

μ-verCraft-II



Projet PIST 2025/26

Leon VOLLES

Luis GÖBES

Professeur Piat

Table des matières

1	Introduction.....	9
1.1	L'idée principale des aéroglissoirs.....	9
1.2	Different types des aéroglissoirs/hovercrafts	9
1.3	Objectif à long terme du projet.....	10
2	Cahier des charges (CdC).....	11
2.1	Cahier des charges fonctionnel	11
2.2	Cahier des charges technique / détaillées	12
2.3	Choix de type d'hovercraft	12
3	Composantes principales	13
3.1	Choix des composantes	13
3.1.1	ESP32-S3	13
3.1.2	ESC	13
3.1.3	Moteurs	14
3.1.4	Hélices	15
3.1.5	Batterie	15
3.1.6	IMU/10DOF	16
3.1.7	IR	16
3.1.8	Materiaux pour le chassis.....	17
3.2	Commande	17
3.2.1	Liste des composantes/commande.....	17
3.2.2	Fournisseurs principaux.....	17
3.3	CAO	18
4	Construction	19
4.1	Fabrication du châssis en atelier	19
4.1.1	Découpage laser	19
4.2	Connections électrique.....	21
4.2.1	GPIO.....	21
4.2.2	Schéma électrique	22
4.2.3	Longeurs des cables.....	22
4.3	Assemblage.....	23
4.3.1	Soudage	23
4.3.2	Coller les pièces	23
4.3.3	Assemblage finale.....	24
4.4	Calibration du boussole/magnetometer	25

4.4.1	Protocole de Calibration.....	25
4.5	Paramètres pour l'ESC.....	26
5	Validation des composantes	28
5.1	Test des moteurs	28
5.2	Capteur 10 DOF mems (ACCEL + GYRO + MAG + ENV)	28
5.2.1	Validation des capteurs	28
5.3	Validation du filtre complémentaire	30
6	Programmation	32
6.1	Modes de pilotage.....	32
6.1.1	Structure « commande manuel ».....	32
6.1.2	Structure « commande autonome »	33
6.2	RTOS (FreeRTOS on ESP32)	34
6.2.1	For people that are new to multi-threading	34
6.2.2	Main idea.....	34
6.2.3	What an RTOS is (and why we use it).....	34
6.2.4	Two cores on the ESP32 (what that means).....	34
6.2.5	Scheduling (very short reminder).....	34
6.2.6	RTOS building blocks used in this project (high level).....	34
6.3	Hovercraft_variables (global project settings)	35
6.3.1	Main idea	35
6.3.2	Where it is used / how it is called	35
6.3.3	Define all IO-Pins	35
6.3.4	Battery related limits and variables	35
6.3.5	Motor control variables.....	36
6.3.6	Web piloting / UI presets	37
6.3.7	Wifi SSID, PW, IP Adresses.....	37
6.3.8	Gyro/IMU/Complementary filter settings.....	37
6.3.9	Control loop constants	37
6.3.10	PID controller constants	38
6.4	Main (application entry point)	39
6.4.1	Main idea	39
6.4.2	Where it is used / how it is called	39
6.4.3	Project structure: what runs when.....	39
6.4.4	Functionalities implemented (project-level).....	41
6.4.5	Methods / functions overview	41
6.4.6	Parameters used (configuration & tuning).....	41

6.4.7	Schematic	42
6.5	Motor control (DShot ESC output)	43
6.5.1	Main idea	43
6.5.2	Where it is used / how it is called	43
6.5.3	Functionalities implemented.....	43
6.5.4	Methods (overview)	43
6.5.5	Parameters used.....	44
6.6	Motor mixer (lift + thrust + yaw differential)	45
6.6.1	Main idea.....	45
6.6.2	Where it is used / how it is called	45
6.6.3	Functionalities implemented.....	45
6.6.4	Methods (overview)	45
6.6.5	Parameters used.....	45
6.6.6	Schematic	46
6.7	IMU (ADXL345+ITG3205+QMC5883+BMP280) + yaw estimation	47
6.7.1	Main idea.....	47
6.7.2	Where it is used / how it is called	47
6.7.3	Functionalities implemented.....	47
6.7.4	Heading convention / mounting correction (important)	47
6.7.5	Methods (overview)	48
6.7.6	Parameters used.....	48
6.7.7	Schematic	48
6.8	PID controller (generic, rate-control oriented)	49
6.8.1	Main idea.....	49
6.8.2	Where it is used / how it is called	49
6.8.3	Functionalities implemented.....	49
6.8.4	Methods (overview)	49
6.8.5	Parameters used.....	49
6.8.6	Schematic	50
6.9	Battery monitor (ADC voltage + current + used capacity)	51
6.9.1	Main idea	51
6.9.2	Where it is used / how it is called	51
6.9.3	Functionalities implemented.....	51
6.9.4	Methods (overview)	51
6.9.5	Parameters used.....	51
6.9.6	Schematic	52

6.10	WiFi manager	53
6.10.1	Main idea.....	53
6.10.2	Where it is used / how it is called	53
6.10.3	Functionalities implemented.....	53
6.10.4	Methods (overview)	53
6.10.5	Parameters used.....	53
6.11	Network piloting (Web UI + WebSocket control/telemetry)	54
6.11.1	Main idea.....	54
6.11.2	Where it is used / how it is called	54
6.11.3	Functionalities implemented.....	54
6.11.4	Methods (overview)	55
6.11.5	Parameters used.....	55
6.11.6	Schematic	56
6.12	IR sensors (line detection).....	57
6.12.1	Main idea.....	57
6.12.2	Where it is used / how it is called	57
6.12.3	Functionalities implemented (global view)	57
6.12.4	How the algorithm works (conceptual).....	57
6.12.5	Methods overview.....	58
6.12.6	Parameters used (internal + extern)	58
6.13	Heading controller (outer loop for cascaded yaw control)	60
6.13.1	Very important note	60
6.13.2	Note about autonomous mode (M-Mode)	60
6.13.3	Where it is used / how it is called	60
6.13.4	Functionalities implemented.....	60
6.13.5	Configuration parameters	60
6.13.6	Notes / assumptions.....	60
6.14	Autonomous_sequence (M-Mode Sequencer).....	62
6.14.1	Purpose.....	62
6.14.2	Integration (where it runs)	62
6.14.3	Public interface.....	62
6.14.4	Inputs (to update()).....	63
6.14.5	Outputs (polled by the caller).....	63
6.14.6	State machine (current implementation).....	64
6.14.7	Stop/abort behavior and safety	66
6.14.8	Implementation notes / gotchas	66

6.14.9	Files.....	66
7	Validation avec CdC.....	67
7.1	Dimensions	67
7.2	Poids final	67
7.3	Batterie	68
7.4	Stabilité angulaire.....	68
7.5	Positionnement angulaire	68
7.6	CdC fonctionnel avec validation	69
7.7	CdC technique avec validation	69
7.8	Autres réussites	71
8	Manuel d'utilisation.....	72
8.1	Utiliser le μ -verCraft-II.....	72
8.2	Charger les batteries	72
8.3	Adoption de la commande	72
9	Apprentissages principaux	74
10	Conseils pour commencer comme nouveau groupe	75
10.1	Télécharger le projet de GitHub	75
10.2	VS-Code/Cursor IDE.....	75
10.3	Platform IO	75
10.4	Git / GitHub	75
10.5	GitHub Copilot/Cursor /Google Antigravity	76
10.6	Calibration nécessaire	76
10.7	Lisez notre rapport, nous y avons travaillé fortement pour vous aider.....	76
11	Tâches ouvertes et idées pour la continuation	77

1 Introduction

1.1 L'idée principale des aéroglisseurs

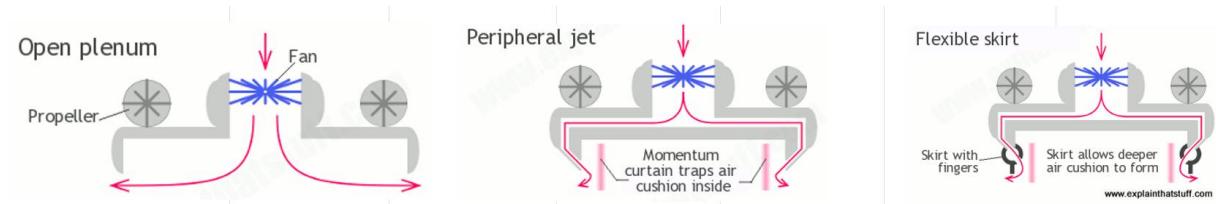
Un aéroglisseur (anglais: Hovercarft) est un véhicule amphibie qui se déplace sur un coussin d'air, ce qui lui permet de circuler sur l'eau, la terre, la boue, la glace et d'autres surfaces planes et imperméables, sans être en contact direct avec le sol. Ce coussin d'air est créé par de puissants ventilateurs qui soufflent de l'air sous le châssis, soulevant ainsi l'appareil à petite distance au-dessus de la surface. Certains aéroglisseurs sont équipés d'une jupe souple qui permet de maintenir l'air sous la structure, ce qui les rend plus économies en énergie et moins sensibles aux fuites d'air sur un terrain irrégulier. Comme il ne repose ni sur des roues ni sur une coque de bateau conventionnelle, l'aéroglisseur peut facilement passer d'un type de terrain à un autre, ce qui en fait un moyen de transport tout-terrain utilisable en toute saison. La première application pratique a été développée dans les années 1950 en raison du manque de moteurs suffisamment puissants, bien que les premières mentions de ce concept datent du XVIII^e siècle.

Les principaux intérêts et avantages des aéroglisseurs sont leur indépendance au regard du terrain, leur grande vitesse sur les zones peu profondes ou encombrées, ainsi que leur capacité à évoluer là où les bateaux ou les voitures ne peuvent pas aller. Ils peuvent traverser des champs de mines, des plages, des vasières, l'eau ou encore de la glace, ce qui les rend utiles pour les missions de secours, les opérations militaires amphibies ou le transport dans des régions isolées ou mal équipées en infrastructures.

Cependant, les aéroglisseurs présentent également des inconvénients importants. Ils sont bruyants et relativement coûteux à construire et à exploiter, notamment en raison de la maintenance générale et de l'usure de leurs jupes. Ils consomment beaucoup d'énergie, sont difficiles à piloter et restent sensibles aux vents forts et aux terrains en pente.

1.2 Différent types des aéroglisseurs/hovercrafts

Il existe trois types principaux d'aéroglisseurs qui fonctionnent tous en créant une poche d'air à haute pression sous le véhicule à l'aide de ventilateurs.



<https://www.explainthatstuff.com/hovercraft.html>

L'aéroglisseur à plenum ouvert à un fond plat simple. De l'air sous pression remplit une cavité ouverte sous la coque et s'échappe librement sur les bords. Ce type est simple, solide et peu coûteux, mais il perd beaucoup d'air, nécessite beaucoup de puissance pour flotter et fonctionne principalement sur des surfaces très lisses et planes.

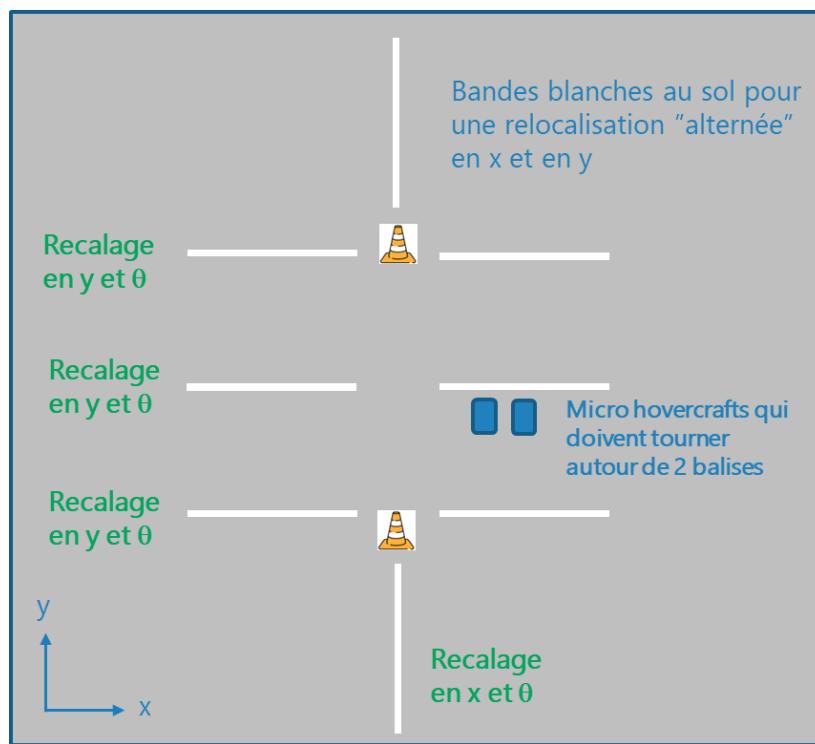
Un aéroglisseur à jet périphérique résout ce problème en envoyant de l'air par une fente étroite ou un anneau tout autour de la coque afin de former un « rideau » d'air rapide qui maintient la pression à l'intérieur du coussin. Ce design offre une meilleure portance pour une puissance identique et une étanchéité accrue du coussin. Toutefois, les jets situés sur les bords sont plus complexes sur le plan mécanique et peuvent perdre en efficacité sur un terrain très irrégulier.

Un aéroglissoir à jupe souple ajoute un rideau ou un anneau en tissu ou en caoutchouc souple autour de la base, qui descend pour suivre le relief et conserver la majeure partie de l'air sous l'appareil. Cela permet de planer plus haut et de franchir des vagues et des obstacles avec moins de puissance. Toutefois, la jupe est lourde et complexe, s'use et doit être changée régulièrement. Elle peut également augmenter la traînée lorsqu'elle frotte ou tape sur la surface.

1.3 Objectif à long terme du projet

À long terme, l'objectif est de faire concourir deux aéroglissoirs micro autonomes ou plus dans le hall de l'ENSMM, en suivant une piste connue définie par des lignes perpendiculaires au sol. En détectant ces lignes aux positions prédefinies, les aéroglissoirs recalibrent leur angle d'orientation θ ainsi qu'une des composantes de position x, y , calculées par exemple par des capteurs comme un centrale inertie (IMU). Cette opération pourra être compensée de dériver temporellement les valeurs de l'IMU et permet de calculer une localisation approximative pendant la conduite autonome entre des lignes.

L'objectif du projet actuel est de concevoir et de construire la version 2 du micro-aéroglissoir en tenant compte des enseignements tirés du prototype 1 de l'équipe précédente. Nous avons décidé de recommencer tout le projet à zéro. Nous avons pris cette décision en raison des défauts majeurs de la version 1 (voir le rapport 1 pour plus de détails), qui pouvaient être grandement améliorés. Pour ce nouvel aéroglissoir, nous avons développé et validé un mode de commande à distance basé sur le Wi-Fi et contrôlé par PID qui ne nécessite pas d'application, ainsi qu'une stratégie de localisation et de commande initiale qui lui permet d'effectuer un tour de la piste en mode autonome.



La piste est définie comme indiqué en haut. Chaque ligne mesure deux mètres de long et les lignes sont espacées d'un mètre, ce qui donne un terrain de 4 par 6 mètres.

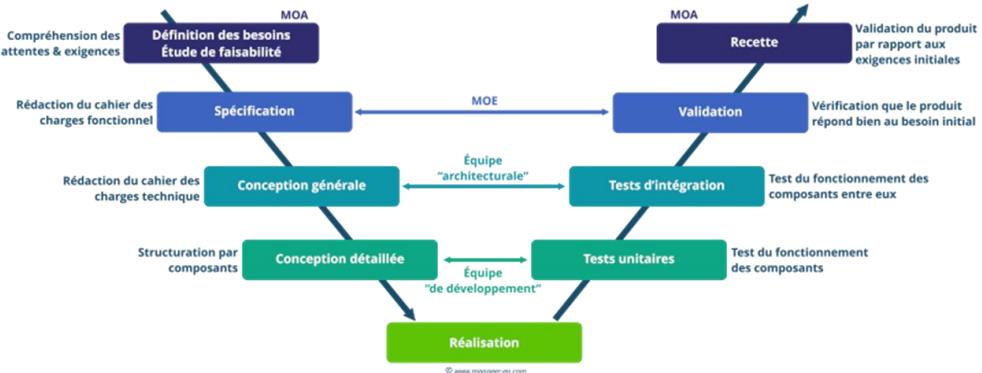
2 Cahier des charges (CdC)

2.1 Cahier des charges fonctionnel

Prof. Piat Construction et gestion de produits SUPMICROTECH-ENSM				Liste des exigences pour μ-verCraft				Commande: XXX Numéro de commande: / Semestre d'hiver 2025/26			
Données organis.		Données du processus		Exigences		Valeur - Données					
Nb.	Nom	Type	Phase			Réalisation minimale	Réalisation satisfaction	Réalisation idéale	Unité		
Fonction physique et technique											
F01		E		Dimensions		<200x100	<180x90	<140x90	mm		
F02		E		Poids		<500	<300	<100	g		
F03		E		Autonomie de la batterie		>2	>4	>6	min		
F04		E		Fond de route		Plane	Plane	Légères irrégularités			
F05		S		Course autonome sur circuit		Télécommandée	Autonome 180° autour d'un porte	Autonome un tour complet			
F06		S		Tolerance rotation angulaire		+~20% de la comand	+~10% de la comand	+~5% de la comand			
F07		E		Localisation		Marquages au sol	Marquages au sol	Marquages + Suivi des mouvements			
F08		S		Capteurs marquages		2 capteur IR	3 capteurs IR	3 capteur de flux optique			
F09		E		IMU		Capture 6-DOF	6-DOF + Filtre intégré	6-DOF + Filtre + Suivi de vie			
F10		E		Recalibrage de la pose avec des marquages		1 paramètre	2 paramètre	3 paramètre			
F11											
F12											
F13											
F14											
F15											
Technologie - Fabrication et assemblage											
T01											
T02											
Économie											
E01				Prix des composants		<300	<200	<100	€		
Relation homme-produit											
R01				Télécommande		BT + application	BT/WiFi	WiFi + site web			
R02				Indicateur de niveau de batterie		-	LED pulse/PWM	LED couleur			
Utilisation, entretien, maintenance											
U01				Mises à jour micrologicielles		Câble USB	Câble USB	Sans fil			
U02											
<i>Types d'exigences: O/N-Oui/Non; E-Exigence; S-Souhait Phase de conception: P -Principe ; C -Concept ; D -Design ; R -Réalisation</i>											
Remplace l'édition: / de: /							Édition actuelle: de: 17/01/2026				

2.2 Cahier des charges technique / détaillées

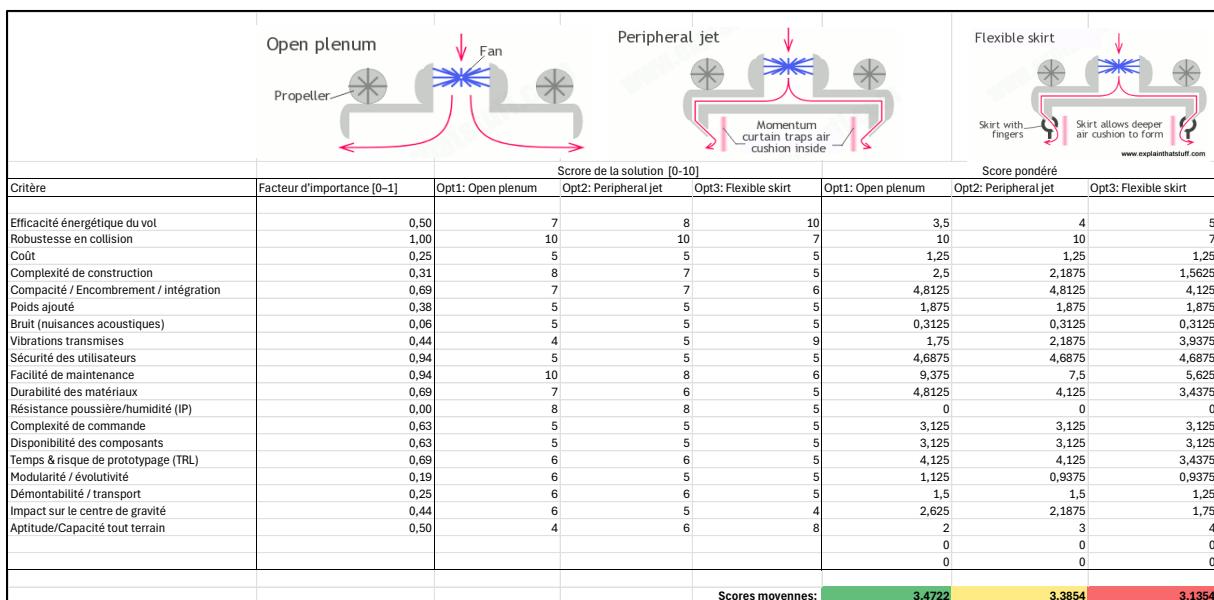
Cela était développé avec le cycle en V en tête :



Fonction	Test de validation	Demande valeur nécessaire
Définition des besoins (cdc fonctionnel / fonctions souhaitées) Aller tout droit jusqu'à la ligne, tourner ~180° autour du poteau Mode "télécommand": Suivre des commandes à distance le mieux possible	Recette validation qualitative, résultat de toutes les autres fonctions enchaînées validation qualitative, suffisamment réactive, que c'est faisable sans entraînement	validation qualitative, idéal: +20° bien tourné après validation qualitative
Spécification (cdc f -> cdc t) "architecture de haut niveau" Rotation autour axe z avec une angle défini Télécommande des mouvement Approximer l'orientation/angle relative par un ligne passée Positionnement dans l'espace Informer utilisateurs si tension faible pour LED & WiFi	Validation commander 90°, voir si le résultat et +9° dans les valeur demandé Envoyer des signaux type "avance vitesse 10%, ...", vérifier avec µC (déjà par camera) appareil/méthode expliquée au TD passer une ligne, comparer position évaluée par µC avec réalité/measure externe brancher une alimentation de laboratoire, valider que LED change l'état	+/-10% erreur +/-1% erreur, délai < 250ms +/-10% erreur +50% erreur voir CDC
Conception générale ("modules") Génération de poussée verticale (Schub/Hub erzeugen) Génération de rotation autour axe z Stabilisation de la rotation autour de l'axe z Mesure d'angle Créer des coussins d'air Envoyer des signaux à distance Mesure du courant de la Source d'énergie Mesure de la tension de la Source d'énergie Positionnement par rapport à une ligne Vitesse dans l'espace	Tests d'intégration >60gf/moteur avec hélice, en cas de 2 moteur verticale envoyer signal, observer vitesse de rotation et dépassement donner de petites perturbations ext. avec un stylo / similaire, voir si la perturbation est corrigée tourner IMU de quelques degrés, vérifier avec Geo-Triangle Allumez l'aéroglyisseur sur une planche. Inclinez la planche à partir de l'horizontale jusqu'à ce que l'aéroglyisseur commence à planer envoyer des signaux par un appareil, recevoir les signaux sur le µC activer le moteur, regarder avec capteur et multimètre multimètre Bouger sur une ligne, regarder si c'est enregistré définir un longeur spécifique au sol, mesurer le temps de déplacement en ligne droite, comparer vitesse théorique et réel	datasheet suffit validation qualitative validation qualitative +/-10% erreur validation qualitative par ex. via serial-out -0.15 A +0.1V +10° -0.25
Conception détaillée (composantes) Moteur électrique IMU Source d'énergie	Test unitaire Faire tourner le moteur dans le sens souhaité Sortir des accélérations sur 6 axes multimètre	ramp RPM 0% -> 100% -> 0% -> -100% -> 0% valider les sens des rotations autour des 3 axes (qualitative) tension entre 3V et 8V
Réalisation		

2.3 Choix de type d'hovercraft

Pour choisir entre des différents types de construction, nous avons fait une matrice de décision. Plus de détails sont disponibles dans la présentation1 et dans « phase1_CahierDesCharges_uVerCraft_2.xlsx » :



3 Composantes principales

3.1 Choix des composantes

Nous avons sélectionné chaque composant pour répondre à des contraintes strictes de poids, de taille et de compatibilité énergétique.

3.1.1 ESP32-S3

Ce choix est le cœur du système embarqué. Il offre le compromis idéal entre puissance de calcul et connectivité sans surcharger l'aéroglisseur.

- **Processeur Dual-Core** : Les deux cœurs, un essentiel pour séparer les tâches critiques. Un cœur est dédié à la boucle de contrôle temps réel (PID, lecture capteurs), tandis que l'autre gère la communication WiFi/Webserver.
- **Connectivité native** : WiFi et Bluetooth Low Energy (BLE) intégrés, permettant la télécommande via navigateur web sans module externe supplémentaire.
- **Facteur de forme** : Le module (format Seeed XIAO) est extrêmement compact et léger, parfait pour un micro-aéroglisseur.
- **I/O suffisants** : Nombre de broches adéquat pour gérer les 4 moteurs (PWM/DShot), l'IMU (I²C) et les capteurs IR (ADC).

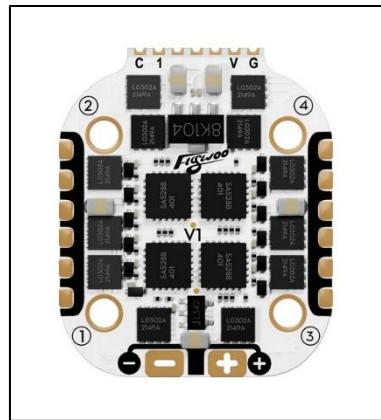


<https://www.gotronic.fr/art-carte-xiao-esp32s3-36954.htm>

3.1.2 ESC

Le choix de l'ESC a été dicté par la contrainte d'espace extrême du châssis.

- **Compacité extrême** : Format de montage 16x16 mm, le standard le plus petit pour les ESC 4-en-1 performants.
- **Courant admissible** : Capacité de 35A par moteur, ce qui est largement surdimensionné pour nos besoins (moteurs consommant < 11A), garantissant une fiabilité thermique et une longue durée de vie.
- **Contrainte de tension** : Ce modèle nécessite une tension minimale de 2S (7.4V), ce qui exclut l'utilisation de batteries 1S et a orienté le choix de toute la chaîne de puissance vers le 2S.



[ESC 4en1 Goku BLS 35A V3 4S | Flywoo](#)

3.1.3 Moteurs

Les moteurs ont été choisis pour leur rapport poids/puissance et leur compatibilité avec la tension 2S imposée par l'ESC.

- **Compatibilité 2S** : Fonctionnement optimal sous ~7.4V.
- **Format 1003** : Stator de 10mm de diamètre par 3mm de hauteur. Ce volume détermine le couple moteur. En cas de remplacement, une plage de taille 0803 à 1202.5 resterait compatible.
- **KV (Constante de vitesse)** : 14 800 KV. Cette valeur élevée est nécessaire pour obtenir une vitesse de rotation suffisante avec de petites hélices sous 2S. Une plage de 12 000 à 18 000 KV est acceptable.
- **Poids plume** : Seulement 4,4g par moteur.

Points-clés :							
• Modèle : Moteur Robo 1003 14800KV	• Marque : Flywoo	• Pour des builds ultra-légers	• Shaft 1.5 mm	• Trous de montage 6.6 x 6.6 mm	• Environ 3.3 g	• Garantie : 2 ans	

Tableau compatibilité

Prop (inch)	Voltages (V)	Throttle (%)	3	Pull (g)	Power (W)	Efficiency (g/W)	Temperature (in full throttle load 3's)
GF1608R*3	7.4	50%	2.4	61	17.8	3.435	59°C
		100%	8.6	141	63.6	2.216	
GF1609R*4	7.4	50%	2.5	67	18.5	3.622	64°C
		100%	9.5	154	70.3	2.191	
GF1635R*3	7.4	50%	2.4	62	17.8	3.491	58°C
		100%	8.7	143	64.4	2.221	
GF1635R*4	7.4	50%	2.8	72	20.7	3.475	64°C
		100%	9.6	154	71.0	2.168	
GF45mm*2	7.4	50%	2.4	65	17.8	3.660	62°C
		100%	8.8	155	65.1	2.380	
GF45mm*3	7.4	50%	2.8	77	20.7	3.716	68°C
		100%	10.5	170	77.7	2.188	

[Flywoo | Moteur Robo 1003 14800Kv](#)

3.1.4 Hélices

L'hélice transforme la puissance moteur en poussée et sustentation.

- **Compatibilité mécanique** : Alésage central ("shaft") de 1,5 mm correspondant exactement à l'axe des moteurs choisis.
- **Dimensions** : Diamètre de 35mm (environ 1.3 pouces). C'est la taille maximale que notre conception CAO permet d'intégrer sans collision avec le carénage.
- **Profil** : Tripales (3 pales) pour un meilleur rendement et une poussée plus stable à bas régime par rapport aux bipales.



<https://www.lacameraembarquee.fr/helices-fpv/18886-helices-gemfan-35mms-triples-1309-13x096x3-shaft-15mm-6936572228471.html>

3.1.5 Batterie

La source d'énergie doit fournir un courant fort instantané tout en restant légère.

- **Chimie et Tension** : Technologie LiPo (Lithium-Polymère) en configuration 2S (2 cellules en série). Tension nominale de 7.4V, plage de fonctionnement 6.0V (vide) à 8.7V (pleine charge HV).
- **Capacité** : 450 mAh. Suffisant pour plusieurs minutes d'autonomie tout en gardant un poids faible. Compatible HV (High Voltage, charge jusqu'à 4.35V/cellule) pour un gain d'énergie.
- **Surveillance Tension** : Un pont diviseur de tension (ratio 3.2, résistances 220kΩ et 100kΩ) permet au microcontrôleur de lire la tension batterie sur une entrée analogique (max 3.3V).
- **Surveillance Courant** : Lecture directe via la broche de télémétrie analogique de l'ESC.



<https://www.lacameraembarquee.fr/batteries-fpv/15924-pack-de-3-batteries-lipo-cnhl-hv-2s-450mah-70c.html>

3.1.6 IMU/10DOF

Indispensable pour la stabilisation automatique et le maintien de orientation/angle/cap.

- **Capteurs intégrés** : Module complet "10 Degrees of Freedom" combinant un accéléromètre (3 axes), un gyroscope (3 axes), baromètre et un boussole/magnétomètre.
- **Magnétomètre / boussole** : La présence d'une boussole électronique (magnétomètre) est critique pour corriger la dérive du gyroscope sur l'axe de lacet (Yaw) et maintenir une orientation absolue stable.
- **Interface** : Communication via bus I²C standard.

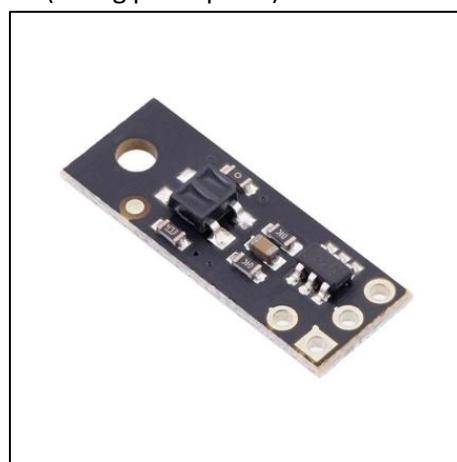


[Module 10 DoF SEN0140 - Gotronic](#)

3.1.7 IR

Ces capteurs permettent la navigation autonome en détectant le contraste de la piste au sol.

- **Technologie** : Capteur infrarouge réflexif à sortie analogique.
- **Plage de détection** : Efficace de 1 mm à 30 mm. Cette large plage est cruciale car la hauteur de vol (lévitation) de l'aéroglisseur peut varier dynamiquement sans perte de signal.
- **Légèreté** : Poids négligeable (< 0.5g par capteur).



<https://www.gotronic.fr/art-suiveur-de-ligne-4241-31140.htm>

3.1.8 Matériaux pour le chassis

La structure doit être rigide pour la mécanique mais ultra-légère pour la lévitation.

- Mousse (Structure principale)** : Dépron de 10mm (polystyrène extrudé). Matériau standard en aéromodélisme, offrant une excellente rigidité pour une densité très faible. Facile à découper et poncer.
- Bois (Renforts et supports moteurs)** : Contreplaqué fin de 3mm (peuplier ou similaire). Utilisé pour les zones subissant des contraintes mécaniques (fixation moteurs). Nous avons choisi par ex. bois de peuplier, mais d'autres types de bois fins/légers, qui peuvent être découpés au laser peuvent également être utilisés.



3.2 Commande

3.2.1 Liste des composantes/commande

La liste avec les liens est dans « phase1_CahierDesCharges_uVerCraft_2.xlsx »

	nb. Pièces/Hovercraft	nb. Pièces/Commande	Option1	Option2	Option3	approx. Weight sum	cost/piece	total cost
ESP32	1	1	Carte XIAO	GoTronic		3,3 g	8,60 €	8,60 €
ESC 4in1, 16mm	1	1	ESC 4en1 Go	CaméraEmb		3,6 g	45,90 €	45,90 €
Capteur IR	3	3	Suiveur de l	GoTronic		1,05 g	2,90 €	8,70 €
IMU (Module 10 DoF SEN0140)	1	1	https://www	GoTronic		1,6 g	5,80 €	5,80 €
7,4V-~5V régulateur	1	1	Régulateur	CaméraEmb		2 g	9,90 €	9,90 €
moteur	4	5	Flywoo 1 M	CaméraEmb	Moteur M2 1	17,6 g	14,90 €	74,50 €
hélices (d=35mm)	1	2	Hélices 35m	CaméraEmb		3,6 g	2,90 €	5,80 €
batterie (2s ~300mAh-550mAh)	1x pack de 3	1	Pack de 3 b	CaméraEmb	BetaFPV 1 Ld	31 -32g (18g-40	16,90 €	16,90 €
Bois de 3mm (chassis)	1	1	Contreplaqu	Conrad	CaméraEmb	17,2 g (CAO)	10,99 €	10,99 €
Mousse 10mm (Polystyrol)	2+1	6	Quadrax 25	Conrad	Nous même	2,8 g (CAO)	- €	- €
Velcro / Bande auto-agrippante	1	1	https://www	Conrad		2 g	1,99 €	1,99 €
resistance 100k Ohm	1	3	ENSM	1M Ω	1:2 grand resistance	1 ~g	- €	- €
resistance 220k Ohm	1	3	ENSM	2M Ω	1:2 grand resistance	1 ~g	- €	- €
Adaptateur pour charger XT60/XT	1	2	Adaptateur	CaméraEmb	Adaptateur	- -	1,90 €	3,80 €
cables	1x noir	1	Câble silico	CaméraEmb		2 g en totale	1,30 €	1,30 €
cables	1x rouge	1	Câble silico	CaméraEmb		- -	1,30 €	1,30 €
cables	1x jaune	1	Câble silico	CaméraEmb		- -	1,30 €	1,30 €
cables	1x blanc	1	Câble silico	CaméraEmb		- -	1,30 €	1,30 €
						SOMME:	89,75 g	198,08 €

3.2.2 Fournisseurs principaux

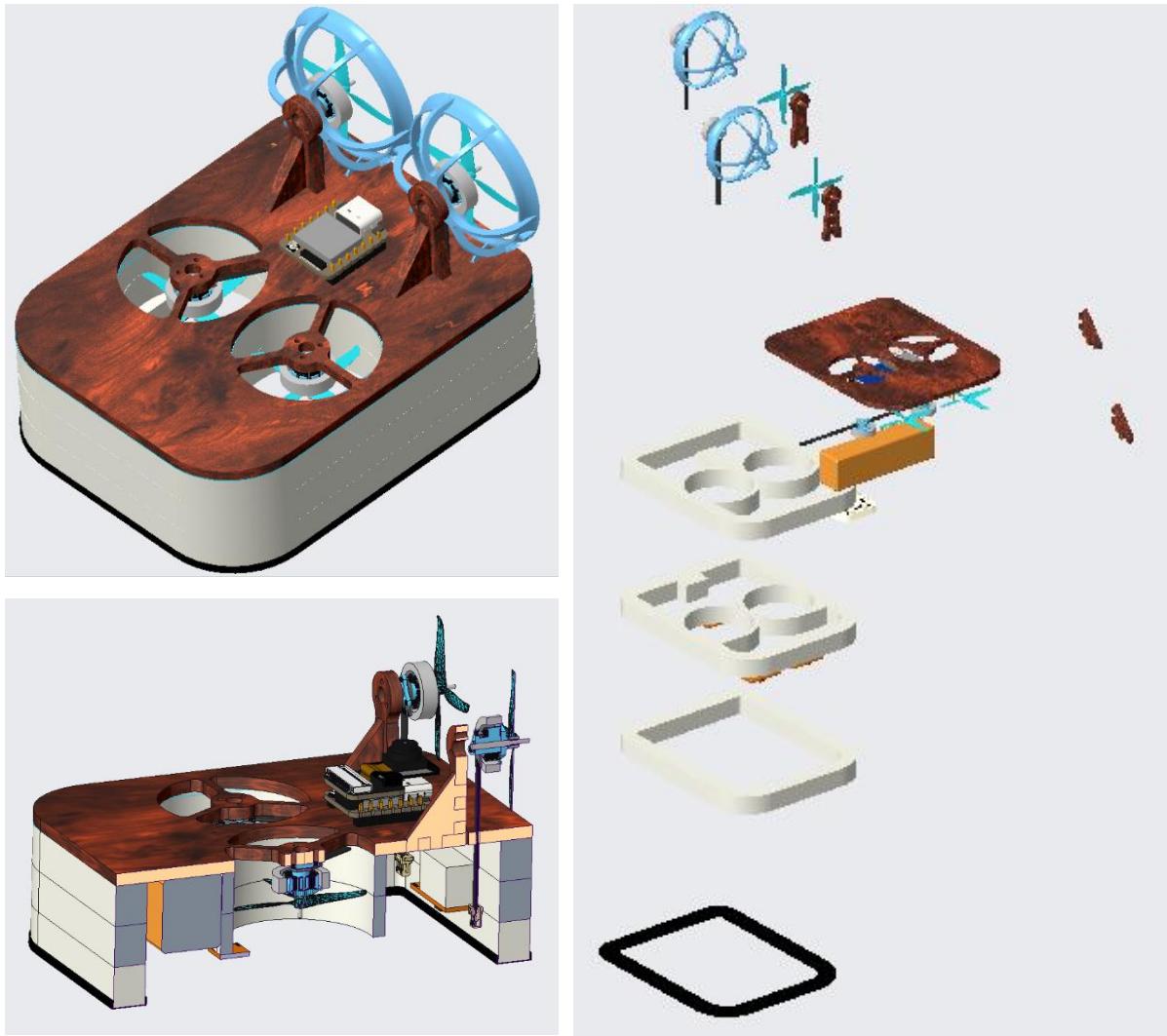
Les sources/fournisseurs principales sont :

- [GoTronic | Robotique et composants électroniques https://www.gotronic.fr/](https://www.gotronic.fr/)
- <https://www.lacameraembarquee.fr/>
 - Alternatives :
 - [Drone-Fpv-Racer le spécialiste du vol en FPV, DIY https://www.drone-fpv-racer.com/](https://www.drone-fpv-racer.com/)
 - <https://www.studiosport.fr/>
- <https://www.leroymerlin.fr/>

3.3 CAO

La CAO a été conçue dans Creo 10. Le modèle complet se trouve dans les fichiers du projet sous « mu-verCraft-II\2_Mecanique ». La densité des différents composants a été enregistrée afin de pouvoir estimer le poids total du châssis.

Creo rencontre un problème de chemin d'accès aux fichiers avec certains composants. Pour charger tous les composants lors de l'ouverture de l'assemblage, il faut cliquer avec le bouton droit sur les composants défectueux, sélectionner « Retrieve missing component » (Récupérer le composant manquant) et sélectionner manuellement .asm ou le .prt. Tous les fichiers nécessaires se trouvent dans le dossier sous leur nom correct.



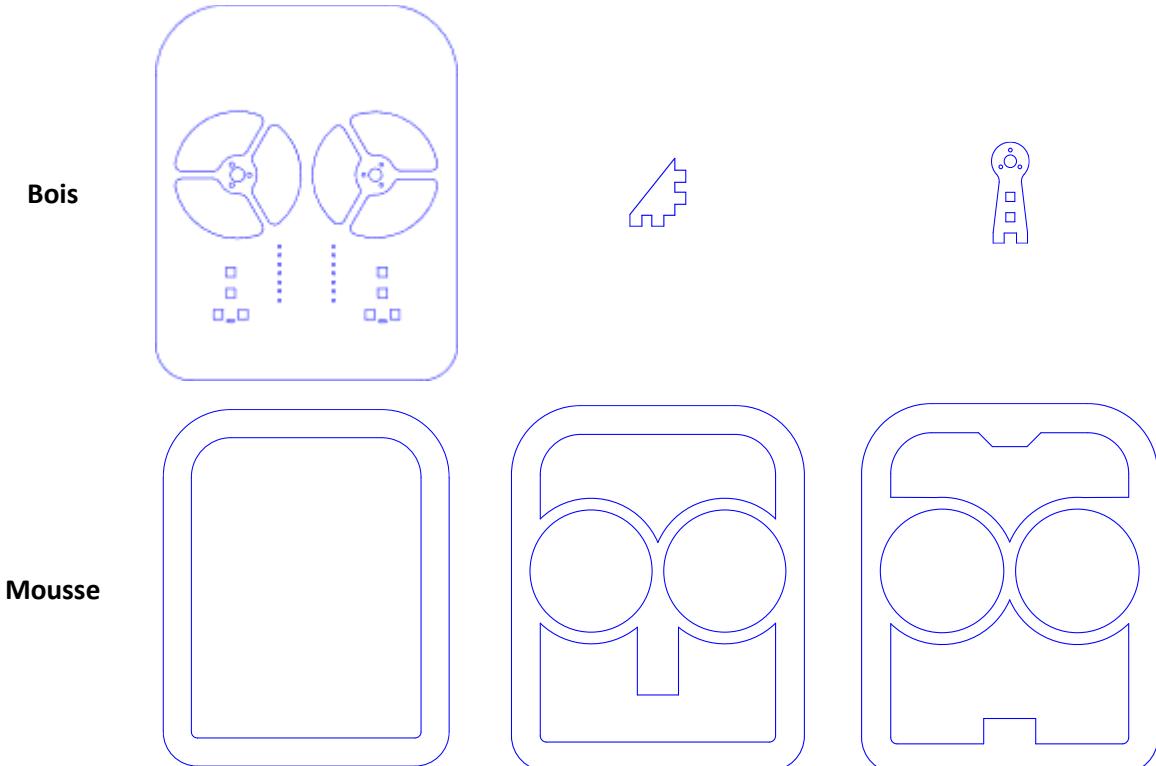
- Haut à gauche : L'aéroglissoir avec protection moteur en option. Celle-ci n'est pas obligatoire, car les hélices ne dépassent pas du châssis. Sans elle, le poids est réduit.
- Bas à gauche : Coupe transversale des conduits du moteur. Au début, les moteurs peuvent nécessiter un léger fraisage dans la mousse si les tolérances sont très serrées. Cela est bénéfique pour l'efficacité des hélices. On peut également voir une caméra optionnelle pouvant être installée.
- À droite : Vue éclatée de l'ensemble de l'assemblage.

4 Construction

4.1 Fabrication du châssis en atelier

4.1.1 Découpage laser

La machine de découpe laser de l'atelier est reconnue par l'ordinateur comme une imprimante. Dans Inkscape, vous pouvez ouvrir les fichiers .svg qui se trouve dans « mu-verCraft-II\2_Meca-nique\svg_laser ». Les différentes couleurs dans le fichier .svg indiquent à la machine de découpe laser le réglage à utiliser pour la découpe.



Le bois peut être découpé avec les réglages standard « bois 3 mm ».

Pour le Depron, les réglages suivants ont fonctionné pour nous :

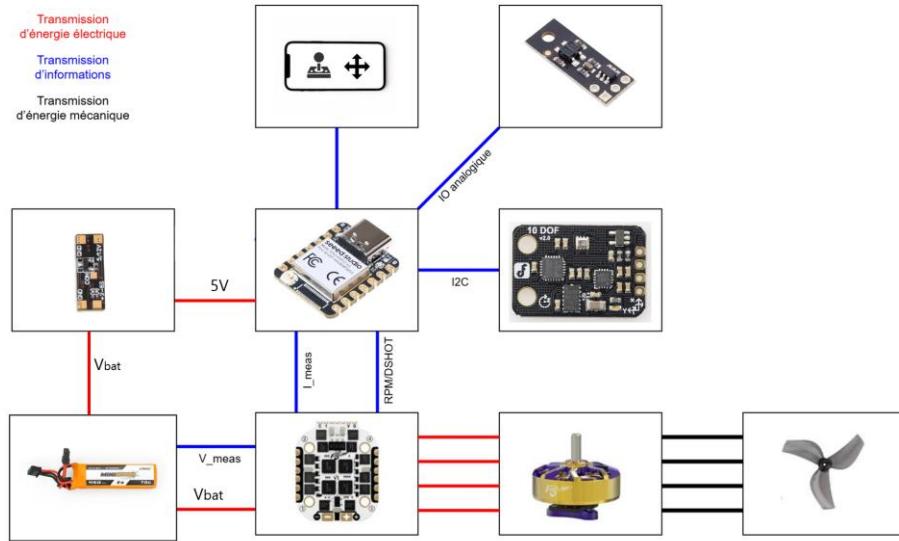
Puissance	Vitesse	PPI/Hz	Passages	Ass. d'air	Z-Offset	Avancé
30.00	8.00	10.000	9	Marche	-2.00	Par Défaut



Tous les composants découpés pour un aéroglyisseur.

4.2 Connections électrique

La figure suivante présente tout d'abord un schéma symbolique pour donner une idée d'ensemble :

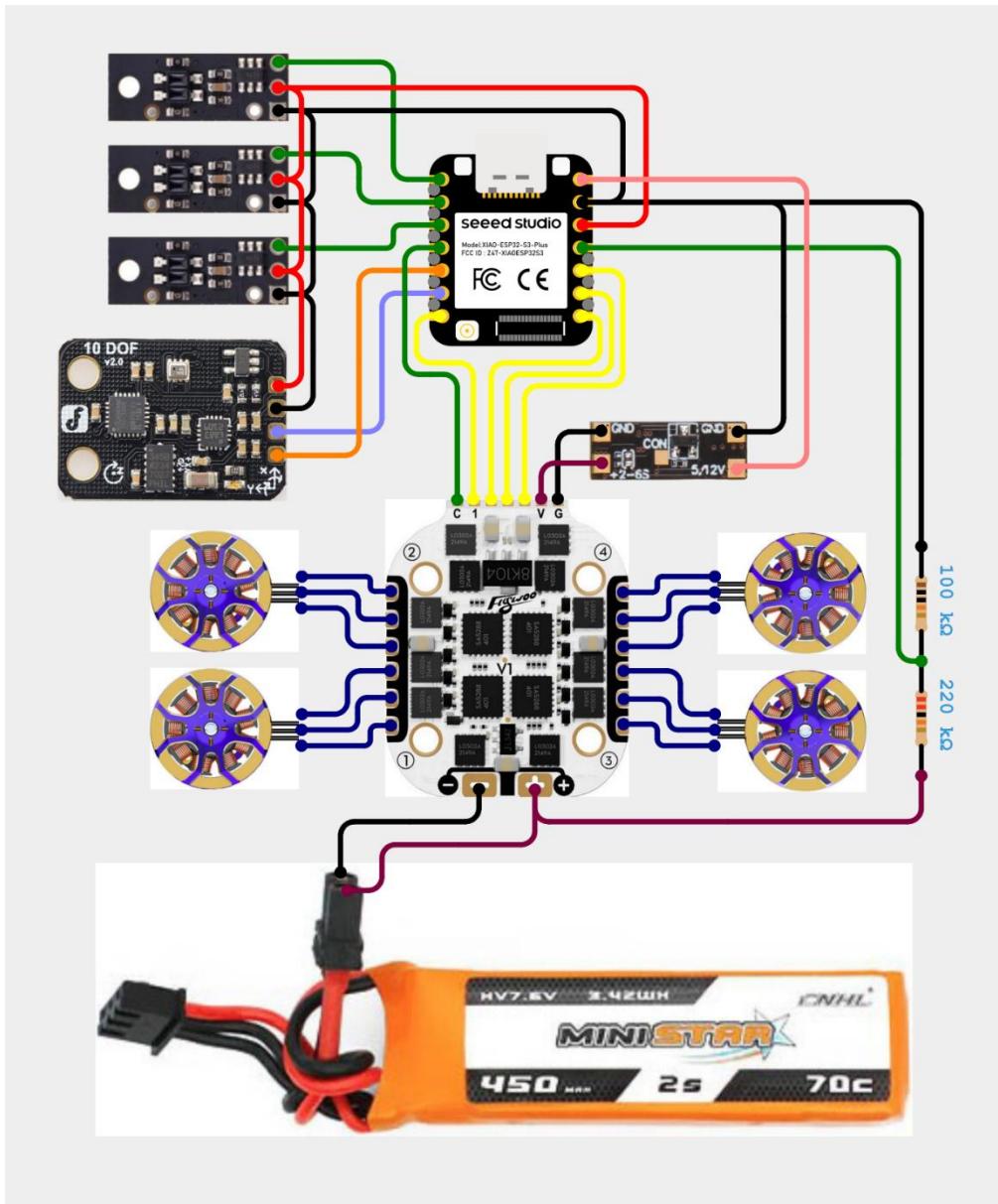


4.2.1 GPIO

Les GPIO sont configurés dans la fiche « hovercraft_variables.cpp ». Plus d'information dans le chapitre « programmation ».

Catégorie	Fonction / Description	Pin (Valeur)
Moteurs	Avant Gauche (FL)	GPIO_NUM_7
	Avant Droit (FR)	GPIO_NUM_43
	Arrière Gauche (BL)	GPIO_NUM_44
	Arrière Droit (BR)	GPIO_NUM_8
Batterie	Surveillance Tension (Voltage)	GPIO_NUM_9
	Surveillance Courant (Current)	GPIO_NUM_4
Capteurs IR	Avant Gauche (FL)	GPIO_NUM_3
	Avant Droit (FR)	GPIO_NUM_2
	Arrière Centre (Back Middle)	GPIO_NUM_1
Communication	I2C SDA (Données)	GPIO_NUM_5
	I2C SCL (Horloge)	GPIO_NUM_6
Divers	LED Statut	LED_BUILTIN

4.2.2 Schéma électrique



4.2.3 Longueurs des câbles

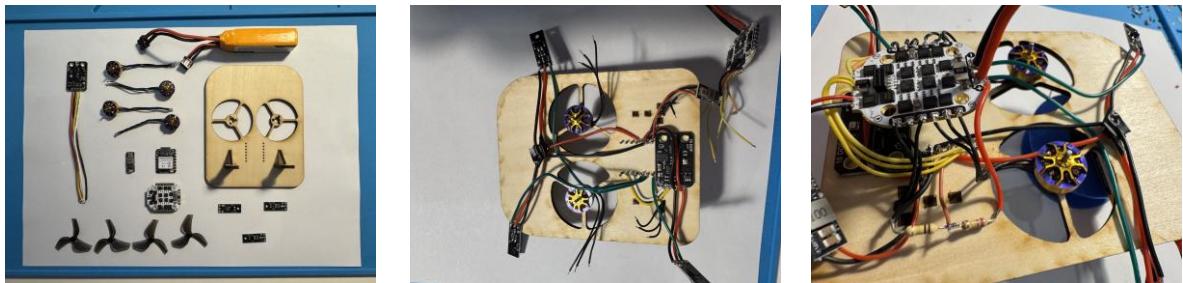
Si un jour on veut construire une autre hovercraft identique, la longer des câbles serait intéressant :

- Longueur Cable ESC avec connecteur xt30 : ~50-55mm
- Longueur Cable ESC -> BEC : 30mm
- Longueur Cable ESC -> ESP (DShot) 55mm
- Longueur Cable Moteur : couper directement a cote de connecteur et laissez la longueur comme ça

4.3 Assemblage

4.3.1 Soudage

Tous les composants doivent être soudés comme indiqué dans le schéma électrique. Cependant, le soudage doit être coordonné avec les étapes de montage. Dans « mu-verCraft-II\1_Electronique\Assemblage », des images montrent toutes les étapes nécessaires dans le bon ordre.



Ci-dessus, les étapes les plus importantes.

4.3.2 Coller les pièces

Nous avons utilisé cette colle à bois et polystyrène pour coller le bois et la mousse ploystyrène avec succès.



4.3.3 Assemblage finale



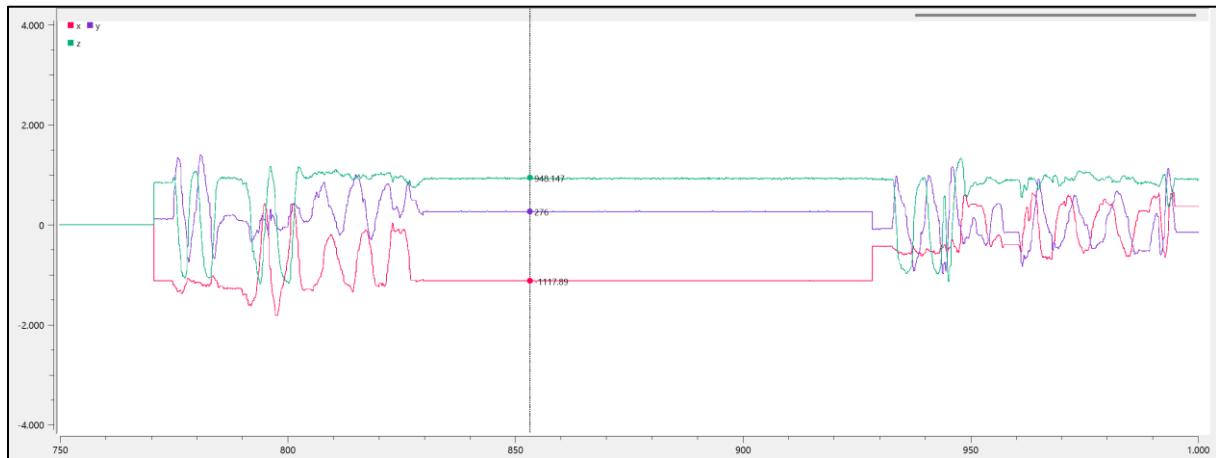
4.4 Calibration du boussole/magnetometer

Le magnétomètre mesure l'intensité du champ magnétique terrestre sur trois axes pour déterminer l'angle/cap (Yaw). Cependant, le capteur est inévitablement influencé par son environnement immédiat : les métaux ferreux du châssis, les batteries et les courants continus créent un champ magnétique parasite constant.

Le magnétomètre mesure l'intensité du champ magnétique terrestre sur trois axes pour déterminer l'angle/cap (Yaw). Cependant, le capteur est inévitablement influencé par son environnement immédiat : les métaux ferreux du châssis, les batteries et les courants continus créent un champ magnétique parasite constant (0,0).

L'objectif est alors de corriger les perturbations magnétiques fixes (liées au châssis et aux composants électroniques) qui décentrent les mesures. On peut voir la différence dans la figure suivante :

- Avant calibration (fig. gauche) : Le signal est décalé, limitant la lecture de l'angle/cap à une plage inexploitable ($\sim 90^\circ$ à 220°).
- Après calibration (fig. droite) : Les données sont centrées sur l'origine (0,0), rétablissant une lecture linéaire sur 360° .



4.4.1 Protocole de Calibration

Pour identifier ces constantes propres à notre matériel, une procédure spécifique a été mise en place via un script dédié. On peut le trouver dans « 3_Firmware >> Calibrate_Magnetometer ». Les résultats des constantes doivent alors être copié en « hovercraft_variables.cpp » dans le projet « Firmware » !

Remarques importantes pour le suivre :

- **Environnement Réel** : La calibration doit être effectuée avec le capteur monté dans sa position finale sur le châssis, avec tous les composants (batteries, ESC) installés, car ils participent à l'empreinte magnétique du véhicule.
- **Qualité du Mouvement** : Les rotations doivent être lentes et fluides pour éviter d'introduire du bruit d'accélération ou de rater les véritables extrêmes (pics de mesure).
- **Interférences Externes** : S'éloigner de toute source magnétique externe majeure (outils en acier, établis métalliques, aimants de haut-parleurs) durant la procédure.

N'oubliez pas de copier les valeurs dans « src/hovercraft_variables.cpp » après !

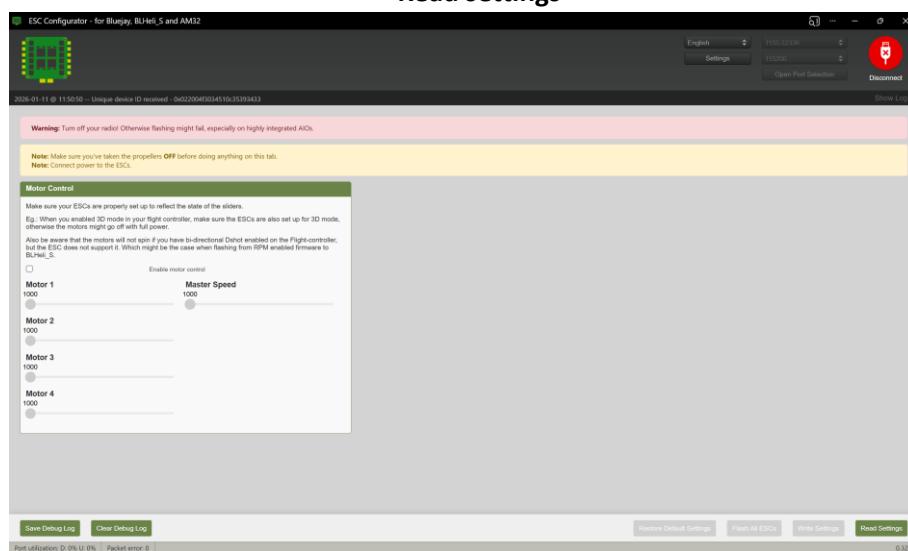
4.5 Paramètres pour l'ESC

Flashé avec "ESC Configurator" la software "BLHeli_S" avec le mode "3D". Si on ne fait pas les changements des paramètres, le ESC ne va pas pouvoir tourner à l'inverse, ni suivre les commandes. Pour cela on a besoin d'un contrôleur type « FC Betaflight ». Nous avons utilisé un privé, car il est utilisé qu'une seul fois, donc nous n'avons pas acheté un nouveau. Si l'école veut construire plusieurs autres, il faut acheter une comme ça ou chercher dans l'internet comment on peut faire autrement. Cela serait une option par exemple, mais il y en a pleines autres (demande ChatGPT ou similaire, il sait expliquer [Stack GOKU GN405 Nano HD 35A V3 BLS ELRS 2.4GHz ICM42688P | Flywoo](#)). Avec une contrôleur comme ça, suivez les étapes suivantes :

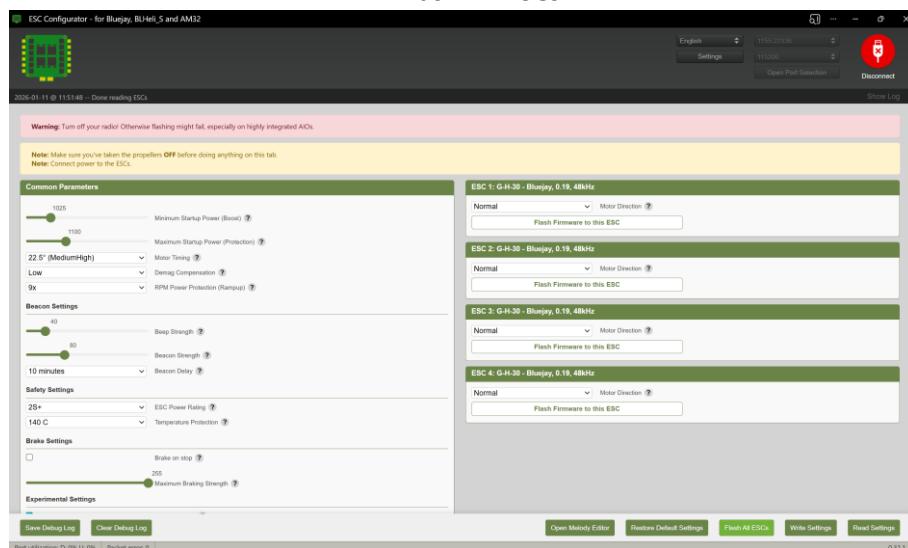
-> Ouvrir [ESC Configurator - for Bluejay, BLHeli_S and AM32](https://esc-configurator.com/) <https://esc-configurator.com/>

-> Brancher la batterie

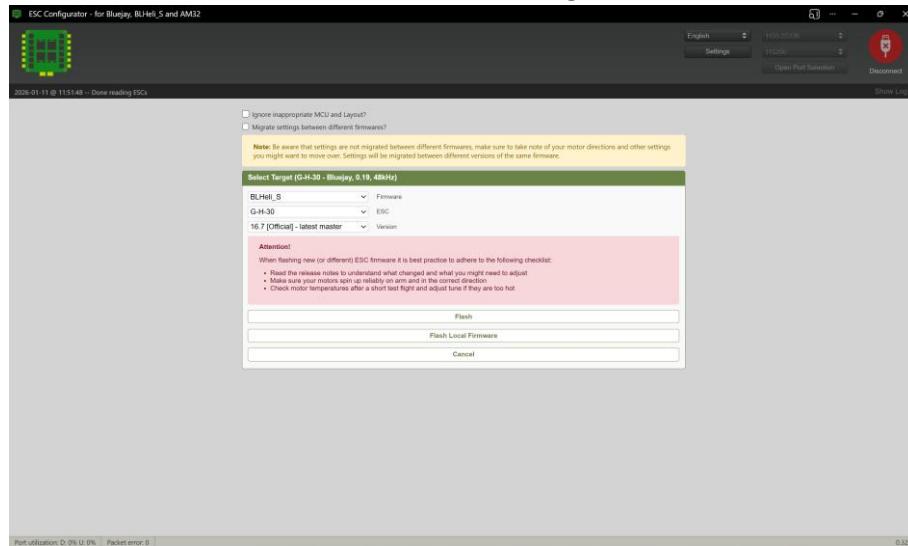
-> Read settings



-> Flash All ESCs



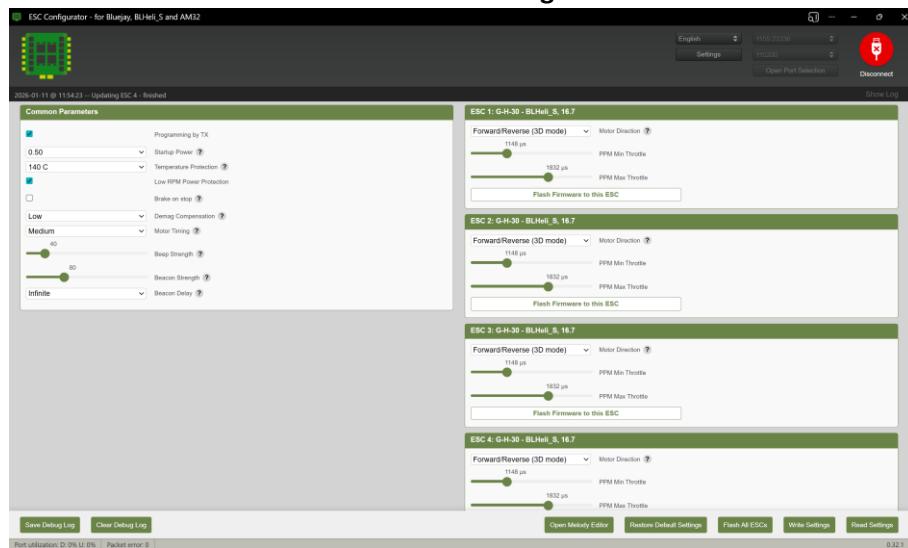
-> Utilizer ce settings



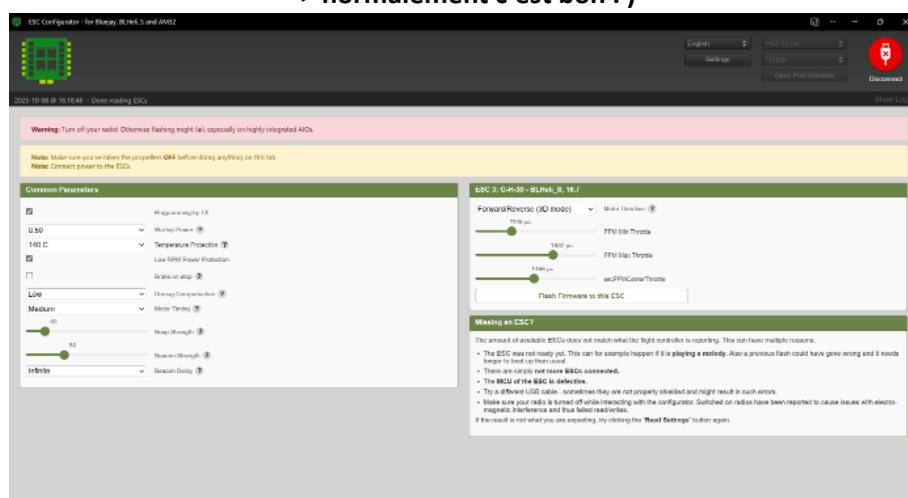
-> Mettez le mode Forward/Reversed (3D Mode)

-> Beacon delay Infinite

-> Write Settings



-> normalement c'est bon :)



5 Validation des composantes

5.1 Test des moteurs

Un fiche pour valider le fonctionnement des moteurs a été mis en place pour valider deux points critiques dès la réception du matériel :

- L'intégrité fonctionnelle des moteurs (vérification qu'ils ne sont pas défectueux).
- La fiabilité de la communication entre le microcontrôleur ESP32 et le contrôleur ESC via le protocole numérique DShot.

Le code pour faire cette vérification est disponible dans l'arborescence du projet sous : 3_Software > ESP32_motorTest2.

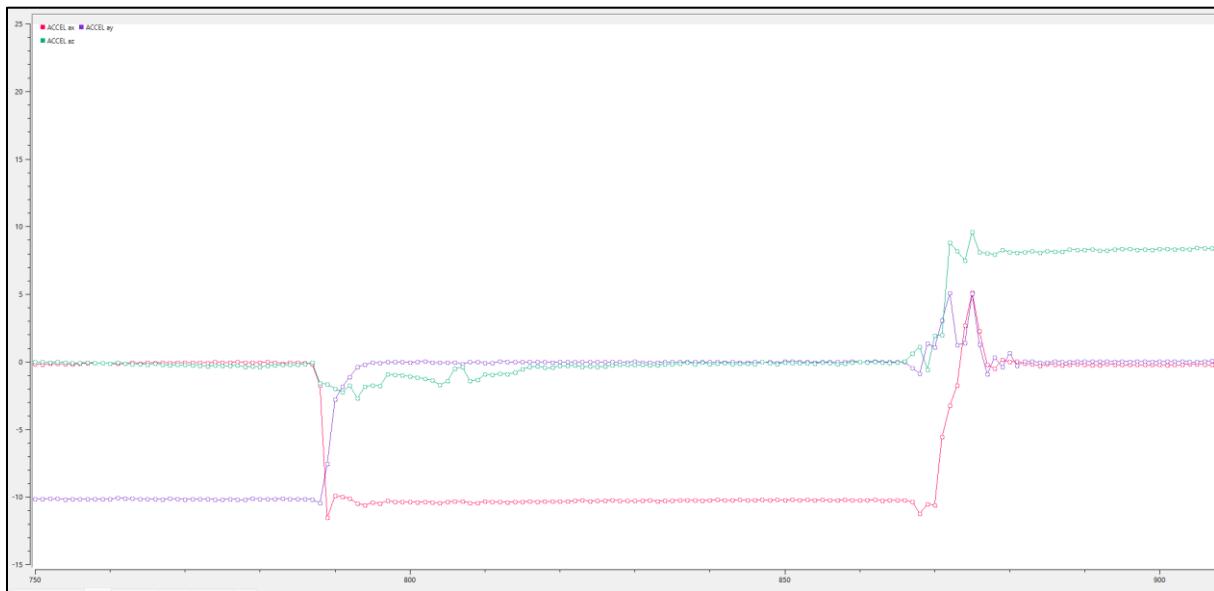
5.2 Capteur 10 DOF mems (ACCEL + GYRO + MAG + ENV)

[Fermion: 10 DOF Mems IMU Sensor-ADXL345/ITG3205/VCM5883L/BMP280 - DFRobot](#)

5.2.1 Validation des capteurs

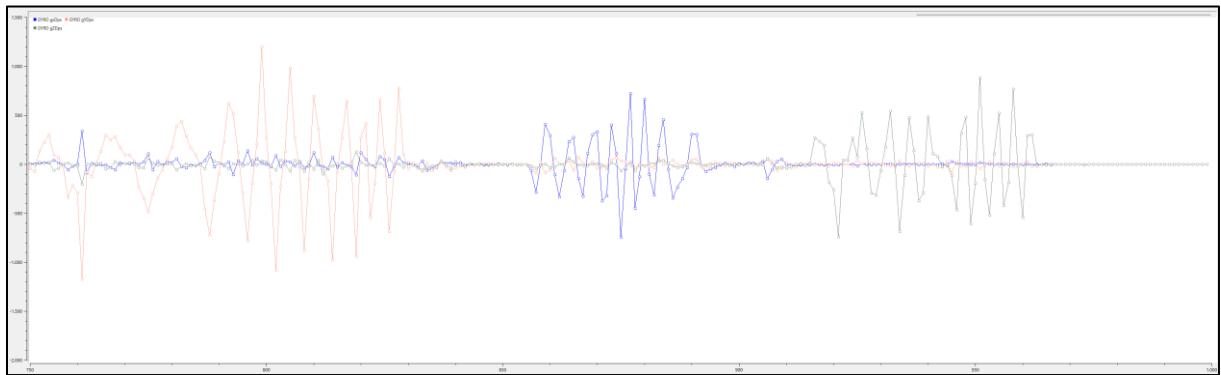
Pour valider utilisé nous avons utilisé [SerialPlot - Realtime Plotting Software | Hackaday.io](#)

5.2.1.1 Accéléromètre (m/s^2)



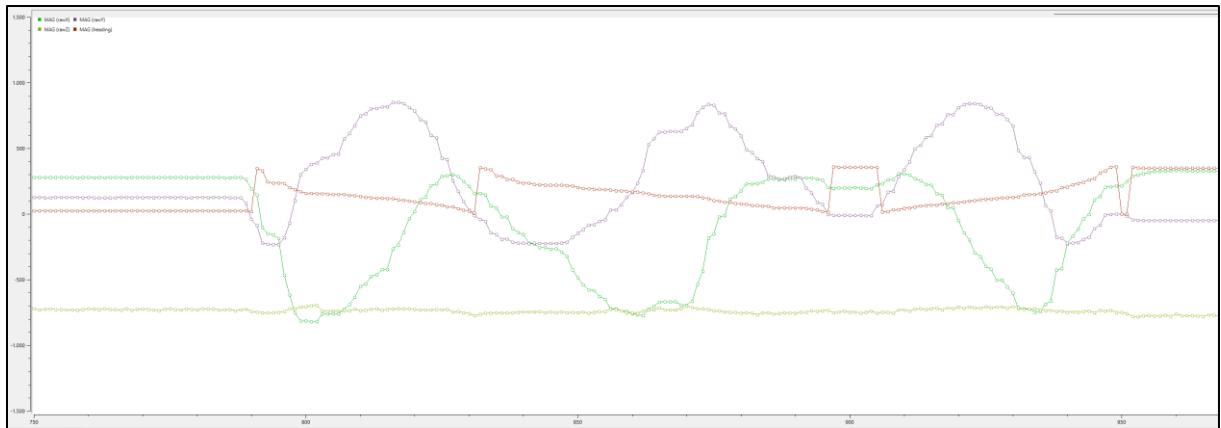
Le capteur était balancé sur ses 3 surfaces, et on voit bien que les valeurs se stabilisent à $\pm 1g$ selon leurs axes.

5.2.1.2 Gyroscope (deg/s)



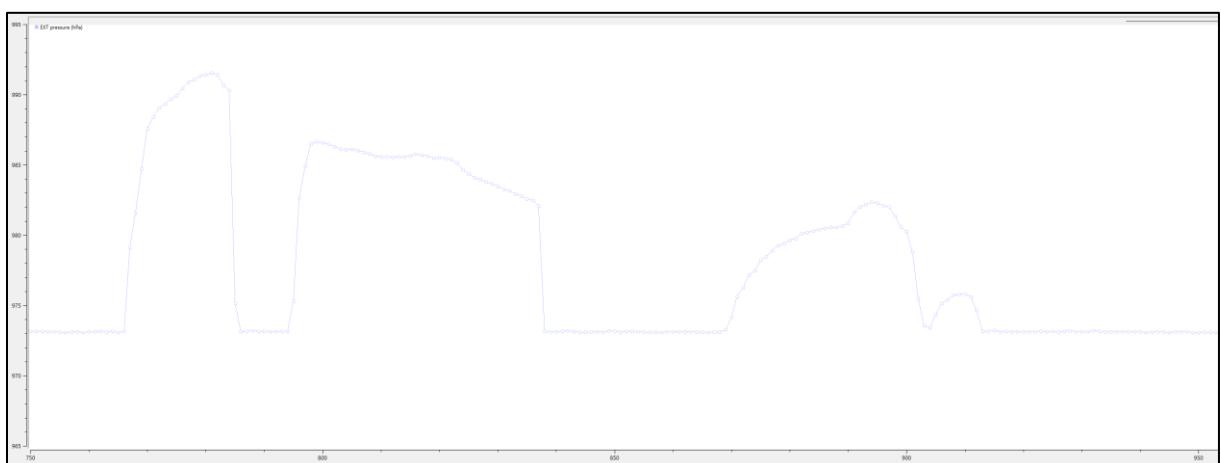
Le capteur était bouché selon ses 3 axes des rotations, et après laisse stationnaire, on voit bien le résultat est cohérent.

5.2.1.3 Magnétomètre / boussole (deg)



Le capteur sort trois valeurs « rawX, rawY, rawZ », avec ces trois il est possible de calculer l'orientation après. Ici le capteur était tourné purement autour d'axe z, c'est pour ça que le valeur jaune (rawZ) quasiment ne changes pas. La trace brune montre bien, que le capteur lit bien l'orientation. Les sautes brunes sont les changements de 0°->360° et inverse.

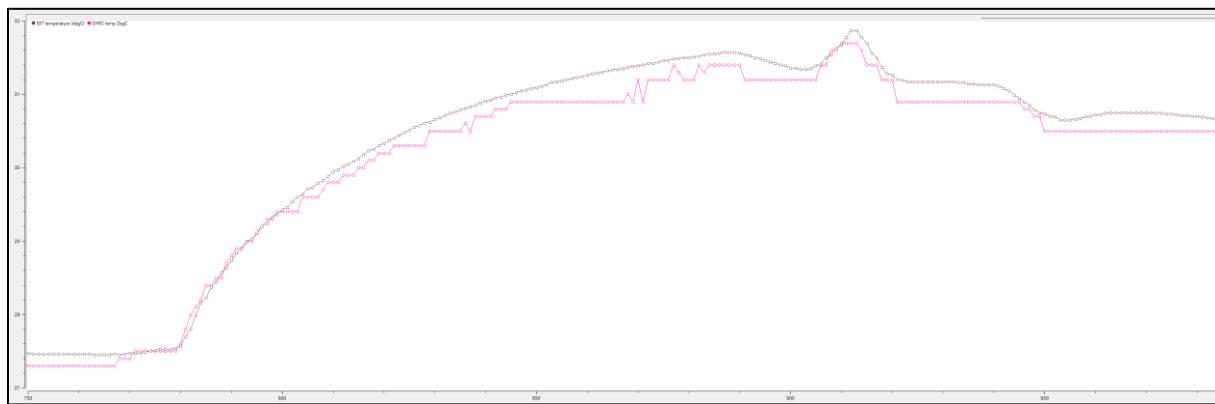
5.2.1.4 Pression exterior (hPa)



Pour ces mesurés de la pression, nous avons poussé sur le capteur avec notre dois. Le résultat est, que la pression monte à l'intérieure, et cela était mesuré avec le capteur. Nous, n'avons pas besoin, et non plus les outils pour vérifier mieux.

5.2.1.5 Temperature

Il y a deux mesurés, une est la température ambiant (brun) et la température du Gyro (rose). On voit, que la résolution de l'extérieure est mieux et est plus réactive à des changements, et celle dans le gyro prends plus de temps pour réagir, vu que le capteur est à l'intérieure et les changements l'extérieure prend plus de temps pour y arriver.



5.3 Validation du filtre complémentaire

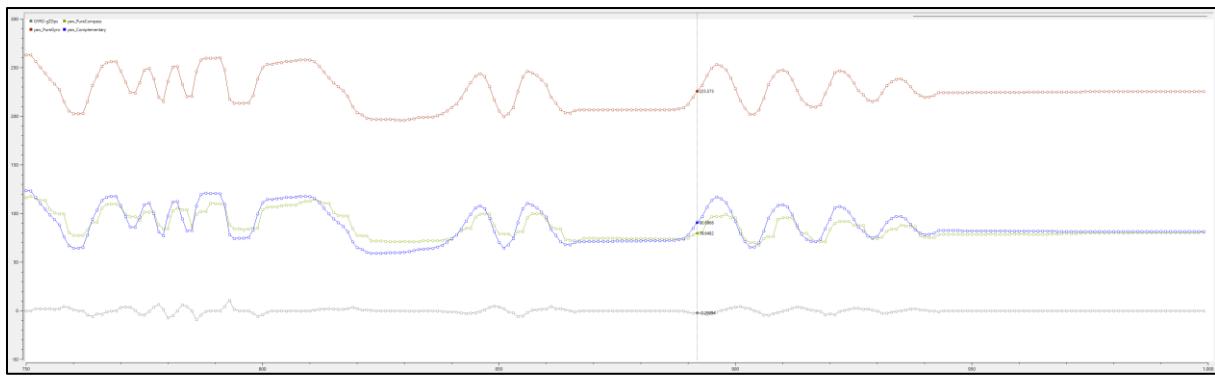
Le filtre complémentaire pour boussole et gyroscope combine les points forts des deux capteurs : le gyroscope est très réactif pour les mouvements rapides mais dérive avec le temps, tandis que la boussole (magnétomètre) est stable à long terme mais lente et sensible aux perturbations magnétiques. Cette fusion est essentielle pour obtenir une estimation de l'angle/cap (yaw) précise et sans dérive, indispensable pour la stabilisation des drones ou la réalité augmentée, où ni le gyroscope seul (trop de dérive) ni la boussole seule (trop de bruit) ne suffisent.

L'implémentation consiste à additionner l'intégration rapide du gyroscope (filtre passe-haut) et la lecture lente de la boussole (filtre passe-bas) via un coefficient de pondération α (ex: 0.98 pour le gyro), selon la formule :

$$\text{Angle} = \alpha * (\text{Angle} + \text{Gyro} * dt) + (1 - \alpha) * \text{Boussole}$$

En résumé, on "fait confiance" au gyroscope pour la fluidité immédiate et on utilise la boussole pour corriger doucement l'orientation absolue en arrière-plan.

Dans la figure suivante, on observe en brun la trace du gyroscope qui, à cause de l'intégration, présente une erreur très importante de plus de quatre-vingt-dix degrés. On constate que la boussole et le résultat du filtre complémentaire se superposent. Lors des mouvements rapides, le filtre complémentaire est plus réactif que la boussole, ce qui est dû à l'influence du gyroscope. Les erreurs qui en résultent sont ensuite corrigées dès que la boussole s'est stabilisée :



Les fonctions pour cela sont implémentées en imu.h et imu.cpp "void IMU::updateYawComplementaryFrom(float gz_rad_s, float mag_heading_deg)".

6 Programmation

L'ensemble du programme se compose de nombreux modules/bibliothèques, qui sont expliqués ci-dessous afin de donner aux utilisateurs une vue d'ensemble. La gestion du projet est faite avec GIT. Le projet utilise FreeRTOS, un système d'exploitation temps réel open source qui permet aux utilisateurs de créer des applications embarquées déterministes à faible latence avec une planification modulaire des tâches et un contrôle matériel précis (<https://www.freertos.org/>).

The screenshot shows a GitHub repository interface. On the left, the 'FIRMWARE' folder structure is displayed, containing subfolders like .pio, vscode, include, lib, and src, along with various C/C++ files such as main.cpp, hovercraft_variables.h, and platformio.ini. On the right, a detailed commit history for the 'Luis' branch is shown, listing numerous commits from various authors (origin, Leon, Luiz, etc.) with their commit messages and timestamps. The commit history spans from 2018 to 2020, indicating active development over two years.

6.1 Modes de pilotage

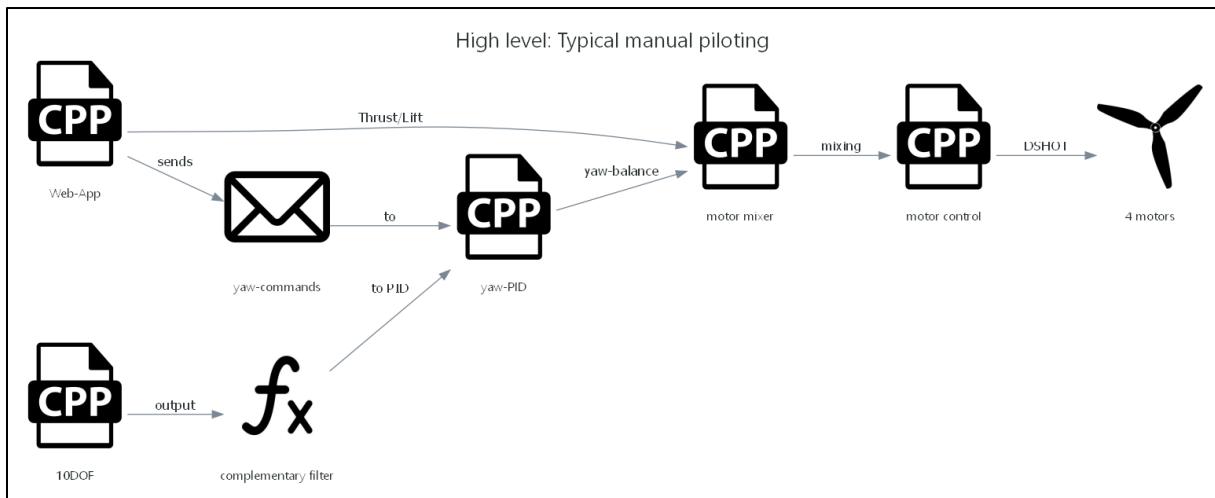
Il existe deux modes de pilotage principaux : manuel et automatique.

- Manuel : Pilotage via une application mobile (smartphone) à l'aide de curseurs virtuels (*sliders*).
- Automatique : L'aéroglisseur exécute un parcours autonome préprogrammé.

6.1.1 Structure « commande manuel »

Le fonctionnement général du code repose sur une structure de pilotage manuel classique. Tout commence par l'application Web, qui transmet les consignes de direction (lacet) au contrôleur PID de stabilisation. Parallèlement, les données issues des capteurs inertIELS (IMU) passent par un filtre complémentaire avant d'alimenter ce même régulateur PID.

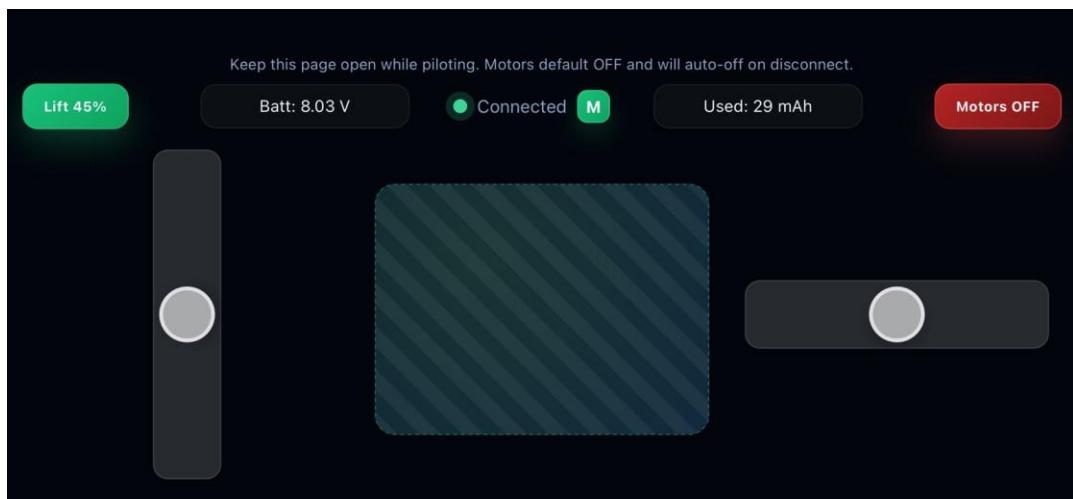
Le mélangeur moteur (Motor Mixer) centralise ensuite ces informations : il reçoit la correction de trajectoire calculée par le PID ainsi que les commandes de puissance (poussée et portance) envoyées par l'utilisateur via l'application. Pour finir, ce mélangeur convertit l'ensemble en signaux pour le contrôleur des moteurs, qui pilote les quatre turbines via le protocole DSHOT.



6.1.2 Structure « commande autonome »

Le concept général repose sur l'exécution du comportement autonome via une machine à états (*state machine*). Chaque état définit un objectif immédiat (par ex. *maintenir un cap*, *avancer pendant N secondes*, *attendre le franchissement d'une ligne*). Les transitions vers l'état suivant sont déclenchées par des conditions spécifiques, telles que des temporisations ou des événements liés aux capteurs infrarouges (par ex. « *maintenir l'angle X jusqu'à la détection d'une ligne, puis changer d'angle* »).

Le processus est enclenché en appuyant sur le bouton « M », suivi d'une attente de 2 secondes pour la calibration. Enfin, l'activation du bouton « Motors On » lance l'exécution du parcours par l'appareil. De plus amples informations sont disponibles dans le chapitre « Autonomous sequence ».



Interface pilotage sous forme de webapp sur le Smartphone.

Dans la suite, les différents blocs de code et bibliothèques utilisé pour faire fonctionner l'hovercraft seront présentés un par un :

(La partie suivante est rédigée en *anglais*, car la documentation originale contenue dans le code VS-code est rédigée dans cette langue (fiches « .md »), comme c'est généralement le cas en programmation.)

6.2 RTOS (FreeRTOS on ESP32)

6.2.1 For people that are new to multi-threading

An RTOS lets you split one big `loop()` into multiple independent “tasks” that run *as if* they were in parallel. The RTOS decides which task runs when, so timing-critical work (e.g. control loops) can keep a stable update rate even while other work (e.g. networking) is happening.

6.2.2 Main idea

This firmware uses FreeRTOS to run multiple subsystems concurrently (WiFi/web, IMU, control loop, sensors, telemetry) with explicit timing and priorities.

6.2.3 What an RTOS is (and why we use it)

FreeRTOS is a small real-time operating system used on microcontrollers. “Real-time” here means we care about predictable timing: a task can run periodically (e.g. every 5ms) and the scheduler helps keep that periodic behavior even when other tasks are active.

Official FreeRTOS website: <https://www.freertos.org/>

6.2.4 Two cores on the ESP32 (what that means)

Many ESP32 chips have two CPU cores. In practice that means the RTOS can run one task on core 0 while another task runs on core 1 at the same time.

In this firmware we *pin* some tasks to a specific core using `xTaskCreatePinnedToCore(...)`: - **Core pinning** is mainly used to isolate “noisy” workloads (like network/web housekeeping) from timing-sensitive workloads (like sensor reading / control loops). - Pinning does not magically make code thread-safe: any data shared between tasks still needs safe handoff (queues, atomic variables, etc.).

6.2.5 Scheduling (very short reminder)

The scheduler is the RTOS component that decides which task runs next based on priority and “readiness”. A task becomes not-ready when it blocks (e.g. delays for some time, waits on a queue), and ready again when that wait condition is satisfied.

Common timing patterns you’ll see in this firmware: - `vTaskDelay(...)`: simple “sleep” for at least a given time. - `vTaskDelayUntil(...)`: periodic scheduling pattern (tries to keep a constant period, useful for control loops).

6.2.6 RTOS building blocks used in this project (high level)

- **Tasks:** each subsystem runs in its own task function.
- **Priorities:** tasks can be assigned different priorities so time-critical tasks run first.
- **Queues:** tasks exchange data without sharing raw variables directly (producer/consumer pattern).

6.3 Hovercraft_variables (global project settings)

6.3.1 Main idea

This file is the single place to tune the hovercraft: pin assignments, control loop tuning, and user-facing presets are centralized here so you don't have to hunt through every subsystem.

6.3.2 Where it is used / how it is called

- Declarations live in [src-hovercraft_variables.h](#) and the actual values are defined in [src-hovercraft_variables.cpp](#).
- Almost every subsystem consumes these values indirectly, but the main wiring is done in [src/main.cpp](#) when objects are constructed and initialized.

6.3.3 Define all IO-Pins

These constants define how the ESP32 is wired to the hardware. If the wiring changes, change values here.

- LED_PIN (int)
 - Onboard status LED used by the blink test task in [src/main.cpp](#).
 - Default is LED_BUILTIN.
- global_PIN_MOTOR_FL, global_PIN_MOTOR_FR, global_PIN_MOTOR_BL, global_PIN_MOTOR_BR (gpio_num_t)
 - DShot/ESC signal pins for the 4 motors (Front-Left, Front-Right, Back-Left, Back-Right).
 - Used in motorCtrl.init(...) in [src/main.cpp](#).
- global_PIN_BATTERY_VOLTAGE_MONITOR, global_PIN_BATTERY_CURRENT_MONITOR (gpio_num_t)
 - ADC pins for battery voltage and current sensing.
 - Passed to batteryMonitor.init(...) in [src/main.cpp](#).
- global_PIN_IR_SENSOR_FL, global_PIN_IR_SENSOR_FR, global_PIN_IR_SENSOR_BM (gpio_num_t)
 - ADC pins for the 3 IR line sensors (Front-Left, Front-Right, Back-Middle).
 - Sampled via analogRead(...) in task_irSensors in [src/main.cpp](#) (mapping/order: index0=BM, index1=FL, index2=FR).
 - Passed into the IRSensors constructor; IRSensors::begin() currently creates the internal sample queue used between the sampling and processing tasks.
- global_PIN_I2C_SDA, global_PIN_I2C_SCL (gpio_num_t)
 - Intended I2C pin definitions for sensors.
 - Note: the current IMU implementation starts I2C via Wire.begin((int)SDA_PIN,(int)SCL_PIN) if the macros SDA_PIN/SCL_PIN are defined, otherwise it calls Wire.begin() defaults (see [lib imu/src imu.cpp](#)). These globals are currently not referenced directly.

6.3.4 Battery related limits and variables

These constants configure how ADC readings are converted into real battery voltage/current, plus optional low-voltage thresholds.

- `global_BatteryVoltage_VoltageDividerRatio` (float)
 - Divider ratio used to reconstruct battery voltage from the ADC pin voltage.
 - The battery monitor computes: $V_{batt} = V_{pin}/ratio$ (see [lib/battery_monitor/src/battery_monitor.cpp](#)).
 - For a simple resistor divider, a typical ratio is $ratio = R_{bottom}/(R_{top} + R_{bottom})$.
- `global_BatteryCurrent_VoltageDividerRatio` (float)
 - Scaling factor applied to the measured current-sense pin voltage.
 - The battery monitor computes `sensor_mV = iPin_mV * ratio`.
 - Keep at `1.0` if you have no divider/scaler on the current sense line.
- `global_BatteryCurrent_SensorScaler_AmpsPerVolt` (float)
 - Betaflight-style current scale, **in units of [0.1 mV]/A** (despite the variable name).
 - The battery monitor converts it to mV/A via `mvPerAmp = scale * 0.1` and computes $I(A) = sensor_{mV}/(mV/A)$.
 - Example: `130` means `13 mV/A`.
- `global_BatteryVoltageLow_WarningLow` (float, Volts)
 - Intended warning threshold for low battery.
 - Note: currently not enforced automatically; battery voltage is mainly sent as telemetry (see the battery task in [src/main.cpp](#)).
- `global_BatteryVoltageLow_MotorCutoffLow` (float, Volts)
 - Intended hard cutoff threshold where motors should stop.
 - Note: currently not enforced automatically; you can implement a failsafe in the motor task or battery task based on this.

6.3.5 Motor control variables

These variables define motor output scaling and direction conventions.

- `global_AllMotorsScalePercent` (float, %)
 - Global power cap applied inside `MotorCtrl` to reduce maximum throttle.
 - Used when `MotorCtrl motorCtrl(global_AllMotorsScalePercent)` is constructed in [src/main.cpp](#).
- `global_NegativeRpmScaleFactor` (float)
 - Reverse-direction compensation factor.
 - Applied only when a motor command is negative, to boost reverse output if reverse thrust is weaker (used in [lib/motor_ctrl/src/motor_ctrl.cpp](#)).
- `global_MotorsReversedFL, global_MotorsReversedFR, global_MotorsReversedBL, global_MotorsReversedBR` (bool)
 - Per-motor direction flags.
 - Passed into `motorCtrl.init(...)` in [src/main.cpp](#) so that “positive thrust” matches the physical direction of the mounted propellers.
 - These are non-const globals, but in the current firmware they are effectively “startup configuration”.

6.3.6 Web piloting / UI presets

These values define what the web UI offers (presets) and how user input is interpreted.

- `global_WebLiftPresetPercent_Array[]` (float array, %)
 - List of lift presets shown/cycled in the web UI.
 - Used by the web UI generation logic in [lib/network_piloting/src/network_piloting.cpp](#).
- `global_WebLiftPresetPercent_Array_len` (size_t)
 - Length of the lift preset array.
 - Used to bound preset iteration in [lib/network_piloting/src/network_piloting.cpp](#).
- `global_WebLiftPresetPercent_Array_startIndex` (int)
 - Which preset becomes the first “non-zero” selection after startup (startup selection is always 0).
 - Used by preset cycling logic in [lib/network_piloting/src/network_piloting.cpp](#).
- `global_WebThrustPresetPercent` (float, %)
 - Scales the maximum thrust command coming from the web UI.
 - Implemented in the thrust callback in [src/main.cpp](#): `scaled = thrustPercent * (preset/100)`.

6.3.7 Wifi SSID, PW, IP Adresses

These parameters configure the ESP32 SoftAP and the webserver port.

- `global_WifiApSsid` (char[])
 - SSID (network name) for the ESP32 Access Point.
 - Used in `wifiManager.startAccessPoint(...)` in [src/main.cpp](#).
- `global_WifiApPassword` (char[])
 - WPA2 password for the Access Point (must be at least 8 characters).
 - Used in `wifiManager.startAccessPoint(...)` in [src/main.cpp](#).
- `global_WebServerPort` (uint16_t)
 - TCP port for the embedded webserver (typically 80).
 - Used to construct the webserver inside NetworkPiloting (see [lib/network_piloting/src/network_piloting.cpp](#)).

6.3.8 Gyro/IMU/Complementary filter settings

These values tune the IMU yaw estimation.

- `global_ComplementaryFilter_yawAlpha` (float, unitless)
 - Complementary filter weight for yaw: higher means “trust gyro more, correct slowly with magnetometer”.
 - Used when constructing the IMU and when configuring the filter in the IMU task in [src/main.cpp](#).
 - Typical range is 0...1.

6.3.9 Control loop constants

These define timing of the main control loop.

- `global_ControlLoopRate_Hz` (float, Hz)
 - Target frequency of the motor/control task loop.
 - Used to compute the periodic delay in `task_motorManagement` in [src/main.cpp](#).

6.3.10 PID controller constants

Yaw-rate control tuning parameters.

- `global_YawRatePid_Kp, global_YawRatePid_Ki, global_YawRatePid_Kd` (float)
 - Gains for the yaw-rate PID controller.
 - Used to initialize `yawRatePid` in `task_motorManagement` in [src/main.cpp](#).
- `global_YawRatePid_OutputLimit` (float)
 - Output clamp for the PID command (expected mixer units are roughly -100...100).
- `global_YawRatePid_IntegratorLimit` (float)
 - Integrator clamp (anti-windup) in the same units as the PID output.
- `global_MaxYawRateSetpoint_dps` (float, deg/s)
 - Maps full steering input ($\pm 100\%$) to a yaw-rate setpoint.
 - Used in [src/main.cpp](#) to compute the yaw-rate setpoint from the user steering slider.
- `global_YawCenterSensitivity` (float, deg/s)
 - “Center sensitivity” for the yaw command curve: sets how responsive yaw feels for *small* steering deflections.
 - Interpreted as the approximate slope around stick/slider center (0%).
 - Used in [src/main.cpp](#) to shape the mapping from steering percent to yaw-rate setpoint.
- `global_YawRateExpo` (float, unitless, 0...1)
 - “Expo” shaping for the yaw command curve: higher expo makes the response softer around center and ramps up more strongly towards full deflection.
 - Used together with `global_YawCenterSensitivity` and `global_MaxYawRateSetpoint_dps` to compute the yaw-rate setpoint from the user steering slider.
 - Used in [src/main.cpp](#) in the yaw setpoint mapping.
 - **Visualization tool:** there is a curve visualizer at [3_Software/rates.html](#) (open it in a browser) to inspect the response curve.

6.4 Main (application entry point)

6.4.1 Main idea

This file is the integration point of the whole firmware: it wires together all subsystems (motors, mixer, IMU, WiFi/web control, sensors, battery) and starts the FreeRTOS tasks that make the hovercraft run.

6.4.2 Where it is used / how it is called

- This is the Arduino entry point compiled by PlatformIO: [src/main.cpp](#).
- `setup()` runs once after boot, then FreeRTOS tasks execute continuously; `loop()` is intentionally unused.

6.4.3 Project structure: what runs when

6.4.3.1 Boot sequence (high level)

1. **Boot → setup()**: initialize peripherals, create the control queue, start WiFi + web control, then create RTOS tasks.
2. **Runtime**: tasks run concurrently on the ESP32's two cores.
3. **Safety default**: motors start disarmed (`motorsEnabled=false`) and the motor task keeps outputs safe until arming is received.

6.4.3.2 Data flow overview (who talks to whom)

- **Web UI → NetworkPiloting callbacks → control queue**
 - WebSocket messages update `g_latestSetpoints` and immediately publish them using `xQueueOverwrite(g_controlQueue, ...)`.
 - The queue length is 1, so the motor task always consumes the **latest** set-points.
- **IMU task → shared yaw-rate → motor task**
 - IMU task updates `g_yawRateMeasured_dps` and `g_lastYawRateUpdate_us`.
 - Motor task checks “freshness” (timestamp age) before using yaw-rate for control.
- **Motor task → mixer → motor controller → ESCs**
 - Motor task computes the yaw-rate control command (`diffCmd`) via `PIDController`.
 - `MotorMixer` converts lift/thrust/diff into 4 motor percentages.
 - `MotorCtrl` sends DShot commands to the ESCs.
- **Battery task → NetworkPiloting telemetry**
 - Battery task publishes voltage/current/mAh via `networkPiloting.sendTelemetry(...)`.

6.4.3.3 Task map (what each task does)

The firmware creates multiple tasks in `setup()` using `xTaskCreatePinnedToCore(...)`.

6.4.3.3.1 `task_wifiManager (core 0)`

- Calls `networkPiloting.loop()` periodically (every ~50ms).

- Purpose: housekeeping (WebSocket client cleanup); the async server handles network I/O callbacks.

6.4.3.3.2 *task_imu (core 1)*

- Initializes IMU, then runs a periodic loop at **200 Hz** (`vTaskDelayUntil(..., 5ms)`).
- Reads gyro continuously; reads magnetometer slower (~20 Hz) and updates yaw complementary filter.
- Publishes yaw rate (deg/s) to the shared variables for the control loop.

Heading sign / offset note: the heading published to the UI and heading-hold (`g_yawMeasured_deg`) is a **mounting-corrected** magnetometer+gyro estimate. The IMU applies a hardcoded transform `heading_corrected = wrap360(180 - heading_raw)` so the displayed heading has the desired 180° offset and increases in the chosen positive rotation direction.

If you later enable a non-zero heading-controller Kd, ensure the yaw-rate signal used for derivative-on-measurement has the **same sign convention** as the heading (positive yaw-rate means heading increasing).

6.4.3.3.3 *task_motorManagement (core 1)*

- Runs the main control loop at `global_ControlLoopRate_Hz` using `vTaskDelayUntil`.
- Non-blocking reads of `g_controlQueue` to get the latest setpoints.
- Implements yaw-rate control:
 - Converts steering percent to yaw-rate setpoint using `global_MaxYawRateSetpoint_dps`.
 - If motors are armed and gyro data is fresh: `PIDController.update(...)` → `diffCmd`.
 - Otherwise: PID reset + `diffCmd = 0`.
- Applies safety gates:
 - Always calls `motorCtrl.setMotorsEnabled(mySetPoint.motorsEnabled)`.
 - If disarmed, forces thrust and diff to zero; lift can be “latched” but not applied until arming.
 - If lift is disabled, calls `motorCtrl.applyLiftOff()` and forces mixer lift to 0.

6.4.3.3.4 *task_irSensors (core 1)*

- Polls for ISR-completed measurements (`irSensors.hasNewMeasurement()`) and consumes them.
- Currently prints debug values (angle to line, lateral velocity). Intended as a future navigation input.
- Typical polling delay is ~20ms.

6.4.3.3.5 *task_batteryMonitor (core 1)*

- Calls `batteryMonitor.update()` and prints voltage/current/mAh.
- Sends telemetry to web clients once per second.

6.4.3.3.6 *task_blink (core 1)*

- Simple LED blink for testing.

6.4.4 Functionalities implemented (project-level)

- **System bring-up:** initialize motor outputs, mixer state, sensor interrupts, battery ADC scaling, WiFi AP, and web control.
- **Real-time scheduling:** run IMU acquisition and control loop periodically with vTaskDelayUntil.
- **Command & telemetry plumbing:**
 - WebSocket commands → queue → control loop.
 - Battery telemetry → WebSocket broadcast.
- **Core separation:** keep network housekeeping on core 0; keep control/sensing on core 1.
- **Safety behaviors:** disarm defaults, lift-off handling, PID reset on stale sensor data.

6.4.5 Methods / functions overview

This file is mostly glue code and task entry points (not a class).

- **setup()**
 - Initializes serial + LED, motors (`motorCtrl.init(...)`), mixer (`motorMixer.init()`), IR sensors (`irSensors.begin()`), battery monitor (`batteryMonitor.init(...)`).
 - Creates the control queue (`g_controlQueue = xQueueCreate(1, sizeof(ControlSetpoints))`).
 - Starts WiFi AP and registers NetworkPiloting callbacks (lift/thrust/steering/arm).
 - Calls `networkPiloting.begin()` and creates RTOS tasks.
- **loop()**
 - Empty by design (everything runs in tasks).
- Task functions (`task_wifiManager`, `task_imu`, `task_motorManagement`, `task_irSensors`, `task_batteryMonitor`, `task_blink`)
 - Each is an infinite loop; timing is controlled by `vTaskDelay(...)` or `vTaskDelayUntil(...)`.

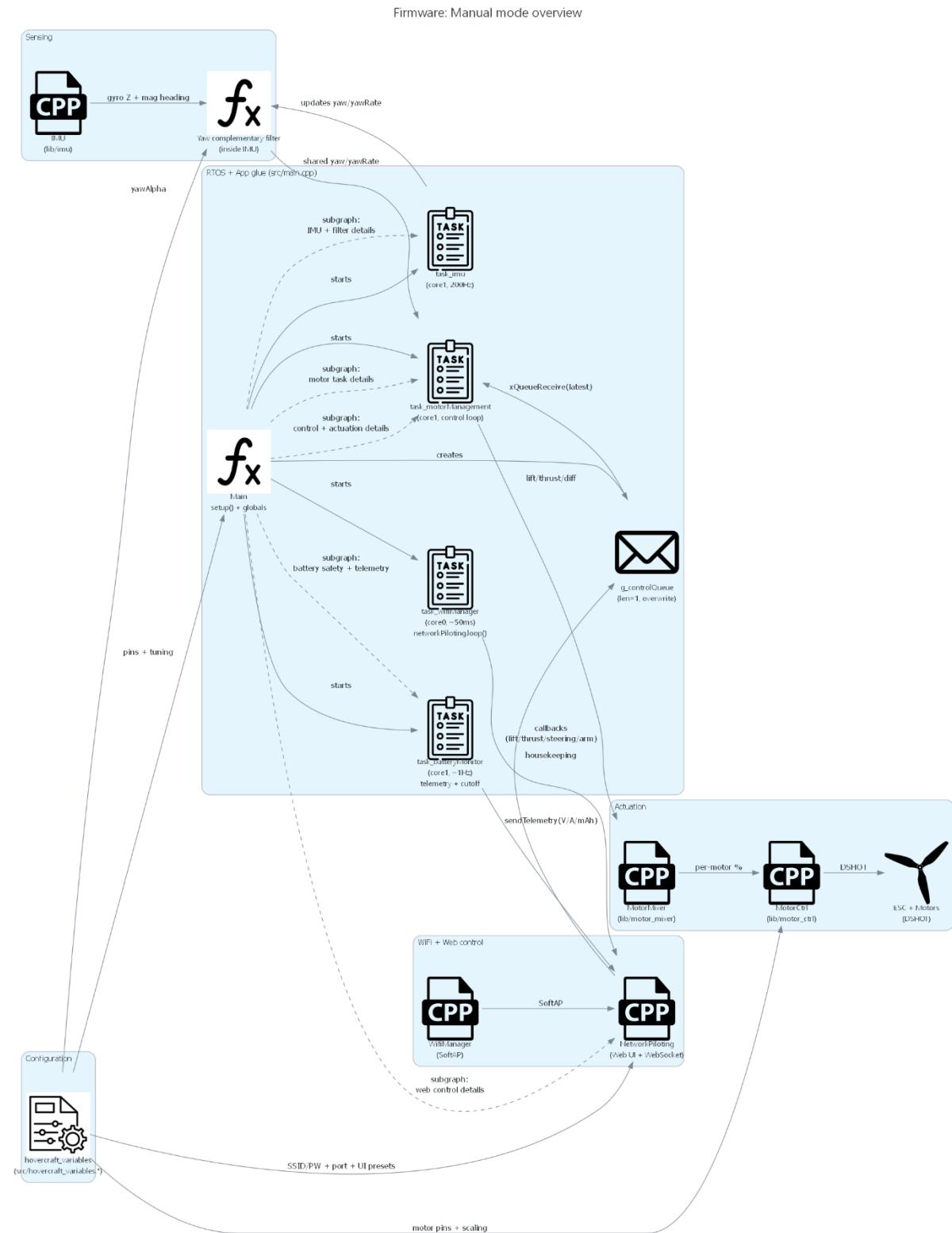
6.4.6 Parameters used (configuration & tuning)

Most parameters come from [src/hovercraft_variables.h](#) (with definitions in [src/hovercraft_variables.cpp](#)). Key groups:

- **Motor hardware & safety:** `global_PIN_MOTOR_*`, `global_MotorsReversed*`, `global_AllMotorsScalePercent`.
- **Control loop:** `global_ControlLoopRate_Hz`, `global_MaxYawRateSetpoint_dps`.
- **Yaw-rate PID:** `global_YawRatePid_Kp`, `global_YawRatePid_Ki`, `global_YawRatePid_Kd`, `global_YawRatePid_OutputLimit`, `global_YawRatePid_IntegratorLimit`.
- **IMU filter:** `global_ComplementaryFilter_yawAlpha`.
- **Web control scaling:** `global_WebThrustPresetPercent`.
- **WiFi:** `global_WifiApSsid`, `global_WifiApPassword`.

- **IR sensors:** `global_PIN_IR_SENSOR_*`, `global_IRSensorDistance_a_meters`.
- **Battery monitor:** `global_PIN_BATTERY_*`, `global_BatteryVoltage_VoltageDividerRatio`, `global_BatteryCurrent_VoltageDividerRatio`, `global_BatteryCurrent_SensorScaler_AmpsPerVolt`.

6.4.7 Schematic



6.5 Motor control (DShot ESC output)

6.5.1 Main idea

This module owns the four ESC outputs and converts high-level motor commands (percent per motor) into DShot3D throttle values, while enforcing safety rules (arm/disarm, emergency off) and some small helpers for real hardware behavior. DShot is used, because it is more save than a PWM. Look in the Setup->ESC settings chapter, to find out how the ESC is configured.

6.5.2 Where it is used / how it is called

- Class/API: [lib/motor_ctrl/include/motor_ctrl.h](#) (implementation in [lib/motor_ctrl/src/motor_ctrl.cpp](#)).
- Instantiated as a global object in [src/main.cpp](#) as `MotorCtrl motorCtrl(global_AllMotorsScalePercent)`.
- Initialized in `setup()` via `motorCtrl.init(...)` using pin + direction flags from [src/hovercraft_variables.h](#).
- During runtime, `task_motorManagement()` in [src/main.cpp](#) controls the safety gate with `motorCtrl.setMotorsEnabled(...)` and applies lift cut via `motorCtrl.applyLiftOff()`.
- The mixer [lib/motor_mixer/include/motor_mixer.h](#) holds a reference to the `MotorCtrl` instance and is the typical caller of the per-motor percent setters.

6.5.3 Functionalities implemented

- **DShot3D output mapping:** converts percent commands [-100,100] into DShot3D throttle commands [-999,999].
- **Global power scaling:** applies `pGeneralMotorPowerScalerPercent` so the project can cap motor power globally (useful to avoid overloading motors/ESC/battery).
- **Emergency safety gate:** `setMotorsEnabled(false)` forces **all** motors to 0 immediately and overrides any mixer/setpoint logic.
- **Lift-only cut:** `applyLiftOff()` stops the front motors (lift) without touching the rear motors (thrust).
- **Startup/arming behavior:** `init()` sends 0-throttle for a long sequence to satisfy typical ESC arming requirements.
- **Kick-start helper:** when commanding from standstill to a “large” throttle, sends a brief small kick (~20ms) before switching to the requested throttle (helps with motors that don’t start reliably from very low commands).
- **Direction handling:** supports per-motor wiring reversal flags, and applies a configurable reverse-direction compensation factor for negative commands.

6.5.4 Methods (overview)

- `MotorCtrl(generalMotorPowerScalerPercent)` – sets internal state to 0 and starts **disabled** (must be armed explicitly).
- `init(f1_pin, fr_pin, bl_pin, br_pin, reversedFL, reversedFR, reversedBL, reversedBR)` – installs + initializes 4 DShot ESCs and performs the 0-throttle arming sequence.
- `setMotorsEnabled(enabled) / motorsEnabled() / EmergencyOff()` – global arming gate / emergency stop.

- `applyLiftOff()` – immediately stops only the lift motors (front left/right).
- `setFrontLeftPercent() / setFrontRightPercent() / setBackLeftPercent() / setBackRightPercent()` – per-motor commands in percent.
- `setAllPercent(f1, fr, bl, br)` – convenience wrapper for the 4 setters.
- `getFrontLeftPercent() / ...` – returns the last commanded percentages.
- `tempForDebug_*` – direct throttle debug helpers (bypass percent mapping, still respects the safety gate).

6.5.5 Parameters used

6.5.5.1 Internal (class state)

- `pPercentFL/FR/BL/BR` – last requested percent commands.
- `pGeneralMotorPowerScalerPercent` – global percent scaling applied to all motors.
- `pMotorsEnabled` – safety gate; when false, outputs are forced to 0.
- `pSetReversed*` – per-motor reversal flags applied in software.
- `KickState` per motor – tracks the kick-start phase (last throttle, active flag, start time, pending desired command).

6.5.5.2 External (project config)

Used from [src/hovercraft_variables.h](#) / [src/hovercraft_variables.cpp](#):

- `global_AllMotorsScalePercent` – passed into the `MotorCtrl` constructor in [src/main.cpp](#).
- `global_PIN_MOTOR_FL/FR/BL/BR` – passed to `motorCtrl.init(...)`.
- `global_MotorsReversedFL/FR/BL/BR` – passed to `motorCtrl.init(...)` to invert commands per motor.
- `global_NegativeRpmScaleFactor` – multiplier applied to **negative** (reverse) motor commands before optional motor reversal. This is to combat insymetrical thrust due to the insymetrical propeller design.

6.6 Motor mixer (lift + thrust + yaw differential)

6.6.1 Main idea

This module converts high-level pilot/controller commands (lift, thrust, differential thrust) into four per-motor percent commands and forwards them to the motor controller.

6.6.2 Where it is used / how it is called

- Class/API: [lib/motor_mixer/include/motor_mixer.h](#) (implementation in [lib/motor_mixer/src/motor_mixer.cpp](#)).
- Instantiated as a global object in [src/main.cpp](#) as `MotorMixer motorMixer(motorCtrl)` (it requires an existing `MotorCtrl`).
- Initialized in `setup()` via `motorMixer.init()`.
- Called periodically from `task_motorManagement()` in [src/main.cpp](#): it sets lift/thrust/diff commands (including PID output) via `setLift()`, `setThrust()`, and `setDiffThrust()`.

6.6.3 Functionalities implemented

- **Lift mapping (front motors):** front-left and front-right always receive the same lift command in [0,100]%.
- **Rear thrust mapping (rear motors):** back-left/back-right receive a base thrust in [-100,100]% plus an *absolute differential* term for yaw control.
- **Differential authority limiting:** `diffThrust` is mapped to a fixed maximum delta (currently 30% of output range) so yaw control cannot fully dominate rear thrust.
- **Deadband around zero:** rear motors are forced to 0 when magnitude is below 5% to avoid “buzzing” around the ESC dead zone.
- **Clamping/saturation:** all commands are constrained to valid percent ranges before sending to `MotorCtrl`.

6.6.4 Methods (overview)

- `MotorMixer(MotorCtrl &motorCtrl)` – stores a reference to the motor controller used for output.
- `init()` – resets internal commands to 0 and sends 0% to all motors.
- `setLift(lift)` – updates lift command (0..100) and applies outputs.
- `setThrust(thrust)` – updates rear base thrust (-100..100) and applies outputs.
- `setDiffThrust(diffThrust)` – updates yaw differential command (-100..100) and applies outputs.
- `setLiftThrustDiff(lift, thrust, diffThrust)` – sets all three at once (avoids intermediate output states).
- `getLift() / getThrust() / getDiffThrust()` – returns last commanded values.

6.6.5 Parameters used

6.6.5.1 Internal (class state)

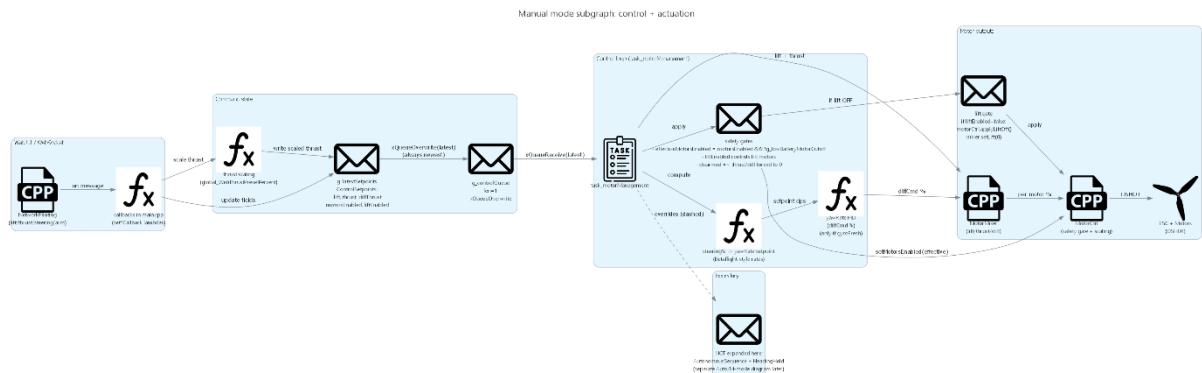
- `m_lift, m_thrust, m_diffThrustBalance` – current command state (percent).
- `m_motorCtrl` – reference to the owned motor output layer.
- Mixer constants in `updateOutputs()`:
 - `diffAbsMaxPercent = 30` (max yaw delta applied to rear motors)

- rear deadband of 5%

6.6.5.2 External (project config / other modules)

- No direct dependency on [src/hovercraft_variables.h](#) in this module.
- `setDiffThrust()` is typically fed by the yaw-rate PID in `task_motorManagement()` in [src/main.cpp](#) (PID constants live in [src/hovercraft_variables.h](#)).
- Final safety gating and scaling are handled in `MotorCtrl` (see [lib/motor_ctrl/include/motor_ctrl.h](#)).

6.6.6 Schematic



6.7 IMU (ADXL345+ITG3205+QMC5883+BMP280) + yaw estimation

6.7.1 Main idea

This module provides a single interface to the hovercraft's IMU sensors (accel/gyro/mag/baro) and maintains a yaw estimate using a complementary filter (fast gyro integration + slow magnetometer correction).

6.7.2 Where it is used / how it is called

- Class/API: [lib imu/include imu.h](#) (implementation in [lib imu/src imu.cpp](#)).
- Instantiated as a global object in [src/main.cpp](#) as IMU imu(global_ComplementaryFilter_yawAlpha).
- Initialized in the RTOS task `task_imu()` in [src/main.cpp](#) via `imu.init()`, then continuously updated.
- The IMU task publishes the measured yaw rate into shared globals (`g_yawRate-Measured_dps`, `g_lastYawRateUpdate_us`) which are consumed by `task_motorManagement()` for yaw-rate control.

6.7.3 Functionalities implemented

- **I2C + sensor bring-up:** starts I2C and probes/initializes each sensor; exposes presence flags (`hasAccel()`, `hasGyro()`, `hasMag()`, `hasBaro()`).
- **Raw sensor access in SI units:**
 - accelerometer in m/s^2 (`getAccel_raw()`)
 - gyro in rad/s (`getGyro_raw()`)
 - magnetometer raw axes + heading in degrees (`getMag_raw()`)
 - barometer temperature + pressure (`getEnv()`)
- **Accel reference calibration (optional):** averages accel samples while the craft is still/level to define a “level frame” and biases; enables corrected accel/angles (`getAccel_corrected()`, `getAngles_corrected()`).
- **Yaw complementary filter:** integrates gyro Z-rate at high rate and corrects drift toward magnetometer heading when available (`updateYawComplementaryFrom()`), using a tunable alpha.
- **Yaw reset to magnetometer:** `resetYawToMag()` forces the filter to re-initialize yaw on the next update.

6.7.4 Heading convention / mounting correction (important)

The magnetometer heading is **not** used “raw”. A hardcoded mounting correction is applied inside `IMU::getMag_raw()` to match the craft's physical mounting and desired positive rotation direction.

- Raw magnetometer heading (after declination) is computed from `atan2(Y, X)` and normalized to [0,360].
- Then we apply the transform:
 - `heading_corrected = wrap360(180 - heading_raw)`

This single expression does two things: - **180° offset** (board mounted reversed: world 0° corresponds to craft 180°) - **Direction inversion** (so the heading increases in the desired “positive” rotation sense)

Because this mapping inverts heading direction, the yaw complementary filter must use a **matching gyro integration sign** so that gyro prediction and magnetometer correction agree. This is handled inside `IMU::updateYawComplementaryFrom()`.

6.7.5 Methods (overview)

- `IMU(yawAlpha)` – constructs the sensor objects and sets yaw filter alpha.
- `init()` – starts I2C, probes sensors, sets ranges/modes, and resets yaw filter state.
- `has*() / isReady()` – capability/status flags.
- `getAccel_raw(), getGyro_raw(), getMag_raw(), getEnv()` – sensor readouts.
- `setDeclinationDeg()` – sets magnetometer declination correction.
- `getAngles_raw()` – roll/pitch from accel (yaw not observable here, returned as 0).
- `calibrateAccelReference() / isCalibrated()` – establish the reference tilt/gravity for corrected accel.
- `getAccel_corrected(), getAngles_corrected()` – accel rotated into the calibrated level frame.
- `updateYawComplementary() / updateYawComplementaryFrom(gz_rad_s, mag_heading_deg)` – yaw estimation update.
- `setYawFilterAlpha(alpha), resetYawToMag(), getYaw_deg(), getYaw-Gyro_deg()` – yaw filter configuration/output.

6.7.6 Parameters used

6.7.6.1 Internal (class state)

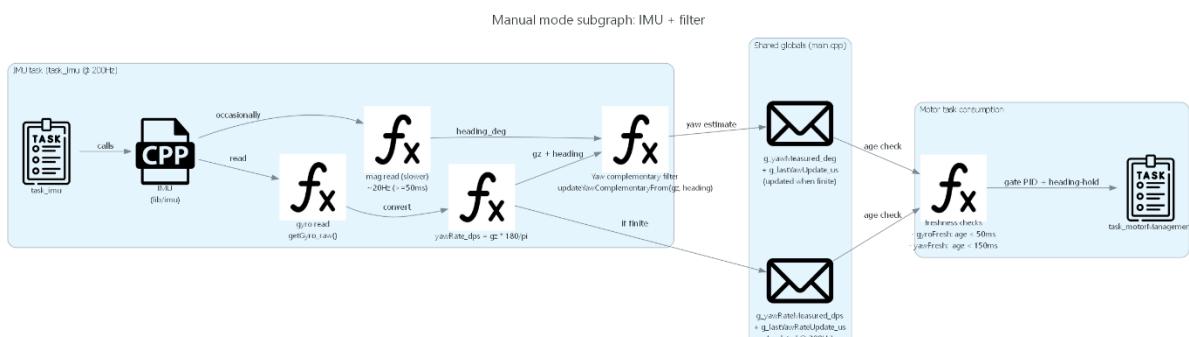
- Sensor presence flags (`_accel0k, _gyro0k, _mag0k, _bmp0k`) and `_ready`.
- Yaw filter state: `_yaw_deg, _yawGyro_deg, _yawAlpha, _lastYawUpdate_us, init` flags.
- Accel calibration: `_roll0_deg, _pitch0_deg, _ax_bias, _ay_bias, _g0`.
- Declination: `_declinationRad`.

6.7.6.2 External (project config)

Used in [src/main.cpp](#) / [src/hovercraft_variables.h](#): - `global_ComplementaryFilter_yawAlpha` – sets how strongly yaw trusts the gyro vs. magnetometer ($\alpha \in [0,1]$).

Note: the IMU module itself starts I2C using `SDA_PIN/SCL_PIN` only if those macros are defined at build time; otherwise it calls `Wire.begin()` with default pins.

6.7.7 Schematic



6.8 PID controller (generic, rate-control oriented)

6.8.1 Main idea

This module provides a lightweight PID controller that turns a setpoint/measurement error into a bounded output command, with practical safeguards (clamping, anti-windup, and derivative-on-measurement).

6.8.2 Where it is used / how it is called

- Class/API: [lib/pid_controller/include/pid_controller.h](#) (implementation in [lib/pid_controller/src/pid_controller.cpp](#)).
- Instantiated locally in the motor/control task (PIDController yawRatePid;) inside `task_motorManagement()` in [src/main.cpp](#).
- Initialized once via `yawRatePid.init(...)` using constants from [src/hovercraft_variables.h](#), then called each control loop with `yawRatePid.update(setpoint, measurement, dt_s)`.
- Called **once per iteration** of `task_motorManagement()`. The task is paced by `vTaskDelayUntil(...)` to a *target* period derived from `global_ControlLoopRate_Hz` (see [src/hovercraft_variables.h](#)).
- The `dt_s` passed into `update()` is computed from `micros()` inside the task (so it reflects real loop timing, not just the requested RTOS delay) and is clamped to [0.001,0.05] seconds before being used.
- The current output is used as a yaw differential command and applied through `MotorMixer::setDiffThrust()` (see [lib/motor_mixer/include/motor_mixer.h](#)).

6.8.3 Functionalities implemented

- **P, I, D terms:** standard PID structure with configurable gains.
- **Derivative on measurement:** uses $-k_d \frac{d(measurement)}{dt}$ to avoid derivative kick from setpoint steps.
- **Output limiting:** clamps controller output to `±outputLimit`.
- **Integrator limiting:** clamps the I-term accumulator to `±integratorLimit`.
- **Simple anti-windup:** if output saturates and the current error would push further into saturation, the integrator is frozen.
- **dt sanity guard:** returns 0 and clears derivative memory if `dt` is invalid or too large (currently `dt > 0.2s`).

6.8.4 Methods (overview)

- `init(kp, ki, kd, outputLimit, integratorLimit)` – sets gains/limits, resets state, marks controller initialized.
- `update(setpoint, measurement, dt_s)` – computes one PID step and returns the saturated output.
- `reset()` – clears integrator and derivative memory (useful when disarming or when measurements are stale).
- `isInitialized()` – indicates whether `init()` has been called.

6.8.5 Parameters used

6.8.5.1 Internal (class state)

- `_kp, _ki, _kd` – gains for yaw.

- `_integrator` – accumulated I term.
- `_prevMeasurement`, `_havePrevMeasurement` – storage for derivative term.
- `_outputLimit`, `_integratorLimit` – clamps.

6.8.5.2 External (current usage in this project)

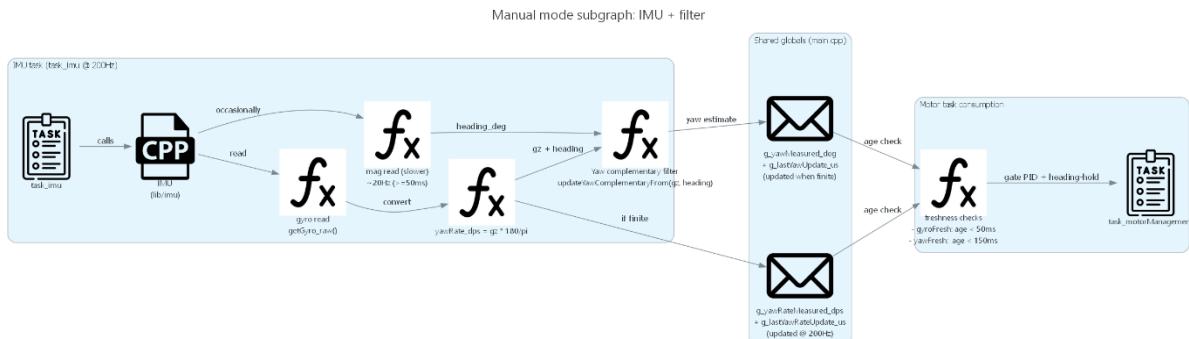
Configured in `task_motorManagement()` in [src/main.cpp](#) from [src/hovercraft_variables.h](#):

- `global_YawRatePid_Kp`, `global_YawRatePid_Ki`, `global_YawRatePid_Kd`
- `global_YawRatePid_OutputLimit` (in mixer output units, typically `diffThrust` in -100..100)
- `global_YawRatePid_IntegratorLimit`
- `global_MaxYawRateSetpoint_dps` – used by the caller to convert the pilot “steering” percent command into a yaw-rate setpoint (deg/s) before feeding it into the PID.

Inputs/units are intentionally generic:

- setpoint and measurement must share units (here: yaw-rate in *deg/s*).
- `dt_s` is seconds.
- output unit is user-defined (here: `MotorMixer::setDiffThrust()` percent command).

6.8.6 Schematic



6.9 Battery monitor (ADC voltage + current + used capacity)

6.9.1 Main idea

This module reads battery voltage and current from ADC pins, applies simple calibration (divider ratios + current scale), and integrates current over time to estimate used capacity in mAh.

6.9.2 Where it is used / how it is called

- Class/API: [lib/battery_monitor/include/battery_monitor.h](#) (implementation in [lib/battery_monitor/src/battery_monitor.cpp](#)).
- Instantiated as a global object in [src/main.cpp](#) and initialized in `setup()` via `batteryMonitor.init(...)`.
- Updated periodically inside the RTOS task `task_batteryMonitor()` in [src/main.cpp](#), which also forwards telemetry through `networkPiloting.sendTelemetry(v, a, mah)`.

6.9.3 Functionalities implemented

- **Voltage measurement (V):** reads an ADC pin in mV and scales it back to battery voltage using a divider ratio.
- **Current measurement (A):** reads an ADC pin in mV, applies an optional sensor divider ratio + optional zero-current offset, then converts sensor mV to amps using a Betaflight-style scale.
- **Used capacity integration (mAh):** integrates $I(t)$ over time using `millis()` delta time to estimate consumed mAh.
- **ESP32 ADC convenience:** on ESP32, uses `analogReadMilliVolts()` and configures ADC resolution + attenuation; other targets fall back to a generic 10-bit/3.3V conversion.
- **Low-voltage handling (system integration):** the RTOS battery task uses the measured battery voltage to drive a low-battery LED indicator and to trigger a motor cutoff when voltage stays below a configured threshold for multiple samples.

6.9.4 Methods (overview)

- `BatteryMonitor()` – sets safe defaults (pins disabled, zeroed outputs).
- `init(voltageAdcPin, currentAdcPin, voltageDividerRatio, currentDividerRatio, betaflightScale_0p1mVPerAmp)` – configures pins and calibration constants; captures initial timestamp.
- `setCurrentOffset_mV(offset_mV)` – removes a fixed sensor offset (for sensors not centered at 0mV @ 0A).
- `update()` – reads ADCs, updates `voltage_V`, `current_A`, and accumulates `used_mAh` based on elapsed time.
- `getVoltage() / getCurrent() / getMAH()` – accessors for the latest computed values.
- `setMAH(mAh)` – allows resetting/overwriting the integrated capacity counter.

6.9.5 Parameters used

6.9.5.1 Internal (class state)

- `_voltageAdcPin, _currentAdcPin` – ADC pins for voltage/current.
- `_voltageDividerRatio` – converts ADC pin voltage to battery voltage.

- `_currentDividerRatio` – converts ADC pin voltage to the sensor voltage (useful if there is an additional divider).
- `_betaflightScale_0p1mVPerAmp` – current conversion constant (see next section).
- `_currentOffset_mV` – optional zero-current offset.
- `_lastUpdateMs` – timestamp used to compute Δt for mAh integration.

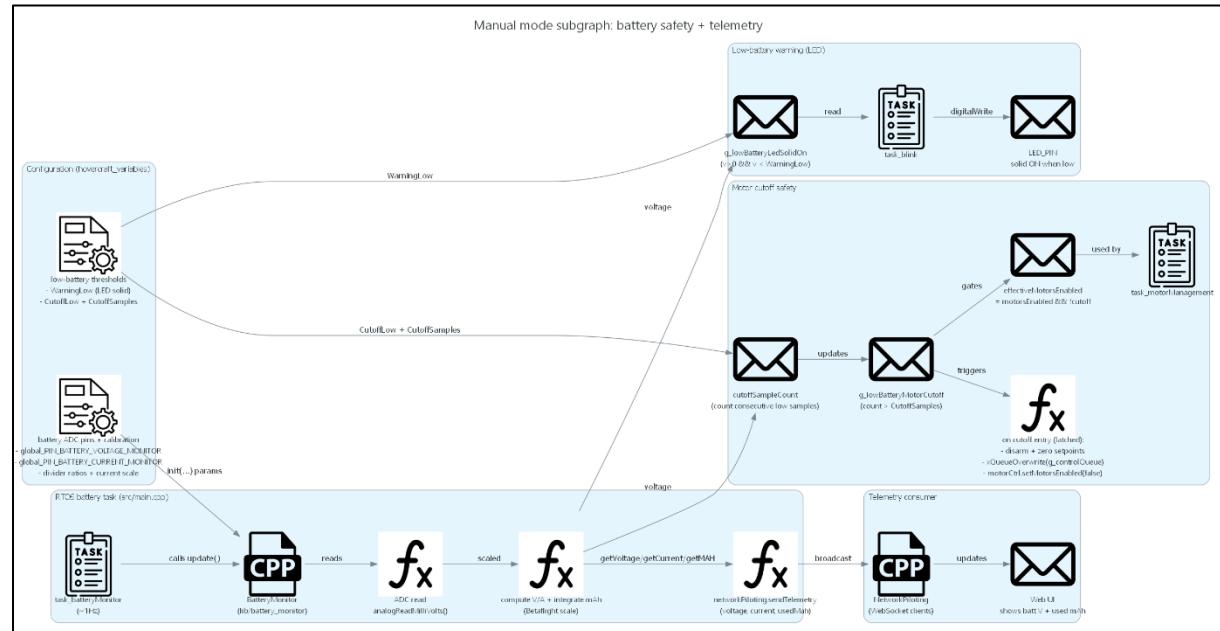
6.9.5.2 External (from hovercraft variables)

Configured in `setup()` in [src/main.cpp](#) using constants declared in [src/hovercraft_variables.h](#): - `global_PIN_BATTERY_VOLTAGE_MONITOR`, `global_PIN_BATTERY_CURRENT_MONITOR` - `global_BatteryVoltage_VoltageDividerRatio` - `global_BatteryCurrent_VoltageDividerRatio` - `global_BatteryCurrent_SensorScaler_AmpsPerVolt`

Low-voltage thresholds used by the battery task in [src/main.cpp](#): - `global_BatteryVoltageLow_WarningLow` – below this battery voltage the status LED is kept solid ON as a low-battery indicator. - `global_BatteryVoltageLow_MotorCutoffLow` and `global_BatteryVoltageLow_MotorCutoffSamples` – motors are disabled if battery voltage is below the cutoff threshold for more than the configured number of consecutive samples.

Note: `BatteryMonitor::init()` expects the last parameter in **Betaflight units: scale in [0.1 mV]/A** (example: 130 means 13 mV/A). The variable name `global_BatteryCurrent_SensorScaler_AmpsPerVolt` is misleading compared to the code/comments; treat it as the Betaflight-style scale value used by this module.

6.9.6 Schematic



6.10 WiFi manager

6.10.1 Main idea

This module wraps the ESP32 Arduino WiFi API to start and manage a WiFi Access Point (SoftAP) used for the hovercraft's web UI / telemetry links.

6.10.2 Where it is used / how it is called

- Class/API: [lib/wifi_manager/include/wifi_manager.h](#) (implementation in [lib/wifi_manager/src/wifi_manager.cpp](#)).
- Instantiated as a global object in [src/main.cpp](#).
- The AP is started once during `setup()` via `wifiManager.startAccessPoint(global_WifiApSsid, global_WifiApPassword)`.
- Runtime “WiFi/Web handling” runs inside `task_wifiManager()` in [src/main.cpp](#), which is created with `xTaskCreatePinnedToCore(..., 0)`, i.e. pinned to **ESP32 core 0** (while most other tasks run on core 1).

6.10.3 Functionalities implemented

- **Start SoftAP:** configures WiFi.mode(WIFI_AP) and starts WiFi.softAP(...) with SSID/password/channel/max connections.
- **Low-latency radio setting:** calls WiFi.setSleep(false) to keep the WiFi link responsive.
- **Connection / IP helpers:** small wrappers to check “connected” state and return the current IP address (AP IP vs station IP).

6.10.4 Methods (overview)

- `startAccessPoint(ssid, password, channel=1, maxConnections=4)` – starts the ESP32 SoftAP and logs IP to Serial.
- `isConnected()` – returns true if AP is active or station is connected.
- `getIp() / getIpString()` – returns IP depending on current WiFi mode.

6.10.5 Parameters used

6.10.5.1 Internal (class state)

- `apActive_` – remembers whether the SoftAP was successfully started.

6.10.5.2 External (project config)

Provided by [src/hovercraft_variables.h](#) / [src/hovercraft_variables.cpp](#):

- `global_WifiApSsid, global_WifiApPassword` (passed to `startAccessPoint()` in [src/main.cpp](#))

Note: `global_WebServerPort` exists in the project config, but it is used by the web-server/network layer (see `network_piloting`), not by `WifiManager` itself.

6.11 Network piloting (Web UI + WebSocket control/telemetry)

6.11.1 Main idea

This module hosts a small, embedded controller webpage and a WebSocket endpoint to send pilot commands (lift/thrust/steering + arm) to the firmware and to stream basic telemetry back to the browser.

6.11.2 Where it is used / how it is called

- Class/API: [lib/network_piloting/include/network_piloting.h](#) (implementation in [lib/network_piloting/src/network_piloting.cpp](#)).
- Instantiated as a global object in [src/main.cpp](#).
- Started in `setup()` via `networkPiloting.begin()` after the AP is created by `wifiManager.startAccessPoint(...)`.
- `setLiftCallback()`, `setThrustCallback()`, `setSteeringCallback()`, `setArmCallback()` are wired in [src/main.cpp](#) to push the received commands into the control queue.
- Runtime maintenance (`networkPiloting.loop() → ws_.cleanupClients()`) is called from `task_wifiManager()` in [src/main.cpp](#), which is pinned to ESP32 core 0.
- Telemetry broadcast is fed from the battery task via `networkPiloting.sendTelemetry(v, a, mah)` in [src/main.cpp](#).

6.11.3 Functionalities implemented

6.11.3.1 Website UI

- **Embedded controller page:** the HTML/CSS/JS UI is stored in flash (PROGMEM) and served for `/`, `/controller.html`, and as a fallback for unknown routes.
- **WebSocket link (`/ws`):** the browser opens a WebSocket and sends JSON messages like `{ "lift": 60, "thrust": -10, "steering": 25, "motorsEnabled": true }`.
- **Telemetry to UI:** the firmware can broadcast compact JSON frames to all clients (currently battery voltage/current/used mAh).
- **Client-side safety UX:** the page springs controls back to zero on pointer release and also resets on tab blur / page hidden / WS disconnect.

6.11.3.2 Control semantics + safety

- **Input shaping on the webpage:** the drag pads map pointer position to a normalized command in $[-100,100]$ with a deadzone and an expo curve (smooth low-sensitivity near center).
- **Lift presets:** the UI toggles lift through a preset list; the preset values and start index are injected by the firmware at page-serve time.
- **Fail-safe on disconnect:** on WebSocket disconnect, the firmware forces `motorsEnabled=false` and calls the user callbacks with `lift=0, thrust=0, steering=0`.
- **Clamping + lightweight parsing (firmware-side):** incoming values are clamped to valid ranges and parsed without a full JSON dependency (simple string searches + `toFloat()`).

6.11.4 Methods (overview)

- `begin()` – configures routes for the embedded page, attaches the WebSocket handler, and starts the async webserver.
- `loop()` – performs periodic housekeeping (`cleanupClients()`).
- `set*Callback(...)` – registers callbacks for lift/thrust/steering/arm updates.
- `sendTelemetry(voltage, current, usedMah)` – broadcasts telemetry JSON to all connected clients (no-op if no clients).
- `getLift(), getThrust(), getSteering(), motorsEnabled()` – returns latest received command state.

6.11.5 Parameters used

6.11.5.1 Internal (class state)

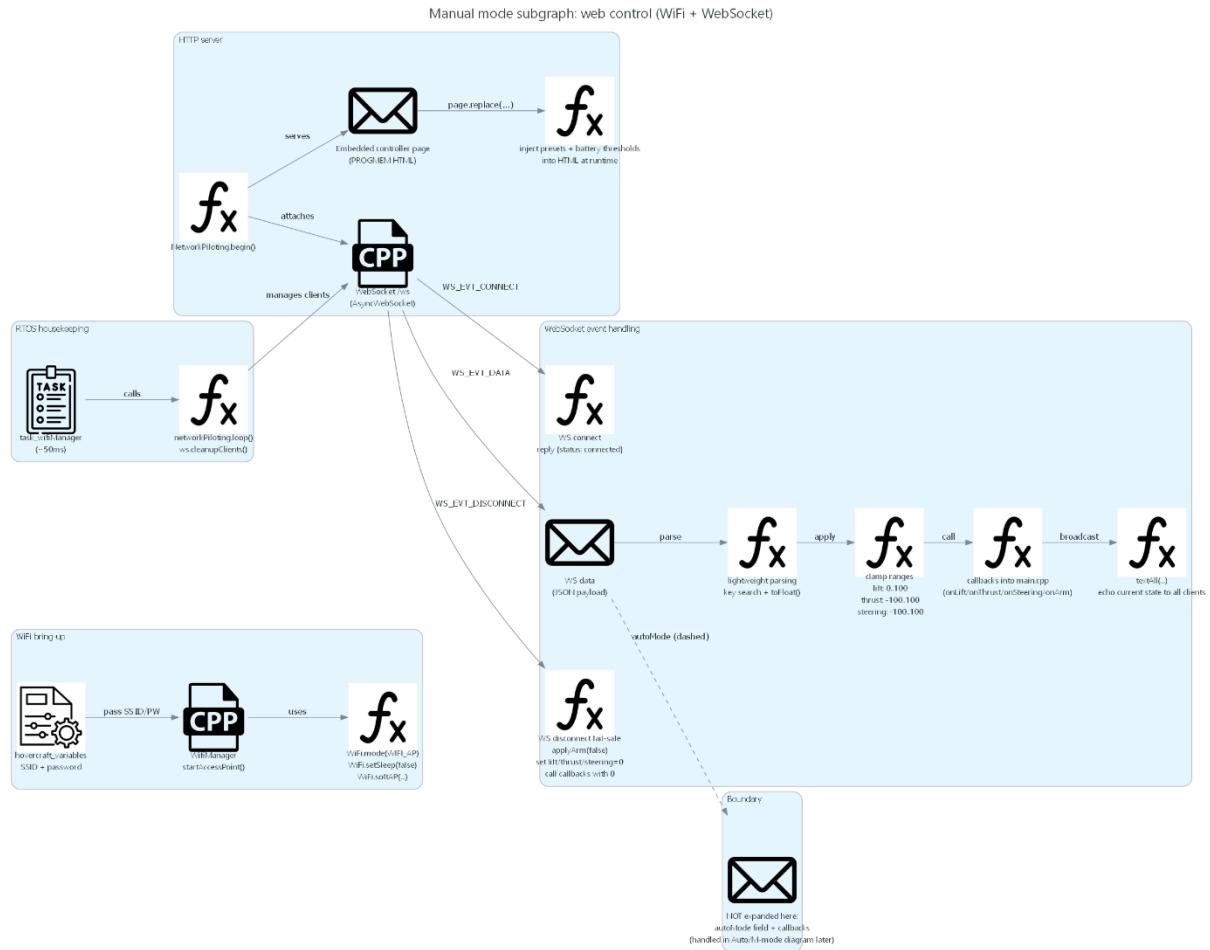
- `server_` – `AsyncWebServer` instance (constructed with the configured port).
- `ws_` – `AsyncWebSocket` at path `/ws`.
- `lift_, thrust_, steering_, motorsEnabled_` – last received command values.
- `onLift_, onThrust_, onSteering_, onArm_` – callback hooks into the rest of the firmware.

6.11.5.2 External (project config)

Used from [src/hovercraft_variables.h](#) / [src/hovercraft_variables.cpp](#):

- `global_WebServerPort` – port used by the HTTP server.
- `global_WebLiftPresetPercent_Array`, `global_WebLiftPresetPercent_Array_len` – lift preset list injected into the webpage.
- `global_WebLiftPresetPercent_Array_startIndex` – which preset becomes active first when toggling lift ON.

6.11.6Schematic



6.12 IR sensors (line detection)

6.12.1 Main idea

This module turns three downward-facing IR sensor signals into (1) an angle α between the hovercraft and the line, and (2) a velocity component perpendicular to the line v_{\perp} . This can later be used for navigation and/or to correct long-term drift from integrated sensors (e.g. yaw).

6.12.2 Where it is used / how it is called

- The class is `IRSensors` in [lib/ir_sensors/include/ir_sensors.h](#) with the implementation in [lib/ir_sensors/src/ir_sensors.cpp](#).
- A global instance is constructed in [src/main.cpp](#) using pins and geometry from [src/hovercraft_variables.h](#).
- Runtime is split into **two RTOS tasks** in [src/main.cpp](#):
 - `task_irSensors`: samples the 3 sensors via ADC and enqueues samples.
 - `task_calcIrSensors`: drains the queue, runs line detection, then reads `alpha` and `v_perp` via getters.

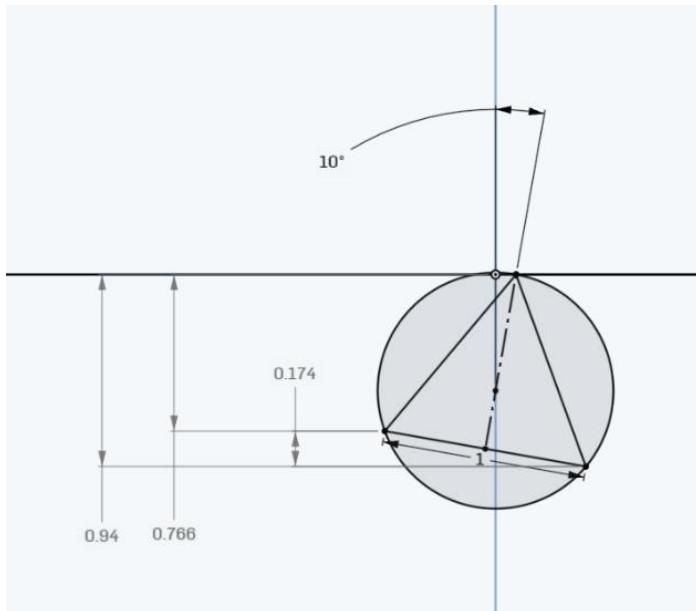
6.12.3 Functionalities implemented (global view)

- **High-rate sampling pipeline:** fast ADC sampling is decoupled from slower processing using a FreeRTOS queue (`enqueueSample` / `processQueue`).
- **Robust line event detection:** per-sensor rising-edge detection with a configurable threshold + hysteresis and an “armed/disarmed” latch to avoid repeated triggers.
- **Timeout handling:** if a line crossing is incomplete (only some sensors triggered) for too long, the partial state is reset.
- **Geometry-based estimation:** when all 3 sensors trigger for a crossing event, the code computes:
 - α (degrees): relative angle of the craft vs. the line.
 - v_{\perp} (m/s): speed perpendicular to the line, with sign inferred from which sensors cross first.

6.12.4 How the algorithm works (conceptual)

4. The sampling task reads 3 ADC values (BM/FL/FR) and timestamps each sample with `micros()`.
5. The processing task runs `processQueue(threshold, hysteresis)` which calls `detectLine(...)` for each sample.
6. `detectLine(...)` looks for a **rising crossing event** per sensor:
 - below threshold → re-arm
 - rising above (threshold + hysteresis) → trigger once and store timestamp
7. When all three sensors have triggered for the same crossing, the module computes α and v_{\perp} using the configured sensor geometry.

To validate the calculations, we tested the same in a small CAD sketch, and got the correct results:



$$\begin{aligned}
 d_{12x} &= \cos(\alpha - 30^\circ), \\
 d_{13x} &= \cos(\alpha + 30^\circ), \\
 d_{23x} &= -\sin(\alpha).
 \end{aligned}$$

6.12.5 Methods overview

- `IRSensors(pinBM, pinFL, pinFR, a_m, b_m)`
 - Stores wiring + geometry. In this project the index mapping is **0=BM, 1=FL, 2=FR**.
- `begin()`
 - Creates the internal FreeRTOS sample queue used to decouple sampling from processing.
- `enqueueSample(values[3], t_us[3])`
 - Pushes one sample triplet into the queue (non-blocking).
- `processQueue(threshold, hysteresis)`
 - Drains the queue and calls `detectLine(...)` for each sample.
- `detectLine(values[3], t_us[3], threshold, hysteresis)`
 - Implements the actual line edge detection, timeout handling, and (when complete) α / v_\perp computation.
- `getAlphaToLine(), getVelocityPerpToLine()`
 - Return the last computed results (may be NAN if no valid crossing happened yet).
- `hasNewMeasurement(), consumeNewMeasurement()`
 - Optional “new data” flag; can be used by callers to only react once per computed crossing.

6.12.6 Parameters used (internal + extern)

From construction / internal state:

- Sensor pins (int): the three ADC pins (BM/FL/FR).
- Geometry (meters):
 - `_a`: side length from BM to FL/FR.
 - `_b`: base length between FL and FR.

- Internal trigger state: per-sensor `_armed[]`, `_prevValue[]`, and timestamps `_t1_us/_t2_us/_t3_us`.

From [`src/hovercraft_variables.h`](#): - `global_PIN_IR_SENSOR_BM`, `global_PIN_IR_SENSOR_FL`, `global_PIN_IR_SENSOR_FR` (pins). - `global_IRSensorDistance_a_meters`, `global_IRSensorDistance_b_meters` (geometry). - `global_IRSensor_Threshold` (ADC threshold) and `global_IRSensor_Hysteresis` (hysteresis band). - `global_IRSensor_Timeout_us` (max allowed time for an incomplete crossing before reset).

Notes for maintainers:

- The current firmware uses **ADC sampling + queue**, not GPIO interrupts, for the IR sensors (see `task_irSensors` / `task_calcIrSensors` in [`src/main.cpp`](#)).
- Queue sizing (`SAMPLE_QUEUE_LEN`) is large in the header; if you run into RAM pressure, reducing it is a first place to look.

6.13 Heading controller (outer loop for cascaded yaw control)

6.13.1 Very important note

The magnetometer needs to be **calibrated** to work as expected. This is explained under chapter 4 in the documentation, and by the use of the files in this GitHub repo under “**3_Firmware >> Calibrate_Magnetometer**”!!! The values than need to be **copied over to hovercraft_variables.cpp!!!**

6.13.2 Note about autonomous mode (M-Mode)

The autonomous “M-Mode” logic (state machine + the 2s calibration/averaging phase) lives in [lib/autonomous_sequence/notesForFunctionalDocumentation_autonomous_sequence.md](#). This file only covers the **heading PID building block** that M-Mode uses. ### Main idea This module implements the **outer loop** of a cascaded yaw controller:

- **Outer loop (this module):** heading error (deg) → yaw-rate setpoint (deg/s)
- **Inner loop (existing):** yaw-rate setpoint (deg/s) → mixer diffThrust command (percent)

This allows commanding an **absolute heading** (relative to North as provided by the magnetometer-corrected complementary filter) instead of commanding yaw rate directly.

6.13.3 Where it is used / how it is called

- Class/API: [lib/heading_controller/include/heading_controller.h](#)
- Implementation: [lib/heading_controller/src/heading_controller.cpp](#)
- Intended call site: the motor/control task (`task_motorManagement()`) in [src/main.cpp](#), when autoMode/heading-hold is enabled.

6.13.4 Functionalities implemented

- **Angle wrap-around handling:** computes smallest signed heading error in [-180,180] degrees.
- **PID structure:** P + I + D terms.
- **Derivative on measurement:** uses measured yaw rate (deg/s) as heading derivative.
- **Output limiting:** clamps yaw-rate setpoint to `outputLimit_dps`.
- **Integrator limiting + anti-windup:** clamps I-term accumulator and freezes it when saturated and the error would push further.

6.13.5 Configuration parameters

From project settings in [src/hovercraft_variables.h](#): - `global_HeadingPid_Kp`, `global_HeadingPid_Ki`, `global_HeadingPid_Kd` - `global_HeadingPid_OutputLimit_dps` - `global_HeadingPid_IntegratorLimit_dps`

6.13.6 Notes / assumptions

- Headings are treated as degrees and normalized to [0,360).
- The controller output is a yaw-rate request (deg/s). The inner yaw-rate controller must still be active for actual actuation.

6.13.6.1 Sign convention (important)

This controller assumes:

- **Positive heading error** means the craft should rotate in the direction that makes the reported heading increase.
- `yawRateMeasured_dps` uses the **same sign convention** as the heading derivative (positive yaw rate = heading increasing).

The IMU heading used by heading-hold is already mounting-corrected (including a hard-coded 180° offset and direction inversion). If you change that convention, update the yaw-rate sign accordingly, especially if using a non-zero Kd (derivative-on-measurement).

6.14 Autonomous_sequence (M-Mode Sequencer)

6.14.1 Purpose

`autonomous_sequence` implements the autonomous competition behavior executed when “M-Mode” (Auto Mode) is enabled. It is implemented as a **small, deterministic state machine** that runs inside the motor/control task.

High-level behavior:

- Calibrate an initial **start heading** by averaging the complementary-filter heading for 2 seconds.
- Wait for motors to be armed.
- Drive “blind” for a short time to pass the start line.
- Then run an **event-driven loop** where each **IR line crossing event** recomputes the next heading setpoint.
- Allow abort at any time (user toggles M off / stop request / motors disarm).

The line-driven pattern after the blind start is intended to approximate the sector pattern:

- $-90^\circ, -90^\circ, 0^\circ, 0^\circ$ (repeat)

6.14.2 Integration (where it runs)

- The UI / mode logic triggers:
 - `AutonomousSequence::requestStart()` when M-Mode is enabled.
 - `AutonomousSequence::requestStop()` when M-Mode is disabled.
- The motor/control task ticks it once per control loop via `AutonomousSequence::update(...)`.
- The motor/control task applies outputs:
 - If `overrideThrust()` is true, it uses `thrustOverride_percent()` instead of user thrust.
 - If `wantsHeadingHold()` is true, it enables heading-hold using `headingTarget_deg()` as setpoint.
 - If `consumeExitRequest()` returns true (one-shot), the caller should force motors off and leave M-Mode.

Important note: `headingTarget_deg()` is a **persistent setpoint**. The implementation intentionally does **not** reset it at the start of each tick; it must remain valid between line events.

6.14.3 Public interface

Entry points:

- `requestStart()`
 - Latches a start request; the next tick in `Idle` enters calibration.
- `requestStop()`
 - Latches a stop request; the next tick transitions to `ExitRequested` (if active).

- `update(nowMs, heading_deg, headingFresh, motorsEnabled, lineEventFresh, lineAlpha_deg, lineVPerp_mps, headingAtLine_deg)`
 - Advances the state machine and updates output commands.

Observability helpers:

- `isActive()`, `state()`
- `startHeading_deg()`
- `lastLineAlpha_deg()`, `lastLineVelocityPerp_mps()` (latched on every `lineEventFresh`)

6.14.4 Inputs (to `update()`)

- `nowMs`: current `millis()`.
- `heading_deg`: current heading estimate (deg). Used mainly for calibration.
- `headingFresh`: when true, the heading sample is accumulated during calibration.
- `motorsEnabled`: whether motors are armed/enabled.
- `lineEventFresh`: one-shot flag indicating a **new** line-crossing event this tick.
- `lineAlpha_deg`: IR-derived line angle α (deg). Valid only when `lineEventFresh==true`.
- `lineVPerp_mps`: IR-derived perpendicular speed (m/s). Currently only latched for debugging/telemetry.
- `headingAtLine_deg`: heading snapshot taken when the IR module generated the line event.

Why `headingAtLine_deg` matters:

- The setpoint recomputation uses the heading that corresponds to the same instant as the IR measurement.
- The caller should ensure `lineAlpha_deg` and `headingAtLine_deg` are from the same event.

6.14.5 Outputs (polled by the caller)

- Thrust override:
 - `overrideThrust()`
 - `thrustOverride_percent()`
- Heading-hold command:
 - `wantsHeadingHold()`
 - `headingTarget_deg()`
- Exit handshake:
 - `consumeExitRequest()` returns true exactly once per exit request.

6.14.6 State machine (current implementation)

States are defined in AutonomousSequence::State.

```
State machine:
switch (state_)
{
    case State::Calibrating:
    case State::WaitingForArm:
    case State::StartBlind:

    case State::DriveStraight_FirstSector:
    case State::DriveStraight_SecondSector:
    case State::DriveCurve_90_FirstSector:
    case State::DriveCurve_90_SecondSector:

    case State::ExitRequested:
    case State::Idle:
    default:
        break;
}
```



6.14.6.1 Idle

- Not running.
- A latched start request transitions to **Calibrating**.

6.14.6.2 Calibrating (2.0 s)

- Motors commanded to stop (thrust override = 0%, heading-hold disabled).
- While `headingFresh==true`, the module accumulates heading samples.
- After 2000 ms:
 - Computes `startHeading` as a circular mean (`sin/cos average`).
 - If motors are armed: sets `headingTarget = startHeading` and transitions to **StartBlind**.
 - Else transitions to **WaitingForArm**.

6.14.6.3 WaitingForArm (indefinite)

- Motors commanded to stop (thrust override = 0%, heading-hold disabled).
- When `motorsEnabled==true`: sets `headingTarget = startHeading` and transitions to **StartBlind**.

6.14.6.4 StartBlind (1.0 s)

Goal: pass the start line without relying on the IR line event.

- Thrust override: **30%**
- Heading-hold: enabled
 - `headingTarget = startHeading`

- After 1000 ms: transitions to **DriveStraight_SecondSector**.

6.14.6.5 DriveStraight_SecondSector (event-driven)

This state holds the current heading target and waits for a line event.

- Thrust override: **10%**
- Heading-hold: enabled
 - headingTarget = headingTarget (keep previous)

On `lineEventFresh==true`:

- Computes next heading setpoint for the first “-90° curve sector” using:

$$\text{headingTarget} = \text{wrap360}(\text{headingAtLine} - \alpha - 90^\circ - |\text{driftFirst}|)$$

where the current hard-coded drift compensation is:

- `driftFirst = 57.0°`

Then transitions to **DriveCurveMinus90_FirstSector**.

6.14.6.6 DriveCurveMinus90_FirstSector (event-driven)

- Thrust override: **10%**
- Heading-hold: enabled
- Waits for the next line event.

On `lineEventFresh==true`:

$$\text{headingTarget} = \text{wrap360}(\text{headingAtLine} - \alpha - 90^\circ - |\text{driftSecond}|)$$

where the current hard-coded drift compensation is:

- `driftSecond = 7.5°`

Then transitions to **DriveCurveMinus90_SecondSector**.

6.14.6.7 DriveCurveMinus90_SecondSector (event-driven)

- Thrust override: **10%**
- Heading-hold: enabled
- Waits for the next line event.

On `lineEventFresh==true` (end of “-90° segment”):

$$\text{headingTarget} = \text{wrap360}(\text{headingAtLine} - \alpha)$$

Then transitions to **DriveStraight_FirstSector**.

6.14.6.8 DriveStraight_FirstSector (event-driven)

- Thrust override: **10%**
- Heading-hold: enabled
- Waits for the next line event.

On `lineEventFresh==true`:

```
headingTarget = wrap360(headingAtLine - α)
```

Then transitions back to **DriveStraight_SecondSector**.

6.14.6.9 Loop summary

After the blind start, the loop is:

- **DriveStraight_SecondSector** → (line) **DriveCurveMinus90_FirstSector**
→ (line) **DriveCurveMinus90_SecondSector** → (line)
DriveStraight_FirstSector → (line) **DriveStraight_SecondSector** (repeat)

6.14.7 Stop/abort behavior and safety

- **requestStop()** can be called at any time.
 - On the next tick, if the sequence is active, it transitions to **ExitRequested**.
- If motors are disarmed while the sequencer is in any “motion” state (**StartBlind**, the straight states, or the curve states), it transitions to **ExitRequested**.
- **ExitRequested** is a one-tick state:
 - Commands thrust override 0% and disables heading-hold.
 - Latches an exit request (**consumeExitRequest()** will return true once).
 - Immediately returns to **Idle** in the same tick.

6.14.8 Implementation notes / gotchas

- Angle wrap-around: all computed headings are wrapped into [0,360) via **wrap360()**.
- Calibration uses a circular mean (sin/cos accumulation) to avoid errors near 0/360.
- Debug prints: the implementation currently emits **Serial.println(...)** messages on transitions and line events.

6.14.9 Files

- lib/autonomous_sequence/include/autonomous_sequence.h
- lib/autonomous_sequence/src/autonomous_sequence.cpp

7 Validation avec CdC

Ce chapitre compare les performances finales du prototype µ-verCraft II avec les exigences initiales définies dans le Cahier des Charges. La validation a été effectuée par des tests unitaires et fonctionnels. Dans la suite, certains sont présenté explicitement, des autres ne sont pas affiché en plus de détail, mais sont documenté dans le fiche « phase1_CahierDesCharges_uVerCraft_2.xlsx ».

7.1 Dimensions

Le CdC imposait des dimensions strictes pour garantir la compacité du micro-aéroglissoeur. L'exigence de réalisation idéale était fixée à < 140x90 mm.

Liste des exigences pour µ-verCraft		Commande: XXX Numéro de commande: / Semestre d'hiver 2025/26				Résultat / Validation	
Exigences		Valeur - Données					
		Réalisation minimale	Réalisation satisfaction	Réalisation idéale	Unité		
Fonction physique et technique							
Dimensions		<200x100	<180x90	<140x90	mm	106x84mm	

Nous avons réussi à concevoir un châssis extrêmement compact de 106 x 84 mm. Cette performance, atteinte grâce à une conception CAO optimisée et l'intégration serrée des composants électroniques, classe ce critère en Réalisation idéale.

7.2 Poids final

Poids	<500	<300	<100	g	79,95g
-------	------	------	------	---	---------------

Le poids total de l'aéroglissoeur (batterie incluse) est de 79,95 g. Ce résultat est nettement inférieur à la limite de 100 g fixée pour la "Réalisation idéale". Cette légèreté exceptionnelle améliore l'autonomie et la réactivité du système, validant ainsi l'objectif le plus ambitieux du projet.



7.3 Batterie

Autonomie de la batterie	>2	>4	>6	min	4-8 min
Indicateur de niveau de batterie	-	LED pulse/PWM	LED couleur		LED + site web

L'autonomie dépend fortement de la vitesse de commande et du style de pilotage, mais nos tests en conditions réelles montrent une autonomie d'environ 4 à 8 minutes, ce qui satisfait l'exigence (R03 > 4 min).

Concernant l'information utilisateur (R02), une charge faible est affichée directement sur l'application de commande web et signalée par la LED sur le microcontrôleur.

Validation des mesures électriques :

- **Mesure du courant :**
 - *Protocole* : Activation du moteur et comparaison entre la valeur du capteur embarqué (ESC) et un multimètre en série.
 - *Demande* : Précision à ± 0.15 A.
 - *Résultat* : Utiliser le capteur de courant intégré dans le ESC. La mesure interne fournie par l'ESC via la télémétrie correspond aux valeurs de référence du multimètre.
- **Mesure de la tension :**
 - *Protocole* : Comparaison entre la valeur lue par l'ADC de l'ESP32 et un multimètre de précision.
 - *Demande* : Précision à $\pm 0,1$ V.
 - *Résultat* : < 0.01 V d'erreur. La précision obtenue est excellente grâce à l'utilisation d'un convertisseur ADC 12-bits associé à un diviseur de tension calibré. La résolution théorique est d'environ 1 mV, et la précision réelle observée est de l'ordre de 10 mV, bien supérieure à l'exigence initiale.

7.4 Stabilité angulaire

Le CdC exigeait une tolérance de rotation angulaire inférieure à $\pm 10\%$ pour une "Réalisation satisfaction".

Tolerance rotation angulaire	+20% de la comand	+10% de la comand	+5% de la comand		PID sans oscillation
------------------------------	-------------------	-------------------	------------------	--	----------------------

Le PID de stabilité angulaire fonctionne très bien, sans des dépassements. La réactivité est variable avec des paramètres « rates », est on peut le prévoir avec un fiche html. La demande est réalisée au maximum.

7.5 Positionnement angulaire

L'objectif était de vérifier la précision d'une rotation commandée par rapport à une orientation de départ, avec une tolérance de $\pm 10^\circ$ pour une satisfaction du cdc.

Validation :

- *Protocole* : Une commande de rotation de -90° a été envoyée au système. L'angle final a été relevé via les données de la boussole et confirmé par mesure physique au sol.
- *Résultat* : Pour une consigne de 90° (départ à 239° , arrivée à 144°), l'aéroglisseur a effectué une rotation réelle de 95° . L'erreur est donc de 5,5% (environ 5°), avec une oscillation résiduelle très faible de $\pm 4^\circ$. Ce résultat est proche de la valeur de $\pm 5\%$ demandée pour « Réalisation idéale », et large dans « Réalisation satisfaction » de $\pm 10\%$.

7.6 CdC fonctionnel avec validation

Voici le tableau récapitulatif des exigences fonctionnelles et leur validation finale.

Prof. Plat Construction et gestion de produits SUPMICROTECH-ENSM			Liste des exigences pour μ-verCraft		Commande: XXX Numéro de commande: / Semestre d'hiver 2025/26				Resultat / Validation
Données organis.		Données du processus	Exigences		Valeur - Données				
Nb.	Nom	Type	Phase		Réalisation minimale	Réalisation satisfaction	Réalisation idéale	Unité	
Fonction physique et technique									
F01	E			Dimensions	<200x100	<180x90	<140x90	mm	106x84mm
F02	E			Poids	<500	<300	<100	g	79,95g
F03	E			Autonomie de la batterie	>2	>4	>6	min	4-8 min
F04	E			Fond de route	Plane	Plane	Légères irrégularités		Variable avec paramètre
F05	S			Course autonome sur circuit	Télécommandée	Autonome 180° autour d'un porteu	Autonome un tour complet		Orientation avec boussole, détection lignes
F06	S			Tolerance rotation angulaire	+20% de la comand	+10% de la comand	+5% de la comand		PID sans oscillation
F07	E			Localisation	Marquages au sol	Marquages au sol	Marquages + Suivi des mouvements		Marquages + boussole
F08	S			Capteurs marquages	2 capteur IR	3 capteurs IR	3 capteur de flux optique		3 capteurs IR
F09	E			IMU	Capture 6-DOF	6-DOF + Filtre intégré	6-DOF + Filtre + Suivi de vie		6-DOF + boussole!!
F10	E			Recalibrage de la pose avec des marquages	1 paramètre	2 paramètre	3 paramètre		Calculer Theta + vPerp
F11									
F12									
F13									
F14									
F15									
Technologie - Fabrication et assemblage									
T01									
T02									
Économie									
E01				Prix des composants	<300	<200	<100	€	198,08 €
Relation homme-produit									
R01				Télécommande	BT + application	BT/WIFI	WiFi + site web		WiFi + site web
R02				Indicateur de niveau de batterie	-	LED pulse/PWM	LED couleur		LED + site web
Utilisation, entretien, maintenance									
U01				Mises à jour micrologicielles	Câble USB	Câble USB	Sans fil		Câble USB
U02									
Types d'exigences: O/N-Oui/Non; E-Exigence; S-Souhait Phase de conception: P-Principe ; C-Concept ; D-Design ; R-Réalisation									
Remplace l'édition: / de: /		Édition actuelle: de: 17/01/2026							

7.7 CdC technique avec validation

Dans la prochaine page la synthèse des tests techniques effectués pour valider les performances unitaires et d'intégration est affiché. La structure est adaptée au développement en cycle V :

Fonction

Test de validation

Demande		Résultats
	valeur nécessaire	
Définition des besoins (cdc fonctionnel / fonctions souhaitées)	Recette	
Aller tout droit jusqu'à la ligne, tourner ~180° autour du poteau Mode "télé commandé": Suivre des commandes à distance le mieux possible	validation qualitative, résultat de toutes autres fonctions enchaînées validation qualitative, suffisamment réactive, que c'est faisable sans entraînement	Séquence simple avec boussole comme point de consigne (après itératif complémentaire), cf Vitesse "luis" compatible :D
Spécification (cdc f -> cdc t) "architecture de haut niveau"	Validation	
Rotation autour axe z avec un angle défini Télécommandant des mouvements Approximation l'orientation/angle relative par un ligne passée Positionnement dans l'espace Informé utilisateur si tension faible par LED & WiFi	commander 90°, voir si le résultat et +5° dans la valeur demandée Envoyer des signaux type "avance vitesse: 10%", ..., "venez avec IC (délai par caméra)" appareil/méthode expliquée au TD passer une ligne, comparer position évaluée par IUC avec réalité/measure extérieure brancher une alimentation de laboratoire, valider que LED change d'état	+10% erreur +1% erreur, délai < 250ms +10% erreur +50% erreur via CDC
Conception générale ("modules")	Tests d'intégration	
Generation de poussée verticale (SchubHub et réfugeon) Stabilisation de rotation autour axe z Mesure d'angle Créer des coussins d'air Envoyer des signaux à distance Mesure du courant de la source d'énergie Mesure de la tension de la source d'énergie Positionnement par rapport à une ligne Vitesse dans l'espace	>60g/Fmoteur avec nécess. en cas de 2 moteur vertical envoyer signal, observer vitesse de rotation et dépassement donner de petites perturbations ext. avec un stylo/ similaire, voir si la perturbation est corrigée tourner IMU de 90° degrés, treiller avec Geo Triangle Allumez l'aéroglisseur sur une planche. Incitez la planche à partir de l'horizontale jusqu'à ce que l'aéroglisseur commence à planer envoyer des signaux par un appareil recevoir les signaux sur le µC activer le moteur, regarder avec capteur et mmètre multimètre Bouger sur un ligne, regarder si c'est enregistré définir un longueur spécifique au sol, mesurer le temps de déplacement en ligne droite, comparer vitesse théorique et réel -0.25	datasheet: ~140g/Fmoteur réglable avec rates/mm validation quantitative Filtre complémentaire: rotation (-90°): départ 239° boussole, arrivée à 144° > 95° >>10mm/140g Pousse, Max/Moteur * 2) Affichage Voltage Application Capteur courant intégré dans ESC <0.1V, ADC 12bit et envoi de tension Validation par règle triangulaire Difficulté de mesurer, vu que angle est bon, c'est aussi valide
Conception détaillée (composantes)	Test unitaire	
Moteur électrique IMU Source d'énergie	Faire tourner le moteur dans le sens souhaité Sortir les senses de rotation autour des 3 axes (qualitative) multimètre tension entre 3V et 5V	temp RPM 0% > 100% > 0% > -100% > 0% valider les senses de rotation autour des 3 axes (qualitative) valide
	Réalisation	

7.8 Autres réussites

Au-delà des exigences initiales, le développement a permis de développer plusieurs innovations techniques clés supplémentaires :

- **Découplage mécanique** : Séparation physique efficace de la poussée verticale (levage) et horizontale (propulsion/rotation).
- **Calibration des capteurs** : Fiches dédiée pour la calibration de la IMU et le magnétomètre.
- **Surpuissance de levage** : Génération de poussée verticale validée à 140gF/moteur (cdc: >60gF).
- **Estimation de l'énergie consommé** : Calculation de la capacité consommée (mAh) par intégration du courant, plus précise qu'une simple lecture de tension.
- **Architecture logicielle temps-réel** : Utilisation de FreeRTOS sur 2 cœurs pour séparer le WiFi (Core 0) de la boucle de stabilisation critique (Core 1).
- **Pilotage WiFi & WebApp** : Contrôle via WebSocket (portée supérieure au Bluetooth) et interface web embarquée "zéro-install" compatible tout smartphone.
- **Protocole moteur numérique** : Implémentation du protocole DShot (vs PWM classique) pour une commande moteur plus précise, rapide et sans calibration. En plus les moteurs sont reversibles
- **Algorithme de contrôle avancé** : Régulateur PID avec protection "Anti-Windup" (anti-saturation) et dérivée sur la mesure pour éviter les à-coups.
- **Fusion de capteurs (Sensor Fusion)** : Filtre complémentaire robuste combinant Gyroscope (réactivité) et Magnétomètre (correction de dérive) pour l'orientation / le cap.
- **Sécurité "Failsafe" active** : Arrêt automatique immédiat des moteurs en cas de perte de connexion WiFi ou fermeture de l'interface web.
- **Logique autonome structurée** : Navigation basée sur une machine à états (State Machine) permettant des séquences complexes (suivi de ligne -> virage -> attente).

8 Manuel d'utilisation

8.1 Utiliser le µ-verCraft-II

Pour utiliser l'aérogissoeur, suivez les étapes suivantes :

1. Branchez la batterie
2. Attendez, pour que la LED orange commence de pulser
3. Utilisez votre téléphone et connectez avec le Wifi "**µ-verCraft-II AP**" avec "**Supmicrotech**"
4. Ouvrez la site web « **192.168.4.1** » dans votre navigateur
5. Attendez pour que « **connecté** » est affiché en vert
6. Activez « **LIFT** » et « **Moteurs** »
7. Go :)

Fonctionnalités :

- ➔ Pour changer le « **LIFT** », tapez sur le bouton plusieurs fois, pour cycler les options
- ➔ Batterie :
 - La tension est affichée sur l'application, elle ne doit pas être inférieur à 6,4V !!
 - À 7,0V la LED va rester constant et ne plus flasher, retournez et changez la batterie !
 - À 6,2V le µCraft ne va plus laisser contrôler les moteurs !

8.2 Charger les batteries

Nous avons acheté des adaptateurs XT30 → XT60 pour permettre l'utilisation du chargeur LiPo disponible à l'ENSMM.

Paramètres de charge critiques :

- Type de batterie : **LiPo/LiHV**
- Configuration : **2S** (2 cellules en série)
- Courant de charge : **0,5 A par batterie**

Attention : Restez à proximité pendant la charge. Les batteries LiPo présentent un risque d'incendie si elles sont mal manipulées (surcharge, court-circuit, dommages physiques). Ne les laissez jamais sans surveillance pendant la charge.

Encore une fois, car c'est important :

- La tension est affichée sur l'application, elle ne doit pas être inférieur à 6,4V !!
- À 7,0V la LED va rester constant et ne plus flasher, retournez et changez la batterie !

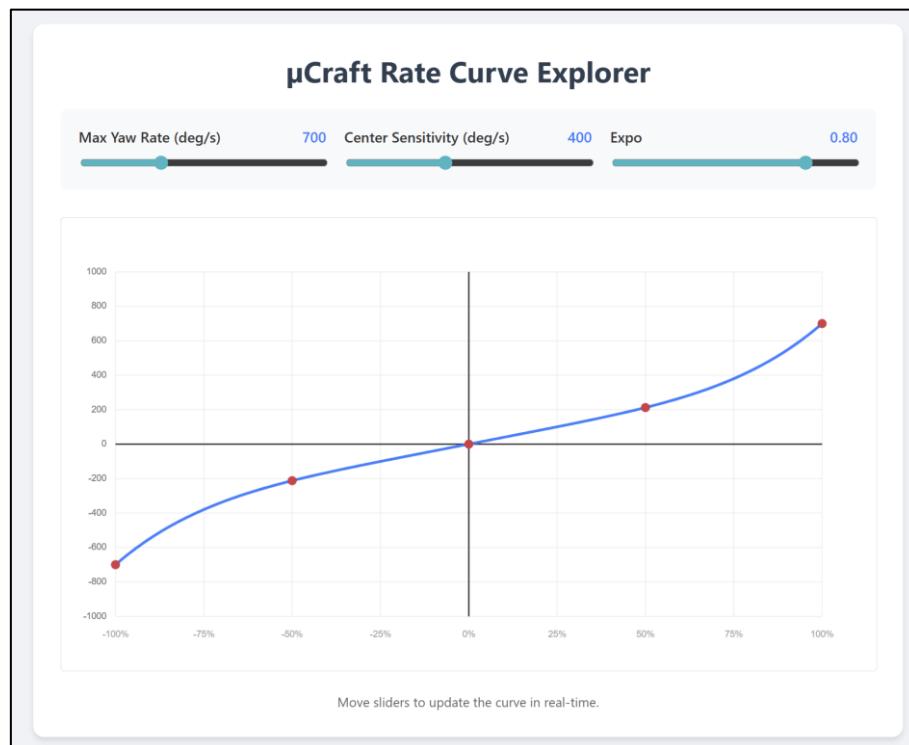
8.3 Adoption de la commande

L'ajustement de la commande de lacet repose sur l'utilisation de courbes de réponse (« Rates ») configurables. L'objectif est de concilier deux exigences contradictoires : assurer une **finesse de pilotage optimale autour du point neutre** pour les corrections légères, tout en permettant une **vitesse de rotation élevée** lorsque la commande est poussée à ses extrémités. Les valeurs sont à changer dans la fiche hovercraft_variables.cpp.

La transformation de l'entrée utilisateur (joystick) en consigne de vitesse angulaire est définie dans src/main.cpp par trois paramètres clés :

- **global_MaxYawRateSetpoint_dps** : Fixe la vitesse de rotation maximale atteinte à pleine déflexion ($\pm 100\%$).
- **global_YawCenterSensitivity** : Règle la sensibilité ("pente") autour du centre, déterminant la réactivité aux petites sollicitations.
- **global_YawRateExpo** : Introduit une composante exponentielle pour adoucir la réponse centrale et l'accélérer progressivement vers les butées.

L'impact de ces paramètres sur la courbe de commande peut être simulé et vérifié via l'outil de visualisation dédié situé dans 3_Software/rates.html comme la suivante :



9 Apprentissages principaux

Nous avons appris beaucoup, mais nos apprentissages principaux sont les suivantes :

- Il était très utile de tout documenter immédiatement, car sinon on oublie facilement certains détails.
- Réfléchir d'abord et lire attentivement la documentation du matériel -> noter les points importants et les enregistrer de manière centralisée, sinon il faut les rechercher plus tard.
- Décomposer le problème en plusieurs petits sous-systèmes et travailler de bas en haut (référence modèle Cycle-en-V).
- Le stockage centralisé des variables facilite l'adaptation ultérieure du système à des conditions cadres modifiées (Hovercraft_variables.cpp).
- Définir précisément à l'avance la structure du logiciel et les interfaces permet de travailler efficace en groupe.
- Souvent, les composants ne sont pas correctement documentés (il faut calibrer soi-même les capteurs, etc.).
- La IMU, le magnétomètre et les autres capteurs ont besoin des calibrations adaptées. Simplement faire les fonctionner est déjà du travail, les faire tourner de manière robuste est une tâche grande, créer des fiches de calibration encore plus.
- La programmation non linéaire (avec plusieurs threads et processus) permet de créer des programmes complexes avec des séquences temporelles définies.
- Utiliser RTOS dès que l'on a plusieurs tâches avec une durée de cycle différente.

10 Conseils pour commencer comme nouveau groupe

10.1 Télécharger le projet de GitHub

La première étape consiste à récupérer le code source complet. Le projet est hébergé publiquement sur GitHub.

- **Lien du dépôt :** <https://github.com/LeonVolles/mu-verCraft-II>
- **Méthode recommandée :** Si vous êtes familiers avec Git, clonez le dépôt directement (git clone ...). Sinon, vous pouvez télécharger le fichier ZIP via le bouton vert "Code" > "Download ZIP" et l'extraire dans votre dossier de travail.

La présente documentation n'est pas exhaustive. La portée du projet est trop vaste pour être intégralement couverte dans un seul document. Pour **plus d'informations**, veuillez consulter les autres fichiers du projet. Surtout « **phase1_CahierDesCharges_uVerCraft_2.xlsx** »

10.2 VS-Code/Cursor IDE

Nous recommandons fortement d'utiliser Visual Studio Code (VS Code) ou Cursor IDE comme éditeur de texte.

- Pourquoi pas l'Arduino IDE ? Le projet est complexe et divisé en plusieurs fichiers .cpp et .h. L'Arduino IDE est mal adapté pour naviguer dans une telle structure, ne gère pas l'autocomplétion avancée et rend le débogage difficile. VS Code offre une bien meilleure visibilité sur l'arborescence du projet.

10.3 Platform IO

VS Code seul ne suffit pas ; il faut lui ajouter l'extension PlatformIO IDE.

- **Rôle :** C'est le moteur qui va gérer la compilation et le téléchargement sur l'ESP32-S3.
- **Avantage majeur :** Contrairement à l'Arduino IDE où il faut installer les bibliothèques manuellement, PlatformIO télécharge automatiquement toutes les dépendances (FreeRTOS, drivers moteurs, etc.) listées dans le fichier platformio.ini. Dès que vous ouvrez le projet, PlatformIO configure tout l'environnement nécessaire pour que cela "fonctionne tout seul".

10.4 Git / GitHub

L'utilisation de Git est fortement recommandée. Pour un projet d'ingénierie en groupe de cette envergure, le partage de fichiers par clé USB ou par e-mail (ex: final_v2_vrai_final.zip) est une source d'erreurs majeure.

- **Sécurité et Historique :** git vous permet de revenir en arrière instantanément si une modification casse le code. C'est votre "filet de sécurité".
- **Collaboration :** Il permet à plusieurs personnes de travailler sur différentes parties du code (ex: l'un sur le PID, l'autre sur le WiFi) sans se gêner, puis de fusionner le travail proprement.
- **Conseil :** Forcez-vous à faire des "commits" réguliers. Si vous ne maîtrisez pas les lignes de commande, l'interface graphique intégrée à VS Code (onglet "Source Control") est très intuitive pour commencer.

➔ **Apprenez avec un jeu très bien fait :** <https://ohmygit.org/>

10.5 GitHub Copilot/Cursor /Google Antigravity

Utilisez des assistants de programmation comme GitHub Copilot ou Cursor. C'est l'avenir du développement et c'est déjà un standard dans l'industrie qu'il est très utile de maîtriser pour vos projets. L'intégration directe dans l'IDE est bien plus efficace que d'utiliser ChatGPT dans une fenêtre séparée, car l'IA a accès au contexte de votre code.

En tant qu'étudiants, vous y avez accès gratuitement (via le GitHub Student Pack). Ne négligez pas cette compétence : lors de mes derniers entretiens d'embauche, on m'a systématiquement demandé si j'avais de l'expérience avec ces outils.

10.6 Calibration nécessaire

Pour des nouveaux hovercrafts, il faut faire une nouvelle calibration du boussole/magnétomètre. Le processus est décrit en chapitre 4, il est important de le faire !

10.7 Lisez notre rapport, nous y avons travaillé fortement pour vous aider

Nous avons consacré beaucoup d'efforts à la rédaction de ce document pour vous fournir des explications détaillées. Le point le plus critique est la partie programmation (que vous retrouverez également dans l'arborescence VS Code). Certains chapitres sont là pour vous aider si vous avez besoin de changer quelque-chose, mais ne pas forcément nécessaire si vous êtes plutôt utilisateur, utilisez le sommaire pour cela.

Nous vous recommandons vivement de lire les fichiers .md inclus dans le projet. Par exemple, pour programmer une section automatique, regardé dans :

lib > autonomous_sequence > notesForFunctionalDocumentation_autonomous_sequence.md

Prenez le temps de les étudier et de les comparer avec les schémas. Il est crucial de comprendre la logique et la structure globale du code avant de commencer à modifier quoi que ce soit. La partie concernant FreeRTOS est également particulièrement importante !

11 Tâches ouvertes et idées pour la continuation

S'appuyant sur notre projet, de nouvelles possibilités de développement s'ouvrent désormais. Les bases techniques sont posées, et nous espérons qu'un futur groupe pourra poursuivre ce travail.

Voici quelques-unes de nos idées :

- En remplaçant l'ESP32-S3 par cela « Carte XIAO ESP32-S3 Sense 102010635 CARTE XIAO ESP32S3 SEN. SOUD »), une caméra peut être connectée en temps réel et utilisée pour de futurs projets.
 - <https://www.gotronic.fr/art-carte-xiao-esp32s3-sen>
 - [Getting Started with Seeed Studio XIAO ESP32S3 Series | Seeed Studio Wiki](#)
- Affichage du streaming de la caméra dans l'application Web (emplacement réservé déjà prévu)
- Détection IA en temps réel sur l'ESP32 ou sur un PC externe par streaming
- Le « framework Arduino » ne peut pas accéder à certaines fonctions de base de l'ESP32. Par exemple, le mode ADC continu ou le DMA nécessitent le « framework ESP-IDF ». Les capteurs analogiques doivent donc être interrogés activement. L'avantage des capteurs analogiques est toutefois qu'il est possible d'ajuster activement la valeur seuil.
Si l'on remplace les capteurs analogiques par des capteurs binaires avec seuil manuel et déclencheur Schmitt, il est possible d'utiliser des interruptions numériques à la place des interrogations.
- Amélioration de la reconnaissance des lignes et calcul plus robuste de l'angle
- Amélioration du mode autonome pour parcourir le circuit de manière robuste
- Développement d'un filtre Kalman pour mieux estimer la rotation entre les lignes
- Estimation de déplacement transversal y par intégration des accélérations orienté
- **Développement d'une stratégie de commande** pour le mode compétitive autonome entre plusieurs aéroglisseurs, par exemple :
 - Approche « drift » : toujours vitesse maximale, « pré-tourner » avant le cône, toujours avec une vitesse d'avancement maximale.
 - Approche « contrôle » : freiner/inverser les hélices en mouvement tout droit avant arriver au cône, pour y arriver avec une vitesse minimale, tourner de manière contrôlée/proche du cône, accélérer tout droit.



