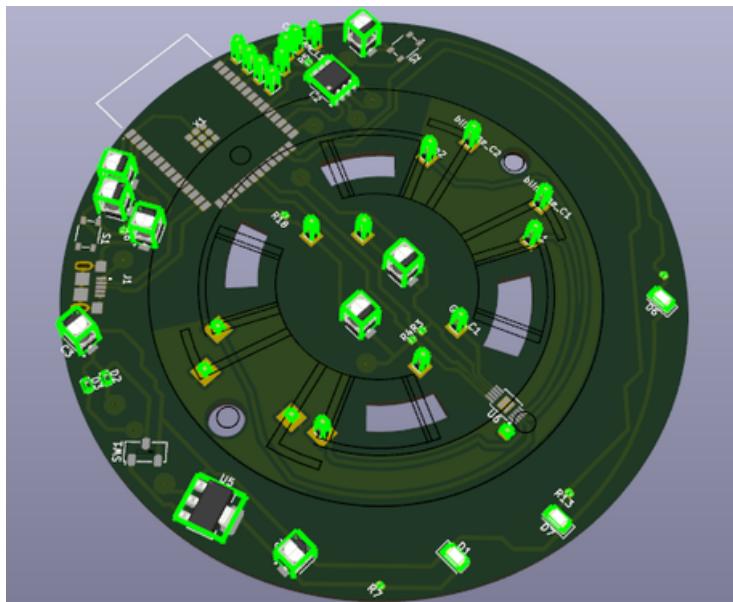


---

# Rapport de projet

## chaîne d'acquisition de mesures pour un capteur de couple imprimé en 3D multi-matériaux



Auteurs • Amar ARRAD  
• Nicolas MORONTA POUPIO

Tuteur : Mr. Maxime Manzano

2025 - 2026

# Table des matières

<b>1</b>	<b>Remerciements</b>	<b>4</b>
<b>2</b>	<b>Introduction</b>	<b>5</b>
2.1	Exosquelette . . . . .	5
2.2	Capteur de couple . . . . .	6
2.3	Cahier des charges . . . . .	7
<b>3</b>	<b>Dimensionnement et choix des composants</b>	<b>9</b>
3.1	Mesure de capacité . . . . .	9
3.2	Protocole CAN . . . . .	9
3.3	Microcontrôleur ESP32 . . . . .	10
3.3.1	Protocoles UART et USB . . . . .	11
3.3.2	Organisation mémoire . . . . .	11
3.4	Alimentation . . . . .	12
<b>4</b>	<b>Conception de la carte</b>	<b>13</b>
4.1	Schéma électrique . . . . .	13
4.1.1	Bloc alimentation : sélection de source, régulation 3.3V et indication de mise sous tension . . . . .	13
4.1.2	Bloc ESP32-S3-WROOM-1 : microcontrôleur principal, démarrage, reset et découplage . . . . .	14
4.1.3	Bloc USB : alimentation et communication . . . . .	16
4.1.4	Bloc UART1 : programmation . . . . .	16
4.1.5	Bloc FDC1004 : acquisition des capacités, I <sup>2</sup> C et blindage actif . . . . .	17
4.1.6	Bloc CAN : interface prévue pour l'intégration système (usage futur) . . . . .	19
4.2	Routage : intégration mécanique et électronique . . . . .	20
4.2.1	Choix technologiques de routage : carte deux couches et largeur de piste . . . . .	21
4.2.2	Masse et blindage . . . . .	22
4.2.3	Routage des voies capteur : réduction des capacités parasites et séparation fonctionnelle . . . . .	23
4.2.4	Routage I <sup>2</sup> C : liaison locale et fiable entre ESP32 et FDC1004	24
4.2.5	Interface CAN : intégration future dans l'exosquelette . . . . .	24
4.2.6	Bilan routage . . . . .	24
4.3	Impression de la carte . . . . .	25

<b>5</b>	<b>Programmation</b>	<b>29</b>
5.1	Traduction du traitement MATLAB en librairie C++ embarquée . . . . .	29
5.1.1	Objectif et principe général . . . . .	29
5.1.2	Choix d'architecture logicielle . . . . .	29
5.1.3	Récupération des mesures : driver FDC1004 . . . . .	30
5.1.4	Calcul de la capacité différentielle . . . . .	30
5.1.5	Estimation de l'angle à partir de la capacité . . . . .	31
<b>6</b>	<b>Conclusion</b>	<b>32</b>
6.1	Technique . . . . .	32
6.2	Personnel . . . . .	32
<b>7</b>	<b>Perspectives</b>	<b>33</b>
<b>8</b>	<b>Annexe</b>	<b>36</b>
8.1	Lien du dépôt GitHub . . . . .	36
8.2	Datasheets des composants . . . . .	36
8.3	Code C++ . . . . .	36

# Table des figures

2.1	Exosquelette pour assister le membre supérieur . . . . .	6
2.2	Capteur de couple . . . . .	7
3.1	Trames protocoles CAN . . . . .	10
4.1	Schéma du bloc alimentation . . . . .	13
4.2	Schéma du bloc ESP32 . . . . .	15
4.3	Schéma du bloc FDC1004 . . . . .	16
4.4	Pins debug . . . . .	17
4.5	Schémas FDC1004 . . . . .	18
4.6	Bloc CAN . . . . .	19
4.7	Dessin du capteur . . . . .	20
4.8	Dessin du PCB . . . . .	21
4.9	Plan de masse . . . . .	22
4.10	Plan de blindage . . . . .	23
4.11	Routage complet . . . . .	25
4.12	Insolation du PCB . . . . .	26
4.13	Révélation du circuit . . . . .	27
4.14	Gravure du PCB . . . . .	27
4.15	Etamage à froid . . . . .	28

# 1 Remerciements

Nous tenons tout d'abord à remercier notre tuteur de projet, Monsieur Maxime MANZANO, pour son aide et la confiance qu'il nous a accordée dans la réalisation de ce projet.

Nous remercions notre tuteur référent, Monsieur Jordane LORANDEL.

Nous remercions particulièrement, Messieurs Régis LEGAVE, François PASTEAU, Éric LE SAINT et Ronan POUSSIER pour leurs conseils et leur aide dans la réalisation de notre circuit imprimé.

## 2 Introduction

Dans le cadre de notre formation de deuxième année de Master EEEA, parcours systèmes embarqués de l'Université de Rennes, nous avons eu l'opportunité de réaliser un projet avec une équipe de l'Inria (Institut national de recherche en sciences et technologies du numérique).

Le sujet de notre projet portait sur la réalisation d'une chaîne d'acquisition de données au niveau d'un capteur de couple associé à un exosquelette.

Dans un premier temps, nous présenterons le contexte du projet, puis dans un second temps, ce rapport portera sur la mission qui nous a été confié en abordant les différentes problématiques, les solutions associées et la méthodologie que nous avons mis en place. Nous finirons par une conclusion portant à la fois sur le plan technique et personnel.

### 2.1 Exosquelette

L'équipe Inria qui nous a proposé le sujet travaille autour d'un exosquelette monté sur un fauteuil roulant

L'exosquelette sur lequel travaille l'équipe (voir la figure 2.1) est destiné à l'assistance du membre supérieur. Il s'agit d'un dispositif robotique conçu pour accompagner les mouvements du bras et réduire l'effort musculaire de l'utilisateur. Ces exosquelettes sont particulièrement utiles dans le cadre de la rééducation ou de l'assistance aux personnes souffrant de troubles moteurs. Cependant, l'un des principaux défis liés à ces systèmes réside dans la manière de contrôler l'exosquelette de façon intuitive et confortable.

Dans ce contexte, certaines études proposent des méthodes de commande innovantes basées sur la force exercée par l'utilisateur. Par exemple, une approche dite Force-Triggered permet de déclencher le mouvement de l'exosquelette à partir d'une impulsion de force dépassant un certain seuil. Une fois ce seuil atteint, l'exosquelette passe dans un mode de déplacement, ce qui évite à l'utilisateur de devoir exercer une force continue pour maintenir le mouvement. Cette méthode améliore ainsi l'ergonomie et réduit la fatigue musculaire, tout en conservant une assistance adaptée à l'utilisateur.

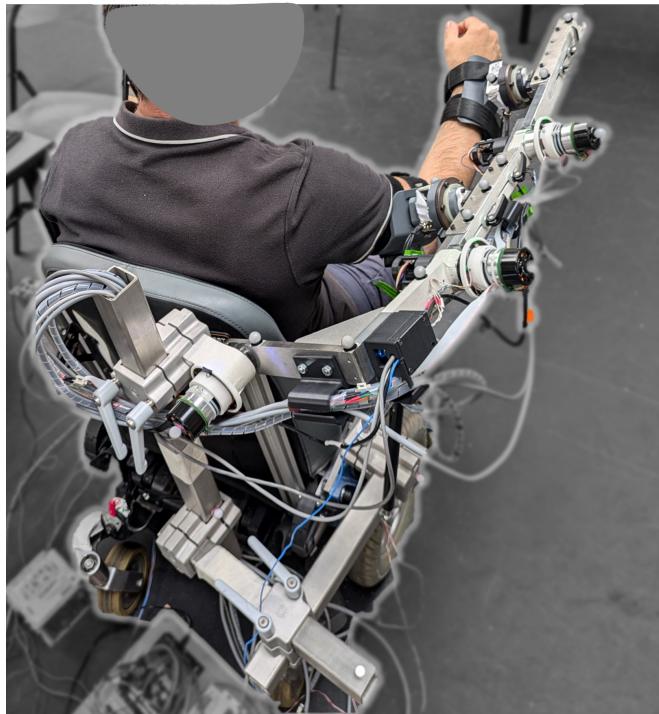


FIGURE 2.1 – Exosquelette pour assister le membre supérieur

## 2.2 Capteur de couple

Dans le cadre de la bonne utilisation de l'exosquelette, il est essentiel de mesurer avec précision les efforts mécaniques appliqués par l'utilisateur, en particulier les couples. Les capteurs de couple classiques sont souvent basés sur des jauge de contrainte ou des éléments mécaniques rigides et présentent ainsi plusieurs limites : ils sont généralement coûteux, encombrants et peu adaptés aux systèmes légers et flexibles.

Une solution innovante consiste à utiliser un capteur de couple imprimé en 3D en multi-matériaux, basé sur un principe capacitif. Ce type de capteur repose sur des structures déformables jouant à la fois le rôle d'élément élastique et de composant de mesure. Le capteur est constitué d'un cadre rigide intégrant quatre structures capacitatives, d'une came et d'un roulement. Lorsqu'un couple est appliqué, la came tourne et comprime certaines structures capacitatives tandis que d'autres se relâchent, provoquant ainsi une variation de capacité mesurable.

Ces structures sont imprimées en une seule étape grâce à une technologie d'impression multi-matériaux. Deux types de TPU (polyuréthane thermoplastique)

sont utilisés : un matériau conducteur pour former les électrodes et un matériau non conducteur et souple pour constituer la partie déformable. Un blindage conducteur (en forme de L sur le capteur) est également intégré afin de réduire les perturbations électromagnétiques. Cette méthode permet d'obtenir un capteur compact, léger et facilement reproductible comme on peut le voir sur 2.2.

Les variations de capacité sont ensuite utilisées pour estimer l'angle de déformation, en déduire le couple appliqué et adapter l'aide de l'exosquelette à l'utilisateur.



FIGURE 2.2 – Capteur de couple

### 2.3 Cahier des charges

L'objectif principal de ce projet est de concevoir et réaliser un circuit électrique embarqué destiné à être intégré au capteur de couple. Ce circuit doit inclure un composant de mesure capacitive ainsi qu'un microcontrôleur permettant l'estimation du couple.

Le dispositif devra être compact, fonctionner en temps réel et transmettre les valeurs mesurées au reste de l'électronique de l'exosquelette via un bus I2C. L'objectif idéal serait également de pouvoir connecter le capteur directement au circuit imprimé lors de sa fabrication par impression 3D.

Missions :

- **Prise en main du capteur et de son fonctionnement** : Étude des travaux existants et compréhension du principe des structures capacitives.
- **Choix d'un microcontrôleur répondant au cahier des charges** : Sélection d'un microcontrôleur adapté aux contraintes techniques du projet.
- **Conception et réalisation du circuit imprimé embarqué** : Choix du composant de mesure de capacité et du système d'affichage, puis conception et réalisation du PCB.
- **Réalisation de l'électronique embarquée du capteur de couple** : Assemblage du capteur avec l'électronique et réalisation de tests de validation.
- **Développement d'une librairie C/C++ pour la récupération des données** : Création d'une librairie permettant à un autre microcontrôleur de récupérer les informations transmises par le capteur.

# 3 Dimensionnement et choix des composants

En prenant en compte l'ensemble des contraintes du sujet, nous avons recherché les composants optimaux pour la création de cette carte. Pour cela, nous avons fait attention aux critères d'alimentation, d'encombrement et d'espace disponible sur la carte, tout en pensant à de futurs développements possibles avec les composants que nous aurons choisis.

## 3.1 Mesure de capacité

Il nous faut dans un premier temps un composant capable de récupérer une mesure de capacité et de la transmettre vers d'autres microcontrôleurs. Sur une carte d'acquisition déjà existante, l'équipe de l'Inria utilisait le composant **FDC1004DGSR**.

Le FDC1004 est un convertisseur capacitif-numérique haute résolution à 4 canaux destinés à la mise en œuvre de solutions de détection capacitive. Le composant intègre également des pilotes de blindage pour les capteurs, qui permettent de réduire les interférences électromagnétiques (EMI) et de concentrer la direction de détection d'un capteur capacitif. Le faible encombrement du FDC1004 permet de l'utiliser dans des applications où l'espace est limité. Ce sont ces raisons qui nous ont convaincus de garder ce composant pour convertir notre mesure de capacité.

## 3.2 Protocole CAN

Il nous a été demandé d'intégrer une communication de données via le protocole CAN pour de futurs développements du capteur. Le capteur recevra et transmettra alors des données en tant qu'esclave au niveau de l'exosquelette.

Le protocole CAN repose sur un principe de diffusion générale : lorsqu'un message est transmis, aucune station n'est ciblée en particulier. Chaque message est identifié par un identificateur unique, que toutes les stations du réseau reçoivent. Les stations sont constamment à l'écoute et utilisent cet identificateur pour reconnaître et traiter uniquement les messages qui les concernent en ignorant les autres.

Cet identificateur détermine également la priorité du message qui régule l'accès au bus lorsque plusieurs stations tentent d'émettre simultanément. Dans sa version de base, l'identificateur est codé au minimum sur 11 bits, permettant ainsi de définir jusqu'à 2048 messages minimum avec différentes priorités. Chaque message

peut contenir jusqu'à 8 octets de données, ce qui correspond par exemple à l'état de 64 capteurs.

La norme CAN définit deux formats de protocole : Standard (Version 2.0A) et Extended (Version 2.0B). La seule différence réside dans la longueur de l'identificateur (ID) : 11 bits en mode Standard, et 11 bits plus 18 bits supplémentaires en mode Extended. Cette extension permet d'augmenter le nombre de variables pouvant être échangées ainsi que le nombre de stations sur le réseau, sans modifier la taille des données par trame, qui reste limitée à 8 octets comme on peut le voir sur figure 3.1.

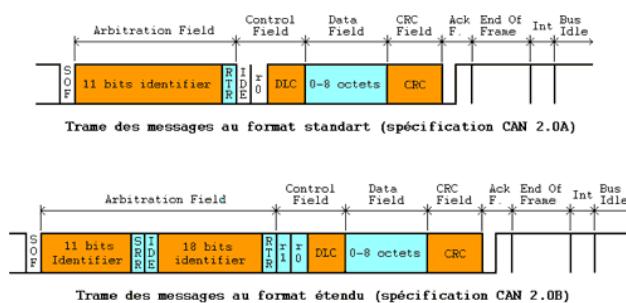


FIGURE 3.1 – Trames protocoles CAN

Pour utiliser le protocole CAN correctement, nous devons choisir un transceiver CAN qui permettra de convertir les trames CAN\_HIGH et CAN\_LOW du BUS et d'échanger des données avec le contrôleur CAN que nous aurons choisi.

Notre choix s'est porté sur le transceiver **TJA1050**, un composant énormément utilisé dans l'industrie automobile et bien documenté. Il s'alimente en 5 volts.

### 3.3 Microcontrôleur ESP32

Nous avons ensuite chercher un microcontrôleur pour notre carte qui puisse acquérir des données, effectuer des traitements sur ces données et les transmettre vers l'extérieur de la carte.

Le Microcontrôleur choisi est l'**ESP32-S3-WROOM-1** car il s'agit d'un composant très complet. En effet, avec ce MCU nous pouvons notamment :

- Communiquer via divers protocoles de communication tels que SPI, I2C et UART,

- Transmettre et recevoir des données avec le transceiver CAN TJA1050 en gérant le TX et le RX du protocole car l'ESP32 intègre un contrôleur CAN,
- Envoyer et recevoir des données sans fil via wifi et/ou bluetooth ce qui peut être intéressant pour des développements futurs.

De plus, ce composant intègre une matrice de GPIO en interne que nous viendrons programmer afin de choisir le rôle des différentes pins de l'ESP32 ce qui nous permettra d'optimiser le routage de la carte qui comporte une contrainte d'encombrement.

### 3.3.1 Protocoles UART et USB

L'ESP32 intègre un protocole USB qui permettra de transmettre des données entre la carte et un ordinateur externe par exemple. Nous pourrons également programmer l'ESP32 via ce protocole en respectant des règles de programmation telles que la gestion de la pin "Enable" et de la pin du GPIO0.

### 3.3.2 Organisation mémoire

Nous avons également choisi ce composant en fonction des dispositions mémoires dont nous avions besoin.

Sur l'ESP32-S3, il existe plusieurs types de mémoire ayant des rôles différents. La ROM est une mémoire interne gravée directement dans la puce lors de la fabrication. Elle contient le programme de démarrage (bootloader) ainsi que certaines fonctions système, et elle ne peut pas être modifiée par l'utilisateur. La SRAM (Static RAM) est la mémoire vive principale utilisée pendant le fonctionnement normal : elle sert à stocker les variables, la pile (stack), le tas (heap) et différents buffers nécessaires au programme. Cette mémoire permet des calculs rapides mais elle est volatile donc elle est perdue lorsque l'alimentation est coupée. Enfin, la RTC SRAM est associée au bloc RTC (Real Time Clock). Elle a l'avantage de pouvoir conserver certaines données lorsque l'ESP32 est placé en mode Deep Sleep, ce qui permet par exemple de sauvegarder un compteur ou un état avant de rentrer dans l'état "Sleep", puis de le récupérer au réveil.

La ROM à une taille de 384 KB, la SRAM de 512KB et la RTC SRAM de 16KB ce qui nous donne suffisamment de mémoire pour appliquer des calculs sur des données. La mémoire est largement suffisante ce qui laisse également plus de

souplesse pour des développements futurs sur la carte. La taille importante de la SRAM pourra notamment permettre de réaliser l'acquisition et le filtrage des données, le calcul du couple, la calibration, la transmission via Wifi/BLE/UART/USB ainsi que du stockage temporaire de données sans être limité par la mémoire. Nous résumons les caractéristiques de la mémoire de l'ESP32-S3-WROOM1-N16R8 dans le tableau 3.1.

Type de mémoire	Taille	Usage
ROM	384 KB	Bootloader ROM + fonctions systèmes
SRAM	512 KB	variables, buffers, piles, tas
RTC SRAM	16 KB	conservation de données en deep sleep

TABLE 3.1 – Organisation mémoire de l'ESP32\_S3\_WROOM1\_N16R8

### 3.4 Alimentation

Nous cherchons maintenant à construire la partie alimentation de notre carte, pour cela nous prenons en compte les différentes tensions d'alimentation des composants choisis précédemment. Nous avons quelques contraintes : tout les composants sont alimentés en 3,3V à part le transceiver CAN TJA1050 qui est alimenté en 5V qui vient directement du protocole CAN.

Comme nous souhaitons utiliser le protocole USB au niveau du microcontrôleur de notre carte, nous avons choisi d'installer un **port micro-USB de type B** afin de pouvoir alimenter la carte en 5V via ce port. De plus, à l'avenir la carte recevra une tension de 5V au niveau du protocole CAN depuis l'exosquelette.

Nous cherchons enfin à convertir cette tension d'entrée de 5V en une tension de 3,3V pour alimenter l'ESP32 et le FDC1004. Pour cela nous utilisons un convertisseur de tension **TLV761** qui prend en entrée 5V et qui fournit en sortie 3,3V. Nous avons choisi ce convertisseur car il peut avoir une tension d'entrée comprise entre 2,5V et 18V et une tension de sortie comprise entre 0,8V et 13,5V. De plus, nous avons fait attention au dropout voltage (ou tension de décrochage) qui correspond à la différence minimale entre la tension d'entrée et la tension de sortie qui est nécessaire pour que le régulateur ait une tension de sortie qui ne décroche pas. Le TLV761 possède un dropout voltage compris entre 0,9V et 1,6V et notre différence de tension entre notre entrée et notre sortie est de 1,7V ce qui veut dire que même dans un cas critique notre tension de sortie régulera correctement.

# 4 Conception de la carte

## 4.1 Schéma électrique

Dans cette partie, nous détaillons le schéma électrique de la carte que nous avons conçue pour interfaçer notre capteur de couple capacitif imprimé en 3D. L'architecture générale repose sur un ESP32-S3-WROOM-1 qui assure le traitement et la communication et sur un FDC1004 qui réalise la conversion capacitée-numérique. L'ensemble est intégré sur un PCB circulaire, pensé pour être monté dans un exosquelette. Nous avons également prévu des interfaces de communication et de test (UART, CAN), afin de conserver une carte polyvalente et facilement intégrable.

### 4.1.1 Bloc alimentation : sélection de source, régulation 3.3V et indication de mise sous tension

L'alimentation est un point central du montage, car la qualité du rail 3.3V influence directement la stabilité de l'ESP32 et la précision des mesures capacitifs. Nous avons donc conçu une alimentation simple, mais robuste, avec la possibilité d'utiliser plusieurs sources 5V.

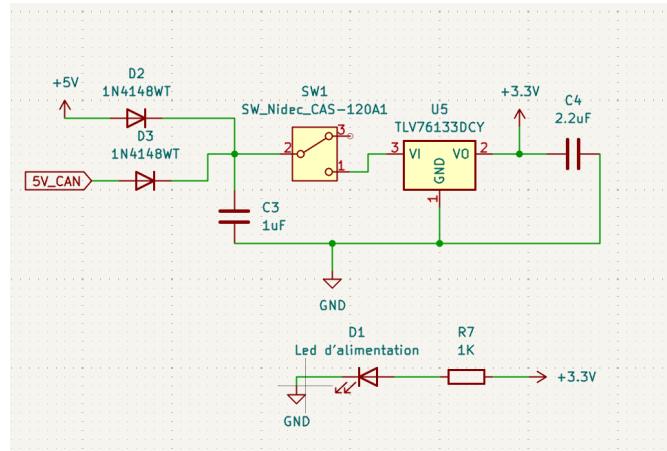


FIGURE 4.1 – Schéma du bloc alimentation

Comme on peut le voir dans la figure 4.1 deux sources de 5V peuvent alimenter la carte, le +5V provenant typiquement de l'USB, et le 5V\_CAN issu d'un environnement externe (par exemple un système embarqué déjà alimenté, comme l'exosquelette). Plutôt que de choisir une seule source, nous avons mis en place

un *diode-OR gate* grâce à deux diodes (D2 et D3). Ce choix nous permet d'éviter qu'une source n'injecte du courant dans l'autre, ce qui sécurise l'électronique en cas de branchements multiples ou de tests en laboratoire. Concrètement, la source qui a la tension la plus élevée alimente naturellement le reste du circuit, et les diodes empêchent tout retour indésirable.

Après cette sélection de source, nous avons ajouté un interrupteur (SW1) afin de pouvoir couper l'alimentation de la carte sans avoir à débrancher les connecteurs. C'est particulièrement pratique lors des essais, des phases de debug ou de montage dans un système final où l'accès physique aux câbles n'est pas toujours simple.

La conversion de 5V vers 3.3V est réalisée par un régulateur linéaire TLV76133DCY. Nous avons choisi un LDO pour obtenir une tension propre et stable, ce qui est important pour la mesure capacitive. Autour du régulateur, nous avons placé les condensateurs nécessaires à la stabilité et au filtrage : un condensateur côté entrée (C3) pour lisser l'alimentation venant de la source 5V et un condensateur côté sortie (C4) pour stabiliser la régulation et fournir localement l'énergie lors des variations rapides de courant. Ce montage assure un 3.3V fiable, indispensable au fonctionnement du microcontrôleur et du convertisseur capacitif.

Enfin, nous avons intégré une LED d'alimentation (D1) avec sa résistance de limitation (R7). Cette LED est reliée au 3.3V et sert d'indicateur simple mais efficace : dès que la carte est correctement alimentée, la LED s'allume, on sait immédiatement si la régulation est active et si le rail 3.3V est présent.

#### 4.1.2 Bloc ESP32-S3-WROOM-1 : microcontrôleur principal, démarrage, reset et découplage

Le bloc ESP32-S3-WROOM-1 constitue le cœur de la carte. C'est lui qui pilote la communication I<sup>2</sup>C avec le FDC1004, exécute l'algorithme embarqué et, à terme, gérera l'intégration du capteur au sein de l'exosquelette via des interfaces de communication.

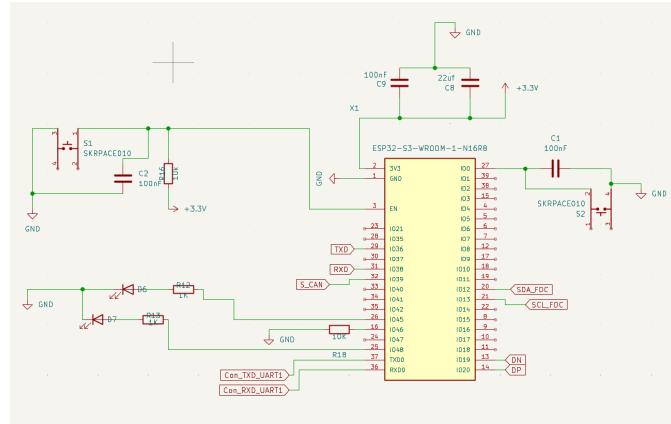


FIGURE 4.2 – Schéma du bloc ESP32

L'ESP32 est alimenté en 3.3V (voir la figure 4.2). Pour garantir une alimentation stable malgré les appels de courant rapides du microcontrôleur, nous avons placé un découplage directement au plus près de l'alimentation du module. Le condensateur de  $22 \mu\text{F}$  (C8) joue le rôle de réservoir d'énergie et amortit les transitions, tandis que le condensateur de  $100 \text{nF}$  (C9) filtre les composantes hautes fréquences. Cette association est classique mais particulièrement importante ici, car le bruit d'alimentation peut perturber indirectement la chaîne de mesure capacitive.

La broche EN (enable) de l'ESP32 est utilisée pour le reset et l'activation du module. Nous avons mis en place un circuit de reset propre avec une résistance de pull-up vers 3.3V et un condensateur vers la masse, ce qui stabilise le démarrage et limite les rebonds. Un bouton poussoir (S1) permet de forcer la broche EN à l'état bas et donc de réinitialiser manuellement le microcontrôleur. Cette possibilité est essentielle pendant le développement : elle permet de relancer rapidement l'exécution sans manipulation logicielle.

Nous avons également intégré un bouton (S2) dédié au *boot mode*. Dans notre cas, S2 est relié à IO0, ce qui correspond au comportement standard de l'ESP32 : maintenir IO0 dans l'état requis au démarrage permet de passer en mode bootloader pour programmer le microcontrôleur. En pratique, cela nous permet de flasher le firmware facilement et de conserver une méthode de programmation fiable directement sur la carte, ce qui est indispensable dans un projet embarqué destiné à être intégré mécaniquement dans un système final.

#### 4.1.3 Bloc USB : alimentation et communication

La carte intègre un connecteur micro-USB (J1). Son rôle principal, dans notre architecture, est d'apporter une source d'alimentation 5V via VBUS. Ce 5V rejoint ensuite le bloc alimentation où il est sélectionné et régulé vers 3.3V.

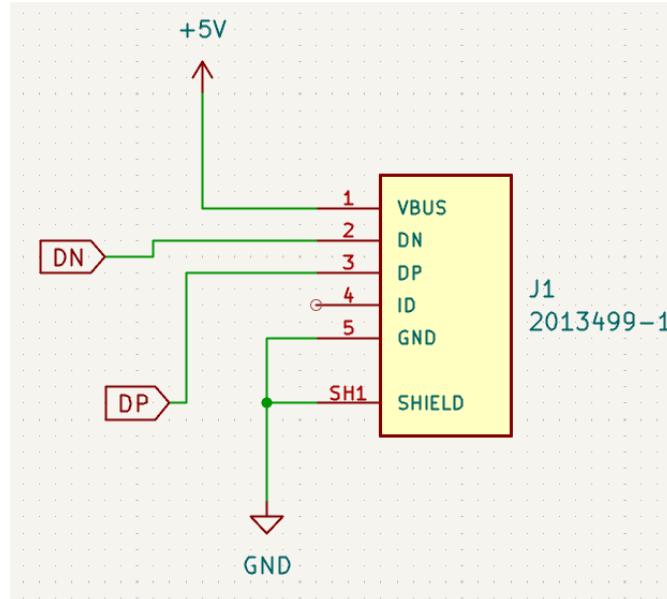


FIGURE 4.3 – Schéma du bloc ESP32

Le connecteur USB expose également les lignes DP et DN (voir la figure 4.3). Même si l'USB n'est pas nécessaire au fonctionnement nominal du capteur, le fait de router ces signaux sur la carte nous laisse la possibilité d'exploiter l'USB natif selon les besoins. Cette approche est cohérente avec l'objectif global de la carte : fournir une plateforme complète et adaptable, plutôt que strictement limitée au minimum fonctionnel.

Nous avons enfin relié la broche SHIELD du connecteur à la masse. Ce choix permet d'améliorer le comportement électromagnétique, en particulier dans un environnement potentiellement bruité comme un exosquelette intégrant moteurs et électroniques de puissance. Ce point est important pour limiter les perturbations qui pourraient se coupler sur le PCB et dégrader la qualité de mesure.

#### 4.1.4 Bloc UART1 : programmation

Nous avons prévu une interface UART dédiée, exposée via les nets Con\_RXD\_UART1, Con\_RXD\_UART1, Con\_3V3\_UART1 et Con\_GND\_UART1 (voir la figure 4.4). Sur notre

carte, ce bloc n'est pas uniquement une sortie de *debug* : il constitue avant tout un moyen de programmer l'ESP32-S3 lors des premières phases du projet. En s'appuyant sur le guide officiel de mise en route de l'ESP-IDF, nous utilisons cette liaison série pour flasher le firmware via les outils standard de l'environnement Espressif (notamment `esptool`), ce qui garantit une méthode de programmation fiable et reproductible.

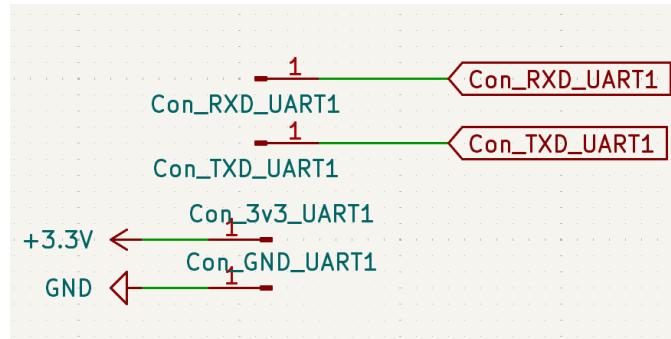


FIGURE 4.4 – Pins debug

Dans la pratique, ce choix nous simplifie énormément le développement. L'UART est l'interface la plus directe pour charger rapidement un nouveau binaire sur le microcontrôleur : on compile le projet, on bascule l'ESP32 en mode bootloader, puis on téléverse le firmware en suivant la procédure décrite par l'ESP-IDF. Ce fonctionnement est cohérent avec notre schéma, car le mode *boot* est assuré par le bouton relié à `I00` (`S2`) tandis que la ligne `EN` permet de redémarrer proprement le module. L'ensemble *I00 + EN + UART* forme donc une chaîne complète qui rend les phases de programmation et de redémarrage très fluides pendant les essais.

Enfin, le fait d'avoir `3V3` et `GND` sur la même interface rend l'utilisation d'un adaptateur USB-UART externe beaucoup plus simple. Nous pouvons alimenter ou référencer correctement l'interface, limiter les erreurs de câblage et effectuer des cycles répétés *flash / reset / observation des logs* de manière sûre et rapide, ce qui correspond exactement aux besoins d'un développement embarqué itératif.

#### 4.1.5 Bloc FDC1004 : acquisition des capacités, I<sup>2</sup>C et blindage actif

Le FDC1004 (U6) est l'élément clé de la mesure. Il réalise la conversion des capacités analogiques issues du capteur en données numériques exploitables par l'ESP32.

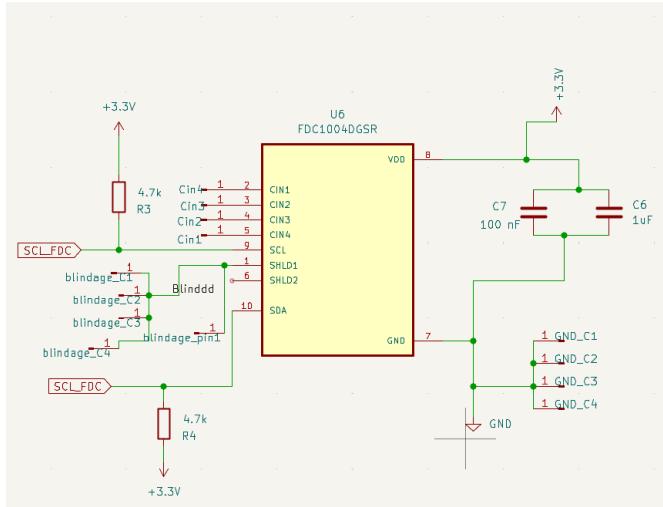


FIGURE 4.5 – Schémas FDC1004

Les quatre électrodes du capteur sont connectées aux entrées CIN1 à CIN4. Notre schéma met en évidence le câblage réel : les nets du capteur sont routés de sorte que CIN1 reçoit Cin4, CIN2 reçoit Cin3, CIN3 reçoit Cin2 et CIN4 reçoit Cin1 (voir la figure 4.5), nous avons malheureusement dû effectuer ce changement à cause des contraintes de routage sur notre PCB. Ce point est important car il doit être pris en compte dans le traitement logiciel : l'ordre physique des électrodes ne correspond pas directement à la numérotation interne du composant.

La communication entre l'ESP32 et le FDC1004 se fait en I<sup>2</sup>C via les lignes SDA\_FDC et SCL\_FDC. Nous avons ajouté des résistances de pull-up de 4.7 kΩ sur ces lignes (R3 et R4) afin de respecter le fonctionnement open-drain de l'I<sup>2</sup>C. Sans ces pull-up, les niveaux logiques ne seraient pas correctement définis et la communication deviendrait instable. Le routage de ces lignes est confirmé comme conforme à notre schéma et à notre PCB.

Un aspect particulièrement important dans notre conception est la gestion du blindage. Le FDC1004 propose des sorties SHLD1 et SHLD2 destinées à générer un blindage actif (*driven shield*). Nous avons utilisé ces sorties pour connecter des nets de blindage (blindage\_C1..C4 et Blindd). L'objectif est de réduire l'influence des capacités parasites dues au routage, à l'environnement proche et aux couplages entre pistes. Dans un capteur capacitif, ces perturbations peuvent être du même ordre de grandeur que la variation utile, donc le blindage actif améliore directement la répétabilité et la précision des mesures.

Enfin, l'alimentation du FDC1004 est découplée localement par un condensateur de 100 nF (C7) et un condensateur de 1  $\mu$ F (C6). Le découplage local est crucial pour ce composant, car il mesure des variations très faibles : une alimentation instable ou bruitée peut introduire du bruit dans la conversion et réduire la résolution effective.

#### 4.1.6 Bloc CAN : interface prévue pour l'intégration système (usage futur)

Même si le CAN n'est pas utilisé dans notre projet actuel, nous avons choisi d'intégrer cette interface car la carte est destinée à être intégrée dans un exosquelette. Dans ce type de système, il est fréquent d'avoir un réseau de communication robuste entre sous-modules (capteurs, actionneurs, unité de calcul), et le bus CAN est particulièrement adapté grâce à sa robustesse aux perturbations et son usage répandu en environnement embarqué.

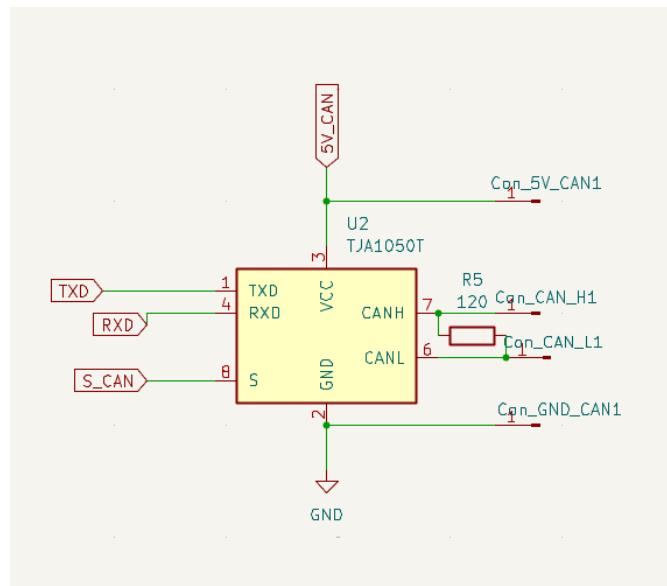


FIGURE 4.6 – Bloc CAN

Le transceiver CAN utilisé est un TJA1050T (U2). Côté microcontrôleur, il reçoit les signaux logiques TXD et RXD provenant de l'ESP32, ainsi qu'une entrée de contrôle S\_CAN qui permet typiquement de gérer un mode veille (voir la figure 4.6). Côté bus, il fournit les lignes différentielles CANH et CANL. Nous avons également prévu une résistance de terminaison de 120  $\Omega$  (R5) entre CANH et CANL. Cette

terminaison est nécessaire lorsque la carte se trouve à une extrémité du bus ; dans le cas contraire, elle peut être laissée non utilisée selon la configuration système.

Le transceiver est alimenté en 5V\_CAN, ce qui correspond au fonctionnement classique de ce type de composant. Le fait de séparer l'alimentation logique (3.3V) et l'alimentation transceiver (5V) permet de respecter les niveaux électriques attendus sur le bus CAN et de simplifier l'intégration dans des systèmes existants.

## 4.2 Routage : intégration mécanique et électronique

Pour commencer la conception du routage, nous sommes partis directement de la géométrie du capteur de couple.

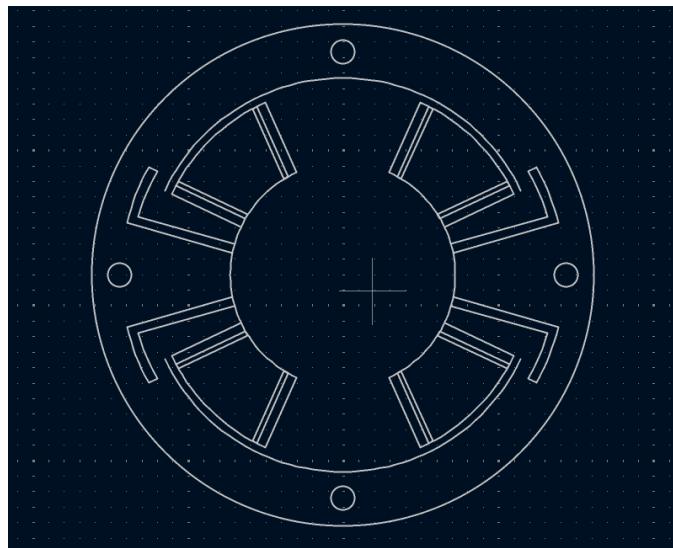


FIGURE 4.7 – Dessin du capteur

Comme on peut le voir sur la figure 4.7 Notre PCB n'est pas une carte ajoutée "à côté" du capteur : il est conçu pour être monté sur celui-ci, donc sa forme doit suivre au mieux le contour mécanique et respecter les zones fonctionnelles du capteur. Nous avons ainsi repris le dessin du capteur comme base de travail dans KiCad afin de garantir une intégration cohérente et reproductible.

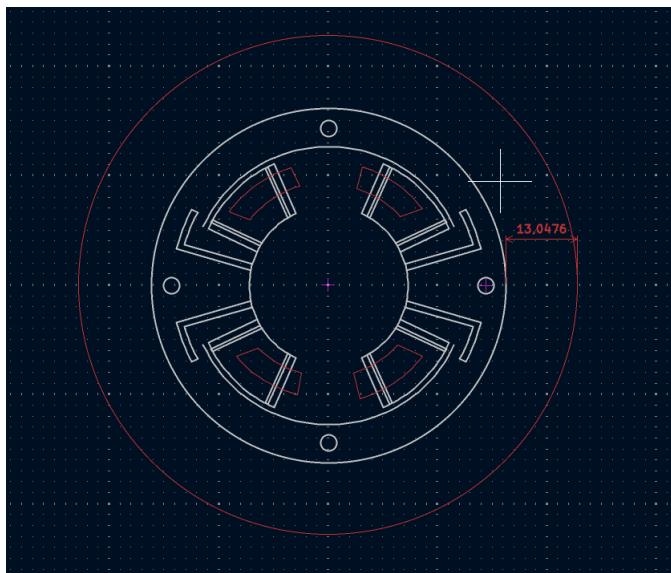


FIGURE 4.8 – Dessin du PCB

À partir de cette référence, nous avons défini le contour principal du PCB sous la forme d'un cercle de 45 mm de diamètre. Comme on peut le voir sur la figure 4.8 ce dimensionnement nous donne une marge par rapport au diamètre du capteur environ 13,0476 mm de plus en diamètre, , ce qui crée une couronne périphérique dédiée à l'électronique. Cette zone supplémentaire nous permet de placer certains des composants sans empiéter sur les zones sensibles au centre du capteur, tout en conservant une carte compacte compatible avec une intégration dans un exosquelette.

Enfin, nous avons prévu des ouvertures directement sur la couche Edge.Cuts. Ces ouvertures donnent un accès physique aux zones capacitatives afin de pouvoir connecter des fils volants directement sur les électrodes et récupérer leurs valeurs.

#### 4.2.1 Choix technologiques de routage : carte deux couches et largeur de piste

Notre PCB est réalisé en deux couches (Top et Bottom), nous avons fixé la largeur de piste à 0,6 mm. Ce choix apporte plusieurs avantages concrets : il améliore la robustesse mécanique des pistes, réduit leur résistance série et augmente la tolérance aux variations de fabrication. Même si cette largeur est supérieure à ce qui est souvent utilisé pour des signaux logiques, elle est cohérente avec notre besoin de fiabilité et avec la volonté de stabiliser au maximum l'ensemble de la

carte, et aussi pour éviter les coupures de pistes lors de la phase d'impression.

#### 4.2.2 Masse et blindage

##### Plan de masse (GND) : stabilité électrique et retours de courant

Le plan de masse joue un rôle essentiel dans notre conception, car il sert à la fois de référence électrique commune et de support pour les retours de courant.



FIGURE 4.9 – Plan de masse

Nous avons donc mis en place un plan de masse étendu, principalement sur la couche Bottom, afin d'éviter une masse routée uniquement en pistes. Un plan de masse continu permet de réduire l'impédance des retours, de limiter les boucles de courant et de diminuer les variations locales de potentiel.

Ce point est particulièrement important dans notre cas, car l'ESP32 peut générer des appels de courant rapides, et ces transitions peuvent perturber la mesure si la masse n'est pas stable. En conservant un plan de masse large et continu, nous limitons le bruit global du système et nous améliorons l'immunité aux perturbations externes, ce qui est cohérent avec l'environnement final qui est l'exosquelette.

##### Plan de blindage (/Blindddd) : protection active des signaux capacitifs

Pour une mesure capacitive fiable, la gestion des parasites est un point critique. Nous avons donc implémenté un plan de blindage dédié, associé au net /Blindddd,

qui correspond au blindage piloté par le FDC1004. Contrairement à un blindage passif relié à la masse, ce blindage est conçu pour être *driven shield* : le FDC1004 fournit des sorties SHLD prévues pour piloter un écran autour des pistes sensibles.

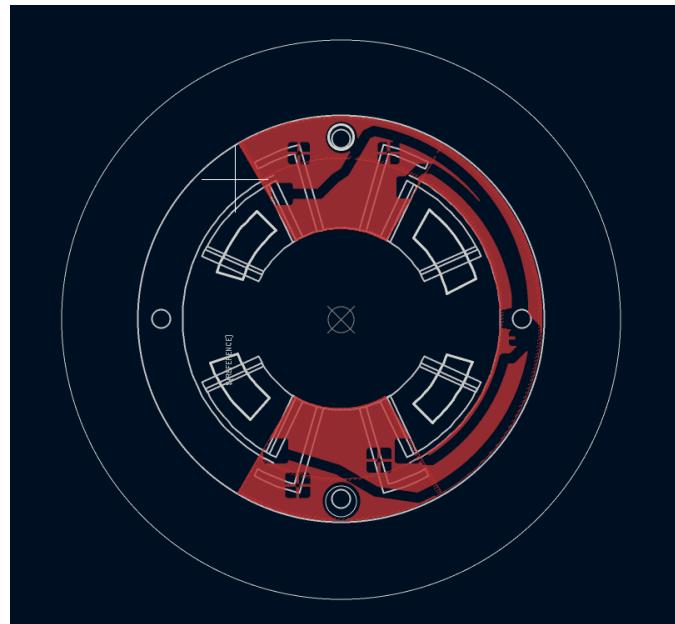


FIGURE 4.10 – Plan de blindage

L’objectif de ce blindage est de réduire l’influence des capacités parasites dues au PCB et à l’environnement (pistes proches, masse, structure métallique, humidité, etc...). Sans blindage, les pistes reliant les électrodes peuvent capter des variations qui se superposent au signal utile. En pilotant un blindage autour de ces zones, nous stabilisons l’environnement électrique des pistes sensibles et nous améliorons la répétabilité des mesures.

#### 4.2.3 Routage des voies capteur : réduction des capacités parasites et séparation fonctionnelle

Les liaisons entre le capteur et le FDC1004 (entrées CIN1..CIN4) constituent les signaux les plus sensibles de la carte. Nous avons donc cherché à limiter autant que possible leur exposition aux perturbations. Notre logique est de garder ces pistes aussi courtes et propres que possible, et d’éviter de les faire passer dans des zones “bruyantes” (alimentation, zones numériques, lignes de communication).

Le placement des composants et l'utilisation du blindage /Blinddd participent à cette stratégie. En réduisant la longueur et en encadrant les pistes sensibles par un blindage, nous limitons l'ajout de capacité parasite qui pourrait fausser la mesure et rendre la calibration instable.

#### 4.2.4 Routage I<sup>2</sup>C : liaison locale et fiable entre ESP32 et FDC1004

La communication entre l'ESP32-S3 et le FDC1004 est réalisée via le bus I<sup>2</sup>C (SDA\_FDC et SCL\_FDC). Nous confirmons que ce bus a été routé conformément au schéma. Même si l'I<sup>2</sup>C n'est pas le signal le plus critique du point de vue analogique, sa fiabilité conditionne directement la disponibilité des mesures.

Nous avons donc privilégié un routage local, direct et robuste. Les résistances de pull-up sont prévues sur le schéma, ce qui garantit des niveaux logiques corrects sur le bus. L'usage d'une largeur de piste de 0,6 mm contribue également à la robustesse mécanique et à la stabilité des connexions.

#### 4.2.5 Interface CAN : intégration future dans l'exosquelette

Même si le CAN n'est pas utilisé dans notre projet actuel, nous l'avons intégré et routé afin d'anticiper l'intégration de la carte dans un exosquelette et d'autres projets futurs.

L'intégration de cette interface sur le PCB nous permet de conserver une marge d'évolution sans refaire une conception matérielle complète. En pratique, cette partie reste inactive dans notre firmware actuel, mais elle prépare la carte à une communication plus "système" lorsque le besoin apparaîtra.

#### 4.2.6 Bilan routage

En synthèse comme on peut le voir sur la figure 4.11 notre routage résulte d'un compromis entre contraintes mécaniques (PCB circulaire monté sur le capteur), contraintes de fabrication (deux couches, pistes de 0,6 mm) et exigences de mesure (plan de masse stable, blindage actif /Blinddd, réduction des capacités parasites). Cette approche nous permet d'obtenir une carte robuste, reproductible et adaptée à l'intégration finale.

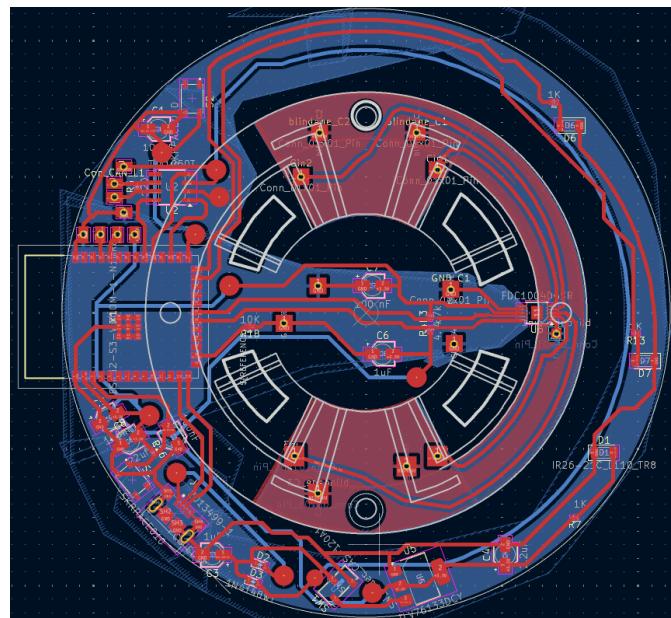


FIGURE 4.11 – Routage complet

### 4.3 Impression de la carte

Afin de fabriquer notre PCB, nous avons suivi une procédure en plusieurs étapes. Chaque étape a un rôle précis : transférer le motif, révéler les pistes, graver le cuivre, puis protéger la surface pour faciliter la soudure et améliorer la durée de vie de la carte.

- **Insolation UV (180 s) :** Nous commençons par insoler la plaque pré-sensibilisée pendant **180 secondes** aux UV, à travers le typon. Cette étape permet de transférer le motif du circuit sur la résine photosensible : les zones exposées réagissent et préparent la carte à la révélation. Comme on peut le voir sur la figure 4.12



FIGURE 4.12 – Insolation du PCB

- **Révélation (hydroxyde de sodium) :** Après l’insolation, nous plongeons la carte dans un bain de **révélateur à base d’hydroxyde de sodium**. Le révélateur enlève la résine sur les zones prévues, ce qui fait apparaître le cuivre qui devra être gravé. On surveille cette étape pour éviter une sous-révélation (motif incomplet) ou une sur-révélation (pistes dégradées), puis on rince pour stopper la réaction (figure 4.13).



FIGURE 4.13 – Révélation du circuit

- **Gravure (perchlorure de fer)** : Une fois le motif révélé, nous réalisons la gravure au **perchlorure de fer**. Le produit attaque uniquement le cuivre mis à nu, tandis que les pistes protégées par la résine restent en place. Cette étape est essentielle pour obtenir des pistes nettes et éviter les courts-circuits. Après gravure complète, nous rinçons soigneusement la carte (figure 4.14).



FIGURE 4.14 – Gravure du PCB

- **Seconde insolation UV (30 s)** : Après la gravure, nous effectuons une deuxième insolation, plus courte (**30 secondes**). Elle permet de finaliser le

traitement de la résine restante et d'améliorer la tenue de la surface avant l'étape de protection.

- **Étamage chimique à froid** : Pour finir, nous réalisons un **étamage chimique à froid**. Cette opération dépose une fine couche d'étain sur le cuivre, ce qui protège les pistes contre l'oxydation et rend la soudure plus facile et plus fiable. Cela augmente aussi la durée de vie de la carte pour les utilisations futures (figure 4.15).

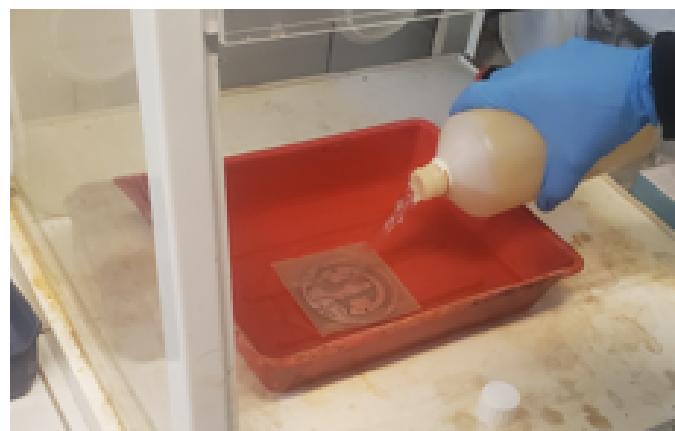


FIGURE 4.15 – Etamage à froid

# 5 Programmation

## 5.1 Traduction du traitement MATLAB en librairie C++ embarquée

Dans cette étape du projet, nous avons remplacé la chaîne de traitement MATLAB par une solution entièrement embarquée sur notre PCB. L'objectif était d'exécuter directement sur l'ESP32-S3 la récupération des mesures de capacité via le FDC1004 et la totalité des calculs nécessaires pour obtenir l'angle estimé et le couple estimé du capteur capacitif imprimé en 3D. Cette transition est essentielle, car elle permet de s'affranchir de MATLAB et de disposer d'un système autonome et exploitable en temps réel dans un environnement embarqué.

### 5.1.1 Objectif et principe général

Le capteur de couple fournit une variation de capacité sur quatre électrodes. Ces quatre capacités sont mesurées par le FDC1004, puis traitées afin d'obtenir une grandeur différentielle. Sous MATLAB, cette grandeur différentielle était ensuite convertie en angle estimé, et un modèle de couple était appliqué pour reconstruire le couple mécanique, en prenant en compte l'hystérésis du capteur ainsi qu'un comportement dynamique lié à la vitesse de rotation.

Nous avons donc cherché à reproduire fidèlement cette solution en C++, avec deux contraintes fortes : (i) conserver exactement les mêmes équations et paramètres que dans le code MATLAB afin de garantir la cohérence des résultats, et (ii) obtenir une implémentation compacte et intégrable facilement dans un projet ESP-IDF.

### 5.1.2 Choix d'architecture logicielle

Pour limiter la complexité et faciliter l'intégration, nous avons choisi une librairie volontairement minimale composée de deux fichiers : `torque_sensor_lib.hpp` et `torque_sensor_lib.cpp` (les codes en annexe). Le premier contient l'API publique (structures de paramètres, structure de sortie, classe principale), tandis que le second regroupe l'implémentation complète (driver FDC1004, calculs de conversion, modèle statique à hystérésis, modèle dynamique et calcul du couple total).

Ce choix nous permet de conserver une séparation claire entre l'interface et le code d'exécution, tout en évitant la multiplication de fichiers, ce qui est plus adapté à un projet embarqué en phase de prototypage.

### 5.1.3 Récupération des mesures : driver FDC1004

La première étape consiste à lire les capacités mesurées par le FDC1004 via une communication I<sup>2</sup>C. Nous avons implémenté un driver minimal mais complet basé sur l'accès registre du circuit. Concrètement, nous avons développé des fonctions de lecture et d'écriture de registres 16 bits car le FDC1004 expose sa configuration et ses résultats via ce format.

Ensuite, nous avons configuré les mesures internes MEAS1 à MEAS4 du FDC1004 pour mesurer chaque entrée CINx par rapport à CAPGND. Cette configuration correspond à notre schéma, où les électrodes du capteur sont connectées directement aux entrées CIN1..CIN4, et où la référence est la masse interne prévue pour ce type de mesure capacitive.

Pour chaque itération de mesure, nous déclenchons les conversions en mode *single-shot*. Après déclenchement, nous attendons la disponibilité des données en surveillant les bits DONE dans le registre MEAS\_CONFIG. Cette approche garantit que la lecture des résultats se fait uniquement lorsque la conversion est terminée, ce qui évite des valeurs incohérentes.

Enfin, nous lisons les registres de résultat associés à chaque mesure et reconstruisons la valeur sur 24 bits signés. Une extension de signe est appliquée pour obtenir un int32\_t correct. Cette valeur brute est ensuite convertie en capacité exprimée en pF à l'aide de la relation associée à la plage ±15 pF :

$$C_{\text{pF}} = \frac{\text{raw}}{2^{19}} + \text{CAPDAC} \times 3.125.$$

Cette conversion permet de manipuler directement des grandeurs physiques, et elle rend la chaîne de traitement plus lisible.

Or, le code MATLAB travaille avec des canaux nommés C1..C4. Pour conserver exactement la même signification dans l'implémentation C++, nous avons appliqué le mapping suivant :

$$C1 = CIN4, \quad C2 = CIN3, \quad C3 = CIN2, \quad C4 = CIN1.$$

Cette étape est indispensable, car une inversion de deux voies suffit à fausser totalement la capacité différentielle et donc toute l'estimation du couple.

### 5.1.4 Calcul de la capacité différentielle

Une fois les quatre capacités correctement associées aux variables MATLAB, nous calculons la grandeur différentielle Ctot avec la même formule que dans le

script :

$$C_{\text{tot}} = \frac{-C1 - C3 + C2 + C4}{1000}.$$

Nous avons conservé la division par 1000 afin de rester strictement cohérents avec l'échelle utilisée sous MATLAB. Cette valeur `Ctot` constitue l'entrée principale pour l'estimation de l'angle et indirectement du couple.

### 5.1.5 Estimation de l'angle à partir de la capacité

Le code MATLAB calcul un angle estimé à partir de `Ctot` via une relation affine, à laquelle s'ajoute une correction proportionnelle à la vitesse. Nous avons reproduit exactement la même structure :

$$\theta_{\text{estim}} = p_1 C_{\text{tot}} + p_0 + s p_{d\theta} \dot{\theta},$$

ù  $p_1$ ,  $p_0$  et  $p_{d\theta}$  sont des paramètres de calibration issus du code MATLAB, et  $s$  est la direction (+1 ou -1) associée à la branche d'hystérésis courante.

Pour calculer  $\dot{\theta}$ , nous utilisons une dérivée numérique simple basée sur deux échantillons successifs et le pas de temps  $dt$  fourni à la fonction `update`. Un état interne (`theta_prev`) est conservé pour effectuer ce calcul de manière continue.

# 6 Conclusion

## 6.1 Technique

Sur le plan technique, ce projet nous a permis de concevoir et développer une chaîne d'acquisition complète pour un capteur de couple capacitif, en intégrant à la fois l'électronique de mesure, l'alimentation et le traitement embarqué. Le choix des composants (FDC1004, ESP32-S3, régulateur TLV761) ainsi que l'intégration d'interfaces comme le CAN et l'USB ont permis de réaliser une carte compacte et évolutive. De plus, le travail sur le routage, le blindage actif et la gestion des capacités parasites a été essentiel afin d'assurer une mesure fiable et stable dans un environnement potentiellement perturbé.

## 6.2 Personnel

Sur le plan personnel, ce projet nous a permis de renforcer nos compétences en conception de PCB, en sélection de composants, ainsi qu'en programmation embarquée. Il nous a également appris à travailler de manière plus rigoureuse en suivant une méthodologie complète, depuis l'analyse du besoin jusqu'à la réalisation et la validation d'une solution fonctionnelle.

Enfin, cette expérience nous a permis de mieux comprendre l'importance de la cohérence entre le matériel et le logiciel dans un système embarqué, tout en développant notre autonomie et notre capacité à travailler en équipe sur un projet technique complet.

# 7 Perspectives

Pour la suite du projet, nous avons identifié quatre évolutions pertinentes qui permettraient d'améliorer à la fois la robustesse de la mesure, la flexibilité logicielle et l'exploitation du capteur dans un système complet.

- **Réintroduction d'une mémoire Flash externe.** Lors de la conception de la première version (V1) de la carte, nous avions envisagé d'ajouter une mémoire Flash externe afin d'augmenter la capacité de stockage disponible. Cependant, au fil des versions et des contraintes d'intégration (notamment la place et le routage), nous avons finalement retiré ce composant pour simplifier la carte. Avec le recul, réintroduire une Flash externe reste une amélioration intéressante. Elle nous donnerait plus de marge pour faire évoluer le firmware (ajout de fonctionnalités, mise à jour plus lourde), mais surtout pour stocker des éléments utiles au projet comme des paramètres de calibration, des coefficients issus d'essais, ou encore des enregistrements de mesures sur une durée prolongée.
- **Ajout d'un filtrage numérique sur les mesures.** Dans notre chaîne de traitement, la mesure capacitive peut être sensible à des perturbations (bruit électronique, environnement proche, mouvements rapides, parasites). De plus, certaines grandeurs calculées, comme la dérivée de l'angle, amplifient naturellement le bruit. Une perspective importante est donc d'intégrer un filtrage numérique léger sur les capacités mesurées ou sur  $C_{tot}$ . L'objectif n'est pas de ralentir la mesure, mais de stabiliser les valeurs et d'améliorer la répétabilité du couple estimé, surtout en conditions réelles d'utilisation.
- **Intégration d'autres capteurs directement sur la carte.** Pour rendre la solution plus complète et plus exploitable dans un exosquelette, nous pouvons envisager l'ajout d'autres capteurs embarqués. Par exemple, un capteur de température permettrait d'étudier la dérive thermique des mesures capacitatives. De la même manière, l'ajout d'un IMU ou d'un capteur d'angle externe pourrait fournir des informations complémentaires sur le mouvement.
- **Ajout de LEDs supplémentaires** pour visualiser l'intensité du couple fournit au capteur. L'idée serait de disposer un plus grand nombre de LEDs tout autour du PCB et d'allumer plus ou moins de LEDs en fonction de l'intensité fournit.



# Bibliographie

- [1] J. E. Aguilar-Segovia, M. Manzano, S. Guégan, R. Le Breton, A. Farhi-Rivasseau, S. Lefebvre, et M. Babel, « Multi-Material Torque Sensor Embedding One-Shot 3D-Printed Deformable Capacitive Structures », *IEEE Sensors Letters*, vol. 8, no. 9, 2024.
- [2] M. Manzano, *[Référence à compléter à partir du PDF Manzano 2024 : titre exact, conférence/journal, lieu, date, pages]*, 2024.
- [3] D. Chiaradia, L. Tiseni, et A. Frisoli, « Compact Series Visco-Elastic Joint for Smooth Torque Control », *IEEE Transactions on Haptics*, 2020.
- [4] A. Badel, J. Qiu, et T. Nakano, « A New Simple Asymmetric Hysteresis Operator and its Application to Inverse Control of Piezoelectric Actuators », *IEEE Transactions on Ultrasonics, Ferroelectrics, and Frequency Control*, 2008.
- [5] S. Bian, M. Liu, B. Zhou, P. Lukowicz, et M. Magno, « Body-Area Capacitive or Electric Field Sensing for Human Activity Recognition and Human-Computer Interaction : A Comprehensive Survey », *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol. (IMWUT)*, 2024.
- [6] M. Schmitz, J. Riemann, F. Müller, S. Kreis, et M. Mühlhäuser, « Oh, Snap ! A Fabrication Pipeline to Magnetically Connect Conventional and 3D-Printed Electronics », *CHI '21 (ACM Conference on Human Factors in Computing Systems)*, 2021.

# 8 Annexe

## 8.1 Lien du dépôt GitHub

Accéder au dépôt GitHub

## 8.2 Datasheets des composants

Dans cette section, nous regroupons les liens vers les fiches techniques (datasheets) et les pages produits des principaux composants utilisés sur notre carte.

- **Microcontrôleur – ESP32-S3-WROOM-1-N16R8 (Espressif)** :  
Lien Mouser (fiche produit / datasheet)
- **Convertisseur capacitif-numérique – FDC1004 (Texas Instruments)** :  
Page produit FDC1004 (TI)
- **Transceiver CAN – TJA1051 (NXP)** :  
Datasheet TJA1051 (PDF)
- **Régulateur LDO – TLV761 (Texas Instruments)** :  
Datasheet TLV761 (PDF)
- **Interrupteur (switch) – Nidec Components (série CAS)** :  
Datasheet série CAS (PDF)
- **Micro USB Type B – 2013499-1 (TE Connectivity)** :  
Page produit TE référence 2013499-1
- **Bouton Poussoir cms – 2337233-1(TE Connectivity)** :  
Page produit TE référence 2337233-1

## 8.3 Code C++

```
1 //header déclarations + structures .hpp
2
3 #pragma once
4 #include <cstdint>
5 #include <cmath>
6 #include "driver/i2c_master.h"
```

```

7
8
9 struct TS_HyperbolaParams {
10     float a, b, c, x0, y0;
11 };
12
13 struct TS_Params {
14     // Terme dynamique
15     float m = 2e-3f;
16     float p = 0.013f;
17
18     // Capa -> angle
19     float p1 = 3.9f;
20     float p0 = 0.4f;
21     float pdtheta = 0.012f;
22
23     TS_HyperbolaParams up;
24     TS_HyperbolaParams down;
25 };
26
27
28 // Sorties principales
29
30 struct TS_Output {
31     float CIN1_pf = 0.f;
32     float CIN2_pf = 0.f;
33     float CIN3_pf = 0.f;
34     float CIN4_pf = 0.f;
35
36     // Mapping
37     // C1=CIN4, C2=CIN3, C3=CIN2, C4=CIN1
38     float C1_pf = 0.f;
39     float C2_pf = 0.f;
40     float C3_pf = 0.f;
41     float C4_pf = 0.f;
42
43     float Ctot = 0.f;
44
45     float theta_estim_deg = 0.f;
46
47     float tau_stat = 0.f;
48     float tau_dyn = 0.f;
49     float tau_tot = 0.f;

```

```

50
51     int direction = -1; // +1 branche montante, -1 branche
52         descendante
53
54
55 // Librairie principale
56
57 // Cette classe encapsule:
58 // - Driver FDC1004 (I2C)
59 // - Traduction de torque_angle_estimator.m (hystérésis)
60 // - Pipeline de calcul (Ctot, angle, couple statique/
61     dynamique)
62 class TorqueSensorLib {
63 public:
64     // Adresse I2C FDC1004
65     static constexpr uint8_t FDC1004_ADDR = 0x50;
66
67     // Constructeur: le handle device I2C est créé c t é
68         application ESP-IDF.
69     TorqueSensorLib(i2c_master_dev_handle_t dev, const
70                     TS_Params& params);
71
72     // Initialisation FDC1004 + configuration des 4 mesures
73         CINx / CAPGND.
74     // capdac: 0..31 (offset hardware).
75     // timeout_ms: délai max d'attente data ready (poll).
76     bool begin(uint8_t capdac = 0, uint32_t timeout_ms = 50);
77
78     // Mise à jour complète:
79     // - lit CIN1..CIN4
80     // - calcule Ctot
81     // - calcule theta_estim + tau_stat + tau_dyn + tau_tot
82     // dt_s: période d'échantillonnage en secondes (ex: 0.01f)
83     TS_Output update(float dt_s);
84
85     // Reset de l'état interne (hystérésis + dérivée).
86     void reset();
87
88 private:
89
90     // Driver FDC1004

```

```

88 enum class FDC_Channel : uint8_t {
89     CIN1    = 0,
90     CIN2    = 1,
91     CIN3    = 2,
92     CIN4    = 3,
93     CAPGND = 4
94 };
95
96     bool fdc_write16(uint8_t reg, uint16_t v);
97     bool fdc_read16(uint8_t reg, uint16_t& v);
98
99     bool fdc_config_measure(uint8_t meas_1_to_4, FDC_Channel
100         cha, FDC_Channel chb, uint8_t capdac);
101     bool fdc_trigger(uint8_t meas_1_to_4);
102     bool fdc_wait_ready(uint8_t meas_1_to_4, uint32_t
103         timeout_ms);
104     bool fdc_read_raw24(uint8_t meas_1_to_4, int32_t& raw24);
105
106 // Conversion datasheet 15 pF:
107 // pF = raw/2^19 + capdac*3.125
108     static float raw24_to_pf(int32_t raw24, uint8_t capdac);
109
110 // hystérésis
111
112     static inline float sgn(float x) { return (x > 0.f) - (x <
113         0.f); }
114
115     static float hyperbola(float x, const TS_HyperbolaParams& p
116         );
117     float fup(float x) const;
118     float fdown(float x) const;
119
120     void compute_major_intersections();
121
122     float hysteresis_update(float angle_deg);
123
124 // Pipeline calcul
125
126     static float compute_Ctot(float C1, float C2, float C3,
127         float C4);
128     float compute_dtheta(float theta, float dt_s);

```

```

126 private:
127     i2c_master_dev_handle_t dev_;
128     TS_Params params_;
129
130     // FDC config
131     uint8_t capdac_ = 0;
132     uint32_t timeout_ms_ = 50;
133
134     // Major loop intersections (fup==fdown)
135     float angle_m_min_ = -5.f;
136     float angle_M_max_ = 5.f;
137     float torque_m_min_ = 0.f;
138     float torque_M_max_ = 0.f;
139
140     // Translation boucle mineure
141     float angle_trans_ = 0.f;
142     float torque_trans_ = 0.f;
143
144     // Etat hystérésis
145     int direction_ = -1;
146     float seuil_rel_ = 0.1f;
147     float seuil_abs_ = NAN;
148     float bidouille_ = 50.f;
149     bool hyst_initied_ = false;
150
151     // Dérivée angle
152     float theta_prev_ = 0.f;
153     bool theta_initied_ = false;
154 };

```

```

1 // l implementation .cpp
2 #include "torque_sensor_lib.hpp"
3 #include "freertos/FreeRTOS.h"
4 #include "freertos/task.h"
5
6 // Registres FDC1004
7 static constexpr uint8_t REG_MEAS1_MSB    = 0x00; // MEAS1 MSB
8     (puis LSB à 0x01, etc.)
9 static constexpr uint8_t REG_MEAS_CONFIG = 0x0C; // trigger +
10    DONE bits
11 static constexpr uint8_t REG_FDC_CONF    = 0x0F; // config
12    globale
13 static constexpr uint8_t REG_MEASx_CFG   = 0x08; // 0x08..0

```

```

11     x0B
12
13 TorqueSensorLib::TorqueSensorLib(i2c_master_dev_handle_t dev,
14     const TS_Params& params)
15 : dev_(dev), params_(params) {
16     reset();
17     compute_major_intersections();
18 }
19
20 // Reset
21
22 void TorqueSensorLib::reset() {
23     angle_trans_ = 0.f;
24     torque_trans_ = 0.f;
25
26     direction_ = -1;
27     seuil_rel_ = 0.1f;
28     seuil_abs_ = NAN;
29     bidouille_ = 50.f;
30     hyst_initiated_ = false;
31
32     theta_prev_ = 0.f;
33     theta_initiated_ = false;
34 }
35
36 // I2C low level
37
38 bool TorqueSensorLib::fdc_write16(uint8_t reg, uint16_t v) {
39     uint8_t buf[3] = { reg, uint8_t(v >> 8), uint8_t(v & 0xFF) };
40     return i2c_master_transmit(dev_, buf, sizeof(buf),
41                               pdMS_TO_TICKS(50)) == ESP_OK;
42 }
43
44 bool TorqueSensorLib::fdc_read16(uint8_t reg, uint16_t& v) {
45     uint8_t r = reg;
46     uint8_t in[2] = {0, 0};
47     if (i2c_master_transmit_receive(dev_, &r, 1, in, 2,
48                                     pdMS_TO_TICKS(50)) != ESP_OK) return false;
49     v = (uint16_t(in[0]) << 8) | uint16_t(in[1]);
50     return true;

```

```

49 }
50
51 // Driver FDC1004 (complet)
52
53 bool TorqueSensorLib::fdc_config_measure(uint8_t meas,
54     FDC_Channel cha, FDC_Channel chb, uint8_t capdac) {
55     if (meas < 1 || meas > 4) return false;
56     if (capdac > 31) capdac = 31;
57
58     const uint8_t reg = uint8_t(REG_MEASx_CFG + (meas - 1));
59
60     // MEASx_CONFIG:
61     // bits 15..13 CHA, 12..10 CHB, 9..5 CAPDAC
62     uint16_t v = 0;
63     v |= (uint16_t(uint8_t(cha) & 0x07) << 13);
64     v |= (uint16_t(uint8_t(chb) & 0x07) << 10);
65     v |= (uint16_t(capdac & 0x1F) << 5);
66
67     return fdc_write16(reg, v);
68 }
69
70 bool TorqueSensorLib::fdc_trigger(uint8_t meas) {
71     if (meas < 1 || meas > 4) return false;
72
73     // Start bits 0..3
74     uint16_t v = 0;
75     v |= (1u << (meas - 1));
76     return fdc_write16(REG_MEAS_CONFIG, v);
77 }
78
79 bool TorqueSensorLib::fdc_wait_ready(uint8_t meas, uint32_t
80     timeout_ms) {
81     if (meas < 1 || meas > 4) return false;
82
83     // DONE bits: DONE1=bit12, DONE2=bit13, DONE3=bit14, DONE4=
84     // bit15
85     const uint16_t done_mask = uint16_t(1u << (11 + meas));
86
87     TickType_t t_end = xTaskGetTickCount() + pdMS_TO_TICKS(
88         timeout_ms);
89     while (xTaskGetTickCount() < t_end) {
90         uint16_t st = 0;

```

```

88     if (!fdc_read16(REG_MEAS_CONFIG, st)) return false;
89     if (st & done_mask) return true;
90     vTaskDelay(pdMS_TO_TICKS(2));
91 }
92 return false;
93 }

94
95 bool TorqueSensorLib::fdc_read_raw24(uint8_t meas, int32_t&
96     raw24) {
97     if (meas < 1 || meas > 4) return false;
98
99     const uint8_t base = uint8_t(REG_MEAS1_MSB + (meas - 1) *
100        2);
101    uint16_t msb = 0, lsb = 0;
102    if (!fdc_read16(base, msb)) return false;
103    if (!fdc_read16(base + 1, lsb)) return false;
104
105    // 24-bit signed: MSB(16) + top8(LSB)
106    int32_t v24 = (int32_t(msb) << 8) | (int32_t(lsb) >> 8);
107
108    // sign extend 24-bit
109    if (v24 & 0x800000) v24 |= ~0xFFFFF;
110    raw24 = v24;
111    return true;
112 }

113 float TorqueSensorLib::raw24_to_pf(int32_t raw24, uint8_t
114     capdac) {
115
116     const float pf = float(raw24) / 524288.0f;
117     const float off = float(capdac) * 3.125f;
118     return pf + off;
119 }

120 // begin() : init FDC1004
121
122 bool TorqueSensorLib::begin(uint8_t capdac, uint32_t
123     timeout_ms) {
124     capdac_ = (capdac > 31) ? 31 : capdac;
125     timeout_ms_ = timeout_ms;
126
127     // Configuration globale

```

```

127     if (!fdc_write16(REG_FDC_CONF, 0x0000)) return false;
128
129 // MEAS1..4 = CIN1..CIN4 vs CAPGND
130 if (!fdc_config_measure(1, FDC_Channel::CIN1, FDC_Channel::
131     CAPGND, capdac_)) return false;
132 if (!fdc_config_measure(2, FDC_Channel::CIN2, FDC_Channel::
133     CAPGND, capdac_)) return false;
134 if (!fdc_config_measure(3, FDC_Channel::CIN3, FDC_Channel::
135     CAPGND, capdac_)) return false;
136 if (!fdc_config_measure(4, FDC_Channel::CIN4, FDC_Channel::
137     CAPGND, capdac_)) return false;
138
139
140
141 // Maths hyperboles
142
143 float TorqueSensorLib::hyperbola(float x, const
144     TS_HyperbolaParams& p) {
145     // y = a*(1 - sqrt(1 + (x-x0)^2/b^2)) + c*(x-x0) + y0
146     const float dx = x - p.x0;
147     const float bb = p.b * p.b;
148     return p.a * (1.f - std::sqrt(1.f + (dx * dx) / bb)) + p.c
149         * dx + p.y0;
150 }
151
152 float TorqueSensorLib::fup(float x) const { return
153     hyperbola(x, params_.up); }
154 float TorqueSensorLib::fdown(float x) const { return
155     hyperbola(x, params_.down); }
156
157 void TorqueSensorLib::compute_major_intersections() {
158
159     // On recherche les changements de signe de (fup - fdown)
160     // sur [-40, 40].
161     const float xmin = -40.f, xmax = 40.f;
162     const int N = 4000;
163
164     float roots[8];
165     int n = 0;

```

```

161
162     float px = xmin;
163     float pf = fup(px) - fdown(px);
164
165     for (int i = 1; i <= N; ++i) {
166         float x = xmin + (xmax - xmin) * (float(i) / float(N));
167         float f = fup(x) - fdown(x);
168
169         if (sgn(f) != sgn(pf)) {
170             float a = px, b = x;
171             float fa = pf;
172
173             for (int it = 0; it < 30; ++it) {
174                 float m = 0.5f * (a + b);
175                 float fm = fup(m) - fdown(m);
176                 if (sgn(fm) == sgn(fa)) { a = m; fa = fm; }
177                 else { b = m; }
178             }
179             if (n < 8) roots[n++] = 0.5f * (a + b);
180         }
181
182         px = x;
183         pf = f;
184     }
185
186     if (n >= 2) {
187         angle_m_min_ = roots[0];
188         angle_M_max_ = roots[n - 1];
189     }
190
191     torque_m_min_ = fdown(angle_m_min_);
192     torque_M_max_ = fdown(angle_M_max_);
193 }
194
195 // Hystérésis
196
197 float TorqueSensorLib::hysteresis_update(float angle_deg) {
198     if (!hyst_initiated_) {
199         seuil_abs_ = angle_deg;
200         hyst_initiated_ = true;
201     }
202 }
```

```

204     float torque = 0.f;
205
206     // Couple statique = min/max entre courbe translatée et
207     // majeure
207     if (direction_ == +1) {
208         const float t_cur = fup(angle_deg - angle_trans_) +
209             torque_trans_;
210         const float t_maj = fup(angle_deg);
211         torque = (t_cur < t_maj) ? t_cur : t_maj;
212
213         // Retour sur majeure => translation remise à zéro
214         if (torque == t_maj) { angle_trans_ = 0.f; torque_trans_
215             = 0.f; }
214     } else {
215         const float t_cur = fdown(angle_deg - angle_trans_) +
216             torque_trans_;
216         const float t_maj = fdown(angle_deg);
217         torque = (t_cur > t_maj) ? t_cur : t_maj;
218
219         if (torque == t_maj) { angle_trans_ = 0.f; torque_trans_
220             = 0.f; }
220     }
221
222     // Détection d'extrema + recalage translation (dichotomie)
223     if (direction_ == -1) {
224         // seuil_abs = min(seuil_abs, angle + seuil_rel)
225         seuil_abs_ = std::fmin(seuil_abs_, angle_deg + seuil_rel_
226             );
226
227         if (angle_deg > seuil_abs_) {
228             // MIN détecté
229             const float angle_m = angle_deg - bidouille_ *
230                 seuil_rel_;
231             const float Torque_m = fdown(angle_m - angle_trans_) +
232                 torque_trans_;
232
233             // Version "test": l'ancien maximum est pris sur la
234             // majeure
234             const float angle_M = angle_M_max_;
235             const float Torque_M = torque_M_max_;
235
236             // Solve: fup(angle_M-x) - fup(angle_m-x) - Torque_M +
237             // Torque_m = 0

```

```

237     float a = -20.f, b = 20.f;
238     const float eps = 0.05f;
239
240     float fa = fup(angle_M - a) - fup(angle_m - a) -
241         Torque_M + Torque_m;
242     for (int it = 0; it < 200 && (b - a) > eps; ++it) {
243         float x = 0.5f * (a + b);
244         float fx = fup(angle_M - x) - fup(angle_m - x) -
245             Torque_M + Torque_m;
246         if (sgn(fx) == sgn(fa)) { a = x; fa = fx; }
247         else { b = x; }
248     }
249
250     const float x_sym = 0.5f * (a + b);
251     angle_trans_ = x_sym;
252     torque_trans_ = Torque_m - fup(angle_m - x_sym);
253
254     direction_ = +1;
255     seuil_abs_ = angle_deg;
256 }
257 } else {
258     // direction_ == +1
259     seuil_abs_ = std::fmax(seuil_abs_, angle_deg - seuil_rel_);
260
261     if (angle_deg < seuil_abs_) {
262         // MAX détecté
263         const float angle_M = angle_deg + bidouille_ *
264             seuil_rel_;
265         const float Torque_M = fup(angle_M - angle_trans_) +
266             torque_trans_;
267
268         // Version "test": l'ancien minimum est pris sur la
269         // majeure
270         const float angle_m = angle_m_min_;
271         const float Torque_m = torque_m_min_;
272
273         // Solve: fdown(angle_M-x) - fdown(angle_m-x) -
274             Torque_M + Torque_m = 0
275         float a = -20.f, b = 20.f;
276         const float eps = 0.05f;
277
278         float fa = fdown(angle_M - a) - fdown(angle_m - a) -

```

```

    Torque_M + Torque_m;
273  for (int it = 0; it < 200 && (b - a) > eps; ++it) {
274      float x = 0.5f * (a + b);
275      float fx = fdown(angle_M - x) - fdown(angle_m - x) -
276          Torque_M + Torque_m;
277      if (sgn(fx) == sgn(fa)) { a = x; fa = fx; }
278      else { b = x; }
279  }
280
281  const float x_sym = 0.5f * (a + b);
282  angle_trans_ = x_sym;
283  torque_trans_ = Torque_m - fdown(angle_m - x_sym);
284
285  direction_ = -1;
286  seuil_abs_ = angle_deg;
287 }
288
289 return torque;
290 }
291
292 // Pipeline couple
293
294 float TorqueSensorLib::compute_Ctot(float C1, float C2, float
295     C3, float C4) {
296     // Formule MATLAB : (-C1 - C3 + C2 + C4)/1000
297     return (-C1 - C3 + C2 + C4) / 1000.f;
298 }
299
300 float TorqueSensorLib::compute_dtheta(float theta, float dt_s
301     ) {
302     if (!theta_initiated_ || dt_s <= 0.f) {
303         theta_prev_ = theta;
304         theta_initiated_ = true;
305         return 0.f;
306     }
307     float v = (theta - theta_prev_) / dt_s;
308     theta_prev_ = theta;
309     return v;
310 }
311

```

```

312 // update() : lecture + calcul complet
313
314 TS_Output TorqueSensorLib::update(float dt_s) {
315     TS_Output out{};
316
317     // 1) Déclenchement des 4 mesures
318     for (int m = 1; m <= 4; ++m) {
319         if (!fdc_trigger(uint8_t(m))) return out;
320     }
321
322     // 2) Attente data ready
323     for (int m = 1; m <= 4; ++m) {
324         if (!fdc_wait_ready(uint8_t(m), timeout_ms_)) return out;
325     }
326
327     // 3) Lecture raw + conversion en pF
328     float cin_pf[4] = {0,0,0,0};
329     for (int m = 1; m <= 4; ++m) {
330         int32_t raw = 0;
331         if (!fdc_read_raw24(uint8_t(m), raw)) return out;
332         cin_pf[m - 1] = raw24_to_pf(raw, capdac_);
333     }
334
335     out.CIN1_pf = cin_pf[0];
336     out.CIN2_pf = cin_pf[1];
337     out.CIN3_pf = cin_pf[2];
338     out.CIN4_pf = cin_pf[3];
339
340     // 4) Mapping a (C1..C4 )
341     // - CIN1 <- Cin4
342     // - CIN2 <- Cin3
343     // - CIN3 <- Cin2
344     // - CIN4 <- Cin1
345
346     out.C1_pf = out.CIN4_pf;
347     out.C2_pf = out.CIN3_pf;
348     out.C3_pf = out.CIN2_pf;
349     out.C4_pf = out.CIN1_pf;
350
351     // 5) Capacité différentielle
352     out.Ctot = compute_Ctot(out.C1_pf, out.C2_pf, out.C3_pf,
353                             out.C4_pf);

```

```

354 // 6) Angle estimé (structure MATLAB)
355 float theta_base = params_.p1 * out.Ctot + params_.p0;
356 float theta_dot = compute_dtheta(theta_base, dt_s);
357
358 // La correction directionnelle utilise la direction
359 // courante avant update hystérésis.
360 const int dir_before = direction_;
361 out.theta_estim_deg = theta_base + float(dir_before) *
362     params_.pdtheta * theta_dot;
363
364 // 7) Couple statique par hystérésis (sur theta_estim_deg)
365 out.tau_stat = hysteresis_update(out.theta_estim_deg);
366 out.direction = direction_;
367
368 // 8) Couple dynamique (équation MATLAB)
369 out.tau_dyn = (params_.m * out.theta_estim_deg - float(out.
370     direction) * params_.p) * std::fabs(theta_dot);
371
372 // 9) Couple total
373 out.tau_tot = out.tau_stat + out.tau_dyn;
374
375 return out;
376 }
```

```

1 // exemple d'utilisation \\
2 TS_Params p;
3 p.up    = {0.5023f, 0.9021f, 1.0109f, -3.2283f, -1.5653f};
4 p.down = {-0.1665f, 0.7224f, 0.7070f, 2.4590f, 0.9289f};
5 p.m = 2e-3f; p.p = 0.013f;
6 p.p1 = 3.9f; p.p0 = 0.4f; p.pdtheta = 0.012f;
7
8 TorqueSensorLib ts(fdc_dev_handle, p);
9 ts.begin(/*capdac=*/0, /*timeout_ms=*/50);
10
11 TS_Output o = ts.update(0.01f);
```