();
MX_TIM2_Init();
MX_USART3_UART_Init();
MX_TIM1_Init();
MX_TIM7_Init();
MX_TIM3_Init();
MX_TIM4_Init();
/* USER CODE BEGIN 2 */
HAL_TIM_Base_Start(&htim2);
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1); // démarrage du moteur 1
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_4); // démarrage du moteur 2
__HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, 0);
__HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_4, 0);
HAL_TIM_Base_Start_IT(&htim6);
HAL_UART_Receive_IT(&huart3, (uint8_t *)buffer, 2);

//distance
HAL_TIM_IC_Start_IT(&htim1, TIM_CHANNEL_2);

//servomoteur
HAL_TIM_PWM_Start(&htim1, TIM_CHANNEL_4);

HAL_TIM_Base_Start_IT(&htim7);
```



```
    commande();

distance = get_sonar_distance();
tourne_droite = 0;
tourne_gauche = 0;
if (distance < 4040 && distance > 1515)
{
    tourner_sonar(800);
    currentState = STATE_EVITE_MUR;
}
else {
    currentState = STATE_FORWARD;
}

break;

case STATE_EVITE_MUR:
    stop_robot();
    rotation_sonar();
    currentState = STATE_ROTATE SONAR;
    break;

case STATE_TURN_LEFT:
tourner_sonar(2400);
    turn_left_auto();
    currentState = STATE_FORWARD;
    break;

case STATE_TURN_RIGHT:
tourner_sonar(2400);
    turn_right_auto();
    currentState = STATE_FORWARD;
    break;

case STATE_ROTATE SONAR:
//rotation_sonar();
if (closest_angle > 2400 && rotation_sonar_en_cours == 0)
{
    tourne_droite = 1;
    currentState = STATE_TURN_RIGHT;
}
else if (closest_angle < 2400 && rotation_sonar_en_cours == 0)
{
    //tourner_sonar(2400);
    tourne_gauche = 1;
    currentState = STATE_TURN_LEFT;
}
```

```

        else if (closest_angle == 2400 && rotation_sonar_en_cours == 0)
        {
            //tourner_sonar(2400);
            tourne_gauche = 1;
            currentState = STATE_TURN_LEFT;
        }
        break;

    }

}

/* USER CODE END WHILE */

/* USER CODE BEGIN 3 */

__HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, 0);
__HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_4, 0);
}

/* USER CODE END 3 */
}

/***
 * @brief System Clock Configuration
 * @retval None
 */
void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

    /** Configure the main internal regulator output voltage
    */
    if (HAL_PWREx_ControlVoltageScaling(PWR_REGULATOR_VOLTAGE_SCALE1) != HAL_OK)
    {
        Error_Handler();
    }

    /** Initializes the RCC Oscillators according to the specified parameters
    * in the RCC_OscInitTypeDef structure.
    */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
    RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_ON;
    RCC_OscInitStruct.PLL.PLLSource = RCC_PLLSOURCE_HSI;
    RCC_OscInitStruct.PLL.PLLM = 1;
}

```

```

RCC_OscInitStruct.PLL.PLLN = 10;
RCC_OscInitStruct.PLL.PLLP = RCC_PLLP_DIV7;
RCC_OscInitStruct.PLL.PLLQ = RCC_PLLQ_DIV2;
RCC_OscInitStruct.PLL.PLLR = RCC_PLLR_DIV2;
if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
{
    Error_Handler();
}

/** Initializes the CPU, AHB and APB buses clocks
*/
RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                            |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_PLLCLK;
RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, FLASH_LATENCY_4) != HAL_OK)
{
    Error_Handler();
}
}

/** 
 * @brief ADC1 Initialization Function
 * @param None
 * @retval None
 */
static void MX_ADC1_Init(void)
{

/* USER CODE BEGIN ADC1_Init_0 */

/* USER CODE END ADC1_Init_0 */

ADC_MultiModeTypeDef multimode = {0};
ADC_AnalogWDGConfTypeDef AnalogWDGConfig = {0};
ADC_ChannelConfTypeDef sConfig = {0};

/* USER CODE BEGIN ADC1_Init_1 */

/* USER CODE END ADC1_Init_1 */

/** Common config
*/
hadc1.Instance = ADC1;
hadc1.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV1;

```

```

hadc1.Init.Resolution = ADC_RESOLUTION_12B;
hadc1.Init.DataAlign = ADC_DATAALIGN_RIGHT;
hadc1.Init.ScanConvMode = ADC_SCAN_DISABLE;
hadc1.Init.EOCSelection = ADC_EOC_SINGLE_CONV;
hadc1.Init.LowPowerAutoWait = DISABLE;
hadc1.Init.ContinuousConvMode = DISABLE;
hadc1.Init.NbrOfConversion = 1;
hadc1.Init.DiscontinuousConvMode = DISABLE;
hadc1.Init.ExternalTrigConv = ADC_SOFTWARE_START;
hadc1.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
hadc1.Init.DMAContinuousRequests = DISABLE;
hadc1.Init.Overrun = ADC_OVR_DATA_PRESERVED;
hadc1.Init.OversamplingMode = DISABLE;
if (HAL_ADC_Init(&hadc1) != HAL_OK)
{
    Error_Handler();
}

/** Configure the ADC multi-mode
*/
multimode.Mode = ADC_MODE_INDEPENDENT;
if (HAL_ADCEx_MultiModeConfigChannel(&hadc1, &multimode) != HAL_OK)
{
    Error_Handler();
}

/** Configure Analog WatchDog 1
*/
AnalogWDGConfig.WatchdogNumber = ADC_ANALOGWATCHDOG_1;
AnalogWDGConfig.WatchdogMode = ADC_ANALOGWATCHDOG_SINGLE_REG;
AnalogWDGConfig.Channel = ADC_CHANNEL_14;
AnalogWDGConfig.ITMode = ENABLE;
AnalogWDGConfig.HighThreshold = 4095;
AnalogWDGConfig.LowThreshold = 3723;
if (HAL_ADC_AnalogWDGConfig(&hadc1, &AnalogWDGConfig) != HAL_OK)
{
    Error_Handler();
}

/** Configure Regular Channel
*/
sConfig.Channel = ADC_CHANNEL_14;
sConfig.Rank = ADC_REGULAR_RANK_1;
sConfig.SamplingTime = ADC_SAMPLETIME_640CYCLES_5;
sConfig.SingleDiff = ADC_SINGLE_ENDED;
sConfig.OffsetNumber = ADC_OFFSET_NONE;
sConfig.Offset = 0;
if (HAL_ADC_ConfigChannel(&hadc1, &sConfig) != HAL_OK)

```

```
{  
    Error_Handler();  
}  
/* USER CODE BEGIN ADC1_Init_2 */  
  
/* USER CODE END ADC1_Init_2 */  
  
}  
  
/**/  
 * @brief TIM1 Initialization Function  
 * @param None  
 * @retval None  
 */  
static void MX_TIM1_Init(void)  
{  
  
    /* USER CODE BEGIN TIM1_Init_0 */  
  
    /* USER CODE END TIM1_Init_0 */  
  
    TIM_ClockConfigTypeDef sClockSourceConfig = {0};  
    TIM_SlaveConfigTypeDef sSlaveConfig = {0};  
    TIM_IC_InitTypeDef sConfigIC = {0};  
    TIM_MasterConfigTypeDef sMasterConfig = {0};  
    TIM_OC_InitTypeDef sConfigOC = {0};  
    TIM_BreakDeadTimeConfigTypeDef sBreakDeadTimeConfig = {0};  
  
    /* USER CODE BEGIN TIM1_Init_1 */  
  
    /* USER CODE END TIM1_Init_1 */  
    htim1.Instance = TIM1;  
    htim1.Init.Prescaler = 46-1;  
    htim1.Init.CounterMode = TIM_COUNTERMODE_UP;  
    htim1.Init.Period = 65535;  
    htim1.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;  
    htim1.Init.RepetitionCounter = 0;  
    htim1.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;  
    if (HAL_TIM_Base_Init(&htim1) != HAL_OK)  
    {  
        Error_Handler();  
    }  
    sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;  
    if (HAL_TIM_ConfigClockSource(&htim1, &sClockSourceConfig) != HAL_OK)  
    {  
        Error_Handler();  
    }  
    if (HAL_TIM_PWM_Init(&htim1) != HAL_OK)
```

```

{
    Error_Handler();
}
if (HAL_TIM_IC_Init(&htim1) != HAL_OK)
{
    Error_Handler();
}
sSlaveConfig.SlaveMode = TIM_SLAVERESET;
sSlaveConfig.InputTrigger = TIM_TS_TI1FP1;
sSlaveConfig.TriggerPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
sSlaveConfig.TriggerPrescaler = TIM_ICPSC_DIV1;
sSlaveConfig.TriggerFilter = 0;
if (HAL_TIM_SlaveConfigSynchro(&htim1, &sSlaveConfig) != HAL_OK)
{
    Error_Handler();
}
sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_RISING;
sConfigIC.ICSelection = TIM_ICSELECTION_DIRECTTI;
sConfigIC.ICPrescaler = TIM_ICPSC_DIV1;
sConfigIC.ICFilter = 0;
if (HAL_TIM_IC_ConfigChannel(&htim1, &sConfigIC, TIM_CHANNEL_1) != HAL_OK)
{
    Error_Handler();
}
sConfigIC.ICPolarity = TIM_INPUTCHANNELPOLARITY_FALLING;
sConfigIC.ICSelection = TIM_ICSELECTION_INDIRECTTI;
if (HAL_TIM_IC_ConfigChannel(&htim1, &sConfigIC, TIM_CHANNEL_2) != HAL_OK)
{
    Error_Handler();
}
sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
sMasterConfig.MasterOutputTrigger2 = TIM_TRGO2_RESET;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim1, &sMasterConfig) != HAL_OK)
{
    Error_Handler();
}
sConfigOC.OCMode = TIM_OCMODE_PWM1;
sConfigOC.Pulse = 0;
sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
sConfigOC.OCIdleState = TIM_OCIDLESTATE_RESET;
sConfigOC.OCNIdleState = TIM_OCNIDLESTATE_RESET;
if (HAL_TIM_PWM_ConfigChannel(&htim1, &sConfigOC, TIM_CHANNEL_4) != HAL_OK)
{
    Error_Handler();
}
sBreakDeadTimeConfig.OffStateRunMode = TIM_OSSR_DISABLE;

```

```

sBreakDeadTimeConfig.OffStateIDLEMode = TIM_OSSI_DISABLE;
sBreakDeadTimeConfig.LockLevel = TIM_LOCKLEVEL_OFF;
sBreakDeadTimeConfig.DeadTime = 0;
sBreakDeadTimeConfig.BreakState = TIM_BREAK_DISABLE;
sBreakDeadTimeConfig.BreakPolarity = TIM_BREAKPOLARITY_HIGH;
sBreakDeadTimeConfig.BreakFilter = 0;
sBreakDeadTimeConfig.Break2State = TIM_BREAK2_DISABLE;
sBreakDeadTimeConfig.Break2Polarity = TIM_BREAK2POLARITY_HIGH;
sBreakDeadTimeConfig.Break2Filter = 0;
sBreakDeadTimeConfig.AutomaticOutput = TIM_AUTOMATICOUTPUT_DISABLE;
if (HAL_TIMEx_ConfigBreakDeadTime(&htim1, &sBreakDeadTimeConfig) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN TIM1_Init_2 */

/* USER CODE END TIM1_Init_2 */
HAL_TIM_MspPostInit(&htim1);

}

/***
 * @brief TIM2 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM2_Init(void)
{

/* USER CODE BEGIN TIM2_Init_0 */

/* USER CODE END TIM2_Init_0 */

TIM_ClockConfigTypeDef sClockSourceConfig = {0};
TIM_MasterConfigTypeDef sMasterConfig = {0};
TIM_OC_InitTypeDef sConfigOC = {0};

/* USER CODE BEGIN TIM2_Init_1 */

/* USER CODE END TIM2_Init_1 */
htim2.Instance = TIM2;
htim2.Init.Prescaler = 1-1;
htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
htim2.Init.Period = 80000;
htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim2.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
if (HAL_TIM_Base_Init(&htim2) != HAL_OK)
{

```

```

        Error_Handler();
    }
sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
if (HAL_TIM_ConfigClockSource(&htim2, &sClockSourceConfig) != HAL_OK)
{
    Error_Handler();
}
if (HAL_TIM_PWM_Init(&htim2) != HAL_OK)
{
    Error_Handler();
}
sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim2, &sMasterConfig) != HAL_OK)
{
    Error_Handler();
}
sConfigOC.OCMode = TIM_OCMODE_PWM1;
sConfigOC.Pulse = 0;
sConfigOC.OCPolarity = TIM_OCPOLARITY_HIGH;
sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_1) != HAL_OK)
{
    Error_Handler();
}
if (HAL_TIM_PWM_ConfigChannel(&htim2, &sConfigOC, TIM_CHANNEL_4) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN TIM2_Init 2 */

/* USER CODE END TIM2_Init 2 */
HAL_TIM_MspPostInit(&htim2);

}

/**
 * @brief TIM3 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM3_Init(void)
{
    /* USER CODE BEGIN TIM3_Init 0 */

    /* USER CODE END TIM3_Init 0 */

```

```

TIM_Encoder_InitTypeDef sConfig = {0};
TIM_MasterConfigTypeDef sMasterConfig = {0};

/* USER CODE BEGIN TIM3_Init 1 */

/* USER CODE END TIM3_Init 1 */
htim3.Instance = TIM3;
htim3.Init.Prescaler = 0;
htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
htim3.Init.Period = 65535;
htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim3.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
sConfig.EncoderMode = TIM_ENCODERMODE_TI12;
sConfig.IC1Polarity = TIM_ICPOLARITY_RISING;
sConfig.IC1Selection = TIM_ICSELECTION_DIRECTTI;
sConfig.IC1Prescaler = TIM_ICPSC_DIV1;
sConfig.IC1Filter = 8;
sConfig.IC2Polarity = TIM_ICPOLARITY_RISING;
sConfig.IC2Selection = TIM_ICSELECTION_DIRECTTI;
sConfig.IC2Prescaler = TIM_ICPSC_DIV1;
sConfig.IC2Filter = 8;
if (HAL_TIM_Encoder_Init(&htim3, &sConfig) != HAL_OK)
{
    Error_Handler();
}
sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN TIM3_Init 2 */

/* USER CODE END TIM3_Init 2 */

}

/**
 * @brief TIM4 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM4_Init(void)
{

/* USER CODE BEGIN TIM4_Init 0 */

/* USER CODE END TIM4_Init 0 */

```

```

TIM_Encoder_InitTypeDef sConfig = {0};
TIM_MasterConfigTypeDef sMasterConfig = {0};

/* USER CODE BEGIN TIM4_Init 1 */

/* USER CODE END TIM4_Init 1 */
htim4.Instance = TIM4;
htim4.Init.Prescaler = 0;
htim4.Init.CounterMode = TIM_COUNTERMODE_UP;
htim4.Init.Period = 65535;
htim4.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
htim4.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
sConfig.EncoderMode = TIM_ENCODERMODE_TI12;
sConfig.IC1Polarity = TIM_ICPOLARITY_FALLING;
sConfig.IC1Selection = TIM_ICSELECTION_DIRECTTI;
sConfig.IC1Prescaler = TIM_ICPSC_DIV1;
sConfig.IC1Filter = 8;
sConfig.IC2Polarity = TIM_ICPOLARITY_RISING;
sConfig.IC2Selection = TIM_ICSELECTION_DIRECTTI;
sConfig.IC2Prescaler = TIM_ICPSC_DIV1;
sConfig.IC2Filter = 8;
if (HAL_TIM_Encoder_Init(&htim4, &sConfig) != HAL_OK)
{
    Error_Handler();
}
sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim4, &sMasterConfig) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN TIM4_Init 2 */

/* USER CODE END TIM4_Init 2 */

}

/**
 * @brief TIM6 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM6_Init(void)
{

/* USER CODE BEGIN TIM6_Init 0 */

```

```
/* USER CODE END TIM6_Init_0 */

TIM_MasterConfigTypeDef sMasterConfig = {0};

/* USER CODE BEGIN TIM6_Init_1 */

/* USER CODE END TIM6_Init_1 */
htim6.Instance = TIM6;
htim6.Init.Prescaler = 1221-1;
htim6.Init.CounterMode = TIM_COUNTERMODE_UP;
htim6.Init.Period = 65520;
htim6.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
if (HAL_TIM_Base_Init(&htim6) != HAL_OK)
{
    Error_Handler();
}
sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim6, &sMasterConfig) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN TIM6_Init_2 */

/* USER CODE END TIM6_Init_2 */

}

/**
 * @brief TIM7 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM7_Init(void)
{

/* USER CODE BEGIN TIM7_Init_0 */

/* USER CODE END TIM7_Init_0 */

TIM_MasterConfigTypeDef sMasterConfig = {0};

/* USER CODE BEGIN TIM7_Init_1 */

/* USER CODE END TIM7_Init_1 */
htim7.Instance = TIM7;
htim7.Init.Prescaler = 8000-1;
htim7.Init.CounterMode = TIM_COUNTERMODE_UP;
```

```

htim7.Init.Period = 5000-1;
htim7.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
if (HAL_TIM_Base_Init(&htim7) != HAL_OK)
{
    Error_Handler();
}
sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim7, &sMasterConfig) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN TIM7_Init 2 */

/* USER CODE END TIM7_Init 2 */

}

/***
 * @brief USART2 Initialization Function
 * @param None
 * @retval None
 */
static void MX_USART2_UART_Init(void)
{

/* USER CODE BEGIN USART2_Init 0 */

/* USER CODE END USART2_Init 0 */

/* USER CODE BEGIN USART2_Init 1 */

/* USER CODE END USART2_Init 1 */
huart2.Instance = USART2;
huart2.Init.BaudRate = 115200;
huart2.Init.WordLength = UART_WORDLENGTH_8B;
huart2.Init.StopBits = UART_STOPBITS_1;
huart2.Init.Parity = UART_PARITY_NONE;
huart2.Init.Mode = UART_MODE_TX_RX;
huart2.Init.HwFlowCtl = UART_HWCONTROL_NONE;
huart2.Init.OverSampling = UART_OVERSAMPLING_16;
huart2.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;
huart2.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;
if (HAL_UART_Init(&huart2) != HAL_OK)
{
    Error_Handler();
}
/* USER CODE BEGIN USART2_Init 2 */

```

```
/* USER CODE END USART2_Init 2 */  
  
}  
  
/**  
 * @brief USART3 Initialization Function  
 * @param None  
 * @retval None  
 */  
static void MX_USART3_UART_Init(void)  
{  
  
/* USER CODE BEGIN USART3_Init 0 */  
  
/* USER CODE END USART3_Init 0 */  
  
/* USER CODE BEGIN USART3_Init 1 */  
  
/* USER CODE END USART3_Init 1 */  
huart3.Instance = USART3;  
huart3.Init.BaudRate = 9600;  
huart3.Init.WordLength = UART_WORDLENGTH_8B;  
huart3.Init.StopBits = UART_STOPBITS_1;  
huart3.Init.Parity = UART_PARITY_NONE;  
huart3.Init.Mode = UART_MODE_TX_RX;  
huart3.Init.HwFlowCtl = UART_HWCONTROL_NONE;  
huart3.Init.OverSampling = UART_OVERSAMPLING_16;  
huart3.Init.OneBitSampling = UART_ONE_BIT_SAMPLE_DISABLE;  
huart3.AdvancedInit.AdvFeatureInit = UART_ADVFEATURE_NO_INIT;  
if (HAL_UART_Init(&huart3) != HAL_OK)  
{  
    Error_Handler();  
}  
/* USER CODE BEGIN USART3_Init 2 */  
  
/* USER CODE END USART3_Init 2 */  
  
}  
  
/**  
 * @brief GPIO Initialization Function  
 * @param None  
 * @retval None  
 */  
static void MX_GPIO_Init(void)  
{  
    GPIO_InitTypeDef GPIO_InitStruct = {0};
```

```

/* USER CODE BEGIN MX_GPIO_Init_1 */
/* USER CODE END MX_GPIO_Init_1 */

/* GPIO Ports Clock Enable */
__HAL_RCC_GPIOC_CLK_ENABLE();
__HAL_RCC_GPIOH_CLK_ENABLE();
__HAL_RCC_GPIOA_CLK_ENABLE();
__HAL_RCC_GPIOB_CLK_ENABLE();

/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(Alert_batt_GPIO_Port, Alert_batt_Pin, GPIO_PIN_RESET);

/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(GPIOB, Cmde_DirG_Pin|trig_sonar_Pin, GPIO_PIN_RESET);

/*Configure GPIO pin Output Level */
HAL_GPIO_WritePin(Cmde_DirD_GPIO_Port, Cmde_DirD_Pin, GPIO_PIN_RESET);

/*Configure GPIO pin : Start_Pin */
GPIO_InitStruct.Pin = Start_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
GPIO_InitStruct.Pull = GPIO_NOPULL;
HAL_GPIO_Init(Start_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pin : Alert_batt_Pin */
GPIO_InitStruct.Pin = Alert_batt_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(Alert_batt_GPIO_Port, &GPIO_InitStruct);

/*Configure GPIO pins : Cmde_DirG_Pin trig_sonar_Pin */
GPIO_InitStruct.Pin = Cmde_DirG_Pin|trig_sonar_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

/*Configure GPIO pin : Cmde_DirD_Pin */
GPIO_InitStruct.Pin = Cmde_DirD_Pin;
GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
HAL_GPIO_Init(Cmde_DirD_GPIO_Port, &GPIO_InitStruct);

/* EXTI interrupt init*/
HAL_NVIC_SetPriority(EXTI15_10_IRQn, 0, 0);
HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);

```

```

/* USER CODE BEGIN MX_GPIO_Init_2 */
/* USER CODE END MX_GPIO_Init_2 */
}

/* USER CODE BEGIN 4 */

void check_button_state()
{
    if (debounce_flag == 1 && (HAL_GetTick() - debounce_time) > DEBOUNCE_DELAY)
    {
        debounce_flag = 0;
    }
}

void avancer_20(void)
{
    HAL_GPIO_WritePin(GPIOC, Cmde_DirD_Pin, GPIO_PIN_SET); // réglage de la direction
    HAL_GPIO_WritePin(GPIOB, Cmde_DirG_Pin, GPIO_PIN_SET); // réglage de la direction
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, 45765); // réglage du rapport cyclique
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_4, 45765); // réglage du rapport cyclique
}

void turn_left(void)
{
    HAL_GPIO_WritePin(GPIOC, Cmde_DirD_Pin, GPIO_PIN_SET);
    HAL_GPIO_WritePin(GPIOB, Cmde_DirG_Pin, GPIO_PIN_RESET);
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, 30000);
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_4, 30000);
}

void turn_right(void)
{
    HAL_GPIO_WritePin(GPIOC, Cmde_DirD_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, Cmde_DirG_Pin, GPIO_PIN_SET);
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, 30000);
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_4, 30000);
}

```

```
}

void stop_robot(void)
{
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, 0);
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_4, 0);
}

void recule_robot(void)
{
    HAL_GPIO_WritePin(GPIOC, Cmde_DirD_Pin, GPIO_PIN_RESET);
    HAL_GPIO_WritePin(GPIOB, Cmde_DirG_Pin, GPIO_PIN_RESET);
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_1, 45765);
    __HAL_TIM_SET_COMPARE(&htim2, TIM_CHANNEL_4, 45765);
}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    UNUSED(huart);

    if (go){
        if (buffer[0] == 'F') {
            com_avancer = 1;
        }
        else if (buffer[0] == 'R') {
            com_droite = 1;
        }
        else if (buffer[0] == 'L') {
            com_gauche = 1;
        }
        else if (buffer[0] == 'B') {
            com_arriere = 1;
        }
        else if (buffer[0] == 'V') {
            com_arret = 1;
        }
    }
}

HAL_UART_Receive_IT(&huart3, (uint8_t *)buffer, 2);

}

void commande(void){
```

```

    if (com_avancer == 1) {
avancer_20();
com_avancer = 0;
}

    if (com_gauche == 1) {
turn_left();
com_gauche = 0;
}

    if (com_droite == 1) {
turn_right();
com_droite = 0;
}

    if (com_arriere == 1){
recule_robot();
com_arriere = 0;
}

    if (com_arret == 1){
stop_robot();
com_arret = 0;
}

}

void HAL_ADC_ConvCpltCallback(ADC_HandleTypeDef *hadc)
{
    UNUSED(hadc);
    measure_batt = 1;

}

//BATTERIE
void HAL_ADC_LevelOutOfWindowCallback(ADC_HandleTypeDef *hadc)
{

if (hadc->Instance == ADC1){
HAL_GPIO_WritePin(GPIOA, Alert_batt_Pin, GPIO_PIN_SET);
}

}

```

```

void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    UNUSED(htim);

    if (htim->Instance == TIM7)
    {
        if (rotation_sonar_en_cours == 1)
        {

            // Calculer la moyenne des 10 mesures prises à cet angle
            average_distance = sonar_sum / NUM_SAMPLES;

            // Stocker ou traiter la moyenne ici selon les besoins
            if (average_distance <= closest_distance)
            {
                closest_distance = average_distance;
                closest_angle = angle;
            }

            // Réinitialiser pour le prochain angle
            sonar_sum = 0;
            sonar_count = 0;

            // Incrémenter l'angle après avoir pris 10 mesures
            angle += 50;

            // Mettre à jour l'angle du servomoteur
            if (angle <= 4000)
            {
                __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_4, angle);
            }
            else
            {
                rotation_sonar_en_cours = 0; // Arrêter la rotation à la fin du sweep
            }
        }

        if (trig_sonar == 1)
        {
            // Déclencher le sonar
            activer_sonar();
        }
    }

    if (htim->Instance == TIM6)

```

```

    {
    HAL_ADC_Start_IT(&hadc1);

}

void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin == Start_Pin && debounce_flag == 0)
    {
        debounce_flag = 1;
        debounce_time = HAL_GetTick();
    }

    if (GPIO_Pin == Start_Pin)
    {
        if (HAL_GPIO_ReadPin(Start_GPIO_Port, Start_Pin) == GPIO_PIN_SET)
        {
            go = !go; // Basculer la variable go entre 0 et 1
        }
    }
}

//SONAR
//mesure distance par sonar
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef *htim)
{
if (htim ->Channel == HAL_TIM_ACTIVE_CHANNEL_2 && trig_sonar == 1)
{
distance = HAL_TIM_ReadCapturedValue(&htim1, TIM_CHANNEL_2);

    if (rotation_sonar_en_cours == 1 && sonar_count < NUM_SAMPLES)
    {
        sonar_sum += distance;
        sonar_count++;
    }
}
}

```

```
}

void activer_sonar()
{
    HAL_GPIO_WritePin(GPIOB, trig_sonar_Pin, GPIO_PIN_SET);
}

// Fonction pour mesurer la distance
int get_sonar_distance(void)
{
    activer_sonar();
    trig_sonar = 1;
    return distance;
}

void tourner_sonar(int angle)
{
    __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_4, angle);
}

void rotation_sonar()
{
    sonar_index = 0;
    sonar_sum = 0;
    sonar_count = 0;
    memset(sonar_readings, 0, sizeof(sonar_readings));
    rotation_sonar_en_cours = 1;
    angle = 800; // Start angle from the beginning
    closest_distance = 5050; // Reset closest distance
    __HAL_TIM_SET_COMPARE(&htim1, TIM_CHANNEL_4, angle); // Ensure sonar starts moving
}

void turn_left_auto()
{
    closest_angle_degre = (90*closest_angle)/2400;

    incr = 10292*(closest_angle_degre)/1800;

    compteur_right = TIM3->CNT;
    lim = incr + compteur_right;
    if(lim < 65535){
        while(TIM3->CNT < lim){


```

```

turn_left();
compteur_right = TIM3->CNT;
compteur_left = TIM4->CNT;
}
}
else{
lim = lim - 65535;
while(TIM3->CNT > compteur_right || TIM3->CNT < lim){
turn_left();
compteur_right = TIM3->CNT;
compteur_left = TIM4->CNT;
}
}

// Arrêter les moteurs
stop_robot();
}

void turn_right_auto()
{

//1800° -> 10292

closest_angle_degre = (90*closest_angle)/2400;

incr = 10292*(closest_angle_degre-90)/1800;

compteur_left = TIM4->CNT;
lim = incr + compteur_left;

if(lim < 65535){
while(TIM4->CNT < lim){
turn_right();
compteur_left = TIM4->CNT;
compteur_right = TIM3->CNT;
}
}
else{
lim = lim - 65535;
while(TIM4->CNT > compteur_left || TIM4->CNT < lim){
turn_right();
compteur_left = TIM4->CNT;
compteur_right = TIM3->CNT;
}
}

// Arrêter les moteurs

```

```
    stop_robot();
}

/* USER CODE END 4 */

/***
 * @brief This function is executed in case of error occurrence.
 * @retval None
 */
void Error_Handler(void)
{
    /* USER CODE BEGIN Error_Handler_Debug */
    /* User can add his own implementation to report the HAL error return state */
    __disable_irq();
    while (1)
    {
    }
    /* USER CODE END Error_Handler_Debug */
}

#ifndef USE_FULL_ASSERT
/***
 * @brief Reports the name of the source file and the source line number
 *        where the assert_param error has occurred.
 * @param file: pointer to the source file name
 * @param line: assert_param error line source number
 * @retval None
 */
void assert_failed(uint8_t *file, uint32_t line)
{
    /* USER CODE BEGIN 6 */
    /* User can add his own implementation to report the file name and line number,
       ex: printf("Wrong parameters value: file %s on line %d\r\n", file, line) */
    /* USER CODE END 6 */
}
#endif /* USE_FULL_ASSERT */
```

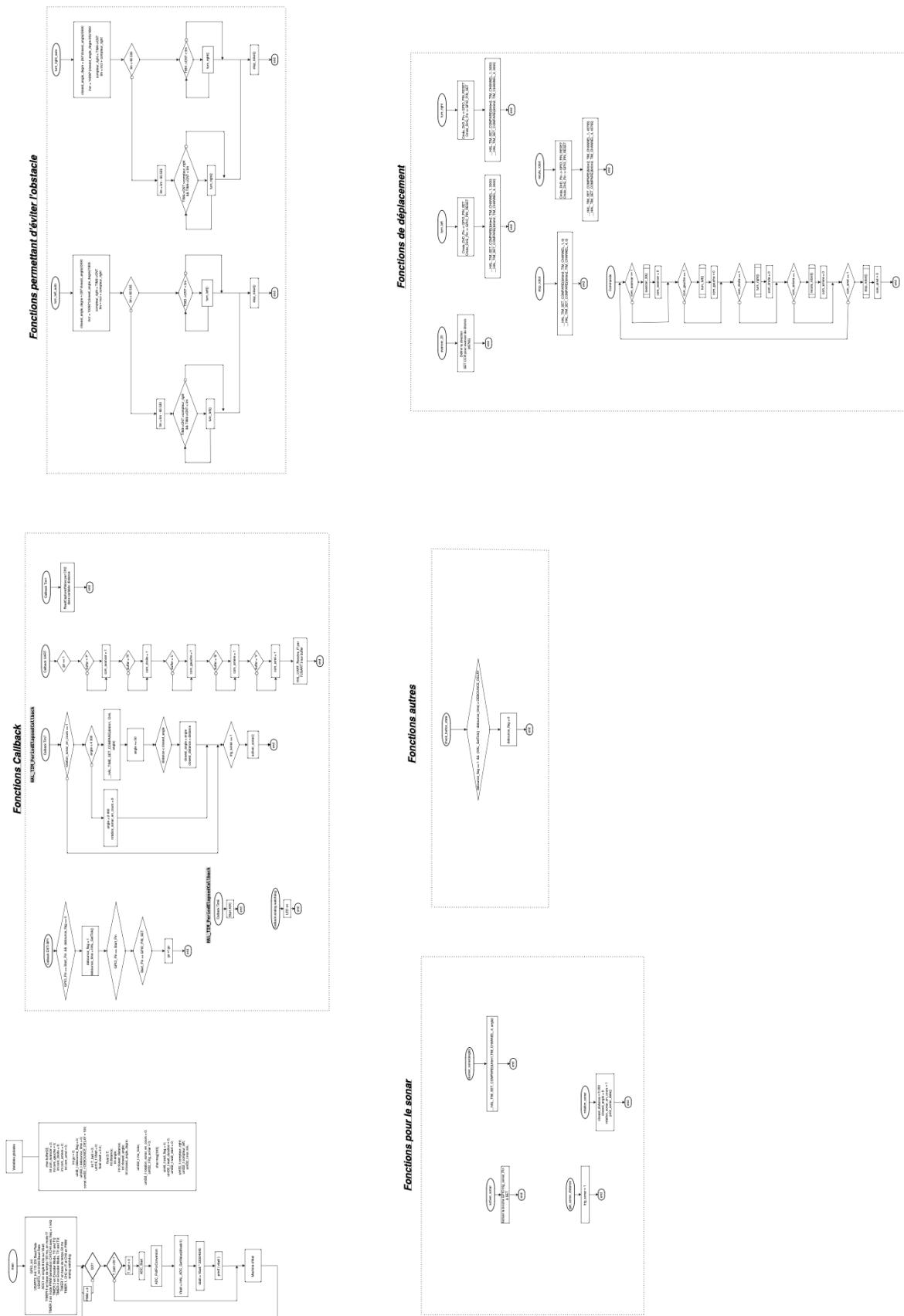


FIGURE 26 – Organigramme complet