

RAPPORT PROJET

**AGONSANOU Lumière
AGOSSA Kewen**

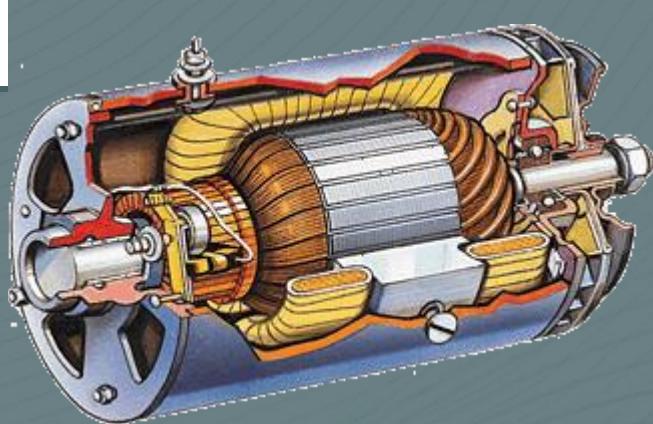
Mars 2025

—
Portage de l'application sur stm32

—
Prof : ARBOUX Valentin

INTRODUCTION

Dans ce projet, nous allons commander un moteur à l'aide d'un signal PWM et implémenter trois timers principaux. Le premier servira à générer le PWM pour piloter le moteur, le second sera dédié à un codeur incrémental afin de mesurer sa position et sa vitesse, et le troisième déclenchera des interruptions périodiques pour l'acquisition de données. Enfin, nous mettrons en place un correcteur PI afin d'asservir le moteur en position et en vitesse.



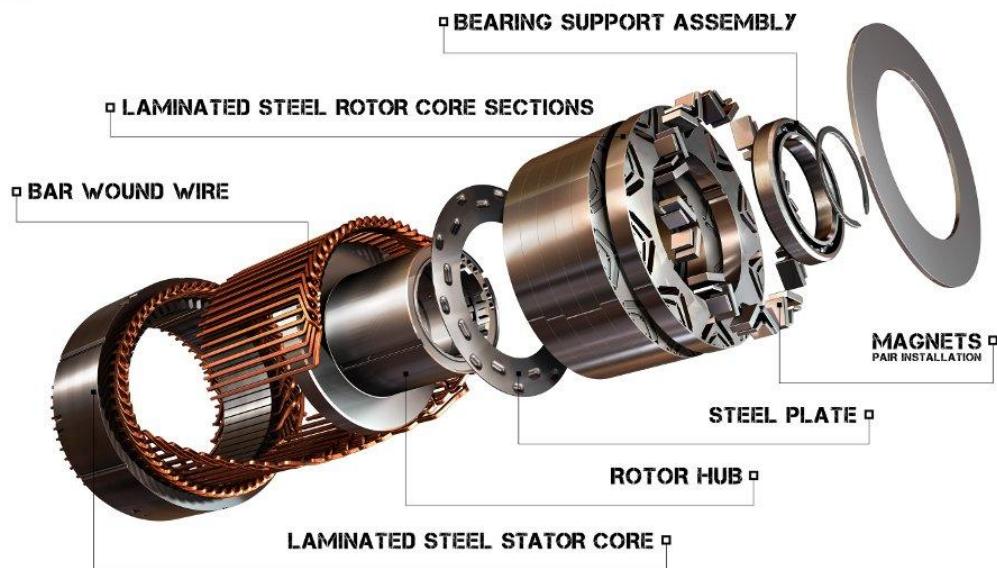


TRAVAIL À FAIRE :

- Faire tourner le moteur à la main sans pwm
 - Générer un pwm avec le timer 1 (sur PA8) pour pouvoir piloter le moteur
 - Monter le moteur et vérifier s'il tourne dans les deux sens
 - Configurer le timer 3 et vérifier s'il déclanche une interruption toutes les 10 ms
 - Lecture de la position avec le codeur incrémental
 - Calculer la vitesse du moteur
 - Implémenter un régulateur PI
 - Tester et valider le fonctionnement avec docklight
-



General Motors Permanent Magnet Electric Motor

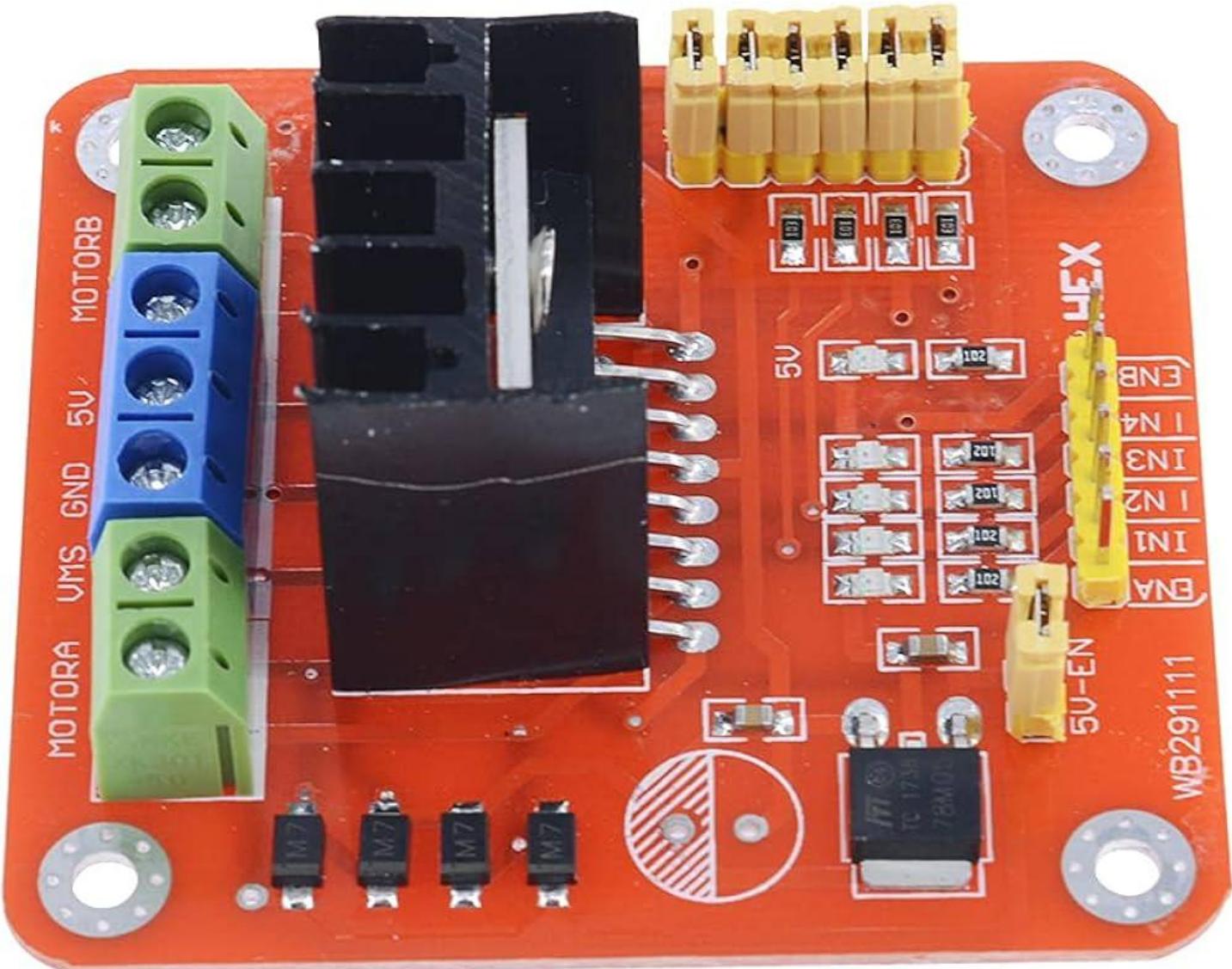


Titre : éclaté d'un moteur mcc

Projet st

Etude de la carte L298 :

La carte mise à notre disposition est un kesyestudio L298N



Il est composé de :

- 2 puces de pilote à double pont H L298N.
- Plage d'alimentation du terminal de la partie moteur VMS : +5 V ~ +35 V
- Courant de crête partiel du pilotage I : 2 A/pont
- La plage d'alimentation de la borne de la partie logique VSS : 4,5-5,5 V.
- Plage de courant de fonctionnement de la partie logique : 0 ~ 36 mA.

Sur cette carte, nous avons remarqué que le UR1 et le UR2 (le deuxième et le troisième cavalier sur le côté) permettent de piloter directement les entrés In1-4. Ainsi, afin d'utiliser ce pont H avec notre nucléo il fallait retirer ces derniers cavaliers (Jumper).

Notre régulateur de puissance ne fonctionnant plus nous avions été contraints de fournir directement du 5V à la carte en reliant VMS à 5V.

Partie 1 : Faire tourner le moteur à la main en lui fournissant un courant continu.

Dans cette partie, nous avons juste monter le moteur sur le pont H, nous lui avons fournis 10V avec l'alimentation et le moteur tombe en roue libre.

Montage :

ENB = 5V

GND = 0V

IN1=cavalier1

IN2=0V car rotation en sens inverse.

Partie 2 : Faire tourner le moteur à l'aide d'un pwm (Timer 1) et vérifier s'il tourne dans les deux sens.

→ Montage L298 vers Nucléo F103 :

ENB = 5V

GND = 0V

IN1=PA8 (Timer 1)

IN2=PA6

En téléversant le code du projet de base, nous avons pu tester la fonction test () ;

Cette fonction permet de générer plusieurs pwm sur plusieurs timer .Ici que le timer 1 nous intéresse donc nous avons tester cette ligne de code :

```
PWM_run(TIMER3_ID, TIM_CHANNEL_1, FALSE, 1000, 50, FALSE);
```

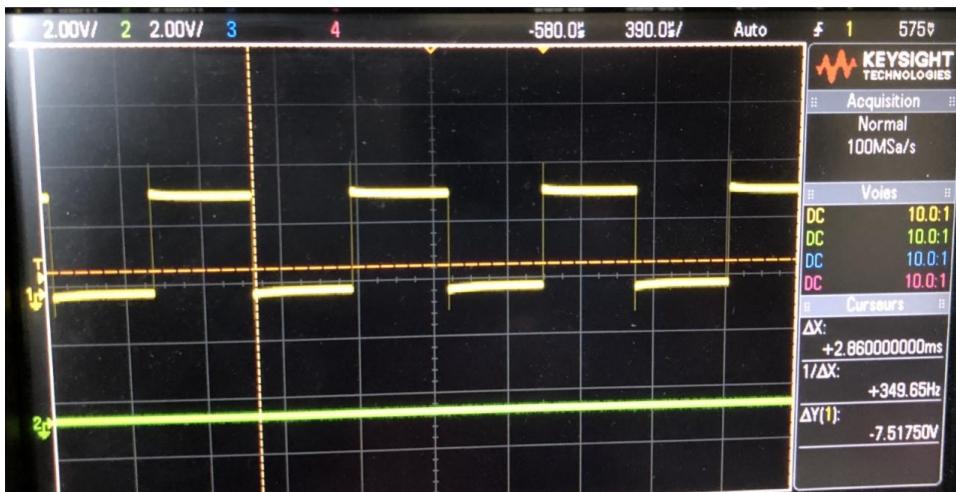
- **TIMER3_ID:** Utilisation du Timer 3.
- **TIM_CHANNEL_1:** Utilisation de la broche **PA6**.
- **1000:** Période de 1ms (1 kHz).
- **50% de duty cycle:** Vitesse moyenne.

Un signal carré de 1 kHz a été observé sur l'oscilloscope.

La fréquence et le duty cycle sont bien contrôlables.

Si le duty cycle varie la vitesse elle aussi varie.

Timer 1	PA6	Effet
PWM	0	Rotation Positif
PWM	3,3	Rotation négative
0%	0	Arrêt



Partie 3 : Validation du traitement de l'interruption (Timer 3) :

- Utiliser Timer 3 pour déclencher une interruption périodique.
- Lire la vitesse du moteur en mesurant les impulsions du codeur incrémental.

Timer 1 → PWM pour le moteur

Timer 3 → Interruptions toutes les 10ms

Afin de générer un timer 3, on utilise la fonction :

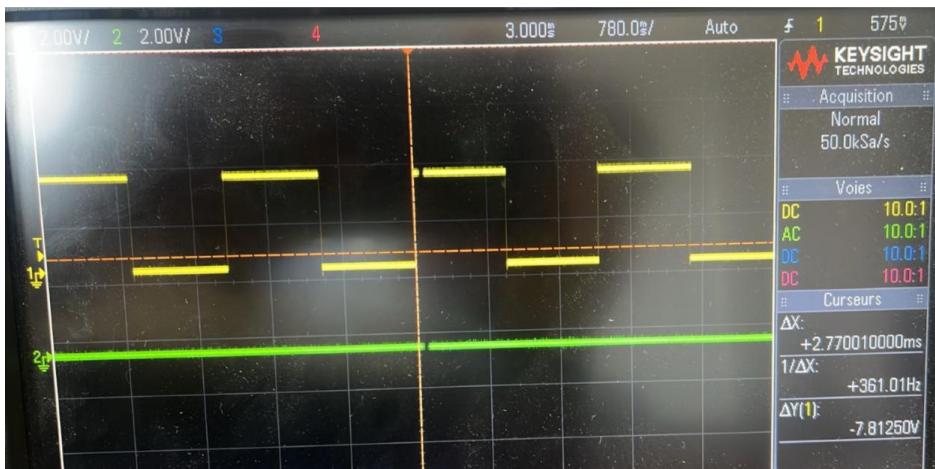
```
TIMER_run_us(TIMER3_ID, 10000, TRUE);
// Changer de sens toutes les 1 secondes (5000ms)
```

Cette fonction arrête le programme chaque 10 ms et rentre dans la tâche de fond. Dans la tâche de fond on inverse la broche pa5 du microcontrôleur et on peut remarquer une impulsion chaque 10 ms.

On n'a pu utiliser la tâche de fond car elle est définie comme Weak.

```
void TIMER3_user_handler_it(void)
{
    HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
}
```

Sur l'oscilloscope, on peut remarquer une interruption chaque 10 ms.



Partie 4 : Mesure de la position avec un codeur incrémental.

Objectif : Faire tourner le moteur dans les deux sens, mesurer sa position avec un codeur incrémental et utiliser un Timer pour générer des interruptions.

Pourquoi utiliser un codeur incrémental ? ☺

- ☺ Le codeur permet de mesurer la rotation du moteur en générant deux signaux en quadrature (A et B).
- ☺ En comptant les impulsions et leur sens, on peut déterminer la position et la vitesse du moteur.

En somme on peut voir le codeur incrémental juste comme une horloge qui compte des ticks chaque seconde. Mais au lieu de mesurer le temps, il mesure **le déplacement angulaire** du moteur.



Color	Function
Red	motor power (connects to one motor terminal)
Black	motor power (connects to the other motor terminal)
Green	encoder GND
Blue	encoder Vcc (3.5 – 20 V)
Yellow	encoder A output
White	encoder B output



Titre : Montage moteur et signaux codeurs

Mode de fonctionnement du codeur :

- En **comptant le nombre de ticks**, on sait **de combien** le moteur a tourné.
- En regardant **le décalage entre A et B**, on sait **dans quel sens** il tourne.

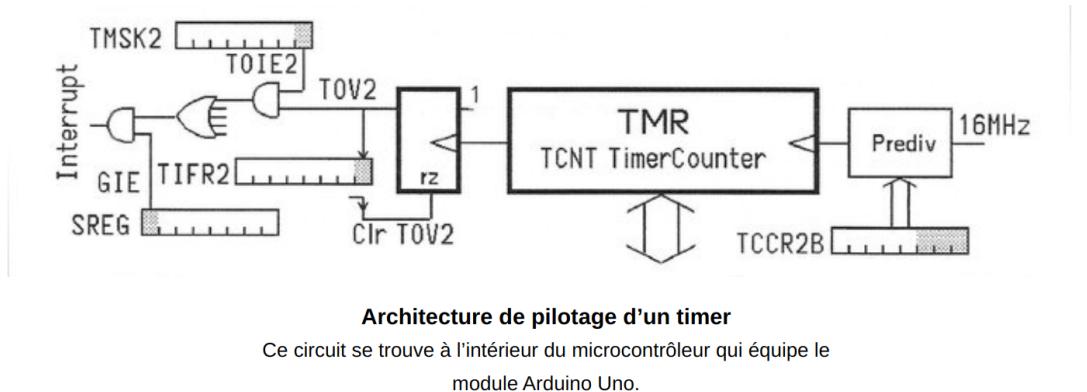
Pourquoi Timer 2 ?

Parce que **Timer 2** sait compter automatiquement ces ticks, sans qu'on ait besoin d'écrire un code compliqué. Il suffit de lire

[_HAL_TIM_GET_COUNTER\(&htim2\)](#) ; pour connaître la position actuelle.
Avec htim2 notre Timer 2.

L'interruption du timer 2

Toutes les demandes d'interruption arrivent sur une porte OU et sont éventuellement bloquées par le signal **GIE** (Global Interrupt Enable) activé par la fonction `sei()`. La figure suivante nous permet de comprendre comment le timer 2 peut faire une demande d'interruption.



Source: <https://www.locoduino.org/spip.php?article88>

Mise en place du codeur incrémental pour mesurer la position :

- **PAo (TIM2_CH1) et PA1 (TIM2_CH2) sont les entrées dédiées pour le mode encodeur sur Timer 2.**
- Ces broches permettent au Timer 2 d'incrémenter ou décrémenter automatiquement en fonction des signaux du codeur.

Mais n'importe quelle broche reliée à un timer aurait fait l'affaire. Nous avons utilisé PAo et PA1 car ils sont reliés directement au timer 2.

- ☺ GPIO PAo et PA1 en mode canal de timer (configuration : voir fonction `void GPIO_Encoder_Init(void)` en fin de document).
- ☺ Les autres paramètres de configurations sont présents dans la fonction `void TIM2_Encoder_Init(void)`. Une grande partie de la fonction nous a été donnée par le sujet. Une analyse approfondie du fichier gpioc.c et gpioc.h nous a inspiré pour la suite.

À savoir :

Définition de la période du compteur

`htim2.Init.Period = 1920 - 1 ;`

Pourquoi 1920 - 1 ?

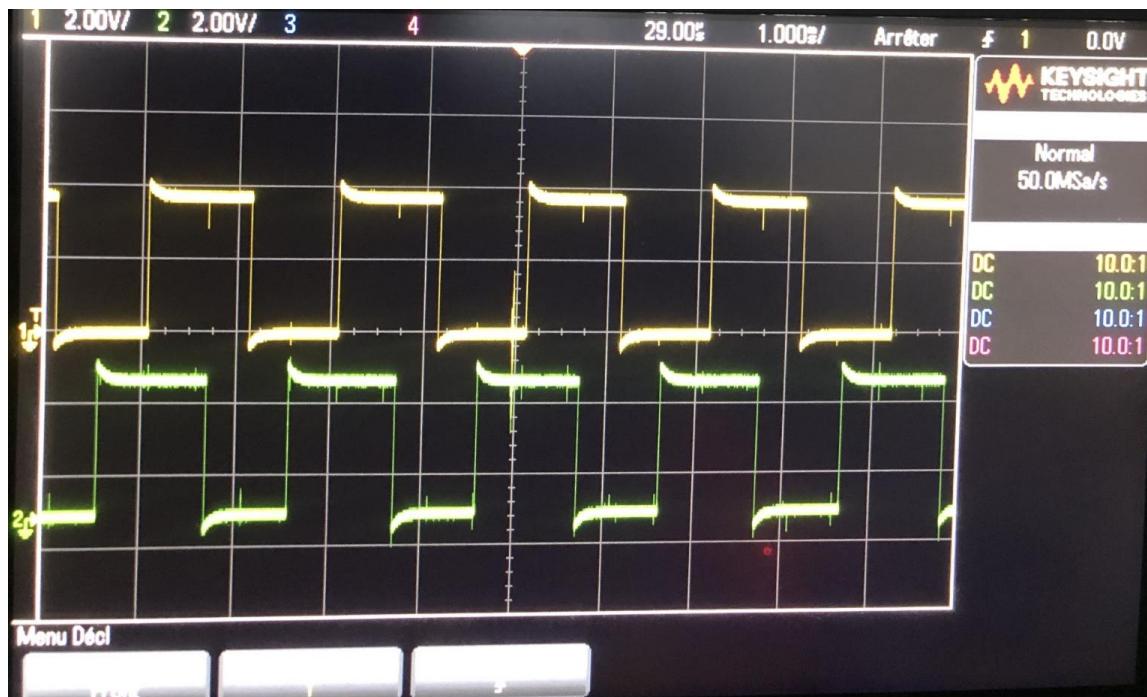
Le codeur génère **1920 impulsions par tour** (30 * 64 selon l'énoncé). TIM2 doit **compter de 0 à 1920-1 (soit 1919), puis revenir à 0**.

Cela permet d'éviter des dépassements de valeur **et de bien suivre les rotations du moteur.**

Lecture de la position :

Nous avons implémenté `uint16_t Lire_Position(void)` qui retourne `_HAL_TIM_GET_COUNTER(&htim2)` ; Cette fonction nous a été donné par l'énoncé elle retourne la position actuelle du moteur.

À l'oscilloscope nous pouvons voir les deux courbes en décalage A et B :



Test : envoie des données séries UART sur docklight :

```
Position moteur : 988<LF>
Position moteur : 1006<LF>
Position moteur : 1022<LF>
Position moteur : 16<LF>
Position moteur : 34nPosition moteur : 578<LF>
Position moteur : 612<LF>
Position moteur : 646<LF>
Position moteur : 679<LF>
Position moteur : 712<LF>
Position moteur : 746<LF>
Position moteur : 781<LF>
Position moteur : 815<LF>
Position moteur : 848<LF>
Position moteur : 881<LF>
Position moteur : 916<LF>
Position moteur : 952<LF>
Position moteur : 987<LF>
Position moteur : 1022<LF>
Position moteur : 36<LF>
Position moteur : 76<LF>
Position moteur : 116<LF>
Position moteur : 154<LF>
Position moteur : 191<LF>
Position moteur : 228<LF>
Position moteur : 264<LF>
Position moteur : 299<LF>
Position moteur : 335<LF>
Position moteur : 371<LF>
Position moteur : 406<LF>
Position moteur : 442<LF>
Position moteur : 479<LF>
Position moteur : 515<LF>
Position moteur : 553<LF>
Position moteur : 591<LF>
Position moteur : 629<LF>
Position moteur : 666<LF>
```

Implémentation du calcul de la vitesse

L'objectif est d'utiliser **les interruptions du timer 3 (toutes les 10 ms)** pour :

- ☺ Lire la position actuelle du codeur.
- ☺ Calculer la différence de position par rapport à la précédente mesure.
- ☺ Convertir cette différence en vitesse en tr/s.
En somme on utilise la formule classique vitesse = distance sur temps

Formule :

- Si N est le nombre d'impulsions comptées en 0.01 s, la vitesse est (en rad/s, l'unité standard, sachant qu'un tour fait 2π radians) : $2\pi N / (0.01 \cdot 1920)$

Cette formule nous permet de déterminer la vitesse en rad/s nous la convertissons en suite en trs/sec.

Mais comment l'implémenter ?

- ✓ D'abord, On va stocker la position précédente du codeur pour pouvoir calculer la différence avec la nouvelle position à chaque interruption.
Un volatil est appréciable car la position va être constamment modifiée lors de chaque interruption.
- ✓ En deuxième position nous allons utiliser le timer3 pour calculer le delta entre les positions chaque 10 ms.
- ✓ En troisième position nous allons utiliser la formule de la vitesse.
- ✓ Et en dernière position nous allons afficher la vitesse grâce à docklight car il ne faut jamais mettre un print dans la tâche de fond cela est interdit pour cause de vitesse d'interruption.

Code : Voir la fonction en fin de document

`void CalculeVitesse(void)`

Explication :

`int32_t current_position = Lire_Position();`

Me permet de lire la position actuelle

`int32_t delta_position = current_position - last_position;`

`delta_position` représente **combien de ticks le moteur a parcourus en 10 ms.**

Si `delta_position` est positif → le moteur tourne dans un sens.

Si `delta_position` est négatif → le moteur tourne dans l'autre sens.

`last_position = current_position;`

On met à jour `last_position` pour la prochaine interruption.

Gestion du débordement du compteur

```
if (delta_position > 960) delta_position -= 1920;  
else if (delta_position < -960) delta_position += 1920;
```

Le **Timer 2** fonctionne sur **1920 ticks par tour**.

Si le compteur dépasse **960 ticks** (moitié d'un tour) :

- ☺ **On soustrait 1920** → Cela corrige un éventuel dépassement.
Si le compteur passe en négatif sous -960 :
- ☺ **On ajoute 1920** → Cela corrige un retour à zéro mal interprété.

Exemple pratique :

Dépassement du compteur sens 1 :

Last Position	Current Position	Delta (Avant Correction)	Delta (Après Correction)
1910	1920	$1920 - 1910 = 10$	10
1920	5	$5 - 1920 = -1915$ (pas bon)	$-1915 + 1920 = 5$
5	20	15	15

```
if (delta_position < -960) delta_position += 1920;
```

Si delta_position est trop négatif, on le ramène à une valeur correcte en ajoutant 1920.

Dépassement inverse (Retour à 1920)

Last Position	Current Position	Delta (Avant Correction)	Delta (Après Correction)
20	10	$10 - 20 = -10$	-10
10	1910	$1910 - 10 = 1900$ (pas bon)	$1900 - 1920 = -20$

1910	1900	1900 - 1910 = -10	-10
------	------	-------------------	-----

Observation sur la terminale série docklight

Sens négatif :

```
Position moteur : 1611<LF>
Vitesse : -6.22 rad/s, Direction : Arrière<LF>
Delta : -19<LF>
Position moteur : 647<LF>
Vitesse : -6.22 rad/s, Direction : Arrière<LF>
Delta : -18<LF>
Position moteur : 1603<LF>
Vitesse : -5.89 rad/s, Direction : Arrière<LF>
Delta : -19<LF>
Position moteur : 639<LF>
Vitesse : -6.22 rad/s, Direction : Arrière<LF>
Delta : -19<LF>
Position moteur : 1594<LF>
Vitesse : -6.22 rad/s, Direction : Arrière<LF>
Delta : -19<LF>
Position moteur : 632<LF>
Vitesse : -6.22 rad/s, Direction : Arrière<LF>
Delta : -19<LF>
Position moteur : 1587<LF>
Vitesse : -6.22 rad/s, Direction : Arrière<LF>
Delta : -19<LF>
Position moteur : 621<LF>
Vitesse : -6.22 rad/s, Direction : Arrière<LF>
Delta : -19<LF>
Position moteur : 1576<LF>
Vitesse : -6.22 rad/s, Direction : Arrière<LF>
Delta : -19<LF>
Position moteur : 612<LF>
Vitesse : -6.22 rad/s, Direction : Arrière<LF>
Delta : -19<LF>
Position moteur : 1566<LF>
Vitesse : -6.22 rad/s, Direction : Arrière<LF>
Delta : -19<LF>
Position moteur : 602<LF>
Vitesse : -6.22 rad/s, Direction : Arrière<LF>
```

Sens Positif :

```
Position moteur : 554<LF>
Vitesse : 6.22 rad/s, Direction : Avant<LF>
Delta : 18<LF>
Position moteur : 1502<LF>
Vitesse : 6.22 rad/s, Direction : Avant<LF>
Delta : 19<LF>
Position moteur : 530<LF>
Vitesse : 6.22 rad/s, Direction : Avant<LF>
Delta : 19<LF>
Position moteur : 1482<LF>
Vitesse : 6.22 rad/s, Direction : Avant<LF>
Delta : 19<LF>
Position moteur : 514<LF>
Vitesse : 6.22 rad/s, Direction : Avant<LF>
Delta : 19<LF>
Position moteur : 1468<LF>
Vitesse : 6.22 rad/s, Direction : Avant<LF>
Delta : 19<LF>
Position moteur : 504<LF>
Vitesse : 5.89 rad/s, Direction : Avant<LF>
Delta : 19<LF>
Position moteur : 1453<LF>
Vitesse : 6.22 rad/s, Direction : Avant<LF>
Delta : 19<LF>
Position moteur : 479<LF>
Vitesse : 6.22 rad/s, Direction : Avant<LF>
Delta : 18<LF>
Position moteur : 1422<LF>
Vitesse : 5.89 rad/s, Direction : Avant<LF>
Delta : 19<LF>
Position moteur : 447<LF>
Vitesse : 6.22 rad/s, Direction : Avant<LF>
Delta : 18<LF>
Position moteur : 1394<LF>
Vitesse : 6.22 rad/s, Direction : Avant<LF>
```

Conclusion :

Sens 1 : On peut remarquer la position du moteur qui va en décrémentant le delta qui est négatif ce qui fait qu'on a une vitesse négative.

Sens 2 : Contraire au sens 1.

Implémentation d'un correcteur PI pour corriger l'erreur :

Code : Voir [void CorrecteurPI\(void\)](#) en fin de document.

Fonctionnement :

Le correcteur PI (Proportionnel-Integral) est utilisé pour ajuster dynamiquement la commande moteur afin de minimiser l'erreur entre la consigne de vitesse et la vitesse mesurée. Dans notre projet, il permet de stabiliser et de réguler précisément la vitesse du moteur.

Il se compose de deux termes :

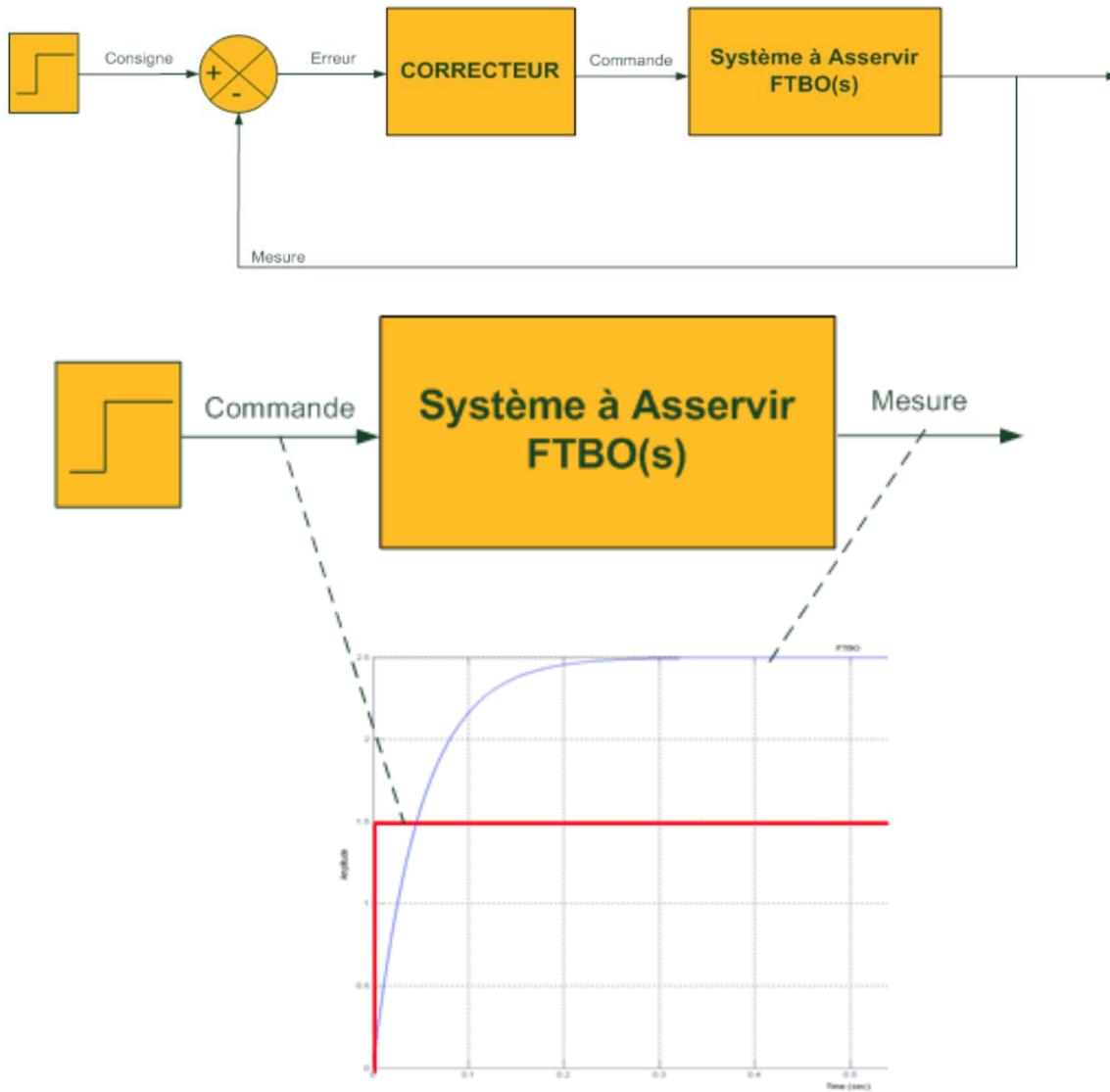
- **Le terme proportionnel (P)** réagit instantanément à l'erreur actuelle : plus l'erreur est grande, plus la correction appliquée est forte.
- **Le terme intégral (I)** cumule les erreurs dans le temps pour corriger les écarts persistants, supprimant ainsi l'erreur statique.

Le signal de commande est donc calculé selon la formule :

$$u(t) = K_p \times e(t) + K_i \times \int e(t) dt$$

où $e(t)$ est l'erreur entre la consigne et la mesure, K_p le gain proportionnel et K_i le gain intégral.

Ce correcteur a été implémenté dans une boucle d'asservissement périodique, avec une mise à jour de la commande moteur toutes les 100 ms, en s'appuyant sur le calcul de la vitesse instantanée issue du retour encodeur.



source: <https://web.enib.fr/~kerhoas/automatique-robotique/correcteur-pi/>

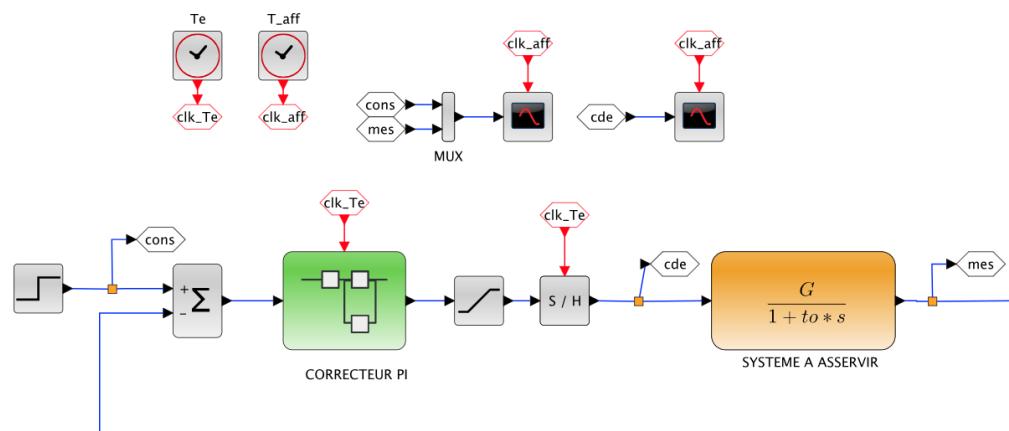
- **Stabilité** : respect d'une marge de phase à déterminer (30 degrés par exemple)

- **Précision** : La mesure rejoint-elle la consigne ?
- **Rapidité** : Un correcteur PI possède une intégration, ce qui permet de régler le problème de l'erreur statique. En contrepartie il ralentit le système, on veillera donc à respecter une marge de phase suffisante mais pas trop grande non plus (plus la marge de phase est grande, plus la réponse est lente et les risques de saturation augmentent). (“Vincent Kerhoas - Réglage (Rapide) d'un Correcteur PI”)

Mise en œuvre du correcteur PI :

Nous avons défini 04 variables l'erreur et l'erreur intégrale nous ont été donné par notre enseignant. En temps normal, il faut les chercher avec Matlab en utilisant le croquis suivant :

sys_corr.zcos
contexte.sce



source: <https://web.enib.fr/~kerhoas/automatique-robotique/correcteur-pi/>

```
#define K_P 0.5 // Gain proportionnel (ajuste rapidement l'erre
#define K_I 10 // Gain intégral (compense les erreurs statiques
#define U_MAX 100 // Limite max du PWM
#define U_MIN -100 // Limite min du PWM
```

☺ **Calcule de l'erreur :**

Nous avons fait la différence entre une consigne qu'on définit nous-même par exemple 5 trs/sec et la vitesse.

☺ **Calcule de l'erreur intégrale :**

Nous avons fait la somme des erreurs.

☺ **Limitation :**

Nous limitions l'erreur intégrale entre 100 et -100 car si l'erreur devient trop grande on arrive plus à corriger.

☺ **Calcule :** utilisation de la formule

Après calcul de la commande nous limitons la commande entre 10 et -10 pour éviter sa saturation.

Application :

Nous appelons la fonction CorrecteurPi dans l'interruption du timer3 ainsi il s'exécute chaque 10 ms quand le timer est déclenché.

Visualisation sur le terminal série :

```
Erreurs : -0.20 | Erreur intégrale : 11.69 | U : 1.07<LF>
Position moteur : 1791<LF>
Vitesse : 1.20 tr/s, Direction : Avant<LF>
Erreurs : -0.20 | Erreur intégrale : 2.01 | U : 0.10<LF>
Position moteur : 1041<LF>
Vitesse : 1.20 tr/s, Direction : Avant<LF>
Erreurs : 0.01 | Erreur intégrale : 0.20 | U : 0.03<LF>
Position moteur : 136<LF>
Vitesse : 1.04 tr/s, Direction : Avant<LF>
Erreurs : 0.01 | Erreur intégrale : 0.21 | U : 0.03<LF>
Position moteur : 1117<LF>
Vitesse : 1.04 tr/s, Direction : Avant<LF>
Erreurs : -0.04 | Erreur intégrale : 0.11 | U : -0.01<LF>
Position moteur : 181<LF>
Vitesse : 0.99 tr/s, Direction : Avant<LF>
Erreurs : 0.01 | Erreur intégrale : 0.14 | U : 0.02<LF>
Position moteur : 1163<LF>
Vitesse : 0.99 tr/s, Direction : Avant<LF>
Erreurs : 0.01 | Erreur intégrale : 0.20 | U : 0.03<LF>
Position moteur : 225<LF>
Vitesse : 0.99 tr/s, Direction : Avant<LF>
Erreurs : 0.01 | Erreur intégrale : 0.10 | U : 0.02<LF>
Position moteur : 1207<LF>
Vitesse : 0.99 tr/s, Direction : Avant<LF>
Erreurs : 0.06 | Erreur intégrale : 0.11 | U : 0.04<LF>
Position moteur : 271<LF>
Vitesse : 0.99 tr/s, Direction : Avant<LF>
Erreurs : 0.01 | Erreur intégrale : 0.18 | U : 0.02<LF>
Position moteur : 1252<LF>
Vitesse : 1.04 tr/s, Direction : Avant<LF>
Erreurs : 0.01 | Erreur intégrale : 0.14 | U : 0.02<LF>
Position moteur : 314<LF>
Vitesse : 0.94 tr/s, Direction : Avant<LF>
Erreurs : -0.04 | Erreur intégrale : 0.10 | U : -0.01<LF>
Position moteur : 1297<LF>
Vitesse : 1.04 tr/s, Direction : Avant<LF>
```

Conclusion :

Notre moteur est effectivement asservi en vitesse si je freine mon moteur en sachant par exemple que je suis à une consigne de 2 trs/sec au lieu que la vitesse baisse, le moteur force ce qui fait que je reste toujours à 2 trs/sec.

Asservissement position :

Objectif : Piloter précisément le moteur afin qu'il atteigne et maintienne une position angulaire cible, définie par une consigne. Il s'agit de contrôler le déplacement du moteur en minimisant l'erreur entre la position réelle mesurée (via l'encodeur) et la position désirée, tout en assurant une trajectoire stable, rapide et sans dépassement excessif.

Conclusion :

Ce projet a été particulièrement formateur et complet. Il nous a permis d'apprendre et de mettre en œuvre plusieurs aspects essentiels du contrôle moteur :

- ✓ Génération du signal PWM avec le Timer 1 pour piloter le moteur
- ✓ Montage du moteur et vérification du sens de rotation
- ✓ Configuration du Timer 3 pour générer une interruption toutes les 10 ms (validée à l'oscilloscope)
- ✓ Lecture de la position du moteur via le codeur incrémental, avec affichage dans Docklight

Nous avons ensuite calculé la **vitesse du moteur** :

- → À partir des valeurs du compteur du codeur, en calculant la différence de position entre deux interruptions
- → Conversion de cette variation en **tours par seconde (tr/s)**

Enfin, nous avons implémenté un **correcteur PI** :

- → Régulation de la vitesse moteur avec un contrôleur proportionnel-intégral
- → Ajustement des gains Kp et Ki pour une réponse stable et rapide
- → Application de la commande PI sur le PWM pour piloter efficacement le moteur

Nous avons ainsi réalisé avec succès un **asservissement moteur en vitesse**, ouvrant la voie vers un asservissement en position .

Code source :

```
/*
*****
* @file main.c
* @author Ac6
* @version V1.0
* @date 01-December-2013
* @brief Default main function.
*****
*/
#include "stm32f1xx_hal.h"
#include "stm32f1xx_hal_tim.h"
#include "stm32f1xx_nucleo.h"
#include "stm32f1_uart.h"
#include "stm32f1_sys.h"
#include "macro_types.h"
#include "stm32f1_gpio.h"
#include "stm32f1_timer.h"
#include "test.h"
#include <stdint.h>
#include <math.h>

TIM_HandleTypeDef htim3;
TIM_HandleTypeDef htim2;

volatile int32_t last_position = 0;
volatile float speed = 0;
volatile int32_t debug_delta = 0;
float erreur;
int8_t rotation_direction;
```

```

volatile float erreur_integrale = 0;
float U = 0;

int32_t Lire_Position(void); // Déclaration explicite
int8_t Lire_Direction(void);
int16_t Correcteur_PI(float consigne, float mesure);

int main(void)
{
// Initialisation de la couche logicielle HAL (Hardware Abstraction Layer)
// Cette ligne doit rester la première étape de la fonction main().
HAL_Init();

// Initialisation du moteur et de PA8 pour le sens

// Initialisation de l'UART2 à la vitesse de 115200 bauds/secondes (92kbits/s) PA2 : Tx | PA3 : Rx.
// Attention, les pins PA2 et PA3 ne sont pas reliées jusqu'au connecteur de la Nucleo.
// Ces broches sont redirigées vers la sonde de débogage, la liaison UART étant ensuite encapsulée sur l'USB vers le PC
// de développement.
UART_init(UART2_ID,115200);

// Indique que les printf sortent vers le périphérique UART2."
SYS_set_std_usart(UART2_ID, UART2_ID, UART2_ID);

// Pour le timer 2
GPIO_Encoder_Init();
TIM2_Encoder_Init();

// Initialisation du port de la led Verte (carte Nucleo)
BSP_GPIO_PinCfg(LED_GREEN, GPIO_MODE_OUTPUT_PP, GPIO_NOPULL, GPIO_SPEED_FREQ_HIGH);

// Initialisation du port du bouton bleu (carte Nucleo)
BSP_GPIO_PinCfg(BLUE_BUTTON, GPIO_MODE_INPUT, GPIO_PULLUP, GPIO_SPEED_FREQ_HIGH);

// test();

BSP_GPIO_PinCfg(GPIOA, GPIO_PIN_6, GPIO_MODE_OUTPUT_PP, GPIO_NOPULL, GPIO_SPEED_FREQ_HIGH);
// HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_RESET); // PA8 = 3.3V
// PWM_run(TIMER1_ID, TIM_CHANNEL_1, FALSE, 1000, 50, FALSE); // Timer 1 - Canal 1 Positive (1.CH1) -> PA8

TIMER_run_us(TIMER3_ID, 10000, TRUE); // Changer de sens toutes les 10 millisecondes (5000ms)

while (1)
{
// Affichage pour debug

```

```

printf("Erreur : %.2f | Erreur integrale : %.2f | U : %.2f\n", erreur, erreur_integrale, U);

int32_t position = Lire_Position();

// printf(" Delta : %ld\n", debug_delta);
printf("Position moteur : %ld\n", position);
printf("Vitesse : %.2f tr/s, Direction : %s\n", speed, (rotation_direction == 1) ? "Avant" : "Arriere");
HAL_Delay(500);
}

}

void Test_Moteur(void)
{
// Sens 1 - Augmenter le PWM progressivement
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_SET); // PA6 = 3.3V (Sens 1)
PWM_run(TIMER1_ID, TIM_CHANNEL_1, FALSE, 1000, 20, FALSE);
HAL_Delay(3000);

PWM_run(TIMER1_ID, TIM_CHANNEL_1, FALSE, 1000, 50, FALSE);
HAL_Delay(3000);

PWM_run(TIMER1_ID, TIM_CHANNEL_1, FALSE, 1000, 80, FALSE);
HAL_Delay(3000);

// Stop moteur avant d'inverser le sens
PWM_run(TIMER1_ID, TIM_CHANNEL_1, FALSE, 1000, 0, FALSE);
HAL_Delay(2000);

// Sens 2 - Augmenter le PWM progressivement
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_RESET); // PA6 = 0V (Sens 2)
PWM_run(TIMER1_ID, TIM_CHANNEL_1, FALSE, 1000, 20, FALSE);
HAL_Delay(3000);

PWM_run(TIMER1_ID, TIM_CHANNEL_1, FALSE, 1000, 50, FALSE);
HAL_Delay(3000);

PWM_run(TIMER1_ID, TIM_CHANNEL_1, FALSE, 1000, 80, FALSE);
HAL_Delay(3000);

// Stop moteur
PWM_run(TIMER1_ID, TIM_CHANNEL_1, FALSE, 1000, 0, FALSE);
HAL_Delay(2000);
}

void TIMER3_user_handler_it(void)
{
HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
}

```

```

CalculeVitesse ();
CorrecteurPI();
}

void GPIO_Encoder_Init(void)
{
__HAL_RCC_GPIOA_CLK_ENABLE(); // Activer l'horloge pour GPIOA

GPIO_InitTypeDef GPIO_InitStruct = {0};

GPIO_InitStruct.Pin = GPIO_PIN_0 | GPIO_PIN_1; // TIM2_CH1 = PA0, TIM2_CH2 = PA1
GPIO_InitStruct.Mode = GPIO_MODE_AF_INPUT; // Mode Alternate Function Input
GPIO_InitStruct.Pull = GPIO_NOPULL;
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;

HAL_GPIO_Init(GPIOA, &GPIO_InitStruct); // Appliquer la config
}

void TIM2_Encoder_Init(void)
{
__HAL_RCC_TIM2_CLK_ENABLE(); // Active l'horloge de Timer 2

TIM_Encoder_InitTypeDef encoderConfig = {0};

htim2.Instance = TIM2;
htim2.Init.Prescaler = 0; // Pas de prédivision (compte à la fréquence d'entrée)
htim2.Init.CounterMode = TIM_COUNTERMODE_UP;
htim2.Init.Period = 1920 - 1; // Ajuste selon ton codeur (ex: 1024 impulsions par tour)
htim2.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;

// Configurer le mode encodeur
encoderConfig.EncoderMode = TIM_ENCODERMODE_TI12; // Utilise les 2 canaux
encoderConfig.IC1Polarity = TIM_ICPOLARITY_BOTHEDGE;
encoderConfig.IC1Selection = TIM_ICSELECTION_DIRECTTI;
encoderConfig.IC1Prescaler = TIM_ICPSC_DIV1;
encoderConfig.IC1Filter = 0x00;
encoderConfig.IC2Polarity = TIM_ICPOLARITY_BOTHEDGE;
encoderConfig.IC2Selection = TIM_ICSELECTION_DIRECTTI;
encoderConfig.IC2Prescaler = TIM_ICPSC_DIV1;
encoderConfig.IC2Filter = 0x00;

HAL_TIM_Encoder_Init(&htim2, &encoderConfig);
HAL_TIM_Encoder_Start(&htim2, TIM_CHANNEL_ALL);
}

int32_t Lire_Position(void)
{
return (int32_t)_HAL_TIM_GET_COUNTER(&htim2);
}

```

```

}

int8_t Lire_Direction(void)
{
return (_HAL_TIM_IS_TIM_COUNTING_DOWN(&htim2)) ? -1 : 1;
}

void CalculeVitesse (void)
{
int32_t current_position = Lire_Position();
int32_t delta_position = current_position - last_position;
rotation_direction = Lire_Direction();

// Gestion du débordement
if (rotation_direction == 1 && delta_position < -960)
{
delta_position += 1920;
}
else if (rotation_direction == -1 && delta_position > 960)
{
delta_position -= 1920;
}

last_position = current_position;

speed = (float)delta_position / (1920.0f * 0.01f); // Vitesse en tr/s
debug_delta = delta_position;
}

// Paramètres du correcteur PI
#define K_P 0.5 // Gain proportionnel (ajuste rapidement l'erreur)
#define K_I 10 // Gain intégral (compense les erreurs statiques)
#define U_MAX 100 // Limite max du PWM
#define U_MIN -100 // Limite min du PWM

void CorrecteurPI (void)
{
int consigne = 1;
// --- Correcteur PI ---
erreur = consigne - speed;
erreur_integrale += erreur;

// Limitation de l'intégrale (anti-windup)
if (erreur_integrale > 100) erreur_integrale = 100;
if (erreur_integrale < -100) erreur_integrale = -100;

float U_temp = (K_P * erreur) + (K_I * erreur_integrale * 0.01);
}

```

```
// Saturation de la commande
if (U_temp > 10) U_temp = 10;
if (U_temp < -10) U_temp = -10;

U = U_temp;

if (U >= 0)
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_SET); // Sens 1
else
HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_RESET); // Sens inverse

PWM_run(TIMER1_ID, TIM_CHANNEL_1, FALSE, 1000, U * 10, FALSE);
}
```