Lycée Alphonse BENOÎT
Cours Victor HUGO
84800 L'ISLE SUR LA SORGUE
Téléphone : 04-90-20-64-20

ACADÉMIE D'AIX-MARSEILLE
Liberté Égalité Fraternité

PROJET ECG

ÉLECTROCARDIOGRAMME

Projet ECG
Simuler. Connecter. Analyser.

BTS CIEL

Épreuve E6.2 – Session 2026

Dossier de revue finale

Valentín BUFFARD (IR 1)

Ivan GOUIN (IR 2)

Raphael SABBEN (IR 3)

Rafael FERNANDES (ER 1)

Paul MICHEL (ER 2)

AVIGNON UNIVERSITÉ
CERI
Centre d'Enseignement et de Recherche en Informatique

Dépot framagit du projet : https://framagit.org/ValNoSekyu/projet-bts-ciel-ecg-2026

TABLE DES MATIÈRES

1. PARTIE COMMUNE **8**
1.1. Expression du besoin 8
1.2. Analyse de l'existant 9
1.3. Solutions et évolutions technologiques 10
1.4. État de l'art et enjeux 11
1.4.1. Équipe et encadrement 11
1.4.2. Perspectives 11
1.5. Problématique 12
1.5.1. Acquisition et traitement de données physiologiques 12
1.5.2. Transmission et supervision en temps réel & Sécurité et gestion des données médicales 12
1.5.3. Exploitation et analyse intelligente des données 12
1.6. Environnement de développement 13
1.6.1. Outils utilisés côté IR 14
1.6.1.1. Outils commun 14
1.6.1.2. Outils côté holter 14
1.6.1.3. Outils côté simulateur 15
1.6.2. Outils côté ER 15
1.7. Planification Globale 17
2. PARTIE PERSONNELLE - VALENTIN BUFFARD **20**
2.1. Objectifs 20
2.2. Planification prévisionnelle 21
2.3. Outils utilisés 22
2.4. Environnement de développement python 22
2.4.1. Mise en place de l'environnement python 22
2.4.2. Librairies utilisées 23
2.4.3. Test de Dash & de FastAPI 23
2.4.3.1. FastAPI 24
2.5. Programmation de l'ESP32 25
2.5.1. Architecture Globale 25
2.5.2. FreeRTOS 26
2.5.3. Fonctionnement du programme 27
2.5.3.1. TaskAcquisition (Cœur 1 - Haute Priorité) 27
2.5.3.2. TaskSerial (Cœur 0 - Basse Priorité) 28
2.5.3.3. Communication SPI avec le MAX11100 28
2.5.3.4. Organisation de la Mémoire et Flux 28
2.5.3.5. Mapping des broches 30
2.6. Développement de l'application web 31
2.6.1. Fonctionnement de l'application 32
2.6.1.1. L'Architecture MVC et l'Injection de Dépendances 33
2.6.1.2. L'Acquisition de Données (Couche Matérielle) 34

2.6.1.3. La Gestion des Communications Temps Réel (WebSockets) 36
2.6.1.4. Persistance et Traitement des Données 38
2.6.1.5. Configuration et Environnement 38
2.7. Test du système d'acquisition 39
2.7.1. Validation de la période d'échantillonnage 42
2.7.2. Validation de la valeur reçue (Vérification de la trame SPI) 43
2.8. Conclusion 44
3. PARTIE PERSONNELLE - IVAN GOUIN 46
3.1 Objectifs 46
3.2 Outils Utilisés 47
3.3 Diagrammes de classes 48
3.4 Diagrammes de séquence 49
3.5 Code de l'application traitement des données 50
3.5.1 Difficultés rencontrées et solutions apportées 50
3.5.1.1 Problème d'encodage des données (Hexadécimal) 50
3.5.1.2 Communication WebSocket unidirectionnelle 50
3.5.1.3 Gestion de la base de données SQLite 51
3.5.1.4 Complexité du code initial 51
3.5.1.5 Communication avec le module CubeCell 52
3.5.1.6 Comptage des pics trop proche 52
3.5.2 CONFIG 53
3.5.3 APP 53
3.5.4 MODEL 53
3.5.4.1 Filtrage du signal ECG 54
3.5.4.2 Découpage en fenêtres glissantes 54
3.5.4.3 Distance physiologique minimale entre pics 54
3.5.4.4 Seuil adaptatif de détection 54
3.5.4.5 Détection des pics R 54
3.5.4.6 Fusion et nettoyage des pics détectés 54
3.5.4.7 Calcul des intervalles RR 55
3.5.4.8 Calcul de la fréquence cardiaque 55
3.5.5 VIEW 55
3.5.6 CONTROLLER 55
3.5.7 SERVICES 55
3.6 Validation du traitement ECG et tests unitaires 57
3.7 Node-RED 57
3.7.1 DATA 57
3.7.1.1 Réception des données 58
3.7.1.2 Normalisation des données 58
3.7.1.3 Visualisation en temps réel 59
3.7.1.4 Simulation des données (mode démonstration) 59
3.7.1.5 Simulation des données (par envoi de trames via TTN) 59
3.7.1.6 Stockage des données 59
3.7.2 HISTORY 60

3.7.2.1 Sélection de la période 60
3.7.2.2 Requête SQL 60
3.7.2.3 Visualisation graphique 60
3.7.2.4 Indicateurs statistiques 61
3.7.3 SETTINGS 61
3.7.3.1 Paramètres 61
3.7.3.2 Logique d'alertes 61
3.7.3.3 Stockage des paramètres 61
3.7.4 IHM finale et architecture globale 62
3.8 Partie Physique 63
3.8.1 Architecture du réseau LoRaWAN 63
3.8.2 Propagation radio et contraintes de transmission 63
3.8.3 Relation entre fréquence et longueur d'onde 64
3.8.4 Compromis entre portée, débit et consommation énergétique 64
3.8.5 Modulation Chirp Spread Spectrum (CSS) 65
3.9 Conclusion 65
ANNEXES TECHNIQUES 66
ANNEXE A 67
Environnement Python : 67
installation et bibliothèques utilisées 67
ANNEXE B 68
Codes application Traitement et analyse 68
1) Fichier config.py 68
2) Fichier app - main.py 68
3) Ficher ecg_metrics.py 69
4) Ficher hex_encoder.py 70
5) Fichier ecg_analyzer.py 71
6) Fichier console_view.py 74
7) Fichier ecg_controller.py 75
8) Fichier database_service.py 76
9) Fichier serial_service.py 77
10) Fichier websocket_service.py 77
ANNEXE C 80
Tests unitaires 80
ANNEXE D 83
Visualisation / Supervision 83
1) JS - normalize payload 83
2) debug uplink 85
3) HTML - Heart 86
4) JS - build simulated uplink 88
5) Simulation TTN 89
6) JS - build INSERT SQLite 91
7) HTML - History controls mockup 91
8) JS - build SELECT SQLite 92

9) History chart 92
10) HTML - History summary panel 93
11) HTML - Settings note 94
12) HTML - Alert 95
13) JS - save settings 95
4. PARTIE PERSONNELLE - Raphaël SABBEN 98
4.1 Objectifs 98
4.2 Planification prévisionnelle 99
4.3 Architecture système et choix matériels (Hardware) 100
4.3.1 Le microcontrôleur : Raspberry Pi Pico (RP2040) 100
4.3.2 Choix du DAC8551 pour la génération analogique 100
4.3.3 Protocole matériel SPI vers le DAC8551 100
4.3.4 La chaîne de communication physique (8N1) 101
4.3.5 Métrologie : Oscilloscope Analog Discovery / WaveForms 101
4.4 Modélisation SysML du système 102
4.4.1 Diagramme de cas d'utilisation (Use Case UML) 102
4.4.2 Diagramme de Définition de Blocs (BDD) 102
4.4.3 Diagramme de Bloc Interne (IBD) et Flux 103
4.4.4 Diagramme de séquence UML 104
4.4.5 Diagramme de Classes UML 105
4.5 Développement logiciel : L'interface C++ (Qt 5.15) 106
4.5.1 Rendu graphique médical 106
4.5.2 Algorithme de parsing CSV (jusqu'à 3600 points) 107
4.5.3 Conversion Float → Uint16 avec écrêtage de sécurité 107
4.6 Protocole de communication binaire 107
4.6.1 Sérialisation Little-Endian 107
4.6.2 Trame de communication 107
4.6.3 Checksum Modulo 256 107
4.7 Développement du firmware embarqué (MicroPython / Pico) 107
4.7.1 Assembleur PIO — Génération SPI sans CPU 107
4.7.2 Transfert DMA — 2 canaux chaînés 108
4.7.3 Protocole de communication — Commandes UART 108
4.7.4 Organisation des données médicales : structure en dossiers CSV 108
4.8 Problèmes rencontrés et solutions apportées 109
4.8.1 Crash au deuxième clic sur Envoyer 109
4.8.2 L'interface se gelait pendant l'envoi 109
4.8.3 Le signal restait à 0V sur l'oscilloscope 109
4.8.4 Le signal était bloqué à 3,3V en permanence 109
4.9 Protocoles de tests et résultats métrologiques 109
4.10 Conclusion générale et perspectives d'évolution 112
ANNEXE A — Schéma de Câblage SPI : Raspberry Pi Pico → DAC8551 114
ANNEXE B — Code MicroPython Complet : PIO et DMA 114
ANNEXE C — Code C++ Qt : mainwindow.cpp Complet 118
ANNEXE D — Tableau de Métrologie et Résultats de Tests 124

ANNEXE E — Glossaire Technique 124
ANNEXE F — Récapitulatif des Décisions d'Architecture 125
ANNEXE G — Environnement de Développement et Outils Utilisés 125
5. PARTIE PERSONNELLE - RAFAËL FERNANDES 128
5.1 Objectifs 128
6. PARTIE PERSONNELLE - PAUL MICHEL 146
6.1 Introduction personnelle 146
6.2 Organisation du Projet – Diagramme de Gantt 147
6.3 Présentation du travail 148
6.3.1 Mise en place de mon environnement de travail 148
6.3.2 Revue du travail effectué par les étudiants 2024-2025 148
6.3.3 Mise en place et configuration de la Raspberry Pi 149
6.3.4 Test de la carte Holter par les étudiants 2024-2025 149
6.3.5 Création du schéma structurel de la carte 151
6.3.6 Routage de la nouvelle carte Holter 153
6.3.7 Montage de la carte Holter 155

PARTIE COMMUNE

1. PARTIE COMMUNE

1.1. Expression du besoin

Le cahier de charges du projet découle d'un sujet d'activité proposée aux étudiants du CERI d'avignon dans le cadre d'une unité d'enseignement (UE) portant sur l'internet des objets (IoT)

Le CERI d'Avignon développe une intelligence artificielle capable de détecter automatiquement les anomalies cardiaques. Pour que cette IA devienne performante, elle a besoin de s'entraîner sur un très grand nombre de signaux d'électrocardiogrammes (ECG) variés.

Le problème, c'est qu'il est très difficile de trouver des signaux provenant de patients réels ayant des pathologies cardiaques. Pour surmonter cette difficulté, le projet se divise en deux parties complémentaires :

- **Le simulateur de signaux ECG :** Cet appareil est chargé de générer les signaux cardiaques. Pour cela, il s'appuie sur des fichiers MATLAB, provenant de différents instituts de la santé, qui ont été convertis au format CSV. Cela permet de reproduire fidèlement et à la demande un grand nombre de pathologies.
- **Le Holter connecté et son application Web :** Ce boîtier fait l'acquisition du signal généré par le simulateur. Les données récupérées sont ensuite envoyées vers une application web qui permet de visualiser le signal et de l'analyser pour y détecter les anomalies cardiaques.

Grâce à cette solution, le CERI d'Avignon dispose d'une chaîne complète pour simuler, acquérir, visualiser et analyser des pathologies sur mesure, offrant ainsi une base d'entraînement idéale pour leur IA.

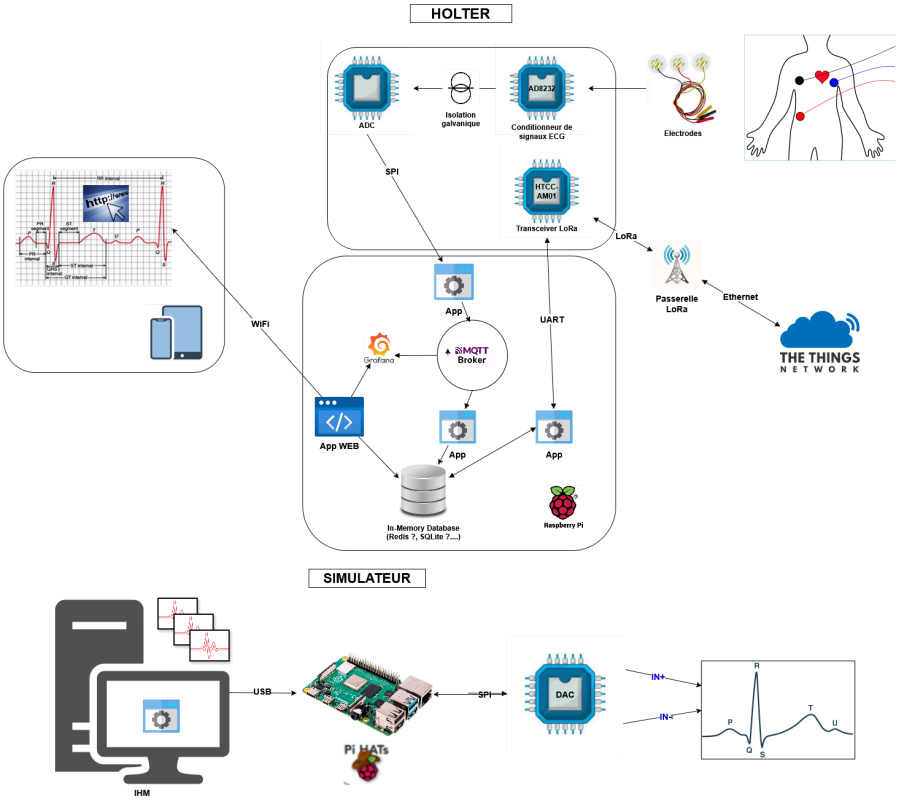
1.2. Analyse de l’existant

Dans le cadre de l'unité d'enseignement dédiée à l'**Internet des Objets (IoT)**, le CERI souhaite moderniser un système de surveillance cardiaque par électrocardiogramme (ECG).-(

Un premier projet avait permis l'année dernière d'acquérir et d'enregistrer les signaux. Cependant, cette solution ne permettait ni le traitement, ni la transmission des données en temps réel, limitant ainsi la supervision à distance.

Le travail réalisé l'an dernier a permis de poser les bases du système, mais l'utilisation d'un système d'exploitation non temps réel (Raspberry Pi OS) a révélé des limites critiques :

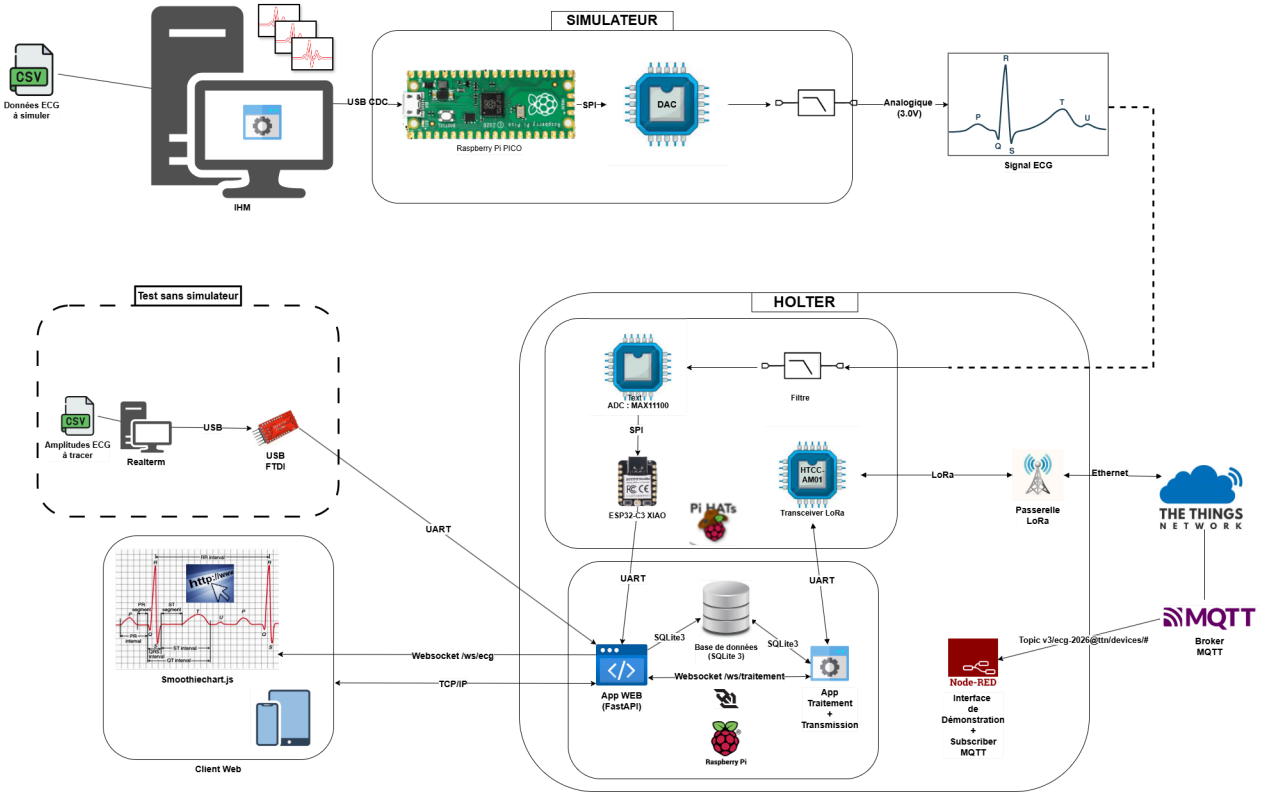
- **Instabilité temporelle :** La principale contrainte vient de l'utilisation d'un OS non "temps réel". La Raspberry Pi, gérant plusieurs tâches simultanées, ne pouvait pas garantir une fréquence stable pour la génération et l'acquisition des points ECG. Cela créait des irrégularités dans l'échantillonnage.
- **Saturation de l'affichage :** L'outil de visualisation (Grafana) atteignait ses limites face au flux important de données. Une fréquence d'échantillonnage élevée entraînait des dépassements de capacité, rendant l'affichage instable ou incomplet.
- **Rigidité matérielle :** Le simulateur dépendait exclusivement d'une extension spécifique (HAT) pour Raspberry Pi, ce qui limitait son utilisation et sa portabilité.



Ancienne topologie ECG de l’an dernier

1.3. Solutions et évolutions technologiques

Pour corriger ces défauts, nous avons choisi de faire évoluer l'architecture vers des solutions matérielles et logicielles plus spécialisées :



- **Utilisation de microcontrôleurs dédiés :** Pour garantir la précision du signal, nous avons séparé les tâches. Un **Raspberry Pi Pico** est désormais dédié à la génération du signal (simulateur) et un **ESP32** à l'acquisition. Ces composants fonctionnent sans OS complexe, ce qui assure une fréquence d'échantillonnage parfaitement stable.

- **Nouvelle architecture de traitement :** Pour remplacer Grafana, Nous avons développé une solution personnalisée en **Python** utilisant le protocole **WebSocket**. Contrairement au MQTT utilisé précédemment, cela permet un affichage plus fluide et une meilleure gestion des flux de données à haute fréquence.

- **Amélioration du simulateur :** Le nouveau simulateur est désormais reconnu comme un **périphérique USB**. Cette approche le rend beaucoup plus flexible : il suffit de le brancher à un ordinateur sur lequel est **installée l'application graphique (IHM)**. Cette interface permet à l'utilisateur de choisir et de charger des signaux issus de bases de données d'instituts médicaux renommés. Ces enregistrements réels, préalablement **convertis au format CSV**, sont ensuite transmis au simulateur pour être générés, rendant le système mobile et parfaitement adapté à des tests en conditions réelles.

1.4. État de l'art et enjeux

Bien qu'il existe déjà de nombreux dispositifs ECG connectés, ils restent souvent peu accessibles au grand public. Leurs capacités d'analyse en temps réel sont limitées et leur exploitation demande généralement une expertise médicale spécifique pour interpréter les données.

Notre projet s'inscrit donc dans une démarche d'innovation mêlant **IoT et télémédecine**. Il répond à des enjeux technologiques majeurs :

- **Traitement de données :** Gestion et analyse des signaux physiologiques.
- **Communication :** Transmission sécurisée via les protocoles **WebSocket** et **LoRa**.

- **Cybersécurité :** Protection des données de santé.

1.4.1. Équipe et encadrement

Ce projet est développé dans un cadre pédagogique par une équipe de cinq étudiants (Valentin Buffard (IR), Ivan Gouin (IR), Raphael Sabben (IR), Rafael Fernandes (ER) et Paul Michel (ER)), spécialisés en informatique et électronique. Nous travaillons sous la supervision de C. Defrance, C. Hortolland et F. Escuret.

1.4.2. Perspectives

À terme, cette solution répondra aux besoins du **CERI d'Avignon**. Elle servira de plateforme expérimentale pour la recherche sur l'intelligence artificielle appliquée à la santé dans le cadre d'une unité d'enseignement (UE) portant sur l'internet des objets (IoT).

1.5. Problématique

Le développement d'un système de surveillance cardiaque connecté soulève plusieurs problématiques techniques et fonctionnelles. En effet, bien que des solutions d'acquisition de signaux ECG existent déjà, elles présentent encore certaines limites, notamment en matière de **traitement des données en temps réel, de transmission sécurisée et d'exploitation des informations collectées**.

De plus, un projet réalisé l'année précédente avait permis de mettre en place un premier système d'enregistrement des signaux ECG. Toutefois, cette solution ne permettait pas une **transmission et une analyse en temps réel des données**, ce qui limitait les possibilités de surveillance instantanée et d'exploitation pédagogique.

Dans ce contexte, plusieurs questions se posent.

1.5.1. Acquisition et traitement de données physiologiques

Comment concevoir un dispositif capable de **capturer et traiter de manière fiable les signaux électrocardiographiques** d'un patient, tout en garantissant une qualité de mesure suffisante pour permettre une analyse pertinente de l'activité cardiaque ?

1.5.2. Transmission et supervision en temps réel & Sécurité et gestion des données médicales

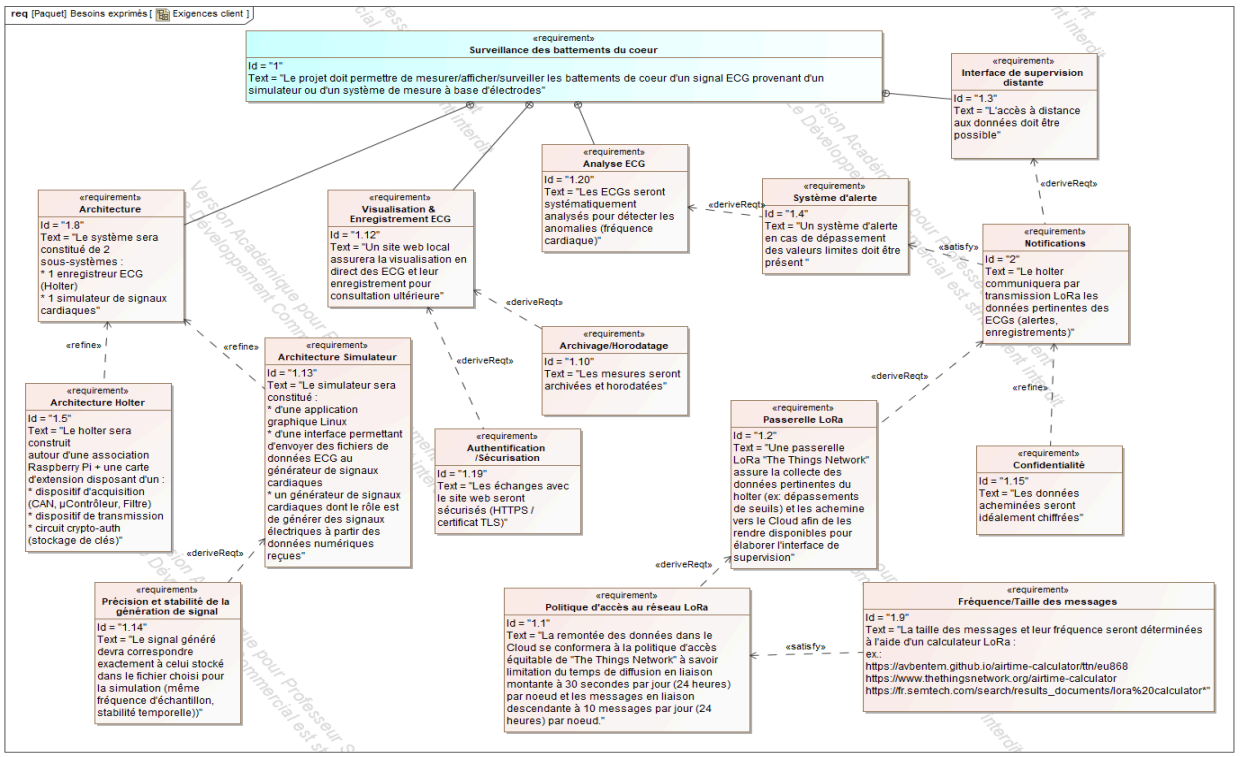
Comment assurer la **transmission des résultats d'analyse ECG en temps réel** vers une infrastructure distante, tout en utilisant des technologies de communication adaptées aux contraintes des systèmes IoT.

Comment garantir la **sécurité, l'intégrité et la confidentialité des données physiologiques transmises**, notamment dans un contexte de communication via des réseaux connectés et des infrastructures cloud ?

1.5.3. Exploitation et analyse intelligente des données

Comment faciliter l'**analyse et l'interprétation des données collectées**, notamment grâce à l'intégration d'outils d'**intelligence artificielle** permettant de détecter automatiquement certaines anomalies du rythme cardiaque ?

Pour répondre à ces problématiques voici le diagrammes des exigences du projet.



1.6. Environnement de développement



Bien que la cible finale du Holter soit une **Raspberry Pi 5**, une grande partie du développement a été réalisée sur une machine virtuelle **Debian 13**.

Ce choix s'est avéré particulièrement avantageux pour l'organisation de notre travail :

- **Centralisation des outils :** Bien qu'aucune plateforme ne nous ait été imposée, nous avons choisi de développer également le simulateur sur cette même machine virtuelle. Cela nous a permis de regrouper tous nos outils au sein d'un écosystème unique.
- **Rationalisation de l'environnement :** En évitant la multiplication des environnements de travail, nous avons limité les risques d'incompatibilité logicielle et simplifié les phases de tests entre le simulateur et l'application d'acquisition.

- **Confort et productivité :** Le développement sur PC offre une puissance de calcul supérieure et une exécution plus rapide des outils de compilation. Cela nous a permis d'avancer efficacement tant que l'accès au matériel physique n'était pas indispensable.

- **Flexibilité :** Cette approche a facilité le travail en autonomie, permettant à chaque membre de l'équipe de progresser sur le code sans dépendre de la disponibilité immédiate des Raspberry Pi 5.
- **Stabilité et compatibilité :** En utilisant la même version de distribution (Debian 13) sur nos PC et sur la Raspberry Pi 5, nous garantissons la cohérence des dépôts et des versions de paquets. Cela réduit drastiquement les risques d'incompatibilité et les problèmes de dépendances lors du déploiement final.

Ainsi, cette homogénéité entre l'environnement de développement et de production sécurise le cycle de vie du projet, tout en optimisant le temps de réalisation.

1.6.1. Outils utilisés côté IR

1.6.1.1. Outils commun

Framagit :

Pour centraliser et sécuriser l'ensemble de nos développements, nous avons utilisé **Framagit**.

Il s'agit d'une **forge logicielle** qui intègre un système de gestion de version basé sur **Git**. Cet outil a été indispensable pour plusieurs raisons :

- **Suivi des modifications :** Git nous a permis de conserver un historique complet et daté de chaque modification du code, facilitant le débogage en cas de régression.
- **Collaboration efficace :** Grâce au système de branches, chaque membre de l'équipe (IR1 et IR2 et IR3) a pu développer ses fonctionnalités en parallèle, sans risquer d'écraser le travail des autres.

1.6.1.2. Outils côté holter

Raspberry Pi 5 :

Si le développement initial du **holter** a été réalisé sur PC pour plus de confort, le passage sur **Raspberry Pi** a été une étape indispensable pour intégrer la chaîne d'acquisition physique. En effet, c'est sur cette plateforme que nous avons pu interfacer le système avec l'**ESP32** et l'**ADC** (Convertisseur Analogique-Numérique).

Cette phase de transition a permis de valider plusieurs aspects critiques :

- **Acquisition et traitement réel :** La Raspberry Pi a servi de plateforme de réception pour les données envoyées par l'ESP32. Cela nous a permis de tester en conditions réelles la conversion des signaux via l'ADC et leur traitement immédiat par les modules IR1 et IR2.
- **Interfaçage matériel :** Contrairement au PC, la Raspberry Pi nous a offert l'environnement nécessaire pour gérer les communications et les protocoles d'échange avec les microcontrôleurs tels que **l'ESP 32** et le **HeltecCubecell,** garantissant ainsi la continuité entre la capture du signal ECG et son exploitation logicielle.

- **Exécution multi-profils :** Pour simuler un environnement de production sécurisé, nous avons fait fonctionner les programmes d'acquisition et de traitement simultanément sur deux profils utilisateurs différents. Nous avons ainsi pu vérifier que les flux de données circulent correctement entre les processus sans interférence.
- **Centralisation du projet :** Framagit a servi de dépôt unique et accessible à distance, garantissant que nous disposions toujours de la version stable la plus récente pour nos phases d'intégration sur la Raspberry Pi.

Python 3 :

Le langage Python a été retenu pour le développement du **holte**r en raison de notre expérience collective, mais également pour sa polyvalence et la richesse de son écosystème de bibliothèques, notamment pour le traitement de données et la communication réseau.

N'oublions pas PiP, celui-ci sert à l'installation des librairies Python dans un environnement virtuel.

1.6.1.3. Outils côté simulateur

Raspberry pico :

QtCreator :

Framework C++ ayant servi pour le développement du simulateur offrant un système d'IHM plus moderne et plus performant que Python.

1.6.2. Outils côté ER

Kicad :

KiCad est une suite logicielle de référence dans le domaine de l'électronique, utilisée pour concevoir des circuits imprimés (**PCB** - *Printed Circuit Board*) de manière professionnelle. Il s'agit d'un environnement de **CAO** (Conception Assistée par Ordinateur) spécialisé, permettant de passer d'une idée théorique à une carte électronique prête à être produite.

Le processus de conception au sein de KiCad s'articule autour de trois phases fondamentales :

1. La saisie du schéma électrique

Cette étape constitue le socle du projet. Il s'agit de modéliser le circuit théorique en utilisant des symboles normalisés. Vous puisez dans des bibliothèques de composants (telles que des microcontrôleurs type ESP32, des convertisseurs comme le MAX11100 ou des composants passifs) et vous établissez les connexions électriques entre eux pour définir le fonctionnement logique du système.

2. Le routage du PCB

Une fois le schéma validé, le logiciel permet de basculer vers l'environnement de routage. Ici, vous passez à la conception physique de la carte. Vous définissez l'implantation optimale des composants sur le support, puis vous réalisez le tracé des pistes en cuivre qui assurent la continuité électrique du

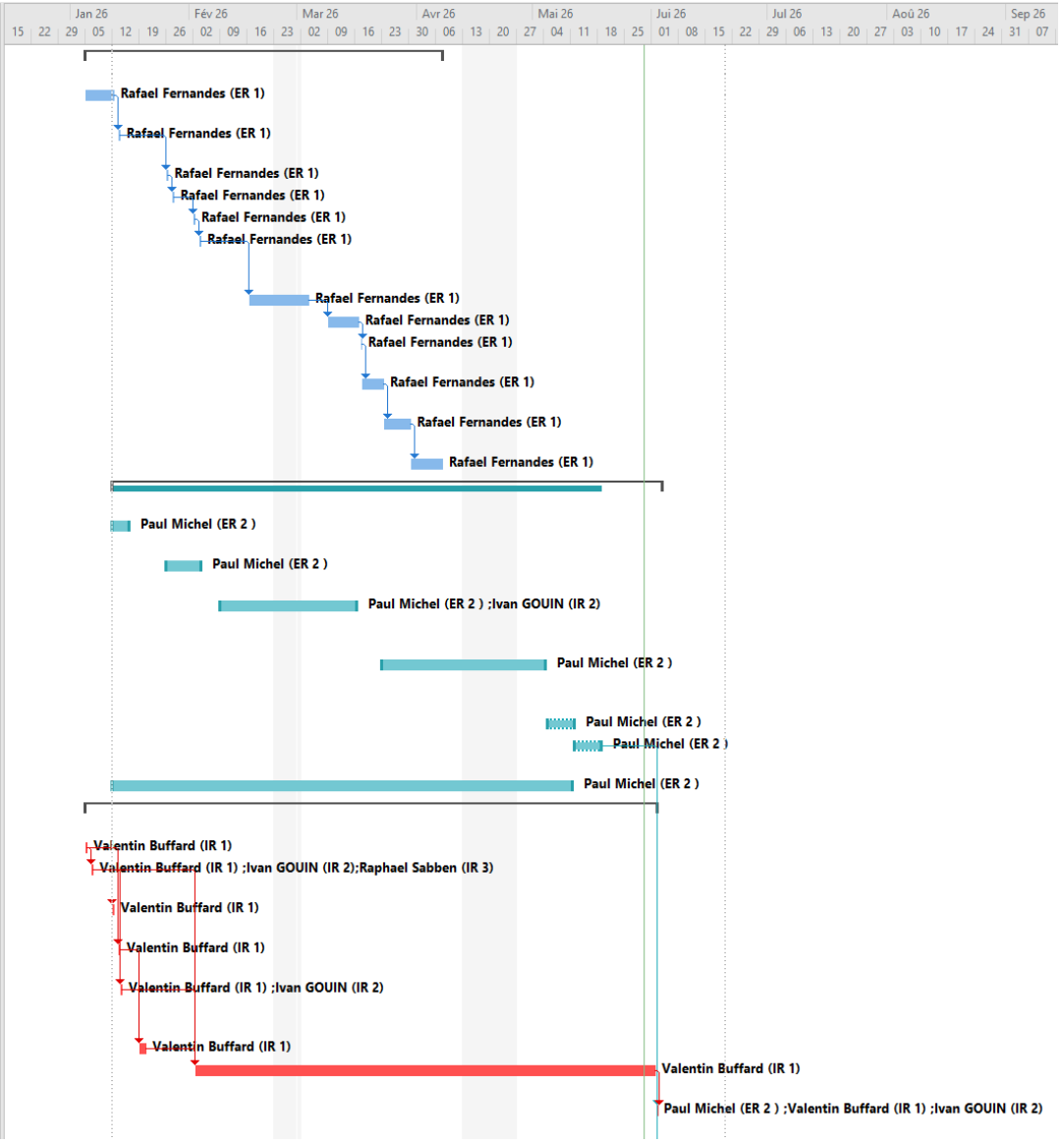
circuit. C’est une étape critique qui demande une attention particulière à la gestion des signaux et aux contraintes mécaniques.

3. La visualisation 3D et l'exportation

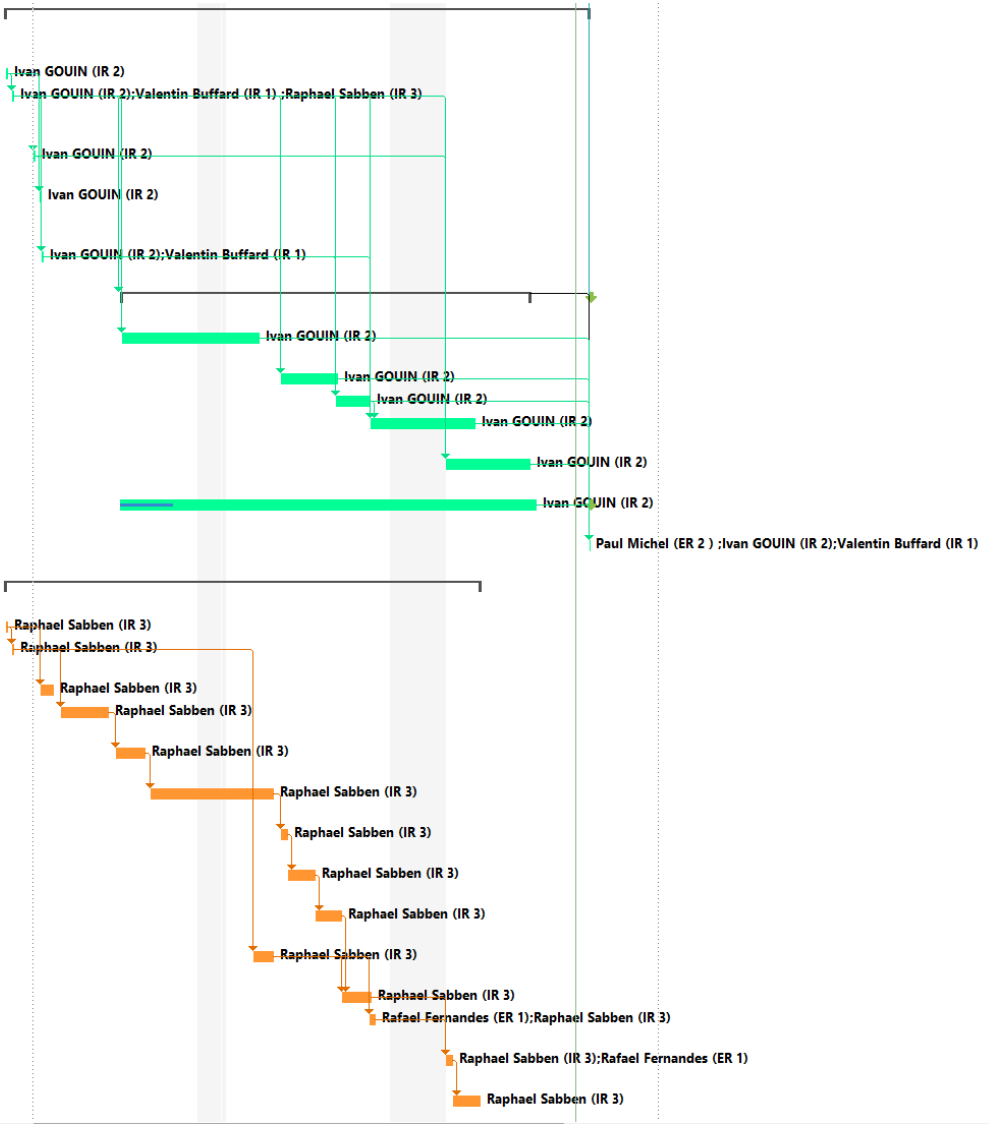
KiCad propose un outil de visualisation 3D haute fidélité. Cette fonctionnalité permet de vérifier l'encombrement physique et l'intégration mécanique de la carte au sein de son boîtier avant toute fabrication. Enfin, le logiciel génère les **fichiers Gerber**, des fichiers standardisés reconnus par l'industrie, qui contiennent toutes les instructions nécessaires aux usines pour la production et le perçage de vos circuits.

1.7. Planification Globale

	Mode Tâche	Nom de la tâche	Durée	Début	Fin
2		**Conception de la carte simulateur (Rafael Fernandes ER1)**	**107 hr**	**Lun 05/01/26**	**Mar 07/04/26**
3		Etude du dossier et des revues de projet de l'année dernière	9 hr	Lun 05/01/26	Lun 12/01/26
4		Mise en oeuvre de l'environement pour la Rasberi PiCo	4 hr	Mar 13/01/26	Mar 13/01/26
5		Test matériels & logiciels	4 hr	Lun 26/01/26	Lun 26/01/26
6		Redaction dossier	4 hr	Mar 27/01/26	Mar 27/01/26
7		Redaction dossier	4 hr	Lun 02/02/26	Lun 02/02/26
8		Préparation de l'oral du 10/03 (Revue de projet n°2) & modification du dossier	4 hr	Mar 03/02/26	Mar 03/02/26
9		Modification du schéma structurel	10 hr	Lun 16/02/26	Mar 03/03/26
10		Routage de la carte	12 hr	Lun 09/03/26	Lun 16/03/26
11		Rédaction de la liste des composants	2 hr	Mar 17/03/26	Mar 17/03/26
12		Dernières modifications à faire sur le routage et le schéma	4 hr	Mar 17/03/26	Lun 23/03/26
13		Brasage de la carte & rédaction du dossier	8 hr	Lun 23/03/26	Lun 30/03/26
14		Rédaction du projet	8 hr	Lun 30/03/26	Mar 07/04/26
15		**Conception de la carte holter (Paul Michel ER 2)**	**150 hr**	**Lun 12/01/26**	**Mer 03/06/26**
16		Prendre connaissance du sujet final de la version 2025	16 hr	Lun 12/01/26	Ven 16/01/26
17		Analyser/tester la carte holter ECG de la version finale 2025	16 hr	Lun 26/01/26	Mar 03/02/26
18		Proposé une nouvelle version du holter, avec base de Rasberry Pi, puis la validé avec étudiant IR2	30 hr	Lun 09/02/26	Lun 16/03/26
19		Routage de la nouvelle carte, et produire les fichers pour permettre la fabrication du PCB	30 hr	Lun 23/03/26	Lun 04/05/26
20		Câbler PCB et effectuer les essais	18 hr	Mar 05/05/26	Lun 11/05/26
21		Mise en service de la carte finalisé, ainsi que sa maintenance	18 hr	Mar 12/05/26	Lun 18/05/26
22		Rédaction du dossier	120 hr	Lun 12/01/26	Lun 11/05/26
23		**Développement du système d'acquisition (Valentin Buffard IR 1)**	**150 hr**	**Lun 05/01/26**	**Mar 02/06/26**
24		Découverte du projet	4 hr	Lun 05/01/26	Lun 05/01/26
25		Mise en place de l'environnement de développement	4 hr	Mar 06/01/26	Mar 06/01/26
26		Recherche d'une solution temps réel pour le traçage de la courbe ECG	4 hr	Lun 12/01/26	Lun 12/01/26
27		Recherches sur le format de la trame SPI de l'ADC MAX11100	4 hr	Mar 13/01/26	Mar 13/01/26
28		Mise en place de la stratégie de stockage et transmission des données	4 hr	Mer 14/01/26	Mer 14/01/26
29		Programmation et Test de l'ESP32	6 hr	Lun 19/01/26	Mar 20/01/26
30		Développement de l'app d'acquistion	101 hr	Lun 02/02/26	Lun 01/06/26
31		Rédaction d'une documentation d'utilisation du Holter	1 hr	Mar 02/06/26	Mar 02/06/26



32	Développement du système de traitement et supervision Holter (Ivan GOUIN-IR 2)	150 hr	Lun 05/01/26	Mar 02/06/26
33	Découverte du projet	4 hr	Lun 05/01/26	Lun 05/01/26
34	Mise en place de l'environnement de développement	4 hr	Mar 06/01/26	Mar 06/01/26
35	Recherches sur le fonctionnement de LoRa	4 hr	Lun 12/01/26	Lun 12/01/26
36	Recherches sur le format des trames hexadécimales du CubeCell HTCC-AB01	4 hr	Mar 13/01/26	Mar 13/01/26
37	Mise en place de la stratégie de stockage et échange des données	4 hr	Mer 14/01/26	Mer 14/01/26
38	**Développement de l'application de traitement.**	**84 hr**	**Mar 03/02/26**	**Lun 18/05/26**
39	Développement de l'application de calcul basique	28 hr	Mar 03/02/26	Mar 10/03/26
40	Passage en architecture MVC	20 hr	Lun 16/03/26	Lun 30/03/26
41	Intégration du protocole LoRa	10 hr	Lun 30/03/26	Mar 07/04/26
42	Développement Websocket / Synchronisation avec IR1	12 hr	Mar 07/04/26	Lun 04/05/26
43	Développement du tableau de bord Node-RED	25 hr	Lun 27/04/26	Lun 18/05/26
44	Tests intégration et tests chaine complète	104 hr	Mar 03/02/26	Mar 19/05/26
45	Rédaction d'une documentation d'utilisation du Holter	1 hr	Mar 02/06/26	Mar 02/06/26
46	Développement du Simulateur (Raphael Sabben IR 3)	123 hr	Lun 05/01/26	Mar 05/05/26
47	Découverte du projet	4 hr	Lun 05/01/26	Lun 05/01/26
48	Mise en place de l'environement de développement	4 hr	Mar 06/01/26	Mar 06/01/26
49	Constitution du référentiel CSV	15 hr	Mar 13/01/26	Ven 16/01/26
50	Développement IHM Qt — parsing CSV	16 hr	Lun 19/01/26	Ven 30/01/26
51	Développement IHM Qt — rendu graphique médical	12 hr	Lun 02/02/26	Lun 09/02/26
52	Développement IHM Qt — ComboBox scan dossier CSV	20 hr	Mar 10/02/26	Ven 13/03/26
53	Développement IHM Qt — boutons Start / Stop	7 hr	Lun 16/03/26	Mar 17/03/26
54	Conception du protocole de communication binaire	7 hr	Mar 17/03/26	Mar 24/03/26
55	Implémentation sérialisation Little-Endian	7 hr	Mar 24/03/26	Mar 31/03/26
56	Étude du firmware MicroPython PIO/DMA	8 hr	Lun 09/03/26	Ven 13/03/26
57	Tests du protocole Qt ↔ Pico	7 hr	Mar 31/03/26	Mar 07/04/26
58	Collaboration avec ER2 — validation câblage SPI	4 hr	Mar 07/04/26	Mer 08/04/26
59	Intégration physique du DAC8551 sur le Pico	7 hr	Lun 27/04/26	Mar 28/04/26
60	Validation du signal analogique à l'oscilloscope	7 hr	Mar 28/04/26	Mar 05/05/26



PARTIE PERSONNELLE
VALENTIN BUFFARD
(IR 1)

2. PARTIE PERSONNELLE - VALENTIN BUFFARD

2.1. Objectifs

Dans ce projet, je suis responsable de la chaîne d'acquisition et de l'interface utilisateur. Mon travail se décompose en trois étapes clés :

- **Acquisition des données :** Je récupère les signaux issus du simulateur pour garantir une source de données fiable et stable.
- **Transmission et stockage :** Une fois capturées, les données sont transmises à l'architecture globale pour être stockées en base de données et envoyées au module de traitement (IR2).
- **Visualisation temps réel :** J'ai développé une interface web qui affiche les signaux sous forme de courbes ECG. Cet outil permet de superviser l'activité cardiaque en direct et facilite l'interprétation des mesures.

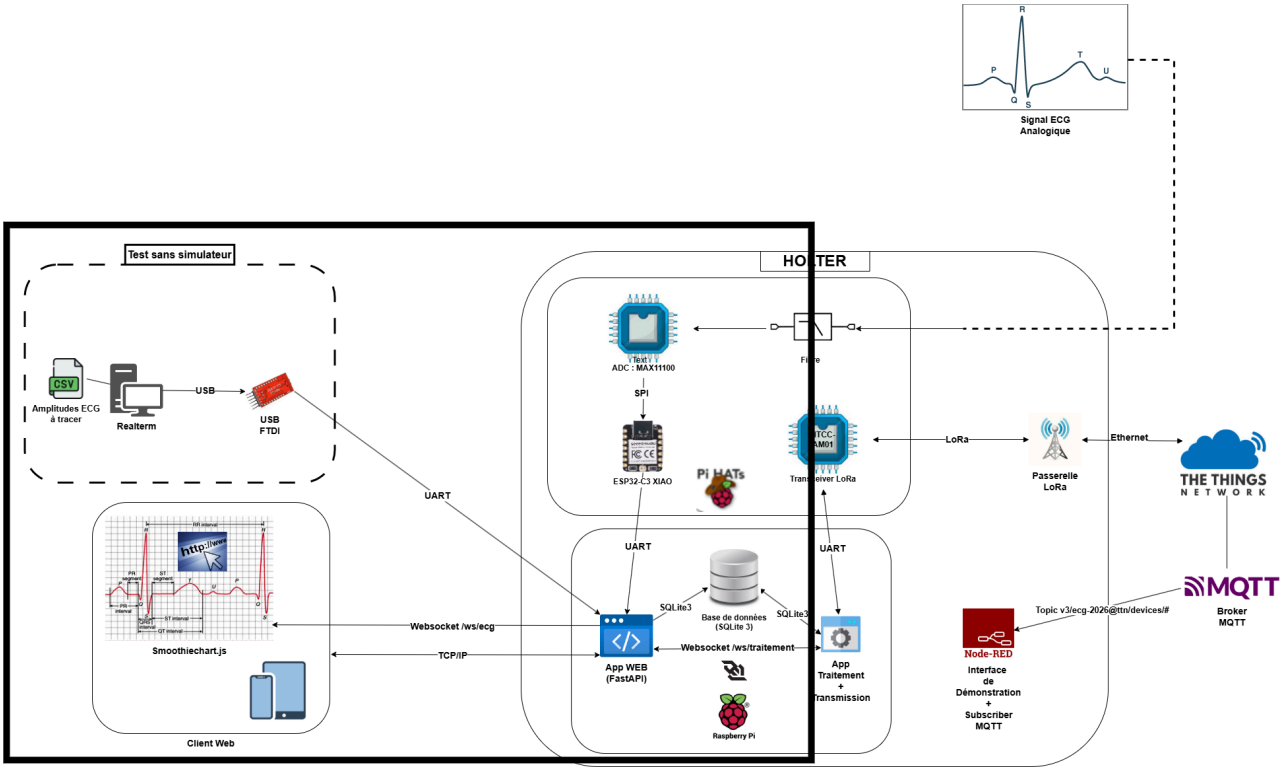


Illustration de la partie visualisation / acquisition

2.2. Planification prévisionnelle

Voici la planification prévisionnelle pour l'organisation du projet.

	Développement du système d'acquisition (Valentin Buffard IR 1)	143 h	Lun 05/01/26	Mer 03/06/26
	Découverte du projet	4 h	Lun 05/01/26	Lun 05/01/26
	Mise en place de l'environnement de développement	4 h	Mar 06/01/26	Mar 06/01/26
	Recherche d'une solution temps réel pour le traçage de la courbe ECG	4 h	Lun 12/01/26	Lun 12/01/26
	Recherches sur le format de la trame SPI de l'ADC MAX11100	4 h	Mar 13/01/26	Mar 13/01/26
	Mise en place de la stratégie de stockage et transmission des données	4 h	Mer 14/01/26	Mer 14/01/26
	Programmation et Test de l'ESP32	6 h	Lun 19/01/26	Mar 20/01/26
	Développement de l'app d'acquistion	101 h	Mar 27/01/26	Lun 01/06/26
	Rédaction d'une documentation d'utilisation du Holter	4 h	Mar 02/06/26	Mer 03/06/26

Lors de la conception de mon diagramme prévisionnel, j'ai structuré l'ordre des tâches pour anticiper les dépendances critiques du projet. Ma stratégie s'est articulée autour de quatre axes prioritaires :

- **Anticipation des risques système :** Ma priorité a été de planifier la mise en place de l'environnement de développement. Il était crucial de prévoir un cadre de travail commun afin d'éviter, dès le départ, tout problème de compatibilité entre mes bibliothèques et celles de mes collaborateurs IR.
- **Validation de la faisabilité graphique :** J'ai ensuite placé l'étude du tracé ECG en début de planning. L'objectif prévisionnel était de garantir que l'affichage en temps réel puisse être fluide avant de lancer le développement lourd de l'interface.
- **Conception du protocole d'échange :** La définition d'une structure de trame série a été programmée tôt dans le projet. Il fallait s'assurer que le pilotage du matériel (démarrage/arrêt) soit verrouillé pour permettre les tests d'acquisition.
- **Coordination inter-modules :** Enfin, j'ai prévu de définir le format de stockage des données en concertation avec Ivan GOUIN (IR2). Cette étape était indispensable pour que nos deux programmes puissent communiquer sans erreur dès leur mise en relation.

Objectif final : Cette organisation prévisionnelle visait à lever tous les verrous techniques majeurs. Une fois ces fondations sécurisées sur le papier, je pouvais sereinement envisager la phase de production logicielle.

2.3. Outils utilisés

Realterm : Logiciel utilisé pour faire des tests en envoyant des données sur la liaison série **UART** simulée pour faire des tests.

Adaptateur FTDI : Adaptateur USB anciennement utilisé pour émuler la liaison série **UART** dans le but de procéder à des tests sur **debian**.

ESP32 S3 XIAO : Micro-contrôleur responsable de l'acquisition à une fréquence fixe depuis l'**ADC Max-11100** et l'envoie des échantillons à la **rpi-5**.

ADC Max-11100 : Convertisseur Analogique numérique de **16 bits** permettant de convertir le signal analogique en numérique.

2.4. Environnement de développement python

Installation de **PIP** : Il s'agit d'un utilitaire, qui permet d'installer des librairies et dépendances python.

sudo apt install python3-pip

Installation de VENV : Une librairie qui permet de créer des environnements virtuels python.

sudo apt install python3.13-venv

2.4.1. Mise en place de l'environnement python

Afin d'installer et de gérer les librairies Python, il est désormais recommandé d'utiliser un **environnement virtuel**. Un environnement virtuel Python permet d'isoler les dépendances d'un projet et de définir précisément les librairies ainsi que leurs versions utilisées pour le développement.

Cette approche présente plusieurs avantages : elle évite les conflits entre différentes versions de librairies installées sur un même système et permet d'exporter facilement la configuration de l'environnement. Il devient ainsi plus simple de **partager le projet avec d'autres développeurs** ou de **déployer l'application dans un environnement de production** en garantissant la reproductibilité de l'environnement de développement.

1. Pour commencer il est nécessaire de créer le répertoire de notre environnement.

   mkdir nom_du_repertoire_environement

2. Puis taper la commande suivante :

   python3 -m venv nom_du_repertoire_environement

3. On peut ensuite activer l'environnement avec la commande suivante :

   source nom_du_repertoire_environement/bin/activate

4. Finalement on peut désactiver l'environnement avec la commande suivante : deactivate

2.4.2. Librairies utilisées

Pyserial : Une librairie qui permet d'exploiter une ou plusieurs liaisons série dans le code Python.

SQLite3 : Permet de superviser et créer des bases de données SQLite via le code python en utilisant des requêtes.

Threading : Ajout de fonctions permettant de faire tourner plusieurs processus dans un même programme.

Asyncio : Possibilité de faire du code asynchrone dans Python .

2.4.3. Test de Dash & de FastAPI

Dans le but de visualiser le tracé de l'ECG, deux approches ont été étudiées : l'utilisation de Dash et celle de FastAPI.

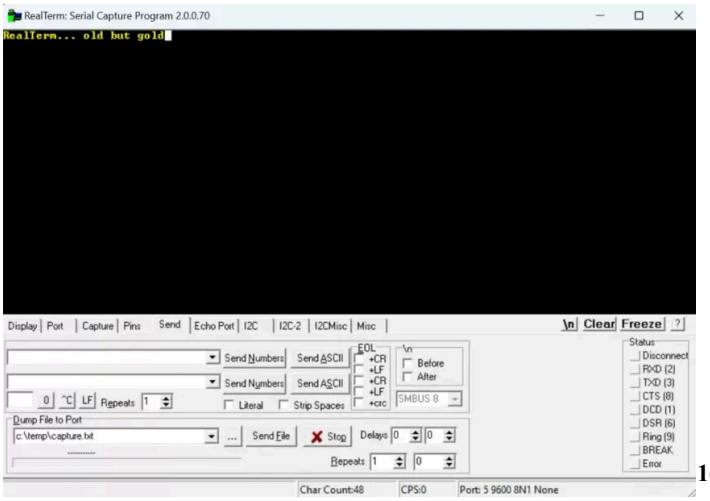
Dans notre cas, l'objectif était de recevoir **point par point les amplitudes d'un signal ECG au format CSV via la liaison série**, puis de tracer la courbe progressivement afin de visualiser l'évolution du signal en temps réel.

Pour réaliser les tests de communication série, le logiciel **RealTerm** a été utilisé. Dans un premier temps, la vitesse de transmission des données a été configurée à **115200 bauds**.

Cette valeur est couramment utilisée dans les communications série car elle offre un **bon compromis entre vitesse de transmission et fiabilité**. Avec un débit de 115200 bauds, il est possible de transmettre un grand nombre de données par seconde, ce qui est particulièrement adapté à notre application où les **valeurs d'amplitude de l'ECG doivent être envoyées rapidement et de manière continue** afin de permettre leur affichage en temps réel. De plus, cette vitesse est **largement supportée par la majorité des microcontrôleurs et interfaces USB-série**, ce qui facilite les tests et l'intégration.

Afin de simuler la liaison **UART**, un **adaptateur USB-série basé sur le composant CH340** a été utilisé. Celui-ci permet de convertir la communication USB de l'ordinateur en communication série UART, rendant possible l'envoi et la réception de données via RealTerm comme si elles provenaient directement d'un microcontrôleur.



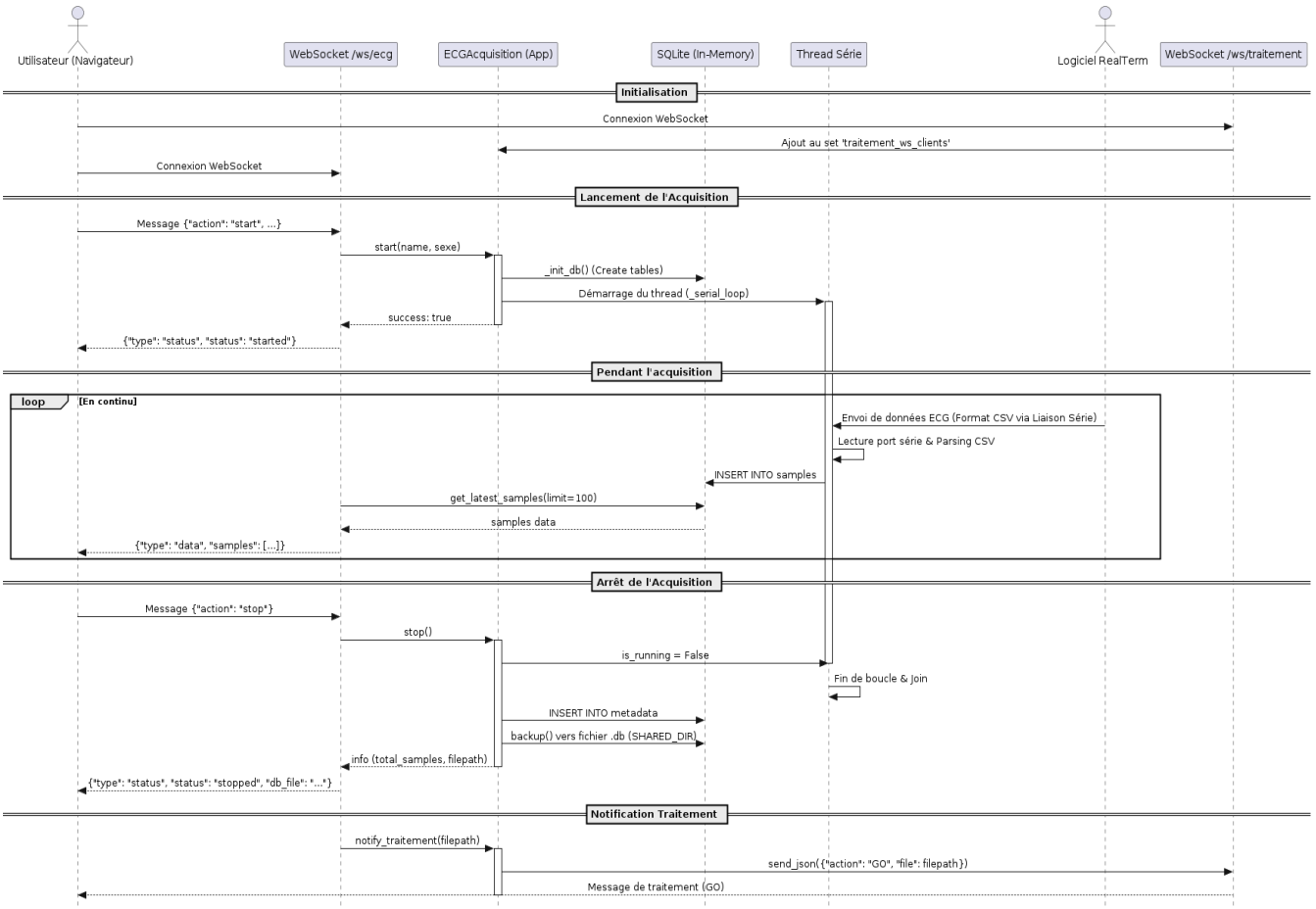


Finalement, Dash s'est révélée inadaptée à nos besoins en temps réel. En effet, notre objectif était de transmettre les données point par point au format CSV via la liaison série afin de tracer la courbe de l'ECG en direct, ce que la solution ne permettait pas efficacement.

2.4.3.1. FastAPI

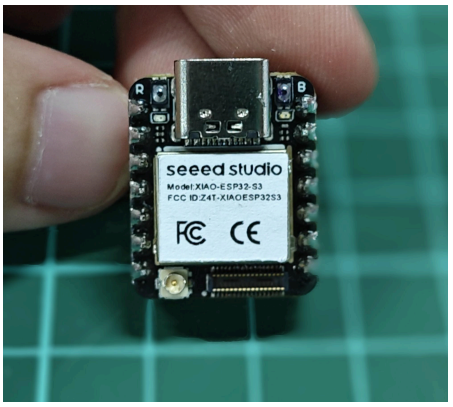
FastAPI est un framework Python moderne permettant de créer des API web rapides et performantes. Contrairement à **Dash**, FastAPI facilite la gestion de données en **temps réel** grâce aux **WebSockets**, qui permettent de transmettre les points de données de l'ECG instantanément. Pour l'affichage graphique, nous utilisons une librairie JavaScript telle que **Smoothie Chart**, offrant un tracé fluide et dynamique des courbes, ce qui rend l'ensemble plus réactif.

Voici un diagramme de séquence qui explique comment fonctionne mon application de test avec **FastAPI** et **Smoothie Chart**.



2.5. Programmation de l’ESP32

En parallèle du développement de l’application web, j’ai également pris en charge la **programmation de l’ESP32**, un composant central du système de holter ECG. Ce microcontrôleur joue un rôle clé dans la chaîne d’acquisition des données.



En effet, l’ESP32 permet de **piloter le convertisseur analogique-numérique (ADC) ADCMax11100EUB+**, chargé de convertir le signal analogique provenant du simulateur en un **signal numérique exploitable**. Les données ainsi acquises sont ensuite transmises au reste du système afin d’être **affichées sur l’application web, puis traitées**.

Cette étape est cruciale : elle garantit une fréquence d'acquisition fixe de **360 Hz**. Ce cadencement rigoureux est indispensable pour rester en parfaite conformité avec les spécifications du simulateur conçu et développé par mes collaborateurs, **Rafael Fernandes (ER 1) et Raphael Sabben (IR2).**

Pour cela, j’ai utilisé **Arduino IDE**, un environnement de développement intégré dédié à la **programmation de l’ESP32**, permettant l’écriture du code source, sa compilation ainsi que son téléversement vers le microcontrôleur.

2.5.1. Architecture Globale

Le programme repose sur un modèle **Producteur/Consommateur** réparti sur les deux cœurs de l'ESP32. L'objectif est d'assurer une fréquence d'échantillonnage de **360 Hz** (période de 2780µS) de manière stable, sans que l'envoi des données sur le port série ne vienne perturber la capture.

Les composants clés :

- **Capteur :** ADC externe **MAX11100** (16 bits) piloté en SPI.
- **Communication :** Liaison série (UART) vers un Raspberry Pi.
- **Synchronisation :** Une file d'attente (**Queue**) FreeRTOS.

2.5.2. FreeRTOS

FreeRTOS (*Real-Time Operating System*) est un noyau de système d'exploitation open-source conçu pour les systèmes embarqués. Contrairement à un programme Arduino classique qui s'exécute en boucle fermée (`void loop`), FreeRTOS permet de découper le code en plusieurs unités indépendantes appelées **Tâches** (Tasks).

Les caractéristiques clés :

- **Multitâche (Multitasking) :** Capacité à exécuter plusieurs fonctions en parallèle (ou par alternance rapide).
- **Ordonnancement (Scheduling) :** Un algorithme décide quelle tâche doit s'exécuter en fonction de sa priorité.
- **Déterminisme :** Il garantit que les tâches critiques (comme l'acquisition SPI) s'exécutent à des intervalles de temps prévisibles.

L'ESP32 possède deux cœurs (Dual-core). FreeRTOS est utilisé par le framework de l'ESP32 pour gérer cette puissance :

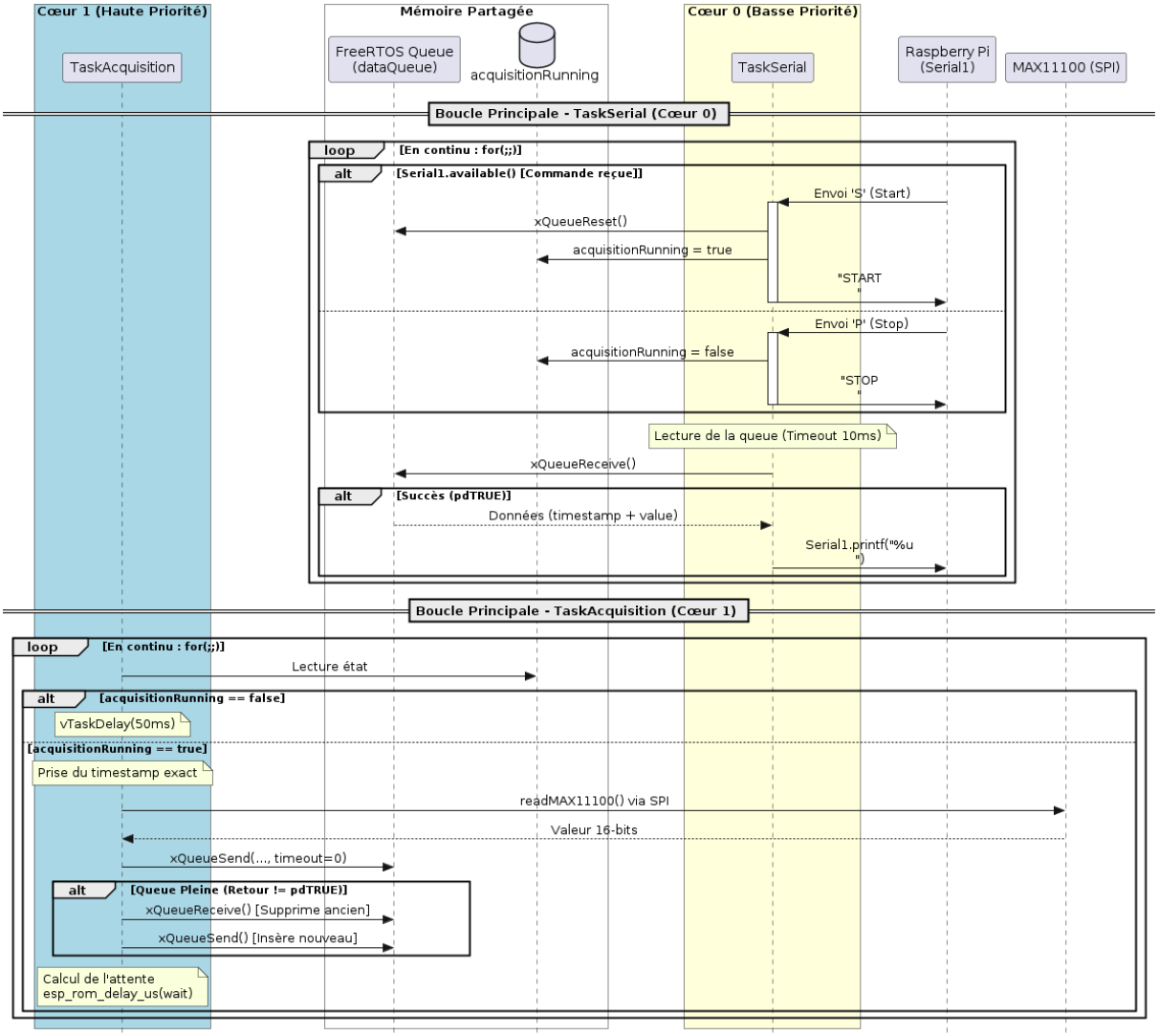
- **Répartition de charge :** Il est possible de dédier le **Cœur 1** aux tâches de calcul/acquisition et le **Cœur 0** à la communication (Wi-Fi, Bluetooth, Série…).
- **Gestion des priorités :** Une tâche importante (Acquisition) peut interrompre une tâche moins urgente..

L'utilisation de **FreeRTOS** est indispensable dans le projet pour garantir le **déterminisme** de l'acquisition. Dans un programme classique, l'envoi de données sur le port série (opération lente) bloquerait l'exécution du code, ce qui rendrait impossible le maintien d'une fréquence d'échantillonnage stable de **360 Hz**.

En utilisant FreeRTOS, j'isole l'acquisition sur le **cœur 1** (priorité haute) et la communication sur le **cœur 0** (priorité basse).

Cette architecture multitâche, couplée à une **Queue** (file d'attente), permet aux deux processus de travailler de manière asynchrone : l'ESP32 continue de capturer chaque échantillon à l'intervalle précis de **2780 µs**, sans jamais être ralenti par le flux de données envoyé vers la Raspberry Pi.

2.5.3. Fonctionnement du programme



2.5.3.1. TaskAcquisition (Cœur 1 - Haute Priorité)

C'est le "Producteur". Elle tourne sur le cœur 1 pour être isolée des interruptions Wi-Fi/Bluetooth (souvent sur le cœur 0).

- **Précision temporelle :** Elle utilise `esp_timer_get_time()` pour un calcul de délai en microsecondes. Plutôt que d'utiliser un simple `delay()`, elle calcule le temps restant avant la prochaine échéance (`nextTime`) pour compenser le temps d'exécution du code lui-même.
- **Lecture SPI :** Elle récupère la valeur 16 bits du MAX11100 via la fonction `readMAX11100()`.
- **Gestion de la Queue :** Si la file est pleine, le programme supprime l'échantillon le plus ancien (`xQueueReceive`) pour insérer le nouveau. Cela garantit que les données transmises sont toujours les plus récentes.

2.5.3.2. TaskSerial (Cœur 0 - Basse Priorité)

C'est le "Consommateur".

- **Contrôle :** Elle écoute les commandes 'S' (Start) et 'P' (Stop) venant du Raspberry Pi pour basculer la variable `acquisitionRunning`.
- **Transmission :** Elle récupère les données de la Queue et les envoie via `Serial1.printf`.

2.5.3.3. Communication SPI avec le MAX11100

Le programme utilise l'instance matérielle **SPI** de l'ESP32-S3, un contrôleur dédié qui permet de communiquer avec l'ADC MAX11100 sans ralentir le reste du processeur.

Pour que l'échange soit fiable, on définit un objet `SPISettings` qui fixe les paramètres de configuration de liaison SPI : une vitesse de **4 MHz** pour la stabilité sur plaque d'essai, l'ordre des bits (**MSB**) et le **Mode 0** (polarité et phase de l'horloge) pour être en parfaite phase avec la datasheet du capteur.

L'intérêt majeur ici est l'utilisation de **transactions SPI** (`beginTransaction`) : cela verrouille la configuration du bus pendant toute la lecture des 16 bits.

C'est une sécurité indispensable dans un environnement multitâche comme FreeRTOS, car cela garantit que la tâche d'acquisition dispose d'un accès exclusif et stable au capteur, évitant ainsi toute corruption des mesures par une autre partie du système.

2.5.3.4. Organisation de la Mémoire et Flux

Les données transitent via une structure personnalisée :

```
// =====================

// CONFIG

// =====================

const int QUEUE_SIZE = 1000;

tile bool acvolaquisitionRunning = false;

QueueHandle_t dataQueue;

// =====================

// STRUCT DATA

struct AdcData

{

    uint16_t value;       // valeur ADC

};
```

Élément	Rôle / Justification technique
Structure AdcData	Encapsule la mesure brute sur 16 bits renvoyée par le convertisseur analogique-numérique (MAX11100) après la reconstruction des octets reçus sur le bus SPI.
Queue (dataQueue**)**	Tampon (*buffer*) configuré pour stocker jusqu'à **1000 structures** AdcData. Définie en mémoire partagée, elle permet d'absorber les variations de débit si la liaison série ralentit temporairement par rapport au rythme strict de l'acquisition (360 Hz).
acquisitionRunning	**Drapeau logique (*flag*) global déclaré comme** volatile. Il sert d'indicateur d'état pour ordonner le démarrage ou la suspension de l'acquisition entre les tâches. Le mot-clé volatile force le compilateur à lire la variable directement en mémoire RAM (et non dans les registres locaux d'un cœur), garantissant que le Cœur 1 (Acquisition) voit instantanément la modification physique effectuée par le Cœur 0 (Série).

2.5.3.5. Mapping des broches

```
// ======================

// PINS

// ======================

const int PIN_CS   = D0;

const int PIN_SCK  = D8;

const int PIN_MISO = D9;

const int PIN_MOSI = D10;



const int RX_PIN = D7;

const int TX_PIN = D6;
```

La configuration des Entrées/Sorties (GPIO) n'est pas arbitraire ; elle répond à une double contrainte matérielle et documentaire.

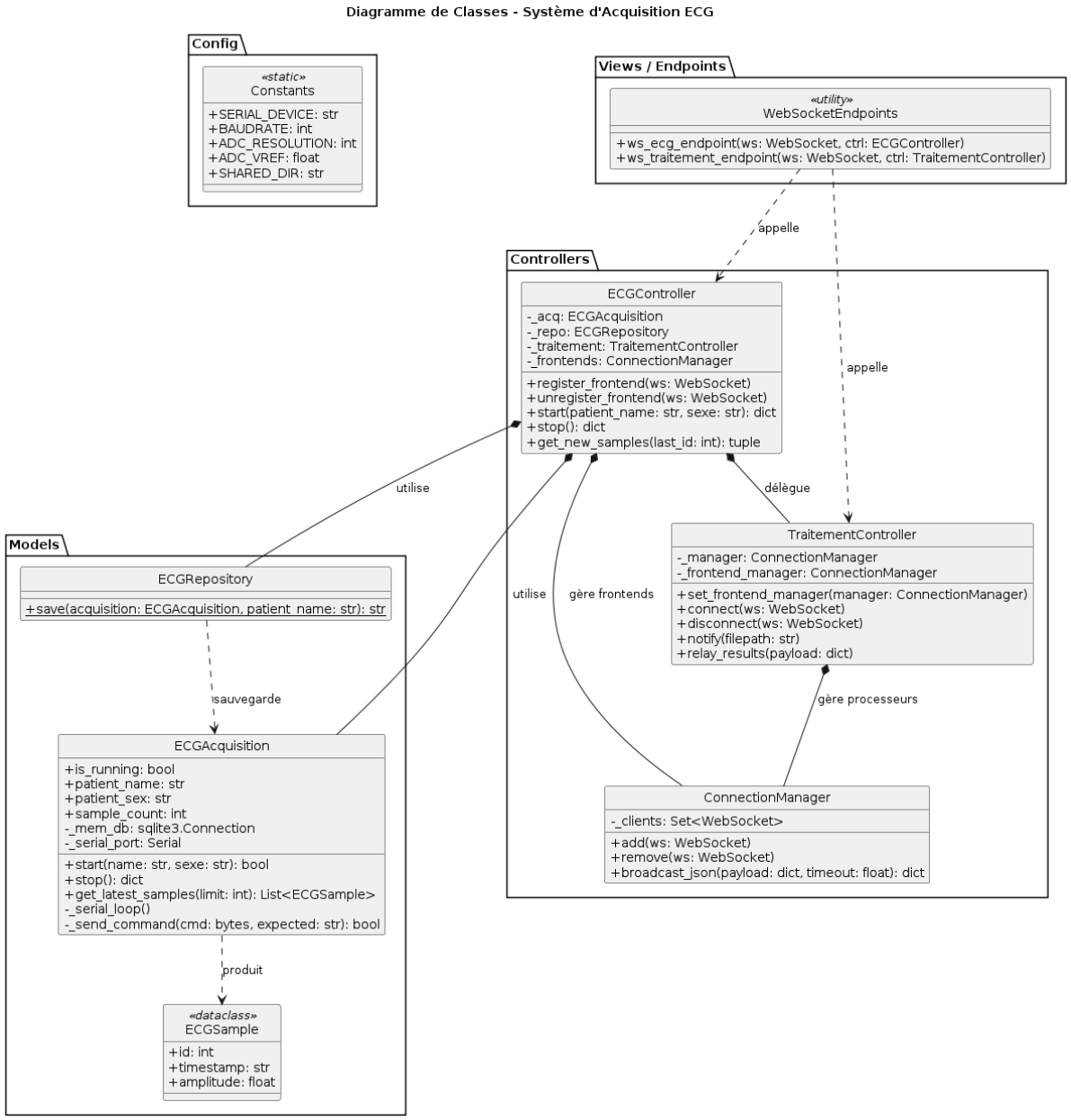
- **Conformité au schéma de Paul Michel (ER 1) :** Le brochage (PIN_CS, PIN_SCK, RX, TX) a été rigoureusement choisi pour correspondre au schéma structurel établi par **Paul Michel**. Cela assure l'interopérabilité entre la carte fille (shield/PCB) et le microcontrôleur, garantissant que les signaux SPI et UART arrivent sur les bonnes pistes physiques.
- **Compatibilité ESP32-S3 XIAO (Seeed Studio) :** L'utilisation des constantes type `D0`, `D8`, `D9` fait référence au nommage spécifique de la carte **Seeed Studio XIAO ESP32-S3**.
    - Le code utilise le bus **FSPI** (Fast SPI), qui est le bus périphérique par défaut de l'ESP32-S3 permettant d'atteindre des vitesses d'horloge élevées (ici 4 MHz) pour l'ADC.
    - Les pins `D7` (RX) et `D6` (TX) sont les broches standards de l'UART secondaire du format XIAO, optimisant le routage vers le Raspberry Pi.

2.6. Développement de l’application web

Pour éviter d'aboutir à un code désorganisé et difficile à maintenir, j'ai choisi de structurer l'application autour du design pattern **MVC (Modèle-Vue-Contrôleur)**. L'idée centrale est de bien séparer les responsabilités au sein du programme :

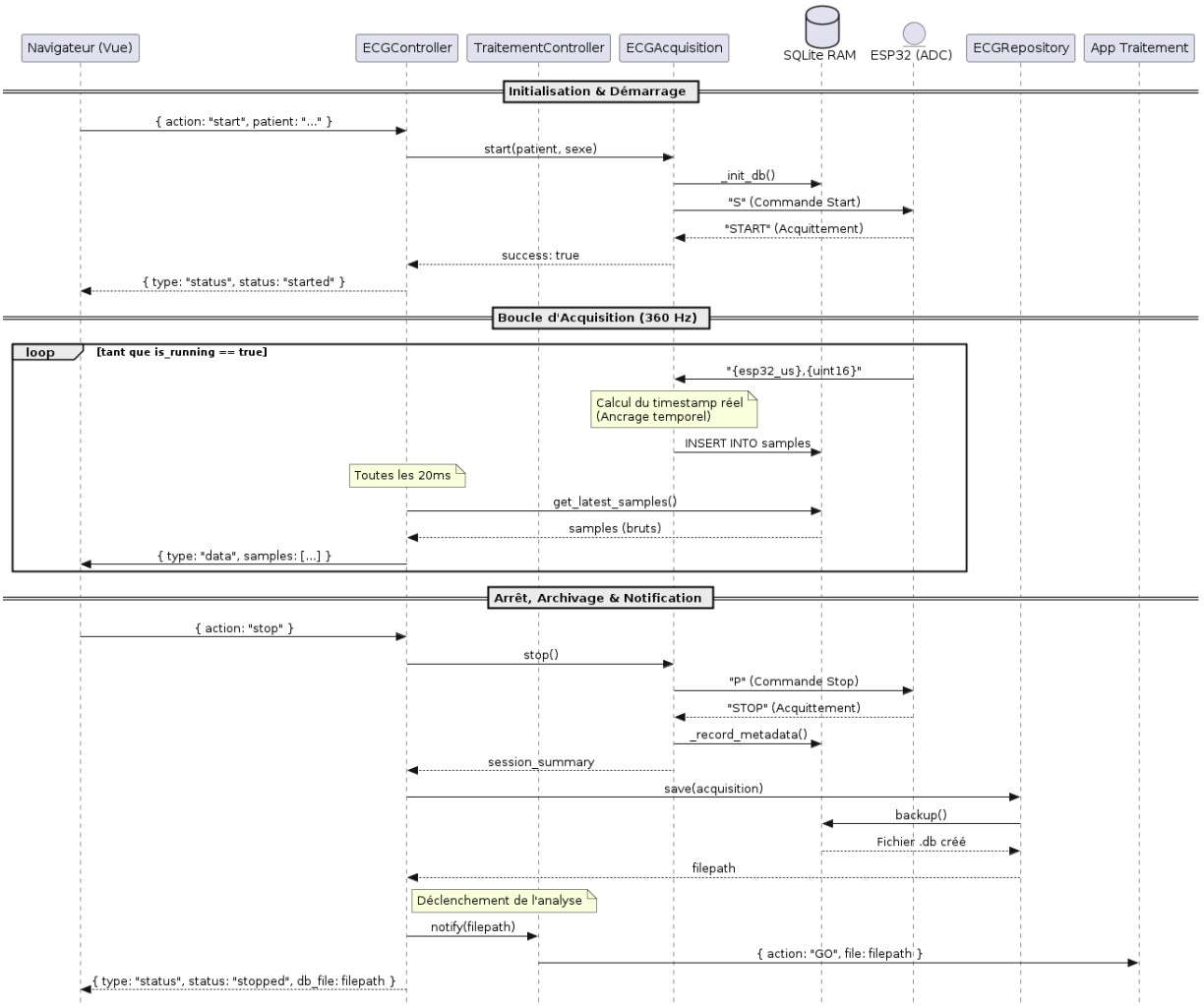
- **La logique métier (Modèle)** est isolée de l'**affichage (Vue)**.
- **Le contrôleur** fait le lien entre les deux en gérant les interactions utilisateur.

Cette approche par "séparation des préoccupations" rend le code beaucoup plus modulable. Si je souhaite modifier l'interface graphique sans toucher à la gestion des données, ou inversement, cela se fait sans risquer de casser l'ensemble du système.



2.6.1. Fonctionnement de l’application

Voici le un diagramme de séquence qui résume le fonctionnement du programme dans le temps.



2.6.1.1. L'Architecture MVC et l'Injection de Dépendances

Le fichier `main.py` joue le rôle d'assembleur. Il instancie les composants dans un ordre précis pour qu'ils puissent communiquer entre eux :

- **Le Modèle** (`ECGAcquisition`) gère le matériel (ESP32 via le port série).
- **Le Contrôleur** (`ECGController`) fait le pont entre les données et l'interface utilisateur.
- **La Vue** (`websockets.py`) gère les échanges en temps réel avec le navigateur Web.

```python
import os
from fastapi import FastAPI, WebSocket
from fastapi.responses import FileResponse

from config import BASE_DIR
from models.ecg import ECGAcquisition, ECGRepository
from controllers.ecg_controller import ECGController, TraitementController
from views.websockets import ws_ecg_endpoint, ws_traitement_endpoint

# ---------------------------------------------------------------------------
# Composition des dépendances (Dependency Injection manuelle)
# ---------------------------------------------------------------------------

acquisition = ECGAcquisition()
repository = ECGRepository()
traitement_ctrl = TraitementController()
ecg_ctrl = ECGController(acquisition, repository, traitement_ctrl)

# ---------------------------------------------------------------------------
# Application FastAPI
# ---------------------------------------------------------------------------

app = FastAPI(title="ECG Acquisition API")


@app.get("/")
def serve_index() -> FileResponse:
    return FileResponse(os.path.join(BASE_DIR, "index.html"))


@app.websocket("/ws/ecg")
async def websocket_ecg(ws: WebSocket) -> None:
    await ws_ecg_endpoint(ws, ecg_ctrl)


@app.websocket("/ws/traitement")
async def websocket_traitement(ws: WebSocket) -> None:
    await ws_traitement_endpoint(ws, traitement_ctrl)


# ---------------------------------------------------------------------------
# Lancement
# ---------------------------------------------------------------------------

if __name__ == "__main__":
    import uvicorn
    uvicorn.run(app, host="0.0.0.0", port=8000)
```

2.6.1.2. L'Acquisition de Données (Couche Matérielle)

C'est le cœur du système situé dans `ecg.py`. Le fonctionnement repose sur un **Thread dédié** (`_serial_loop`) pour ne pas bloquer l'application pendant la lecture série :

```python
def _serial_loop(self) -> None:
    """Lit les lignes de données de l'ESP32 et les insère en DB.

    Format attendu : '{esp32_us},{uint16}\\n'
      - esp32_us : timestamp µs de l'ESP32 (uint32, issu de esp_timer_get_time)
      - uint16   : valeur brute ADC 16 bits

    Les timestamps ESP32 sont des µs relatifs au boot. On les convertit
    en timestamps absolus ISO en prenant le timestamp Python à la réception
    du premier échantillon comme référence d'ancrage.

    Les messages de contrôle résiduels (READY, START, STOP) sont ignorés.
    """
    cursor = self._mem_db.cursor()
    # Référence temporelle : on ancre le premier timestamp ESP32 sur l'heure réelle
    t0_esp_us: int | None = None    # timestamp µs ESP32 du premier échantillon
    t0_real_s: float | None = None  # timestamp Unix (s) correspondant
```

Protocole Série : Le programme envoie des commandes binaires (`S` pour Start, `P` pour Stop) à l'ESP32 et attend des acquittements textuels (`START`, `STOP`).

```python
def start(self, name: str, sexe: str) -> bool:
    """Ouvre le port série, envoie 'S' à l'ESP32, démarre la lecture.

    Retourne False si :
      - une acquisition est déjà en cours
      - le port série ne peut pas être ouvert
      - l'ESP32 ne répond pas 'START' dans le délai imparti
    """
    if self.is_running:
        logger.warning("[ECGAcquisition] Acquisition déjà en cours.")
        return False

    self.patient_name = name or "Anonyme"
    self.patient_sex  = sexe or "Non défini"
    self.sample_count = 0

    self._init_db()

    # Ouverture du port
    try:
        self._serial_port = serial.Serial(
            SERIAL_DEVICE, BAUDRATE, timeout=SERIAL_TIMEOUT
        )
    except serial.SerialException as exc:
        logger.error("[ECGAcquisition] Impossible d'ouvrir %s : %s", SERIAL_DEVICE, exc)
        return False

    # Envoi de la commande START à l'ESP32
    if not self._send_command(ESP32_CMD_START, expected=ESP32_MSG_START):
        self._serial_port.close()
        logger.error("Acquitement start non reçu")
        return False

    self.is_running = True
    self._thread = threading.Thread(target=self._serial_loop, daemon=True)
    self._thread.start()
    logger.info("[ECGAcquisition] Acquisition démarrée pour '%s'.", self.patient_name)
    return True
```

Envoi de la commande start

```python
def stop(self) -> dict | None:
    """Envoie 'P' à l'ESP32, arrête la lecture, retourne le résumé de session."""
    if not self.is_running:
        return None

    # Demande d'arrêt à l'ESP32 avant de couper la boucle
    if self._serial_port and self._serial_port.is_open:
        self._send_command(ESP32_CMD_STOP, expected=ESP32_MSG_STOP)

    self.is_running = False
    if self._thread:
        self._thread.join(timeout=3)

    if self._serial_port and self._serial_port.is_open:
        self._serial_port.close()

    end_time = datetime.now().isoformat()
    self._record_metadata(end_time)

    logger.info(
        "[ECGAcquisition] Arrêt — %d échantillons collectés.", self.sample_count
    )
    return {
        "patient_name": self.patient_name,
        "sexe":         self.patient_sex,
        "end_time":     end_time,
        "total_samples": self.sample_count,
    }
```

Envoi de la commande stop

Conversion Analogique-Numérique : Les données reçues sont des entiers de **16 bits (uint16)**. Le programme traite et stocke exclusivement la valeur numérique brute (comprise entre 0 et 65535) afin de préserver l'intégrité totale du signal capturé sans aucune perte de précision liée à des calculs intermédiaires.

```python
            esp_us    = int(parts[0])
            raw_value = int(parts[1])
            amplitude = float(raw_value)  # valeur brute uint16

            # Ancrage : premier échantillon = heure système courante
            if t0_esp_us is None:
                import time as _time
                t0_esp_us  = esp_us
                t0_real_s  = _time.time()

            # Reconstruction du timestamp absolu depuis le timestamp ESP32
            elapsed_s  = (esp_us - t0_esp_us) / 1_000_000
            ts_unix    = t0_real_s + elapsed_s
            timestamp  = datetime.fromtimestamp(ts_unix).isoformat()

            ts_unix_ms = ts_unix * 1000.0
            cursor.execute(
                "INSERT INTO samples (timestamp, ts_unix_ms, amplitude) VALUES (?, ?, ?)",
                (timestamp, ts_unix_ms, amplitude),
            )
            self._mem_db.commit()
            self.sample_count += 1

        except ValueError:
            logger.debug("[ECGAcquisition] Ligne non parseable ignorée : '%s'", line)
        except serial.SerialException as exc:
            logger.error("[ECGAcquisition] Erreur série : %s", exc)
            break
        except Exception as exc:
            logger.error("[ECGAcquisition] Erreur inattendue : %s", exc)
```

Stockage Tampon : Les échantillons sont d'abord insérés dans une base de données **SQLite en mémoire** (`:memory:`) pour garantir une écriture très rapide à 360 Hz.

```
def _init_db(self) -> None:
    self._mem_db = sqlite3.connect(":memory:", check_same_thread=False)
    self._mem_db.executescript("""
        CREATE TABLE metadata (
            id            INTEGER PRIMARY KEY AUTOINCREMENT,
            patient_name  TEXT,
            sexe          TEXT,
            end_time      TEXT,
            total_samples INTEGER
        );
        CREATE TABLE samples (
            id         INTEGER PRIMARY KEY AUTOINCREMENT,
            timestamp  TEXT,
            ts_unix_ms REAL,
            amplitude  REAL
        );
    """)
    self._mem_db.commit()

def _record_metadata(self, end_time: str) -> None:
    if not self._mem_db:
        return
    self._mem_db.execute(
        "INSERT INTO metadata (patient_name, sexe, end_time, total_samples) VALUES (?,?,?,?)",
        (self.patient_name, self.patient_sex, end_time, self.sample_count),
    )
    self._mem_db.commit()
```

2.6.1.3. La Gestion des Communications Temps Réel (WebSockets)

Le fichier `websockets.py` utilise la bibliothèque FastAPI pour maintenir des connexions persistantes :

- **Bi-directionnalité** : Le serveur écoute les commandes (JSON) du client pour démarrer/arrêter l'acquisition tout en "poussant" (push) les nouveaux échantillons d'ECG toutes les 20 ms (`WS_SEND_INTERVAL`).

```
async def sender() -> None:
    """Pousse les nouveaux échantillons au client à intervalle régulier."""
    nonlocal last_id
    while True:
        try:
            if controller.is_running:
                samples, last_id = controller.get_new_samples(last_id)
                if samples:
                    await ws.send_json({"type": "data", "samples": samples})

            await asyncio.sleep(WS_SEND_INTERVAL)

        except Exception as exc:
            logger.error("[WS ECG sender] %s", exc)
            break

try:
    await asyncio.gather(receiver(), sender())
except WebSocketDisconnect:
    logger.info("[WS ECG] Client déconnecté.")
finally:
    controller.unregister_frontend(ws)
```

- **Gestion des clients** : La classe `ConnectionManager` dans `ecg_controller.py` maintient une liste des clients actifs et gère les déconnexions pour éviter d'envoyer des données dans le vide.

```python
class ConnectionManager:
    """Maintient un ensemble de WebSockets et gère les déconnexions silencieuses."""

    def __init__(self) -> None:
        self._clients: set[WebSocket] = set()

    def add(self, ws: WebSocket) -> None:
        self._clients.add(ws)

    def remove(self, ws: WebSocket) -> None:
        self._clients.discard(ws)

    async def _send_one(self, ws: WebSocket, payload: dict, timeout: float) -> bool:
        try:
            await asyncio.wait_for(ws.send_json(payload), timeout=timeout)
            return True
        except asyncio.TimeoutError:
            logger.warning("[ConnectionManager] Timeout (%ss) — client retiré.", timeout)
            return False
        except Exception as exc:
            logger.warning("[ConnectionManager] Erreur envoi : %s — client retiré.", exc)
            return False

    async def broadcast_json(self, payload: dict, timeout: float = 2.0) -> dict:
        """Envoie payload à tous les clients en parallèle avec timeout individuel."""
        if not self._clients:
            return {"ok": 0, "failed": 0}

        clients = list(self._clients)
        results = await asyncio.gather(
            *[self._send_one(ws, payload, timeout) for ws in clients],
            return_exceptions=False,
        )
        dead = [ws for ws, ok in zip(clients, results) if not ok]
        for ws in dead:
            self.remove(ws)

        return {"ok": sum(results), "failed": len(dead)}

    def __len__(self) -> int:
        return len(self._clients)
```

2.6.1.4. Persistance et Traitement des Données

Une fois l'acquisition terminée, un cycle de sauvegarde et d'analyse se déclenche :

- **Archivage** : Le `ECGRepository` effectue un backup de la base de données mémoire vers un fichier `.db` physique sur le disque dans le répertoire `/ecg-data`.

```python
class ECGRepository:
    """Sauvegarde une session en mémoire vers un fichier SQLite sur disque."""

    @staticmethod
    def save(acquisition: ECGAcquisition, patient_name: str) -> str:
        safe_name = "".join(
            c if c.isalnum() or c in ("-", "_") else "_" for c in patient_name
        )
        filename = f"{safe_name}_{datetime.now().strftime('%Y%m%d_%H%M%S')}.db"
        filepath  = f"{SHARED_DIR}/{filename}"

        file_conn = sqlite3.connect(filepath)
        acquisition.db_connection.backup(file_conn)
        file_conn.close()

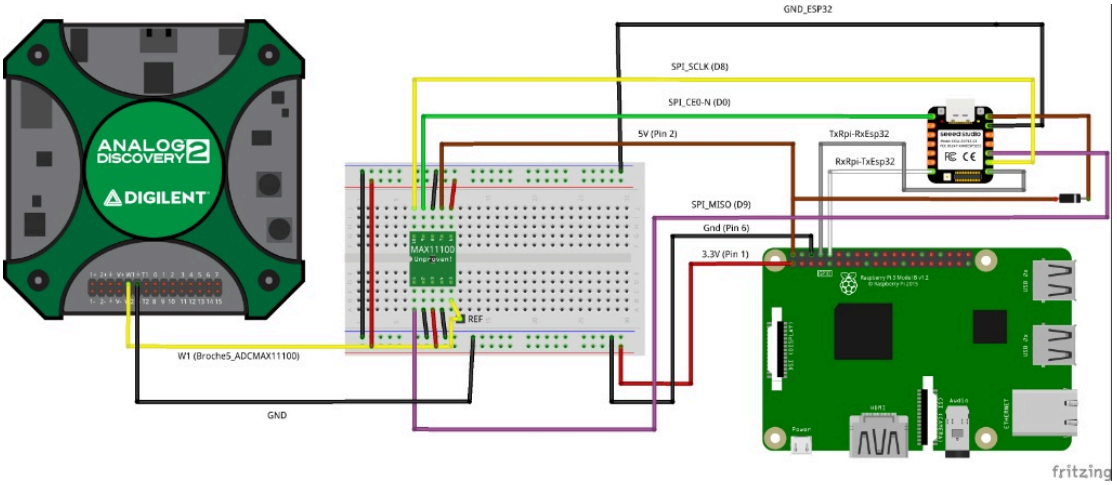
        logger.info("[ECGRepository] Session sauvegardée : %s", filepath)
        return filepath
```

2.6.1.5. Configuration et Environnement

- **Notification d'Analyse** : Le `TraitementController` envoie un message "GO" via un WebSocket spécifique à l' application de traitement de Ivan GOUIN (IR2) pour lui signaler qu'un nouveau fichier est prêt à être analysé.
- **Relais des Résultats** : Une fois l'analyse terminée (calcul de la fréquence cardiaque, HRV, etc.), les résultats sont renvoyés au serveur qui les relaie immédiatement à l'interface graphique du praticien.

2.7. Test du système d'acquisition

Afin de valider l'intégrité du signal SPI et de vérifier le respect des timings de conversion (360 Hz), nous avons utilisé un analyseur logique (Analog Discovery 2) pour comparer le signal physique mesuré avec les trames logiques traitées par l'ESP32.

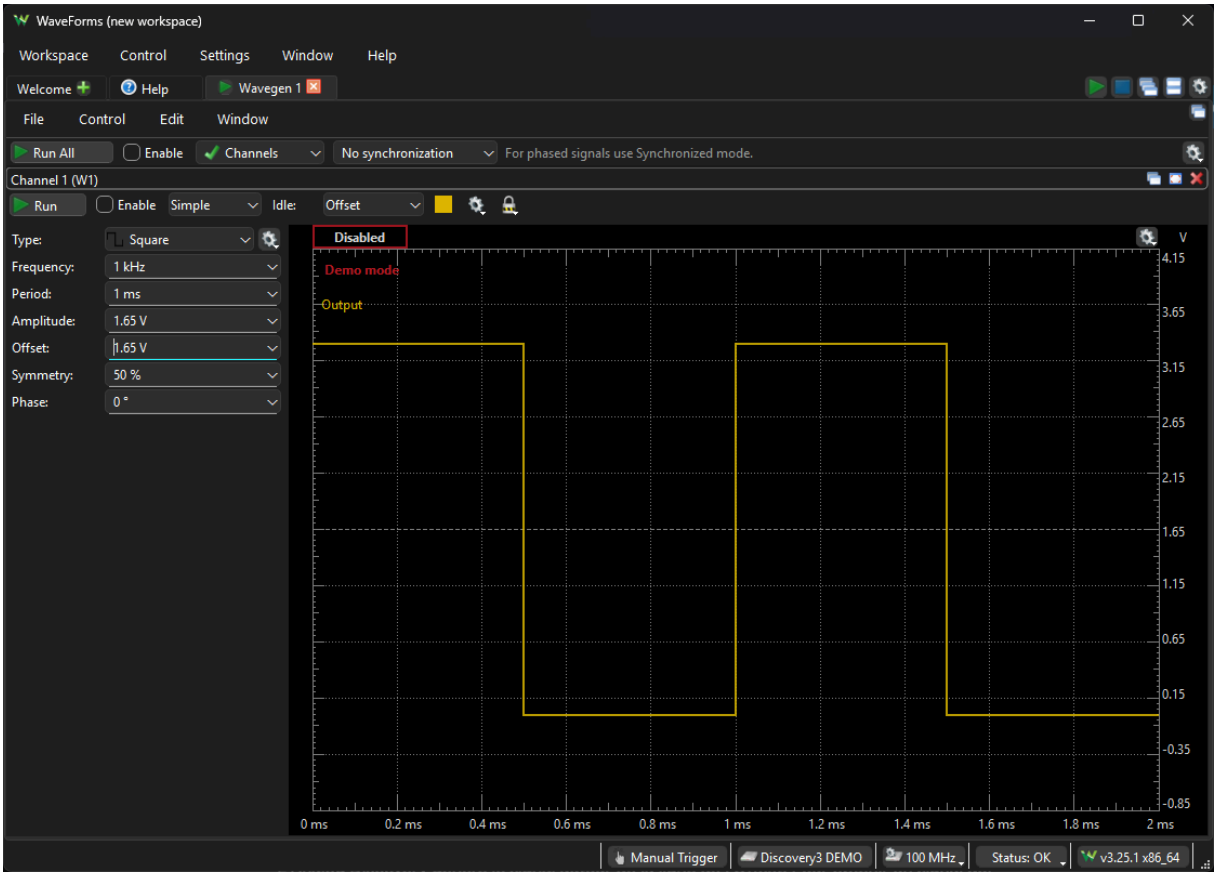


Pour la phase de tests, j'ai utilisé **CuteCom**. Ce logiciel m'a permis de visualiser en temps réel les données qui transitent sur la **liaison entre** la Raspberry Pi et l'ESP32. C'était un outil essentiel pour vérifier que les trames étaient correctement envoyées et reçues, et pour valider la communication entre mes deux équipements.

Pour valider la chaîne d'acquisition, j'ai utilisé un **Analog Discovery 2**, un outil multifonction piloté par le logiciel **waveform**.

J'ai configuré son générateur de fonctions (WAVEGEN) pour injecter un **signal carré oscillant entre 0 V et 3 V**.

Ce choix de tension est crucial car il correspond exactement à la plage dynamique d'entrée du **MAX11100**, alimenté ici en 3 V.



Cette manipulation a permis de vérifier que le système traitait correctement les paliers de tension extrêmes et de valider la précision de la conversion numérique avant d'utiliser des signaux capteurs plus complexes.

Voici un tableau expliquant la configuration de Cutecom

Device	Baudrate	Data bits	Stop bits	Parity
/dev/ttyS0	115200	8	1	None

Device (/dev/ttyS0) : Désigne le port série physique utilisé sur la Raspberry Pi. Le `ttyS0` est le port UART matériel principal relié aux broches GPIO 14 (TX) et 15 (RX).

Baudrate (115200) : Vitesse de transmission exprimée en baud qui symbolise le nombre de symboles par bits. Le réglage doit être identique sur l'ESP32 et CuteCom pour que les données soient lisibles.

Data bits (8) : Chaque trame envoyée contient 8 bits de données (soit 1 octet), ce qui est le standard pour le texte ASCII.

Stop bits (1) : Un bit de fin est ajouté après chaque octet pour indiquer au récepteur que le transfert de ce caractère est terminé.

Parity (None) : Aucun bit de contrôle d'erreur n'est ajouté. On privilégie ici la vitesse et la simplicité, les erreurs étant rares sur une liaison filaire courte.

Voici les résultats obtenus lors des tests :

[Input]	S
[Output]	Start
[Output]	**0**
[Output]	**2.989232**
[Output]	**0**
[Output]	**2.989232**

Input/Command ('S') : Caractère envoyé par l'utilisateur pour basculer la variable `acquisitionRunning` à `true` dans le programme et ainsi lancer le flux de données.

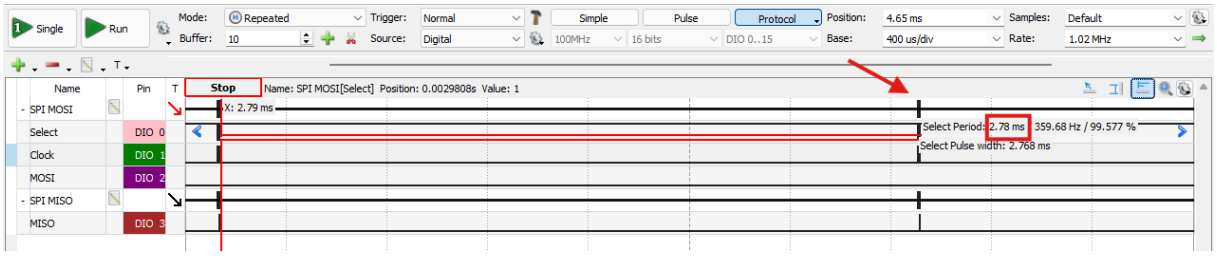
Output (3.289232) : Valeur reçue après conversion. Elle représente la tension en Volts calculée à partir des 16 bits envoyés par le MAX11100.

Bien que la communication série entre l'ESP32 et la Raspberry Pi soit opérationnelle, le test de validation avec le signal carré est considéré comme une **réussite technique**.

- **Observation :** Lors de l'envoi de la commande `'S'`, la console CuteCom affiche une valeur qui alterne entre (`0` et environ `2.989232 V`). Ce qui est cohérent par rapport à notre signal carré.

2.7.1. Validation de la période d'échantillonnage

Pour valider le cadencement de mon application, j'ai mesuré l'intervalle de temps séparant deux activations de la broche de sélection.



Capture d'écran de l'analyseur logique validant la fréquence d'échantillonnage de 360 Hz via la broche Select.

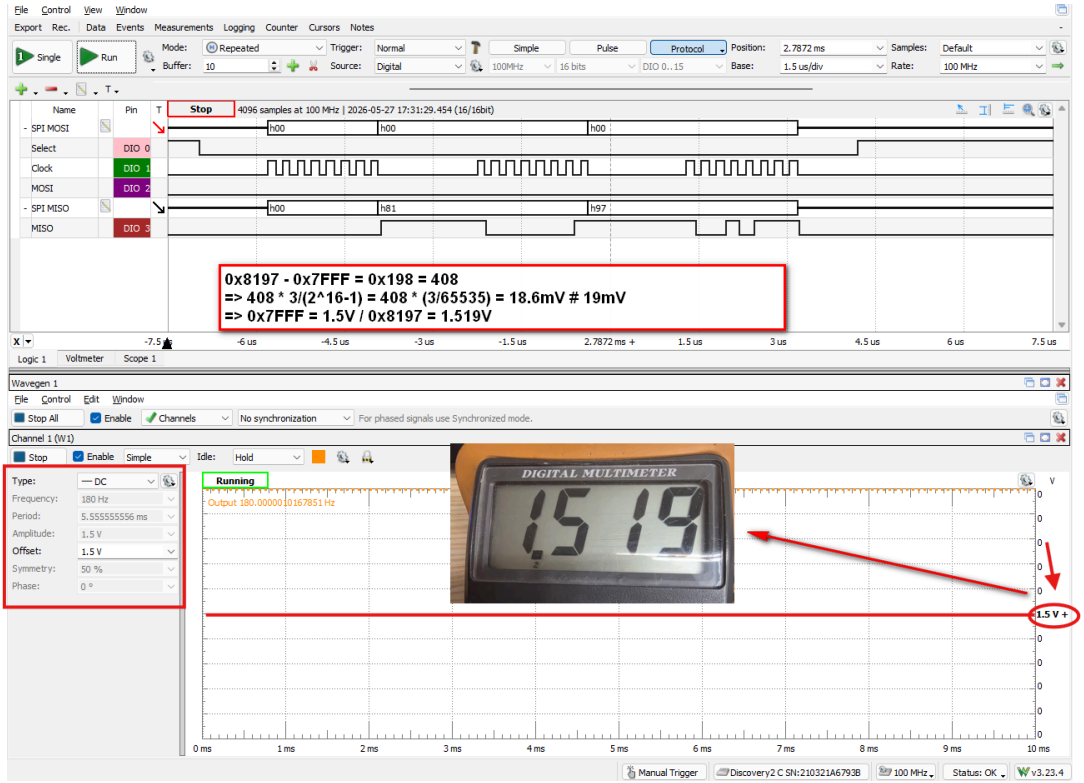
Pour vérifier la régularité de mon cycle d'acquisition, j'ai branché l'analyseur logique de l'Analog Discovery sur la broche de sélection (**Select**).

La période mesurée entre deux fronts descendants est de 2,78 ms, ce qui correspond à une fréquence de 359,68 Hz.

Par rapport aux 360 Hz théoriques, la précision dépasse les 99,9 %. Ce test confirme que la fonction de temporisation `esp_rom_delay_us` que j'ai configurée dans la programmation de l'ESP32 est parfaitement fonctionnelle.

2.7.2. Validation de la valeur reçue (Vérification de la trame SPI)

Pour vérifier que les données reçues en SPI ne sont pas fausses ou corrompues, j'ai fait un test en envoyant une tension théorique de **1,500 V** (ce qui correspond à la moitié exacte de la tension de référence de l'ADC qui est de 3 V).



1. Au multimètre, on s'aperçoit qu'en réalité, la tension physique injectée est de **1,519 V** (à cause des tolérances du matériel et des composants).
2. Sur l'analyseur logique, j'ai regardé ce qui transitait sur la **broche MISO** (le fil par lequel le convertisseur répond). La trame reçue donne la valeur brute 0x8197 en hexadécimal.

Ensuite, j'ai fait un calcul simple pour vérifier si la valeur lue par l'ESP32 correspond bien à la réalité du terrain :

- Mon convertisseur travaille sur 16 bits avec une plage de 0 à 3 V. Son milieu théorique (pour 1,5 V) donne la valeur numérique 0x7FFF.
- La valeur réellement reçue (0x8197) est légèrement au-dessus. L'écart est de **408 pas**.
- Comme le convertisseur a une **résolution** d'environ 45,78 µV par pas (3/65 535), cet écart représente : **408 X 45,78 µV = 18,6mV.**
- Si on ajoute ce décalage à notre milieu théorique (1,5 V + 0,0186 V), on trouve **1,5186 V**.

Le verdict : Le calcul issu de la trame SPI donne **1,519 V**, ce qui est exactement la tension réelle mesurée au multimètre. Cela confirme que ma chaîne d'acquisition fonctionne à la perfection et que les données reçues sont ultra-précises, malgré les légères variations du circuit réel.

2.8. Conclusion

Bien qu'à ce jour le programme d'acquisition soit fonctionnel, le travail fourni m'a permis de réaliser qu'il y a toujours des imprévus qui empêchent de respecter la planification prévue au départ. Entre la théorie et la mise en pratique, j'ai dû faire face à des obstacles que je n'avais pas imaginés.

Ce projet m'a également permis de progresser sur un plan très concret : **la manipulation.** Je me suis familiarisé avec le **câblage sur platine d'essai**, un aspect sur lequel je n'étais pas à l'aise au début. J'ai appris à organiser mes branchements de manière plus rigoureuse et à mieux comprendre l'importance de chaque connexion.

En résumé, cette expérience a été l'occasion d'apprendre à :

- Mieux gérer mon temps face à des blocages imprévus.
- Gagner en autonomie sur le montage de circuits réels.
- Chercher des solutions en testant différentes méthodes de manière logique tel que **Dash et FastAPI**.

PARTIE PERSONNELLE
IVAN GOUIN
(IR 2)

3. PARTIE PERSONNELLE - IVAN GOUIN

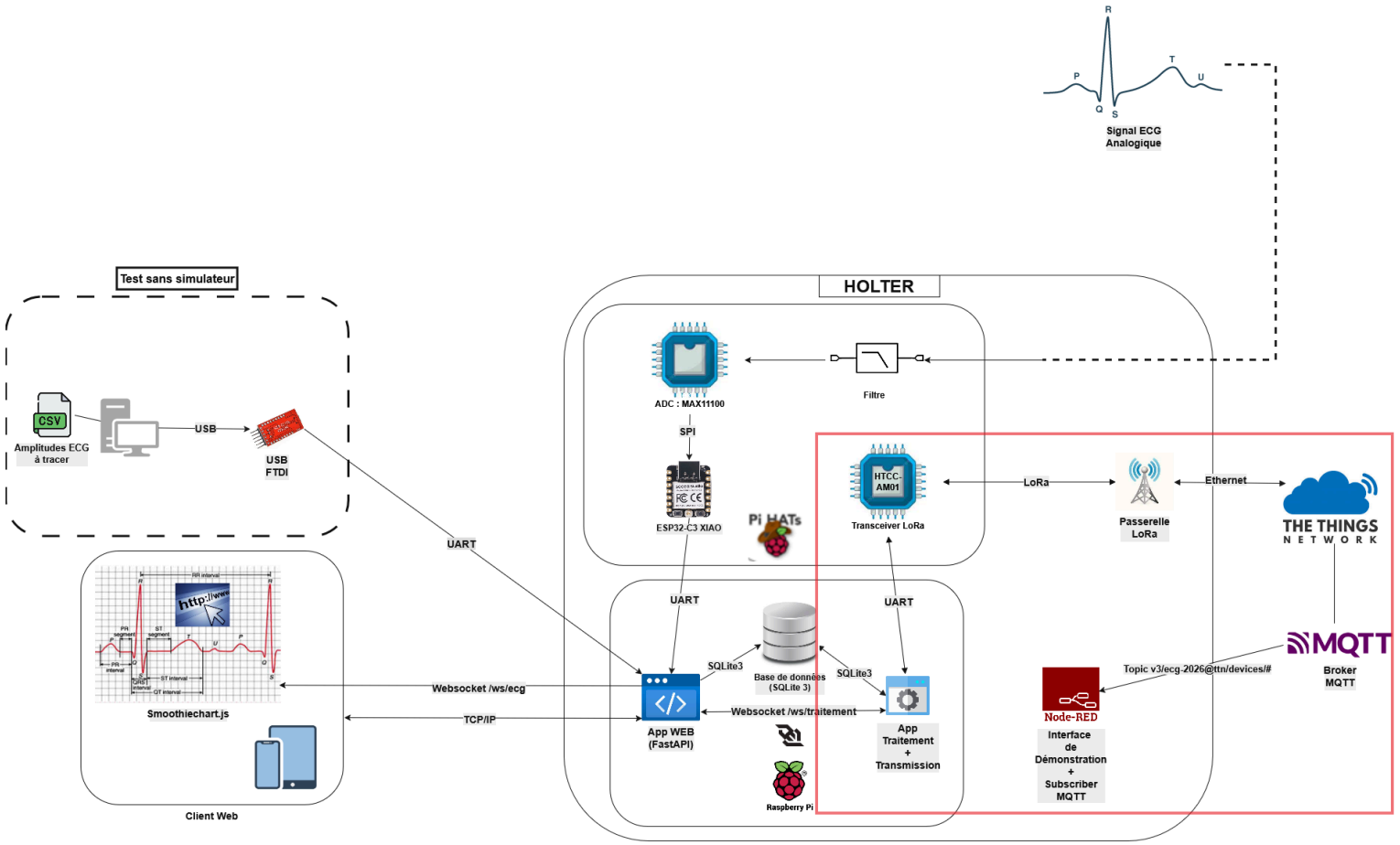
3.1 Objectifs

Dans le cadre de ce projet, une architecture distribuée a été mise en place afin de permettre l'acquisition, le traitement et la transmission de données électrocardiographiques (ECG) en temps réel.

Le système développé s'inscrit dans une architecture IoT distribuée, combinant traitement local et transmission distante.

Le système repose sur une séparation claire entre les différentes couches fonctionnelles : acquisition du signal, traitement des données et visualisation.

Voici le synoptique du projet, et la partie encadrée dans le rectangle rouge sur laquelle je travaille.



Ma partie consiste au **traitement des données**. Le traitement consiste à :

- acquérir les données associées à l'ECG capturé par IR1,
- détecter les pics correspondant aux battements cardiaques, puis calculer plusieurs indicateurs physiologiques tels que la fréquence cardiaque moyenne, minimale et maximale, ainsi que la variabilité du rythme cardiaque (HRV).
- renvoyer le résultat analyse à l'application Web via **WebSocket** pour affichage en temps réel,
- transmettre via LoRaWAN le même résultat d'analyse après encodage vers le cloud The Things Network via le module radio (**CubeCell**).

Les données sont ensuite transmises via le protocole **LoRa** vers une passerelle, puis vers le **cloud** (**The Things Network**), où elles sont accessibles via **MQTT** et visualisées avec **Node-RED**.

Mon travail consistait donc dans un premier temps à analyser les données reçues du programme d'acquisition de **Valentin BUFFARD (IR1)**, par le biais de la base de données que le programme d'acquisition avait créé après chaque échantillonnage.

Une fois que les valeurs sont obtenues, l'objectif étant de pouvoir partager ce résultat graphiquement avec le programme d'acquisition/visualisation de V.Buffard et un tableau de bord qui traduit graphiquement le résultat de l'analyse transmis via LoRa sur l'infrastructure TTN, permettant de déclencher une alerte en cas d'anomalie cardiaque, ceci grâce à une interface réalisée avec **Node-RED**.

Cette application de traitement, développée en Python, a été restructurée selon le modèle architectural **MVC (Model-View-Controller)**. Cette approche permet de séparer la logique métier (analyse ECG), la gestion des entrées/sorties (accès base de données, communication série, WebSocket) et la présentation des résultats.

3.2 Outils Utilisés

En plus des Outils Commun, Ma partie du projet utilise les outils suivants.

SQLite : Le programme accède à la base SQLite via un service dédié (database_service.py), des requêtes SQL sont exécutées afin de récupérer les données nécessaires au traitement.

Oracle VirtualBox : Précédemment, la VM Debian servait en tant qu'environnement de travail, à présent elle sert tout simplement comme serveur hôte temporaire à l'application Node-RED, car toutes les fonctionnalités seront déplacées sur la RPi .

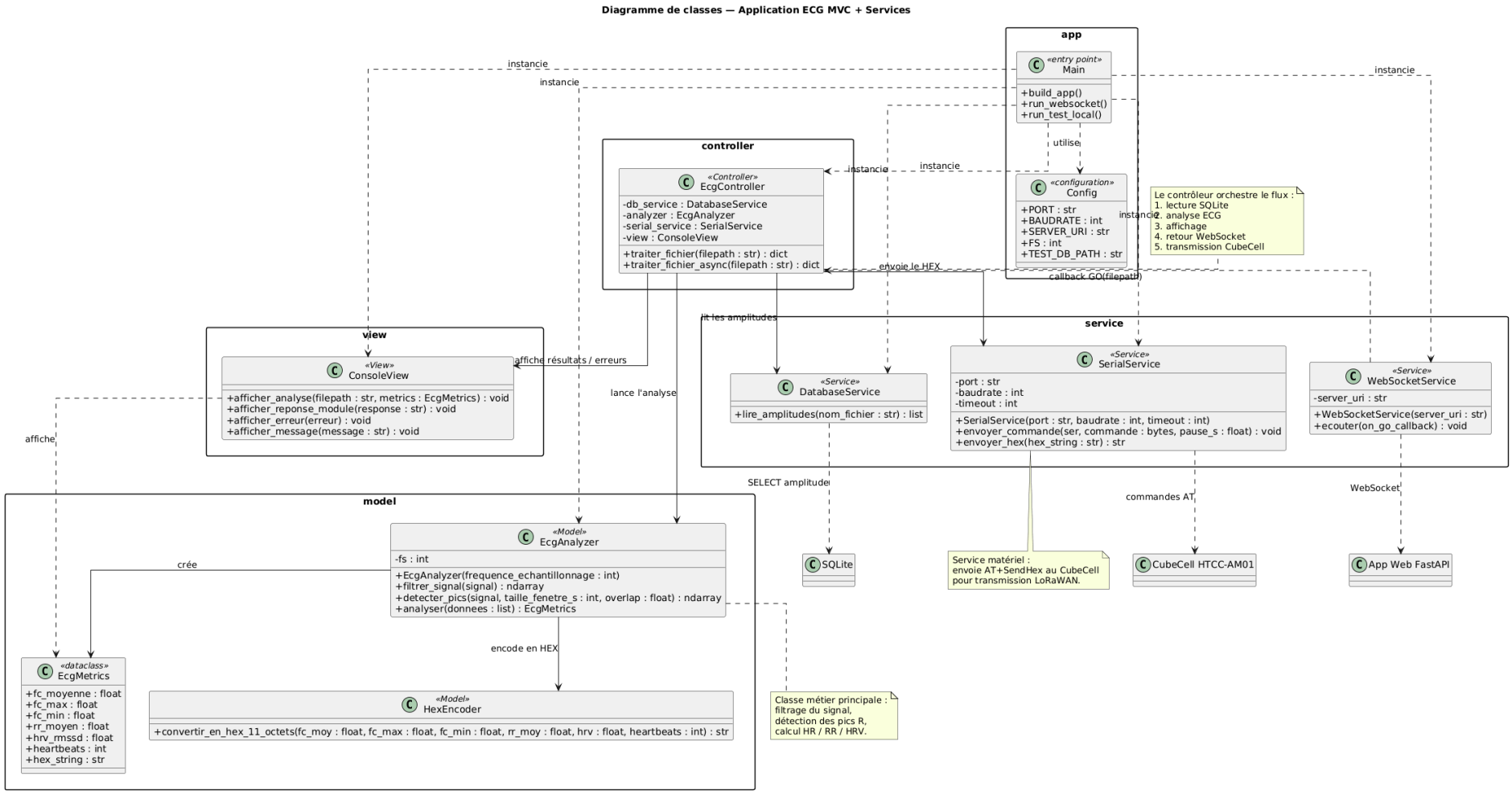
PuTTY : L'outil le plus utile, permettant d'accéder à une RPi en ligne de commande seulement. PuTTY est un outil essentiel à l'exécution d'un ou de plusieurs programmes Python.

Heltec Cube Cell : Le Module Arduino de communication servant à transporter les informations envoyées en Hexadécimal à la passerelle du cloud TTN par le protocole LoRaWAN (Communication basse fréquence faite purement pour les appareils électroniques et les clouds, soit IoT, internet of Things)



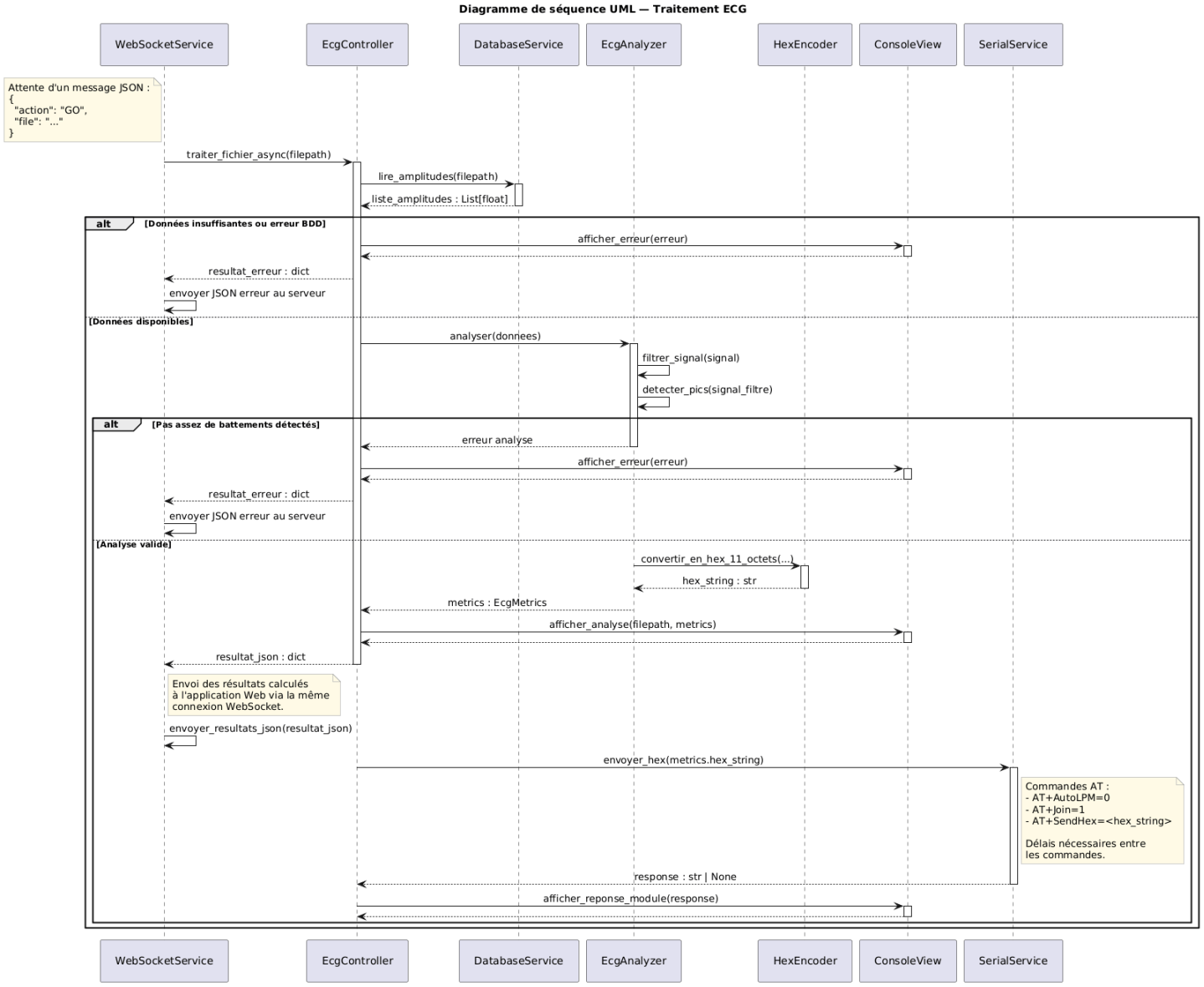
Node-RED : Node-RED est l'outil graphique que j'ai décidé d'utiliser pour la partie tableau de bord, celui-ci ayant la possibilité de connecter par liaison MQTT le tableau de bord au flux de données entrant dans le cloud. **Voir** **Annexe A**

3.3 Diagrammes de classes



Le diagramme de classes illustre l'organisation de l'application selon une architecture MVC complétée par une couche de services. Il met en évidence les responsabilités de chaque composant ainsi que leurs interactions dans le traitement du signal ECG, la communication WebSocket, l'accès SQLite et la transmission LoRaWAN.

3.4 Diagrammes de séquence



Ce diagramme de séquence illustre le fonctionnement global du système lors du traitement d'un signal ECG.

L'application d'acquisition et de visualisation envoie une requête de type GO via WebSocket à l'application de traitement. Celle-ci récupère les données ECG depuis la base de données SQLite, puis effectue l'analyse du signal afin de calculer les indicateurs physiologiques.

Les résultats sont ensuite utilisés pour générer une trame hexadécimale, qui est transmise au module CubeCell via le port série. Parallèlement, les résultats sont renvoyés à l'application Web via WebSocket afin d'être affichés en temps réel.

Ce mécanisme met en évidence une communication bidirectionnelle ainsi qu'une séparation claire entre les différentes couches du systèm

3.5 Code de l'application traitement des données

Lors du développement de cette application, j'ai rencontré plusieurs difficultés pour lesquelles j'ai dû apporter des solutions, soit par des recherches personnelles, soit par des échanges avec mon équipe, ce qui correspond pleinement aux objectifs d'un travail en mode projet.

Dans un premier temps, j'avais développé un code monolithique, que j'ai ensuite fait évoluer vers une architecture de type MVC afin d'améliorer la structuration et la maintenabilité de l'application.

3.5.1 Difficultés rencontrées et solutions apportées

Voici la liste des principaux problèmes que j'ai pu rencontrer tout au long de ce projet, et les différentes solutions apportées.

3.5.1.1 Problème d'encodage des données (Hexadécimal)

Problème

- Lors de l'envoi des données vers le cloud via le module CubeCell, les valeurs reçues sur The Things Network (TTN) ne correspondaient pas aux valeurs calculées par le programme.
- Les données étaient correctement calculées côté Python, mais une fois transmises, elles devenaient incohérentes.

Analyse

- Après analyse, le problème provenait du format d'encodage des données binaires.
- Le programme utilisait initialement un encodage en Big Endian, alors que le système de réception attendait un encodage en Little Endian.

Solution

- Modification du format d'encodage dans la fonction struct.pack :

```
# Avant (incorrect)
'>HHHHHB'

# Après (correct)
'<HHHHHB'
```

3.5.1.2 Communication WebSocket unidirectionnelle

Problème

- Au départ, l'application de traitement recevait bien les ordres (GO) via WebSocket, mais ne renvoyait pas les résultats à l'application Web.
- Cela empêchait l'affichage des résultats côté interface utilisateur.

Analyse

- La communication WebSocket était utilisée uniquement en réception, alors que ce protocole permet une communication bidirectionnelle.

Solution

- Ajout d'un mécanisme d'envoi des résultats après traitement :

3.5.1.3 Gestion de la base de données SQLite

Problème

- L'accès aux données ECG était initialement intégré directement dans le code de traitement, ce qui rendait le code difficile à maintenir et peu modulaire.

Analyse

- Ce couplage entre logique métier et accès aux données compliquait :
  - Les tests
  - Les évolutions futures
  - La lisibilité du code

Solution

- Création d'un service dédié : database_service.py
  - Encapsulation des requêtes SQL
  - Séparation des responsabilités

3.5.1.4 Complexité du code initial

Problème

- Le code de départ était monolithique, mélangeant :
  - Traitement ECG
  - Affichage
  - Accès base de données
  - Communication réseau et série

Analyse

- Ce type de structure rend :
  - Le code difficile à comprendre
  - Les modifications risquées
  - Les tests compliqués

Solution

- Refactorisation complète selon le modèle MVC + Services
  - Model : traitement ECG
  - View : affichage console
  - Controller : orchestration
  - Services : accès technique (BDD, WebSocket, série)

3.5.1.5 Communication avec le module CubeCell

Problème

- La communication avec le module CubeCell via port série nécessitait l'envoi de commandes AT dans un ordre précis, avec des délais spécifiques.

Analyse

- Sans respect de ces contraintes :
  - Le module ne répond pas
  - La transmission échoue

Solution

- Implémentation d'un service dédié serial_service.py avec :
  - Envoi séquencé des commandes AT
  - Gestion des délais (time.sleep)
  - Lecture des réponses du module

3.5.1.6 Comptage des pics trop proche

Problème

- Lors de l'analyse de signaux ECG présentant une fréquence cardiaque élevée, certains battements pouvaient être mal détectés. Dans certains cas, deux pics R proches étaient interprétés comme un seul battement, entraînant une perte de détection

Analyse

- Le problème provenait de l'algorithme de détection implémenté dans `ecg_analyzer.py`.

- La variable `distance_locale`, calculée dynamiquement à partir de la fréquence locale du signal, réduisait parfois excessivement la distance minimale entre deux pics détectables. Dans certaines situations, cette logique pouvait conduire à une mauvaise séparation des battements réels ou à une détection instable selon la fenêtre analysée.

  Cela provoquait :

  - Battements non comptabilisés
  - Calcul erroné des intervalles RR
  - Fréquence cardiaque sous-estimée ou instable
  - Variabilité HRV faussée

Solution

Développement d'une nouvelle version de `ecg_analyzer.py` avec :

- Suppression du calcul adaptatif basé sur `distance_locale`
- Mise en place d'une **distance physiologique minimale fixe de 250 ms** entre deux pics détectés (`distance_physiologique`)
- Conservation d'un **seuil adaptatif local** basé sur la moyenne et l'écart-type du signal

- Ajout d'un **filtrage final des pics détectés** afin d'éliminer les doublons entre fenêtres glissantes

Cette nouvelle approche a permis de stabiliser la détection des pics R, d'éviter les doubles comptages ou les pertes de battements et d'améliorer la fiabilité globale des métriques physiologiques calculées.

Ces différentes difficultés ont permis :

- D'améliorer la compréhension des protocoles de communication
- De structurer le code selon des principes d'architecture logicielle
- D'optimiser la transmission des données dans un contexte IoT contraint

Elles ont contribué à rendre le système plus robuste, modulaire et fiable.

3.5.2 CONFIG

Le fichier config.py regroupe l'ensemble des paramètres de configuration de l'application, tels que les ports de communication, les adresses réseau et les constantes liées au traitement du signal. Cette séparation permet d'éviter la duplication de valeurs dans le code et facilite la maintenance ainsi que l'adaptation du système à différents environnements.

Après la première revue de projet, nous sommes passés à l'étape suivante : utiliser une carte Raspberry, afin de faire fonctionner les programmes IR1 et IR2 sans avoir à passer par le réseau local, c'est pour cela que nous utilisons le préfix "localhost" dans la variable SERVER URI

Voir code en Annexe B (1)

3.5.3 APP

Le fichier main.py constitue le point d'entrée de l'application. Il est responsable de l'initialisation des différents composants du système (services, contrôleur, vue) et de leur assemblage. Il implémente un mécanisme d'injection de dépendances permettant de découpler les différentes couches de l'application. Il assure également le lancement du système, soit en mode test local, soit en mode connecté via WebSocket.

Voir code en Annexe B (2)

3.5.4 MODEL

ecg_metrics.py définit une structure de données représentant les résultats issus de l'analyse ECG. Cette structure permet de regrouper de manière cohérente les indicateurs physiologiques calculés, tels que la fréquence cardiaque moyenne, minimale et maximale, l'intervalle RR moyen, la variabilité du rythme cardiaque ainsi que le nombre de battements détectés.

Voir code en Annexe B (3)

hex_encoder.py assure la conversion des résultats ECG en une trame hexadécimale compacte. Cette étape est nécessaire afin d'adapter les données calculées aux contraintes de transmission du module CubeCell et du réseau LoRa. L'encodage permet de transmettre uniquement les indicateurs essentiels plutôt que le signal ECG brut, ce qui réduit fortement le volume de données échangé.

Voir code en Annexe B (4)

Le fichier ecg_analyzer.py implémente la logique métier principale de l'application. Il est responsable du traitement du signal ECG, incluant le filtrage, la détection des pics correspondant aux battements cardiaques, ainsi que le calcul des indicateurs physiologiques tels que la fréquence cardiaque et la

variabilité du rythme cardiaque (HRV). Ce module produit une structure de données normalisée contenant les résultats de l'analyse, utilisée ensuite par les autres composants du système.

3.5.4.1 Filtrage du signal ECG

Avant l'analyse, le signal ECG doit être filtré afin de supprimer la dérive lente de ligne de base et certaines perturbations basses fréquences.
Dans l'application, un **filtre passe-haut d'ordre 2** est utilisé :

$$H(f) = \frac{Signal_{sortie}(f)}{Signal_{entree}(f)}$$

Ce filtrage permet d'atténuer les composantes inférieures à **0,5 Hz**, correspondant principalement à la dérive de base du signal ECG.

3.5.4.2 Découpage en fenêtres glissantes

L'analyse est réalisée par **fenêtres glissantes** afin d'adapter la détection aux variations locales du signal.
La taille de fenêtre dépend de la fréquence d'échantillonnage :

$$N_{fenetre} = T_{fenetre} \times f_s$$

avec :

- $T_{fenetre}$: durée de fenêtre (s)
- f_s : fréquence d'échantillonnage

3.5.4.3 Distance physiologique minimale entre pics

On utilise une distance minimale fixe basée sur la physiologie cardiaque.
La relation fondamentale entre fréquence cardiaque et période est :

$$T = \frac{60}{FC}$$

Pour une fréquence cardiaque maximale acceptable de 240 bpm :

$$T_{min} = \frac{60}{240} = 0.25\ s$$

Le nombre minimal d'échantillons correspondant vaut :

$$N_{min} = f_s \times T_{min}$$

3.5.4.4 Seuil adaptatif de détection

Afin de limiter les faux pics liés au bruit, un seuil local adaptatif est calculé à partir de la moyenne et de l'écart-type du signal
Formule :

$$Seuil = \mu + 0.6\sigma$$

3.5.4.5 Détection des pics R

La détection est réalisée à l'aide de find_peaks() de SciPy.
Mathématiquement, un pic correspond à un maximum local :

$$x(i) > x(i-1)\ et\ x(i) > x(i+1)$$

3.5.4.6 Fusion et nettoyage des pics détectés

Les pics détectés dans chaque fenêtre sont fusionnés.
Un **ensemble (set)** est utilisé afin d'éliminer naturellement les doublons.

Après tri, un filtrage final élimine les pics trop proches.
Formellement cela est représenté par :

$\Delta n > N_{min}$

où :

- Δn : écart entre deux pics détectés
- N_{min} : distance physiologique minimale

Cette étape supprime les artefacts (accidents) liés au chevauchement des fenêtres.

3.5.4.7 Calcul des intervalles RR

Après détection des pics, les intervalles RR sont calculés.
Formule :

$RR = t_{n+1} - t_n$

3.5.4.8 Calcul de la fréquence cardiaque

La fréquence cardiaque instantanée est déduite des intervalles RR.
Formule :

$FC = \frac{60}{RR}$

Voir code en Annexe B (5)

3.5.5 VIEW

Le fichier console_view.py implémente la couche Vue (View) du modèle MVC. Il est responsable de l'affichage des résultats du traitement ECG ainsi que des messages d'erreur. Cette séparation permet d'isoler la logique de présentation de la logique métier, facilitant ainsi l'évolution de l'interface utilisateur sans impacter le reste du système.

Voir code en Annexe B (6)

3.5.6 CONTROLLER

Le contrôleur, implémenté dans le fichier ecg_controller.py, est instancié par le point d'entrée de l'application (main.py). Il joue un rôle central dans l'orchestration du flux de données, en coordonnant l'accès à la base de données, le traitement du signal ECG et la communication avec les autres composants du système.

Il assure donc la coordination complète du flux de données, depuis la réception de la requête jusqu'à l'envoi des résultats vers les différents systèmes.

Voir code en Annexe B (7)

3.5.7 SERVICES

Les services implémentés dans l'application constituent une couche d'infrastructure chargée de gérer les interactions avec les composants externes du système. Ils ne font pas partie du modèle MVC.

Ils encapsulent les accès à la base de données, les communications réseau ainsi que les échanges avec les modules matériels. Cette approche permet de découpler la logique métier des détails techniques, améliorant ainsi la modularité, la maintenabilité et la testabilité du système.

Le fichier database_service.py implémente un service d'accès aux données. Il encapsule les interactions avec la base SQLite et permet de découpler la logique métier du système des détails liés à

la persistance des données. Cette approche améliore la maintenabilité et facilite le remplacement éventuel du système de stockage.

Voir code en Annexe B (8)

Le fichier serial_service.py implémente le service de communication série avec le module CubeCell. Il utilise la bibliothèque pyserial afin d'ouvrir une liaison série avec le module HTCC-AM01 et d'envoyer les commandes AT nécessaires à la transmission des données. Après le traitement ECG, les résultats calculés sont convertis en trame hexadécimale, puis envoyés au module via une commande de type AT+SendHex. Le module CubeCell se charge ensuite de transmettre ces informations par LoRaWAN vers la passerelle, puis vers le cloud TTN.

Voir code en Annexe B (9)

Le fichier websocket_service.py implémente un service de communication réseau basé sur le protocole WebSocket. Il permet d'établir une connexion persistante avec l'application Web et de gérer des échanges bidirectionnels en temps réel. Ce service est responsable de la réception des commandes de traitement ainsi que de l'envoi des résultats après analyse. Il joue un rôle central dans la synchronisation entre l'interface utilisateur et le module de traitement ECG.

Voir code en Annexe B (10)

À l'issue du traitement des données ECG au sein de l'application structurée en architecture MVC, les résultats calculés sont transmis vers l'extérieur du système via différents canaux de communication.

D'une part, ces résultats sont renvoyés en temps réel à l'application Web via WebSocket, et d'autre part, ils sont convertis en trame hexadécimale puis transmis au module CubeCell pour être envoyés via le réseau LoRaWAN.

Une fois les données arrivées sur le cloud The Things Network, elles sont rendues accessibles via le protocole MQTT. C'est à ce niveau qu'intervient Node-RED, qui permet de récupérer ces données, de les transformer, puis de les exploiter pour la visualisation, le stockage et la gestion des alertes.

Ainsi, Node-RED constitue la continuité logique de l'architecture MVC, en assurant la partie supervision et interaction utilisateur du système IoT.

3.6 Validation du traitement ECG et tests unitaires

Des tests unitaires ont été utilisés afin de valider automatiquement le comportement de l'algorithme de traitement ECG et de garantir la cohérence des résultats après chaque évolution du code.
Réalisés avec `pytest` (), ils permettent de vérifier le comportement de l'algorithme sur plusieurs cas représentatifs : rythme régulier à 60 bpm, rythme plus rapide à 120 bpm, signal trop court, signal plat sans battement, calcul de la HRV et génération de la trame hexadécimale.
Ces tests valident à la fois le fonctionnement nominal de l'application ainsi que la gestion des cas limites et des erreurs.
Les résultats attendus s'affichent sous cette forme à l'écran :

test_ecg_analyzer.py::test_analyse_signal_ecg_60_bpm PASSED
test_ecg_analyzer.py::test_analyse_signal_ecg_120_bpm PASSED
test_ecg_analyzer.py::test_signal_trop_court PASSED
test_ecg_analyzer.py::test_signal_plat_sans_battement PASSED
test_ecg_analyzer.py::test_hrv_rythme_regulier PASSED
test_ecg_analyzer.py::test_generation_hex PASSED

Voir code & résultat sur ligne de commande en Annexe C

À l'issue du traitement des données ECG au sein de l'application structurée en architecture MVC, les résultats calculés sont transmis vers l'extérieur du système via différents canaux de communication.

D'une part, ces résultats sont renvoyés en temps réel à l'application Web via WebSocket, et d'autre part, ils sont convertis en trame hexadécimale puis transmis au module CubeCell pour être envoyés via le réseau LoRaWAN.

Une fois les données arrivées sur le cloud The Things Network, elles sont rendues accessibles via le protocole MQTT. C'est à ce niveau qu'intervient Node-RED, qui permet de récupérer ces données, de les transformer, puis de les exploiter pour la visualisation, le stockage et la gestion des alertes.

Ainsi, Node-RED constitue la continuité logique de l'architecture MVC, en assurant la partie supervision et interaction utilisateur du système IoT.

3.7 Node-RED

Dans le cadre du projet, Node-RED est utilisé comme plateforme centrale de traitement des données. Il permet d'assurer la réception, la transformation, le stockage et la visualisation des données ECG en temps réel.

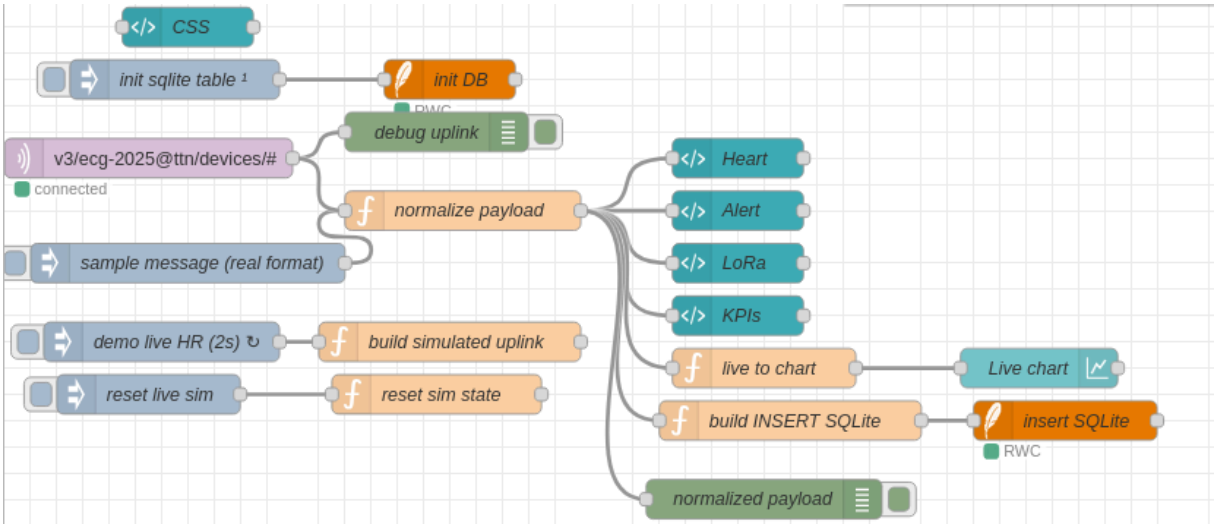
L'architecture que j'ai mise en place repose sur un modèle **orienté flux (event-driven)**, dans lequel chaque message est traité dynamiquement sans stockage intermédiaire inutile, garantissant ainsi une bonne réactivité du système.

Le flow Node-RED est structuré en trois parties principales :

- **DATA** : traitement temps réel
- **HISTORY** : gestion de l'historique
- **SETTINGS** : configuration des seuils d'alerte

3.7.1 DATA

Le bloc DATA constitue le cœur du système. Il assure la réception des données issues du réseau LoRaWAN via le protocole MQTT.



3.7.1.1 Réception des données

Les données sont transmises par le capteur ECG via le réseau The Things Network (TTN) puis récupérées dans Node-RED à l'aide d'un nœud MQTT.

3.7.1.2 Normalisation des données

Les données brutes issues du capteur sont encodées avec un facteur d'échelle (×10).
Une étape de normalisation est donc nécessaire afin d'obtenir des valeurs physiologiques cohérentes.

Un nœud fonction `normalize payload` permet de :

- Convertir les valeurs (division par 10)
- Restructurer les données
- Extraire les informations utiles :
  - Fréquence cardiaque (HR)
  - HR min / max / mean
  - HRV
  - RR mean
  - RSSI / SNR

Cette étape est essentielle pour garantir la cohérence des affichages et des traitements ultérieurs.

Afin de vérifier le bon fonctionnement de la phase de normalisation, les données sont observées dans la fenêtre de débogage de Node-RED.

Celle-ci permet de visualiser à la fois les données brutes issues du réseau LoRaWAN et les données transformées après traitement.

Voir code en Annexe D - (1)

Les données reçues apparaissent à la fois graphiques, et dans l'onglet 'Debug'(les valeurs peuvent être lues dans son entièreté si on déroule le contenu)

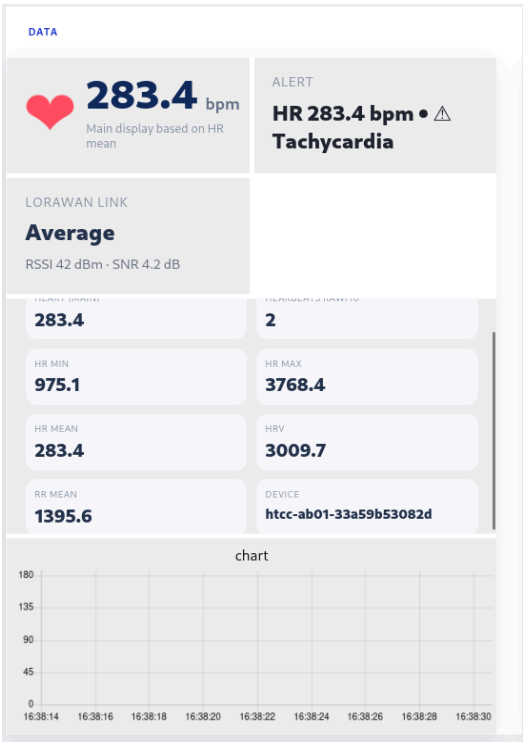
Voir code en Annexe D - (2)

3.7.1.3 Visualisation en temps réel

Les données normalisées sont ensuite utilisées pour alimenter plusieurs composants du Dashboard :

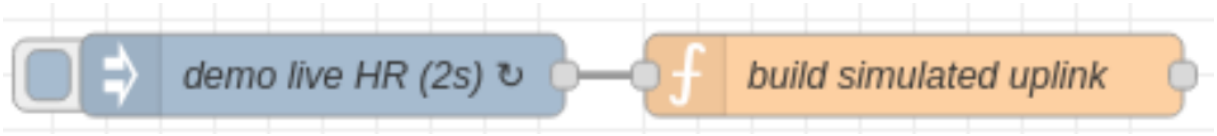
- Affichage de la fréquence cardiaque
- État du réseau LoRa (RSSI / SNR)
- Indicateurs physiologiques
- Graphique en temps réel

Une animation dynamique du cœur a été intégrée, dont la vitesse dépend directement de la fréquence cardiaque mesurée :
durée d'animation = 60000 / BPM
Cette approche permet d'améliorer la lisibilité des données et d'apporter un retour visuel immédiat à l'utilisateur.



Voir code en Annexe D - (3)

3.7.1.4 Simulation des données (mode démonstration)



Le bloc `build simulated uplink` génère :

- Une fréquence cardiaque réaliste (variation sinusoïdale)
- Du bruit physiologique
- Des anomalies ponctuelles (tachycardie / bradycardie)
  L'objectif est de :
- Tester le comportement du système
- Alimenter le graphique temps réel
- Améliorer la démonstration en soutenance
  Voir code en Annexe D - (4)

3.7.1.5 Simulation des données (par envoi de trames via TTN)

Également afin de tester le système indépendamment du capteur physique, une simulation d'uplink est également réalisée via l'interface TTN, en envoyant une trame hexadécimale.

Cela envoi simplement une trame, mais permet de vérifier le bon fonctionnement entre le réseau et le flow Node-RED.

Voir code en Annexe D - (5)

3.7.1.6 Stockage des données

Les données traitées sont enregistrées dans une base SQLite via une requête SQL construite dynamiquement.

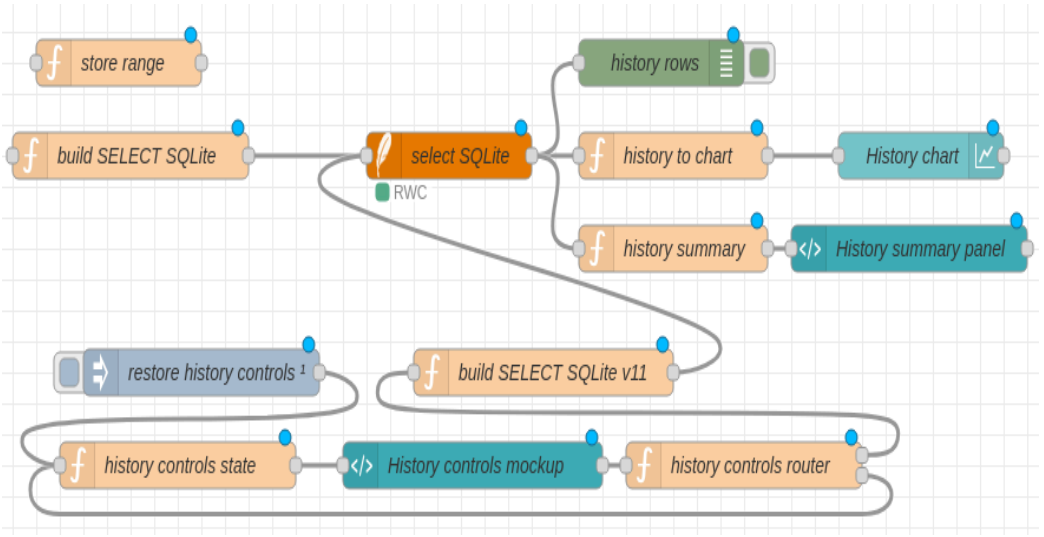
La base contient notamment :

- Timestamp
- Fréquence cardiaque
- Paramètres physiologiques
- Qualité du signal LoRa
- État d'alerte
  Ce stockage permet une exploitation ultérieure des données.

Voir code en Annexe D - (6)

3.7.2 HISTORY

Le bloc HISTORY permet d'afficher les données enregistrées dans la base SQLite sur une période définie.



3.7.2.1 Sélection de la période

Une interface utilisateur permet de sélectionner :

- Une date de début
- Une date de fin
  Les champs incluent à la fois la date et l'heure, permettant une analyse fine des données.

Voir code en Annexe D - (7)

3.7.2.2 Requête SQL

Lors de la validation, une requête SQL est générée dynamiquement :

```
SELECT ts, heart_rate
FROM ecg_measurements
WHERE ts BETWEEN 'start' AND 'end'
ORDER BY ts ASC;
```

Cette requête permet d'extraire les données correspondant à la période sélectionnée.

Voir code en Annexe D - (8)

3.7.2.3 Visualisation graphique

Les résultats sont affichés sous forme de graphique, permettant :

- Une analyse de l'évolution de la fréquence cardiaque
- La détection d'anomalies
- L'observation de tendances

Voir code en Annexe D - (9)

3.7.2.4 Indicateurs statistiques

En complément du graphique, plusieurs indicateurs sont calculés :

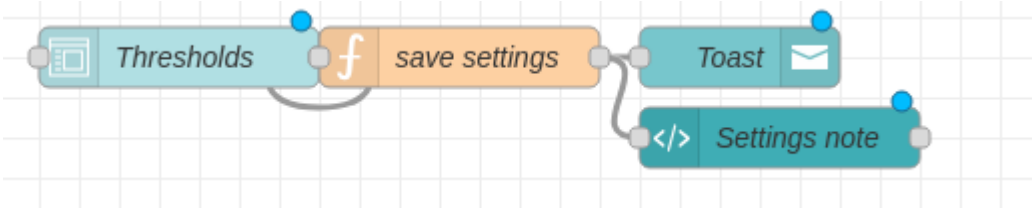
- Nombre de points chargés
- Fréquence cardiaque moyenne
- Valeur minimale et maximale
- Dates de début et fin

Ces informations permettent une interprétation rapide des données.

Voir code en Annexe D - (10)

3.7.3 SETTINGS

Le bloc SETTINGS permet de configurer les seuils d'alerte utilisés par le système.



3.7.3.1 Paramètres

L'utilisateur peut définir :

- Seuil bas (bradycardie)
- Seuil haut (tachycardie)

Voir code en Annexe D - (11)

3.7.3.2 Logique d'alertes

Ces seuils sont utilisés pour déterminer l'état du patient :

- HR < seuil bas → Bradycardie
- HR > seuil haut → Tachycardie
- Sinon → état normal

Voir code en Annexe D - (12)

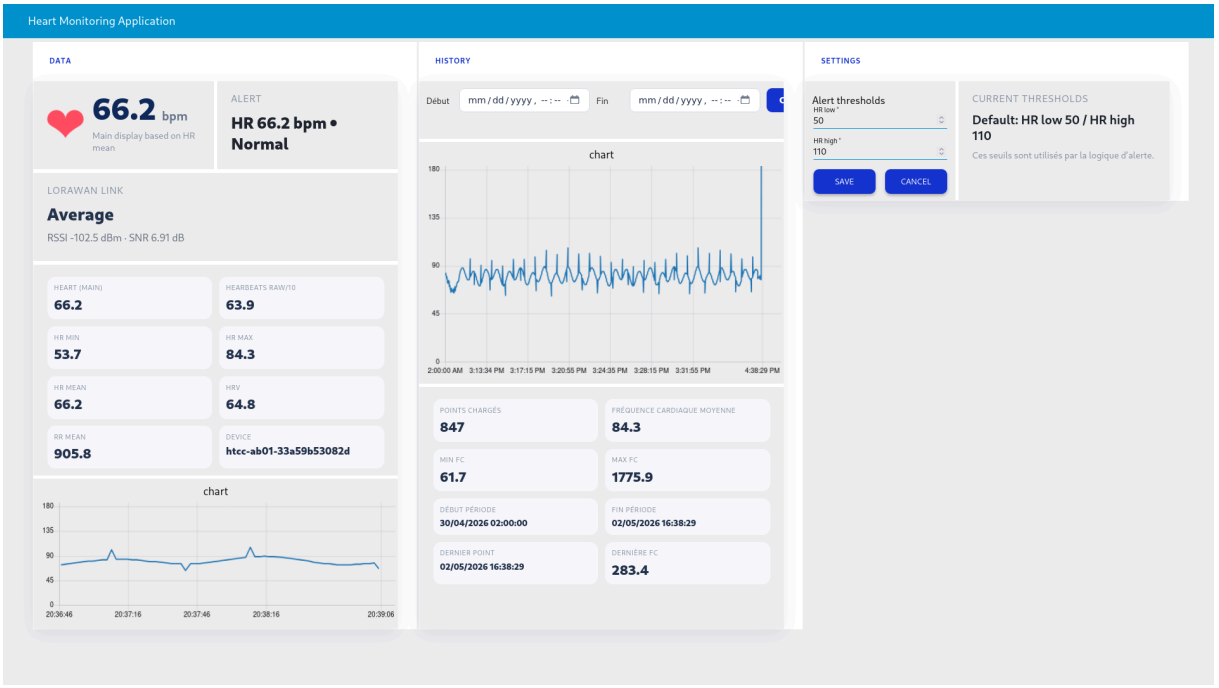
3.7.3.3 Stockage des paramètres

Les seuils sont stockés dans le contexte Node-RED (flow context), ce qui permet :

- Une persistance pendant l'exécution
- Une utilisation globale dans le flow

Voir code en Annexe D - (13)

3.7.4 IHM finale et architecture globale



L'architecture du système peut être résumée comme suit :

Capteur ECG → LoRaWAN → The Things Network → MQTT → Node-RED → SQLite → Dashboard

Cette organisation permet une séparation claire entre :

- Acquisition
- Traitement
- Stockage
- Visualisation

La mise en œuvre de Node-RED m'a permis de développer une chaîne complète de traitement des données ECG, allant de la réception des données jusqu'à leur visualisation en temps réel et en historique.

L'approche orientée flux offre une grande flexibilité et facilite l'intégration de nouvelles fonctionnalités, telles que la simulation des données ou l'ajout d'une interface utilisateur avancée.

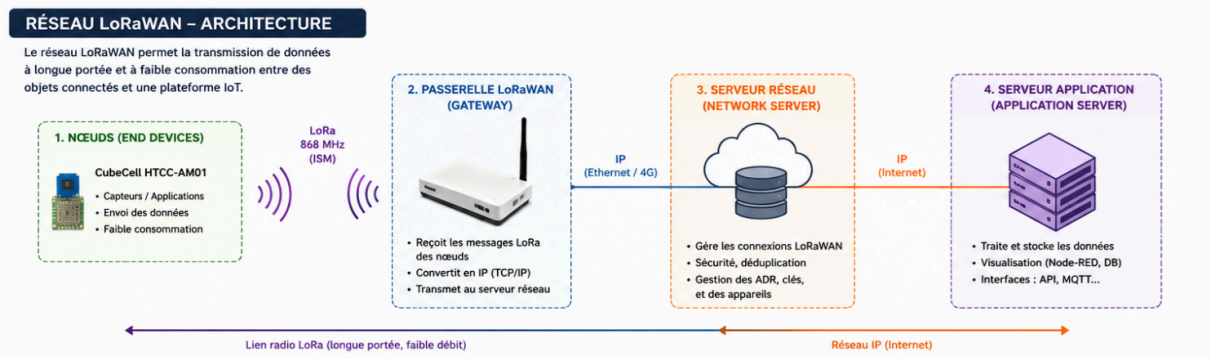
Le système obtenu est à la fois :

- Modulaire
- Évolutif
- Adapté à une utilisation en démonstration ou en conditions réelles

3.8 Partie Physique

Dans le cadre de ma partie du projet, des notions de physique étudiées en BTS CIEL option IR ont été appliquées lors de la transmission des données ECG via le protocole LoRaWAN.

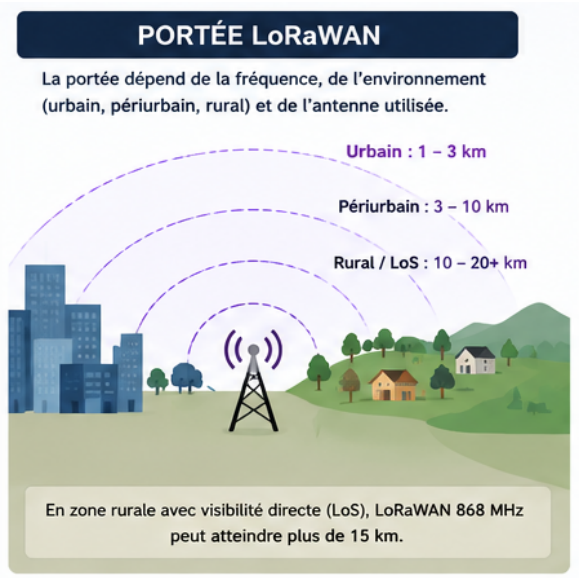
3.8.1 Architecture du réseau LoRaWAN



Ce schéma présente l'architecture générale du réseau LoRaWAN, depuis le module embarqué jusqu'au serveur d'application, en passant par la passerelle et le réseau IP.

Après le traitement du signal ECG et le calcul des métriques physiologiques (fréquence cardiaque, HRV, intervalles RR), les données sont converties en une trame hexadécimale puis envoyées au module CubeCell HTCC-AM01. Celui-ci transmet les informations par ondes radio à une fréquence de 868 MHz vers une passerelle LoRaWAN.

3.8.2 Propagation radio et contraintes de transmission



La transmission met en œuvre plusieurs notions de physique liées aux ondes électromagnétiques, notamment la propagation radio, l'atténuation du signal ainsi que les contraintes de puissance d'émission.

Les essais réalisés durant le projet ont montré que la qualité de la transmission dépend fortement de l'environnement, des obstacles et de la distance entre le module et la passerelle.

3.8.3 Relation entre fréquence et longueur d'onde

Dans le cadre du projet, le module CubeCell communique via le protocole LoRaWAN sur la bande ISM 868 MHz.

En physique, une onde électromagnétique est caractérisée notamment par sa fréquence et sa longueur d'onde. Ces deux grandeurs sont liées par la relation :

$$\lambda = \frac{c}{f}$$

avec :

- λ: longueur d'onde (m),
- c: vitesse de la lumière dans le vide ($3times10^8, m/s$),
- f: fréquence du signal (Hz).

Dans le cas du projet :

$$f = 868\times10^6\, Hz$$

On obtient alors :

$$\lambda\approx0.345\, m$$

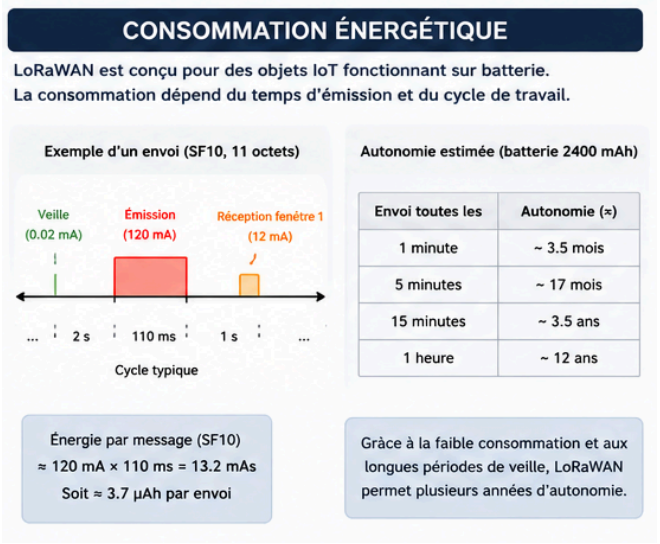
soit une longueur d'onde d'environ **34,5 cm**

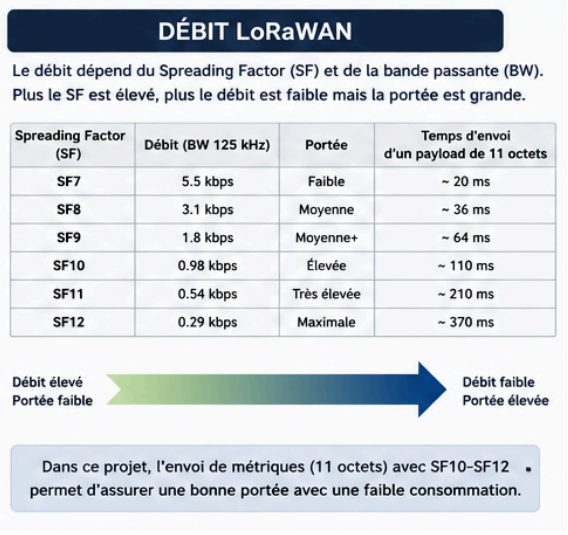
Détail : La Passerelle pour lier le Cube Cell à internet (et donc le cloud) a une longueur d'antenne de ~17cm après avoir été mesuré à la main.

Cela veut donc dire que **nous utilisions une antenne demi-onde,** car :

$$L = \frac{\lambda}{2} = \frac{34.5}{2} = 17.25cm$$

3.8.4 Compromis entre portée, débit et consommation énergétique





Le faible débit offert par le protocole LoRaWAN illustre également les compromis étudiés en physique entre portée, débit et consommation énergétique. Cette contrainte a eu un impact direct sur ma partie du projet, car il n'était pas possible de transmettre le signal ECG brut complet. J'ai donc dû mettre en

place un traitement local permettant de calculer uniquement les métriques essentielles avant transmission.

On peut remarquer que plus le Spreading Factor (SF) est élevé, plus la portée augmente, mais au détriment du débit et du temps d'émission.

3.8.5 Modulation Chirp Spread Spectrum (CSS)

Le protocole LoRa utilise une modulation radio spécifique appelée *Chirp Spread Spectrum* (CSS), permettant d'améliorer la robustesse des transmissions face au bruit et aux perturbations électromagnétiques, tout en conservant une faible consommation énergétique adaptée aux systèmes embarqués et IoT, comme indiqué dans le schéma de la consommation énergétique.

3.9 Conclusion

Pour conclure, cette partie du projet touche bientôt à sa fin, le programme de traitement et tableau de bord étant terminé et performants toutes les tâches demandées dans le cahier des charges.

En revanche, des objectifs comme l'intégration du Node-RED sur carte Raspberry, ou le passage à l'environnement de production (une Raspberry Pi 5) restent à compléter.

Ce projet m'a alors permis l'opportunité d'en apprendre plus sur plusieurs sujets, comme la communication IoT, ou encore la transmission de données couche-par-couche, du module CubeCell physique, au cloud TTN en ligne.

ANNEXES TECHNIQUES

Simulateur ECG — Ivan GOUIN — BTS CIEL IR 2

ANNEXE A
Environnement Python : installation et bibliothèques utilisées

L'environnement virtuel Python permet d'isoler les dépendances d'un projet et de définir précisément les librairies ainsi que leurs versions utilisées pour le développement.

L'utilisation d'un environnement est avantageux, surtout pour un programme situé dans un Raspberry Pi, car cela empêche les conflits de versions librairies déjà installées, ou la probabilité qu'une bibliothèque soit obsolète.

Voici les commandes suivantes pour créer l'environnement python utilisé

```
sudo apt install python3.13-venv
```

```
mkdir nom_du_repertoire_environement
```

```
python3 -m venv nom_du_repertoire_environement
```

Ces bibliothèques couvrent l'ensemble des besoins du projet, allant du traitement du signal à la communication réseau et matérielle.

Sqlite3 : Base de données SQL locale stockée dans un simple fichier.

Numpy : Calcul matriciel et manipulation de tableaux.

Scipy : Algorithmes avancés pour les sciences (calcul intégral, stats).

Asyncio : Exécution de tâches simultanées.

Websockets : Communication bidirectionnelle en temps réel (réseau).

Json : Lecture et écriture de données au format texte JSON.

Struct : Conversion de données Python en données binaires (octets/bytes).

Pyserial : Communication avec du matériel via les ports série (USB/COM).

Time : Gestion des pauses et mesure du temps écoulé.

Dataclasses : Création simplifiée de classes dédiées au stockage de données.

Pytest : Sert à programmer une 'checklist' afin de s'assurer du bon fonctionnement d'un programme.

Les bibliothèques Python comme : sqlite3, asyncio, json, struct, time, dataclasses

Sont internes/natives à Python, ce qui veut dire que nous devons seulement lancer la commande suivante

> pip install numpy scipy websockets pyserial

ANNEXE B
Codes application Traitement et analyse

1) Fichier config.py

```
PORT = "/dev/ttyUSB0"

BAUDRATE = 115200

SERVER_URI = "ws://100.71.80.116:8000"

FS = 360

# Fichier de test local

TEST_DB_PATH = "/ecg-data/Anonyme_20260323_092202.db"
```

2) Fichier app - main.py

```
import asyncio

from config import PORT, BAUDRATE, SERVER_URI, FS, TEST_DB_PATH

from controller.ecg_controller import EcgController

from model.ecg_analyzer import EcgAnalyzer

from service.database_service import DatabaseService

from service.serial_service import SerialService

from service.websocket_service import WebSocketService

from view.console_view import ConsoleView


def build_app():

    db_service = DatabaseService()

    analyzer = EcgAnalyzer(frequence_echantillonnage=FS)

    serial_service = SerialService(port=PORT, baudrate=BAUDRATE)

    view = ConsoleView()

    controller = EcgController(

        db_service=db_service,

        analyzer=analyzer,
```

```
        serial_service=serial_service,
        view=view,
    )

    websocket_service = WebSocketService(server_uri=SERVER_URI)
    return controller, websocket_service
async def run_websocket():
    controller, websocket_service = build_app()
    await websocket_service.ecouter(controller.traiter_fichier_async)
def run_test_local():
    controller, _ = build_app()
    controller.traiter_fichier(TEST_DB_PATH)


if __name__ == "__main__":
    # Choisir un mode :
    # 1) Test local direct
   # run_test_local()

    # 2) Mode websocket
     asyncio.run(run_websocket())
```

3) Ficher ecg_metrics.py

```
class EcgMetrics:
    fc_moyenne: float
    fc_max: float
    fc_min: float
    rr_moyen: float
    hrv_rmssd: float
    heartbeats: int
    hex_string: str | None = None
```

4) Ficher hex_encoder.py

```
import struct

class HexEncoder:

    @staticmethod

    def convertir_en_hex_11_octets(fc_moy, fc_max, fc_min, rr_moy, hrv,
heartbeats):

        """

        Format binaire attendu :

        <HHHHHB

        - fc_moy  : uint16 little-endian

        - fc_min  : uint16 little-endian

        - fc_max  : uint16 little-endian

        - rr_moy  : uint16 little-endian

        - hrv*10  : uint16 little-endian

        - beats   : uint8

        """

        try:

            paquet = struct.pack(

                "<HHHHHB",

                int(fc_moy),

                int(fc_min),

                int(fc_max),

                int(rr_moy),

                int(round(hrv * 10)),

                int(heartbeats),

            )

            return paquet.hex().upper()

        except Exception as e:

            raise ValueError(f"Erreur conversion hex : {e}") from e
```

5) Fichier ecg_analyzer.py

```
import numpy as np
from scipy.signal import butter, filtfilt, find_peaks
from model.ecg_metrics import EcgMetrics
from model.hex_encoder import HexEncoder


class EcgAnalyzer:
    def __init__(self, frequence_echantillonnage=360):
        self.fs = frequence_echantillonnage
    def filtrer_signal(self, signal):
        """
        Filtre passe-haut pour supprimer la dérive de ligne de base.
        """
        b, a = butter(2, 0.5 / (self.fs / 2), btype="high")
        return filtfilt(b, a, signal)
    def detecter_pics(self, signal, taille_fenetre_s=5, overlap=0.5):
        """
        Détection adaptative par fenêtres glissantes.
        """
        taille_fenetre = int(taille_fenetre_s * self.fs)
        if len(signal) < taille_fenetre:
            taille_fenetre = len(signal)

        if taille_fenetre <= 0:
            return np.array([])

        pas = max(1, int(taille_fenetre * (1 - overlap)))
        tous_les_pics = []
        debuts = list(range(0, max(1, len(signal) - taille_fenetre + 1), pas))
        derniere_debut = len(signal) - taille_fenetre
        if derniere_debut >= 0 and (not debuts or debuts[-1] != derniere_debut):
```

```
            debuts.append(derniere_debut)
        for debut in debuts:
            fenetre = signal[debut: debut + taille_fenetre]
            if len(fenetre) < 3:
                continue
            freqs = np.fft.rfftfreq(len(fenetre), d=1 / self.fs)
            fft_amp = np.abs(np.fft.rfft(fenetre))
            masque = (freqs >= 0.8) & (freqs <= 3.5)
            if np.any(masque):
                freq_locale = freqs[masque][np.argmax(fft_amp[masque])]
                distance_locale = int((self.fs / freq_locale) * 0.75)
            else:
                distance_locale = int(self.fs * 0.4)
            distance_locale = max(distance_locale, int(self.fs * 0.2))
            seuil_local = np.mean(fenetre) + 0.6 * np.std(fenetre)
            pics_locaux, _ = find_peaks(
                fenetre,
                distance=distance_locale,
                height=seuil_local,
            )

            for p in pics_locaux:
                idx_global = debut + p
                if (
                    not tous_les_pics
                    or (idx_global - tous_les_pics[-1]) > int(self.fs * 0.2)
                ):
                    tous_les_pics.append(idx_global)

        return np.array(sorted(set(tous_les_pics)))
    def analyser(self, donnees):
        if len(donnees) < 10:
```

```
            raise ValueError("Pas assez de données pour analyse.")
        signal = np.array(donnees, dtype=float)
        signal_filtre = self.filtrer_signal(signal)
        pics = self.detecter_pics(signal_filtre)

        if len(pics) < 2:
            raise ValueError("Pas assez de battements détectés.")

        intervalles_rr_s = np.diff(pics) / self.fs
        intervalles_rr_ms = intervalles_rr_s * 1000
        frequences_bpm = 60 / intervalles_rr_s
        fc_moyenne = float(np.mean(frequences_bpm))
        fc_max = float(np.max(frequences_bpm))
        fc_min = float(np.min(frequences_bpm))
        rr_moyen = float(np.mean(intervalles_rr_ms))
        hrv_rmssd = float(np.sqrt(np.mean(np.square(np.diff(intervalles_rr_ms))))) 
if len(intervalles_rr_ms) > 1 else 0.0
        heartbeats = int(len(pics))

        hex_string = HexEncoder.convertir_en_hex_11_octets(
            fc_moyenne, fc_max, fc_min, rr_moyen, hrv_rmssd, heartbeats
        )
        return EcgMetrics(
            fc_moyenne=fc_moyenne,
            fc_max=fc_max,
            fc_min=fc_min,
            rr_moyen=rr_moyen,
            hrv_rmssd=hrv_rmssd,
            heartbeats=heartbeats,
            hex_string=hex_string,
        )
```

6) Fichier console_view.py

```
class ConsoleView:

    def afficher_analyse(self, filepath, metrics: EcgMetrics):

        print(f"\nAnalyse du fichier : {filepath}")

        print("\n--- Résultats ECG ---")

        print(f"FC moyenne  : {metrics.fc_moyenne:.1f} BPM")

        print(f"FC max      : {metrics.fc_max:.1f} BPM")

        print(f"FC min      : {metrics.fc_min:.1f} BPM")

        print(f"RR moyen    : {metrics.rr_moyen:.1f} ms")

        print(f"HRV RMSSD   : {metrics.hrv_rmssd:.2f} ms")

        print(f"Battements  : {metrics.heartbeats}")

        print(f"Hex (11 B)  : {metrics.hex_string}")

        print("----------------------\n")

    def afficher_reponse_module(self, response):

        if response:

            print("--- Réponse reçue ---")

            print(response)

            print("----------------------")

            if "OK" in response:

                print("Le module répond parfaitement !")

        else:

            print("Le module n'a pas répondu à la commande.")


    def afficher_erreur(self, erreur):

        print(f"Erreur : {erreur}")
```

7) Fichier ecg_controller.py

```
class EcgController:

    def __init__(self, db_service, analyzer, serial_service, view,
websocket_service=None):

        self.db_service = db_service

        self.analyzer = analyzer

        self.serial_service = serial_service

        self.view = view

        self.websocket_service = websocket_service # Ajout du service

    async def traiter_fichier_async(self, filepath):

        try:

            # 1. Analyse

            donnees = self.db_service.lire_amplitudes(filepath)

            metrics = self.analyzer.analyser(donnees)

            # 2. Affichage Console

            self.view.afficher_analyse(filepath, metrics)


            # 3. Envoi au serveur WebSocket (NOUVEAU)

            if self.websocket_service:

                await self.websocket_service.envoyer_metriques(metrics)

            # 4. Envoi au module LoRa/Série

            # Note: envoyer_hex est bloquant (time.sleep).

            # On l'exécute après pour ne pas retarder l'envoi
WebSocket.

            response =
self.serial_service.envoyer_hex(metrics.hex_string)

            self.view.afficher_reponse_module(response)

        except Exception as e:

            self.view.afficher_erreur(e)
```

8) Fichier database_service.py

```
class DatabaseService:

    def lire_amplitudes(self, nom_fichier):

        conn = sqlite3.connect(nom_fichier)

        try:

            cursor = conn.cursor()

            cursor.execute("SELECT amplitude FROM samples ORDER BY id")

            return [row[0] for row in cursor.fetchall()]

        finally:

            conn.close()

class SerialService:

    def __init__(self, port, baudrate, timeout=2):

        self.port = port

        self.baudrate = baudrate

        self.timeout = timeout

    def envoyer_hex(self, hex_string):

        with serial.Serial(self.port, self.baudrate,
timeout=self.timeout) as ser:

            print("Envoi de la commande AT...")

            ser.write(b"AT+AutoLPM=0")

            time.sleep(3)

            ser.write(b"AT+Join=1")

            time.sleep(15)


            ser.write(b"AT+SendHex=" + hex_string.encode())

            time.sleep(5)

            if ser.in_waiting > 0:

                return ser.read(ser.in_waiting).decode('iso-8859-1')

            return None
```

9) Fichier serial_service.py

```
import time
import serial
class SerialService:
    def __init__(self, port, baudrate, timeout=2):
        self.port = port
        self.baudrate = baudrate
        self.timeout = timeout

    def envoyer_commande(self, ser, commande: bytes, pause_s: float):
        ser.write(commande)
        time.sleep(pause_s)

    def envoyer_hex(self, hex_string):
        try:
            with serial.Serial(self.port, self.baudrate,
timeout=self.timeout) as ser:
                print("Envoi de la commande AT...")

                self.envoyer_commande(ser, b"AT+AutoLPM=0", 3)
                self.envoyer_commande(ser, b"AT+Join=1", 15)
                self.envoyer_commande(ser, b"AT+SendHex=" +
hex_string.encode(), 5)

                if ser.in_waiting > 0:
                    response =
ser.read(ser.in_waiting).decode("iso-8859-1")
                    return response
                return None
        except Exception as e:
            raise RuntimeError(f"Erreur série : {e}") from e
```

10) Fichier websocket_service.py

```
class WebSocketService:

    def __init__(self, server_uri):

        self.server_uri = server_uri

        self.ws = None  # Stockage de la connexion active

    async def ecouter(self, callback_async):

        print("Connexion au serveur ECG...")

        async with websockets.connect(self.server_uri) as ws:
```

```
            self.ws = ws  # On garde la main sur le socket
            print("Connecté au serveur")
            while True:
                try:
                    message = await ws.recv()
                    data = json.loads(message)
                except Exception:
                    continue

                if data.get("action") == "GO":
                    filepath = data.get("file")
                    print("Ordre reçu :", filepath)
                    # On passe le service websocket au callback si
besoin,
                    # ou on gère via le contrôleur
                    await callback_async(filepath)

    async def envoyer_metriques(self, metrics: EcgMetrics):
        """Envoie les résultats au serveur WebSocket sous format
JSON."""
        if self.ws:
            payload = {
                "action": "RESULTATS",
                "data": {
                    "fc_moyenne": round(metrics.fc_moyenne, 1),
                    "fc_max": round(metrics.fc_max, 1),
                    "fc_min": round(metrics.fc_min, 1),
                    "rr_moyen": round(metrics.rr_moyen, 1),
                    "hrv_rmssd": round(metrics.hrv_rmssd, 2),
                    "heartbeats": metrics.heartbeats,
```

```
            "hex_string": metrics.hex_string
        }
    }
    await self.ws.send(json.dumps(payload))
    print("Résultats envoyés au serveur WebSocket.")
```

ANNEXE C
Tests unitaires

emplacement

application-de-traitement/ecg_app/test_unit.py

lancement

pytest -v

Code

```
import numpy as np
import pytest
from model.ecg_analyzer import EcgAnalyzer

def generer_signal_pics(fs, duree_s, bpm):
    signal = np.zeros(fs * duree_s)

    intervalle_s = 60 / bpm
    t = intervalle_s

    while t < duree_s:
        index = int(t * fs)
        signal[index] = 1.0
        t += intervalle_s

    return signal


def test_analyse_signal_ecg_60_bpm():
    fs = 360
    signal = generer_signal_pics(fs, duree_s=10, bpm=60)

    analyzer = EcgAnalyzer(frequence_echantillonnage=fs)
    metrics = analyzer.analyser(signal)

    assert metrics.fc_moyenne == pytest.approx(60, abs=5)
    assert metrics.rr_moyen == pytest.approx(1000, abs=100)
    assert metrics.heartbeats >= 8
    assert metrics.hex_string is not None
```

```
def test_analyse_signal_ecg_120_bpm():
    fs = 360
    signal = generer_signal_pics(fs, duree_s=10, bpm=120)
    analyzer = EcgAnalyzer(frequence_echantillonnage=fs)
    metrics = analyzer.analyser(signal)
    assert metrics.fc_moyenne == pytest.approx(120, abs=10)
    assert metrics.rr_moyen == pytest.approx(500, abs=100)
    assert metrics.heartbeats >= 15

def test_signal_trop_court():
    analyzer = EcgAnalyzer()
    with pytest.raises(ValueError, match="Pas assez de données"):
        analyzer.analyser([0, 1, 0])

def test_signal_plat_sans_battement():
    analyzer = EcgAnalyzer()
    signal = np.zeros(360 * 5)
    with pytest.raises(ValueError, match="Pas assez de battements
détectés"):
        analyzer.analyser(signal)

def test_hrv_rythme_regulier():
    fs = 360
    signal = generer_signal_pics(fs, duree_s=10, bpm=60)
    analyzer = EcgAnalyzer(frequence_echantillonnage=fs)
    metrics = analyzer.analyser(signal)
    assert metrics.hrv_rmssd == pytest.approx(0, abs=5)

def test_generation_hex():
    fs = 360
    signal = generer_signal_pics(fs, duree_s=10, bpm=60)
    analyzer = EcgAnalyzer(frequence_echantillonnage=fs)
    metrics = analyzer.analyser(signal)
    assert isinstance(metrics.hex_string, str)
    assert len(metrics.hex_string) > 0
```

Résultat Obtenu :

```
(ecg-env) pi@rpi-holter-ecg:~/projet-bts-ciel-ecg-2026/application-de-traitement/ecg_app $ ls
config.py  controller  main.py  model  programme_mvc_solo.py  __pycache__  README.md  requirements.txt  service  test_unit.py  view
(ecg-env) pi@rpi-holter-ecg:~/projet-bts-ciel-ecg-2026/application-de-traitement/ecg_app $ pytest -v
============================================================ test session starts =============================================================
platform linux -- Python 3.13.5, pytest-9.0.3, pluggy-1.6.0 -- /home/pi/ecg-env/bin/python3
cachedir: .pytest_cache
rootdir: /home/pi/projet-bts-ciel-ecg-2026/application-de-traitement/ecg_app
plugins: anyio-4.13.0
collected 6 items

test_unit.py::test_analyse_signal_ecg_60_bpm PASSED                                                                                  [ 16%]
test_unit.py::test_analyse_signal_ecg_120_bpm PASSED                                                                                 [ 33%]
test_unit.py::test_signal_trop_court PASSED                                                                                          [ 50%]
test_unit.py::test_signal_plat_sans_battement PASSED                                                                                 [ 66%]
test_unit.py::test_hrv_rythme_regulier PASSED                                                                                        [ 83%]
test_unit.py::test_generation_hex PASSED                                                                                             [100%]

============================================================= 6 passed in 0.93s ==============================================================
```

ANNEXE D
Visualisation / Supervision

1) JS - normalize payload

```
const root = msg.payload || {};
const uplink = root.uplink_message || {};
const d = uplink.decoded_payload || {};
const settings = flow.get('ecg_settings') || { hr_low: 50, hr_high: 110
};

function pick(...vals) {
    for (const v of vals) {
        if (v !== undefined && v !== null && v !== "") return v;
    }
    return null;
}
function scale10(v) {
    if (v === null || v === undefined || v === "") return null;
    return Number(v) / 10;
}

// For main display, prefer HRmean which is more stable/plausible than
HearBeats in this dataset.
const heart_rate = scale10(pick(d.HRmean, d.HearBeats, d.heart,
d.heart_rate, d.hr));
const hear_beats = scale10(pick(d.HearBeats));

const hr_min     = scale10(pick(d.HRmin, d.hr_min));
const hr_max     = scale10(pick(d.HRmax, d.hr_max));
const hr_mean    = scale10(pick(d.HRmean, d.hr_mean));
const hrv        = scale10(pick(d.HRV, d.hrv));
const rr_mean    = scale10(pick(d.RRmean, d.rr_mean));

const rssi = uplink.rx_metadata?.[0]?.rssi ?? null;
const snr  = uplink.rx_metadata?.[0]?.snr ?? null;

const device_id = root.end_device_ids?.device_id || "unknown";
```

```
const ts = pick(root.received_at, uplink.received_at, new
Date().toISOString());

let lora_state = "Unknown";
if (rssi !== null && snr !== null) {
    if (rssi > -95 && snr > 5) lora_state = "Good";
    else if (rssi > -110 && snr > 0) lora_state = "Average";
    else lora_state = "Weak";
}

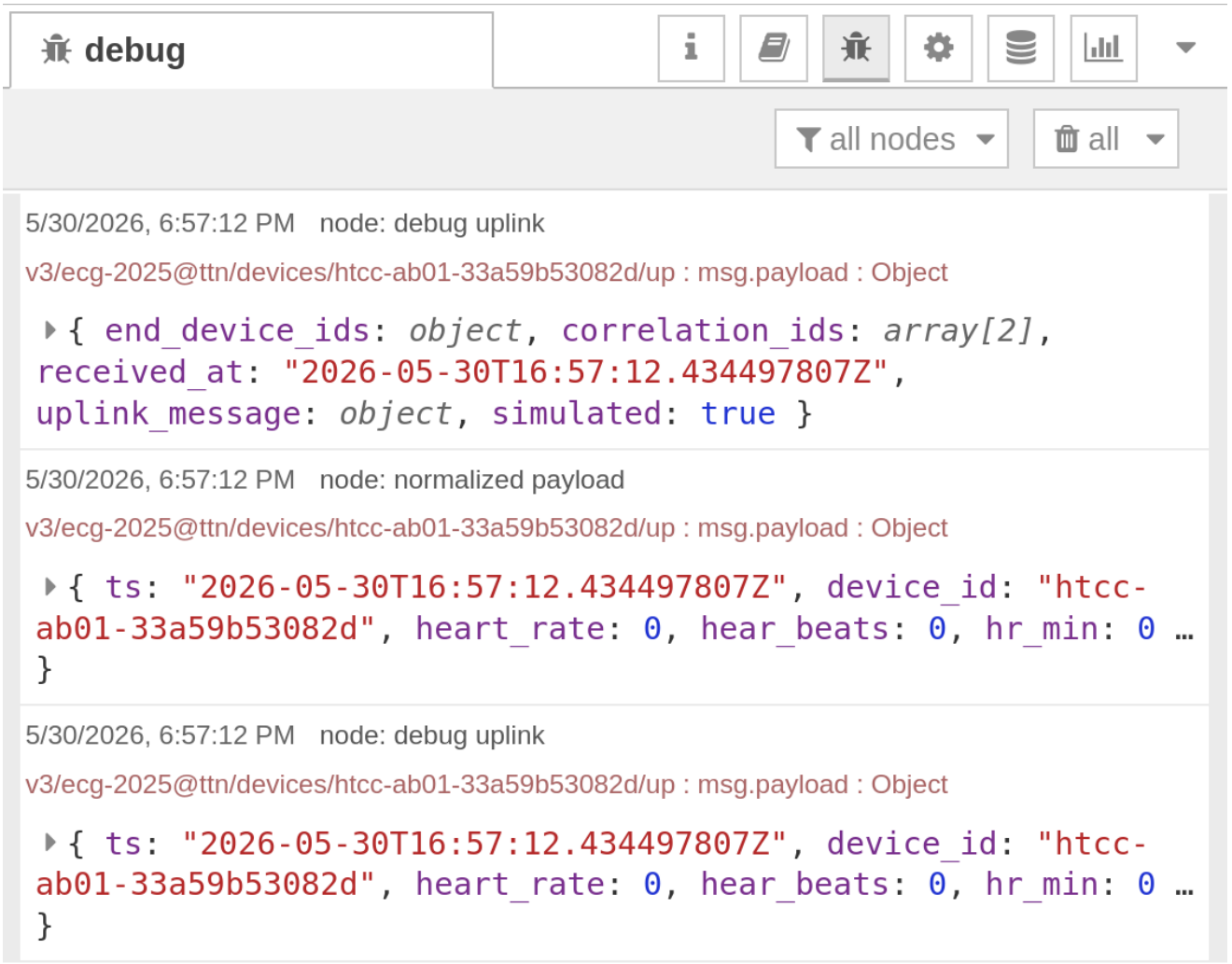
let issues = [];
if (heart_rate !== null) {
    if (heart_rate < settings.hr_low) issues.push("Bradycardia");
    if (heart_rate > settings.hr_high) issues.push("Tachycardia");
}

const alert_text = issues.length
    ? `HR ${heart_rate} bpm • ⚠ ${issues.join(" | ")}`
    : `HR ${heart_rate ?? "-"} bpm • Normal`;

msg.payload = {
    ts,
    device_id,
    heart_rate,
    hear_beats,
    hr_min,
    hr_max,
    hr_mean,
    hrv,
    rr_mean,
    rssi,
    snr,
    lora_state,
    alert_text
};

return msg;
```

2) debug uplink



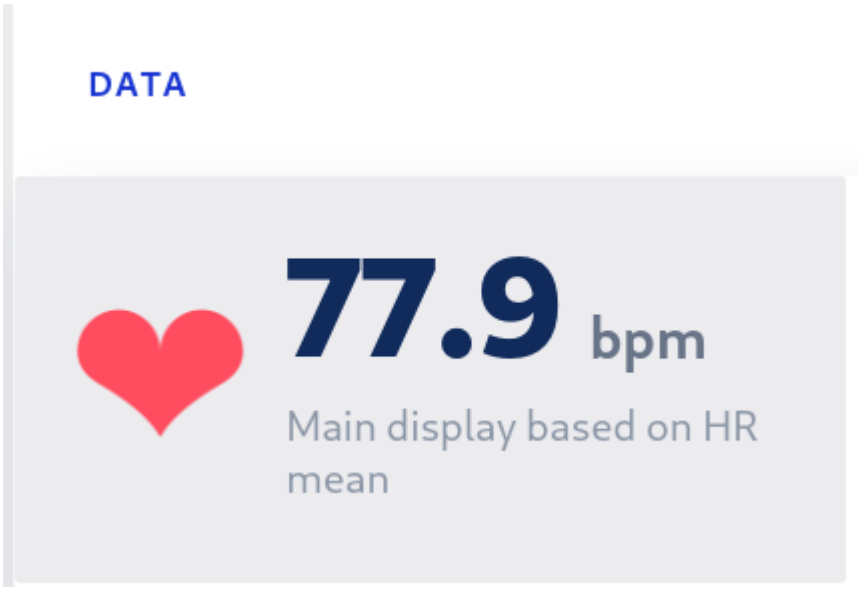
3) HTML - Heart

```
<div class="ecg-card">
  <div class="ecg-row">
    <div class="ecg-heart" id="heart-dynamic">❤</div>
    <div>
      <div class="ecg-row" style="gap:10px;align-items:flex-end;">
        <div class="ecg-value">{{(msg.payload &&
msg.payload.heart_rate) || '--'}}</div>
        <div class="ecg-unit">bpm</div>
      </div>
      <div class="ecg-sub">Main display based on HR mean</div>
    </div>
  </div>
</div>

<style>
.ecg-heart {
  font-size: 76px;
  color: #ff4d5f;
  transform-origin: center;
  display: inline-block;
}
@keyframes beatDynamic {
  0%   { transform: scale(1); }
  20%  { transform: scale(1.18); }
  40%  { transform: scale(1); }
  60%  { transform: scale(1.12); }
  100% { transform: scale(1); }
}
</style>

<script>
(function(scope) {
  scope.$watch('msg', function(msg) {
    if (!msg || !msg.payload) return;
    var hr = Number(msg.payload.heart_rate || 0);
    if (!hr || hr < 30) hr = 60;
    var duration = 60000 / hr;
    var heart = document.getElementById('heart-dynamic');
    if (!heart) return;
    heart.style.animation = 'none';
    setTimeout(function() {
```

```
      heart.style.animation = 'beatDynamic ' + duration + 'ms
infinite';
    }, 10);
  });
})(scope);
</script>
```

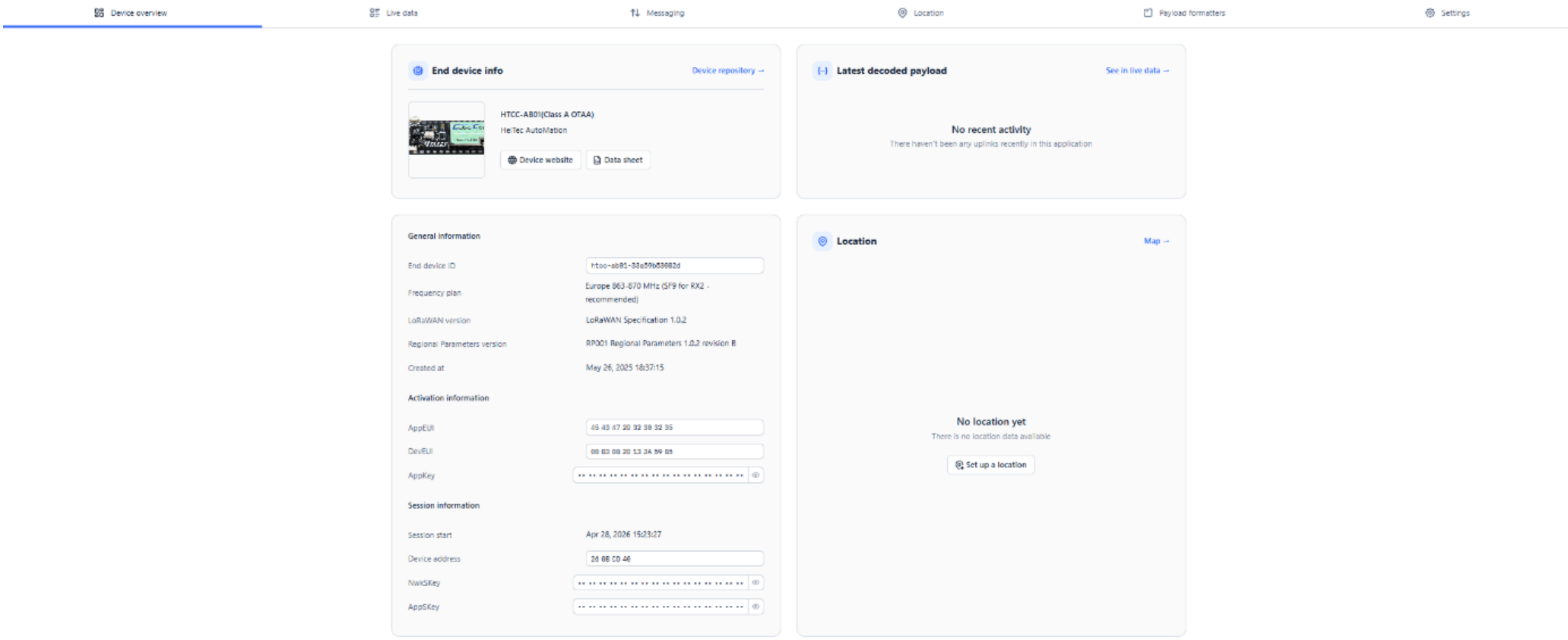


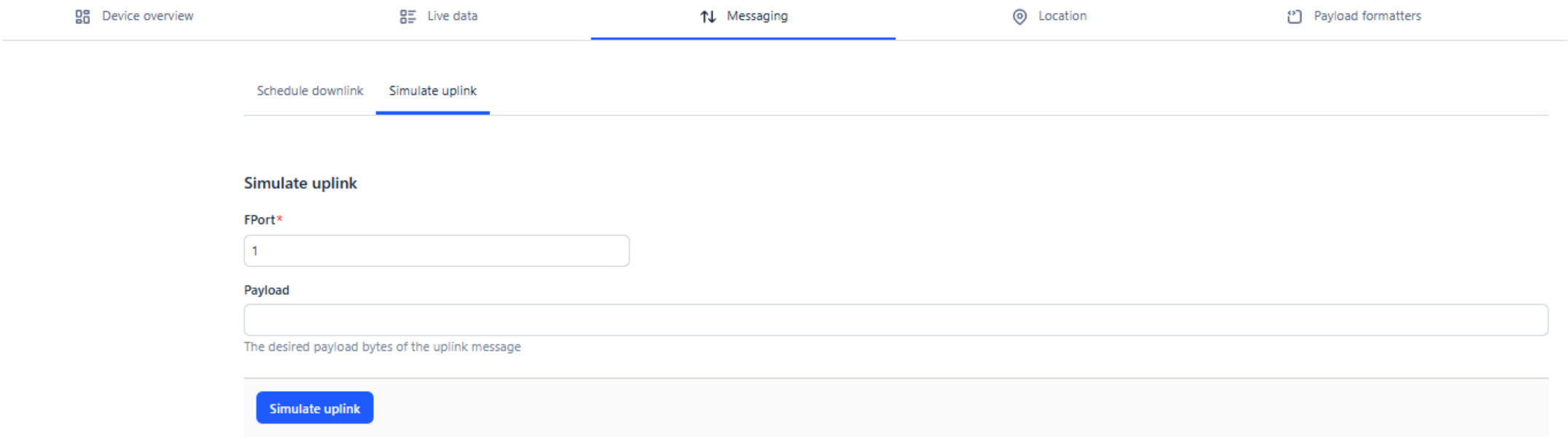
4) JS - build simulated uplink

```
build simulated uplink
let t = flow.get('sim_t') || 0;
t += 1;
flow.set('sim_t', t);
let hrMean = 80 + 6*Math.sin(t/5) + 3*Math.sin(t/11);
if (t % 29 === 0) hrMean += 18;
if (t % 41 === 0) hrMean -= 12;
hrMean = Math.max(55, Math.min(125, hrMean));

let hrMin = Math.max(40, hrMean - (8 + Math.random()*6));
let hrMax = Math.min(170, hrMean + (10 + Math.random()*10));
let hearBeats = hrMean + (Math.random()*8 - 4);
let hrv = Math.max(20, Math.min(180, 75 + 20*Math.sin(t/8) +
(Math.random()*10 - 5)));
let rrMean = 60000 / hrMean;
let rssi = -98 + (Math.random()*10 - 5);
let snr = 5 + (Math.random()*4 - 2);
msg.payload = {
  end_device_ids: {
    device_id: "htcc-ab01-33a59b53082d",
    application_ids: { application_id: "ecg-2025" }
  },
  received_at: new Date().toISOString(),
  uplink_message: {
    decoded_payload: {
      HRV: Math.round(hrv * 10),
      HRmax: Math.round(hrMax * 10),
      HRmean: Math.round(hrMean * 10),
      HRmin: Math.round(hrMin * 10),
      HearBeats: Math.round(hearBeats * 10),
      RRmean: Math.round(rrMean * 10)
    },
    rx_metadata: [{
      rssi: Number(rssi.toFixed(1)),
      snr: Number(snr.toFixed(2))
    }],
    received_at: new Date().toISOString()
  }
};
return msg;
```

5) Simulation TTN




Device overview
Live data
Messaging
Location
Payload formatters
Schedule downlink
Simulate uplink
Simulate uplink
FPort*
1
Payload
The desired payload bytes of the uplink message
Simulate uplink

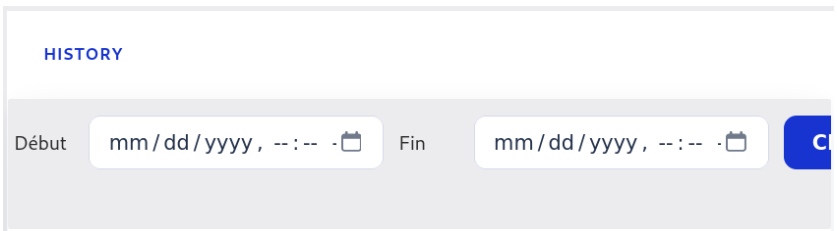
6) JS - build INSERT SQLite

```
const p = msg.payload;
function esc(v){ if (v === null || v === undefined || v === '') return
'NULL'; if (typeof v === 'number') return isFinite(v) ? String(v) :
'NULL'; return "'" + String(v).replace(/'/g, "''") + "'"; }

msg.topic = `INSERT INTO ecg_measurements (ts, device_id, heart_rate,
hr_min, hr_max, hr_mean, hrv, rr_mean, rssi, snr, alert_text,
lora_state) VALUES (${esc(p.ts)}, ${esc(p.device_id)},
${esc(p.heart_rate)}, ${esc(p.hr_min)}, ${esc(p.hr_max)},
${esc(p.hr_mean)}, ${esc(p.hrv)}, ${esc(p.rr_mean)}, ${esc(p.rssi)},
${esc(p.snr)}, ${esc(p.alert_text)}, ${esc(p.lora_state)});`;
return msg;
```

7) HTML - History controls mockup

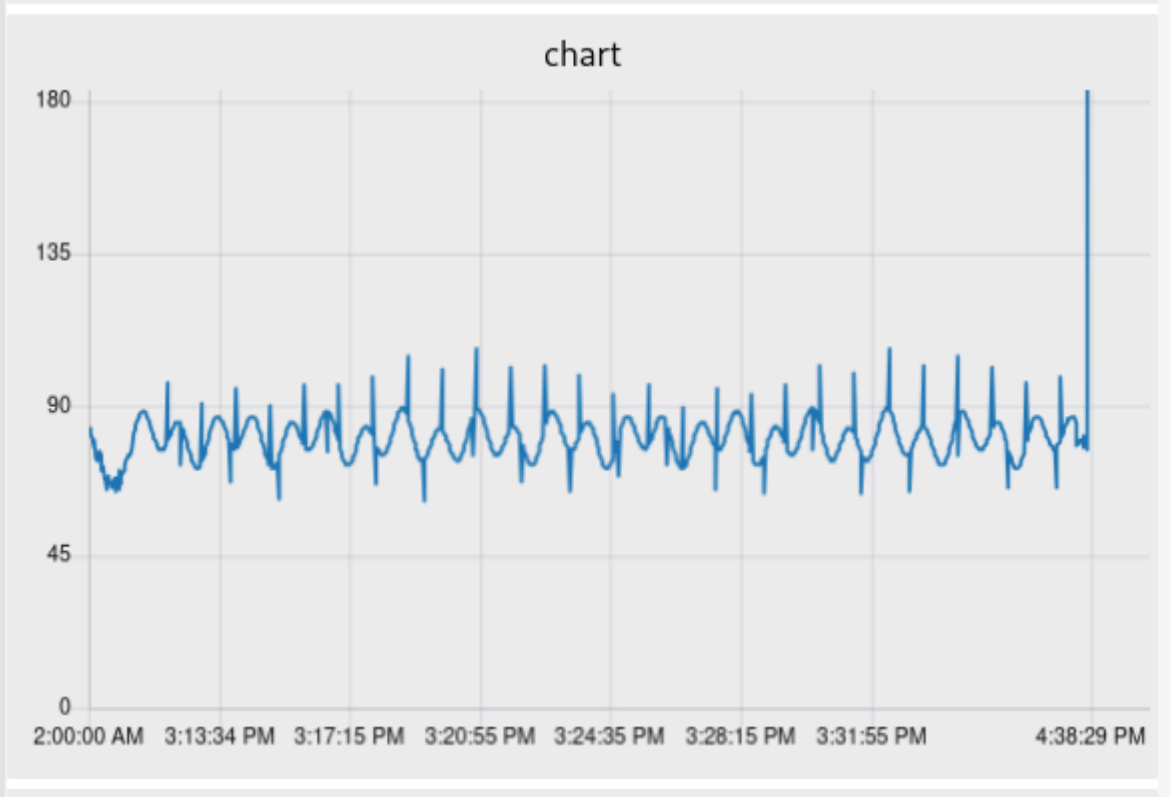
```
<div class="hist-range-wrap" ng-init="
   startVal=''; endVal='';
   fmt=function(v){ if(!v) return ''; var d=new Date(v); var
p=function(x){return ('0'+x).slice(-2)}; return
d.getFullYear()+'-'+p(d.getMonth()+1)+'-'+p(d.getDate())+'T'+p(d.getHou
rs())+':'+p(d.getMinutes()); };
   if(msg && msg.payload && msg.payload.start){
startVal=fmt(msg.payload.start); }
   if(msg && msg.payload && msg.payload.end){
endVal=fmt(msg.payload.end); }
   ">
   <div class="hist-range-label">Début</div>
   <input class="hist-range-input" type="datetime-local"
ng-model="startVal" ng-change="send({topic:'start',
payload:startVal})">
   <div class="hist-range-label">Fin</div>
   <input class="hist-range-input" type="datetime-local"
ng-model="endVal" ng-change="send({topic:'end', payload:endVal})">
   <button class="hist-range-btn" ng-click="send({topic:'load',
payload:{start:startVal,end:endVal}})">CHARGER</button>
</div>
```



8) JS - build SELECT SQLite

```
let range = flow.get('history_range') || {};
let start = range.start ? new Date(range.start) : new Date(Date.now() - 30*60*1000);
let end = range.end ? new Date(range.end) : new Date();
function fmt(d){ const pad = n => String(n).padStart(2,'0'); return d.getFullYear() + '-' + pad(d.getMonth()+1) + '-' + pad(d.getDate()) + ' ' + pad(d.getHours()) + ':' + pad(d.getMinutes()) + ':' + pad(d.getSeconds()); }
msg.topic = `SELECT ts, heart_rate FROM ecg_measurements WHERE ts BETWEEN '${fmt(start)}' AND '${fmt(end)}' ORDER BY ts ASC;`;
return msg;
```

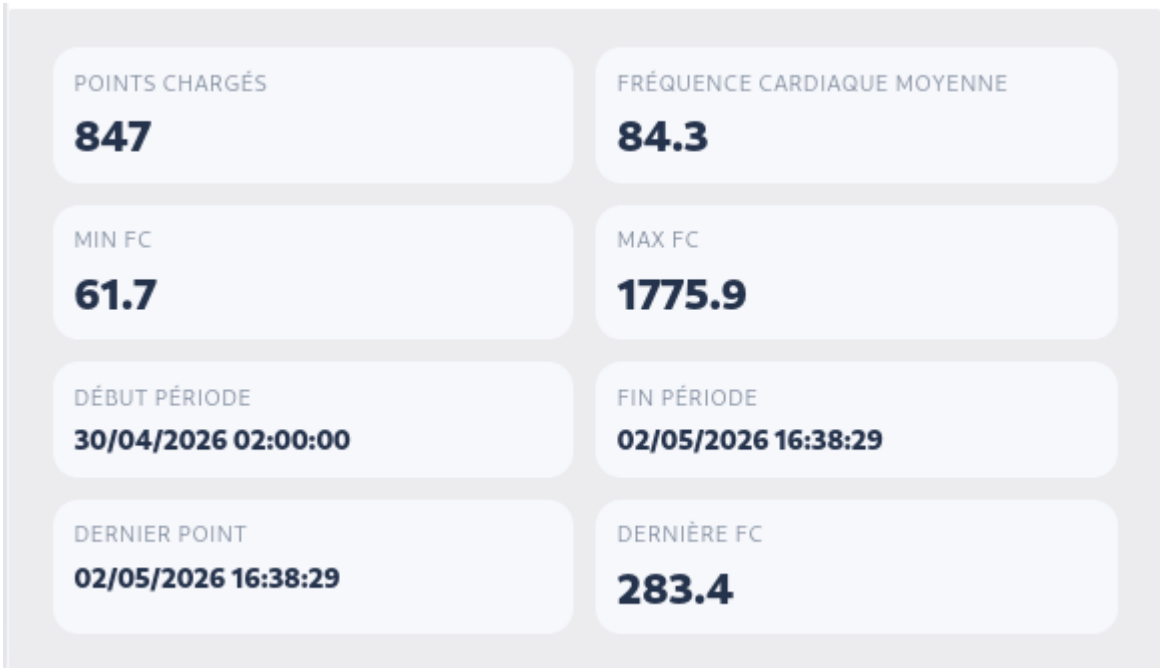
9) History chart



10) HTML - History summary panel

```
<div class='ecg-card'>
   <div class='ecg-grid'>
      <div class='ecg-mini'>
         <div class='lbl'>Points chargés</div>
         <div class='val'>{{(msg.payload && msg.payload.count)
      </div>
      </div>
      <div class='ecg-mini'>
         <div class='lbl'>Fréquence cardiaque moyenne</div>
         <div class='val'>{{(msg.payload && msg.payload.avg)
      </div>
      </div>
      <div class='ecg-mini'>
         <div class='lbl'>Min FC</div>
         <div class='val'>{{(msg.payload && msg.payload.min)
      </div>
      </div>
      <div class='ecg-mini'>
         <div class='lbl'>Max FC</div>
         <div class='val'>{{(msg.payload && msg.payload.max)
      </div>
      </div>
      <div class='ecg-mini'>
         <div class='lbl'>Début période</div>
         <div class='val'
style='font-size:15px;line-height:1.25'>{{(msg.payload &&
msg.payload.start) || '--'}}</div>
      </div>
      <div class='ecg-mini'>
         <div class='lbl'>Fin période</div>
         <div class='val'
style='font-size:15px;line-height:1.25'>{{(msg.payload &&
msg.payload.end) || '--'}}</div>
      </div>
      <div class='ecg-mini'>
         <div class='lbl'>Dernier point</div>
         <div class='val'
style='font-size:15px;line-height:1.25'>{{(msg.payload &&
msg.payload.last_time) || '--'}}</div>
      </div>
      <div class='ecg-mini'>
```

```
        <div class='lbl'>Dernière FC</div>
        <div class='val'>{{(msg.payload && msg.payload.last_hr) ||
'--'}}</div>
      </div>
   </div>
</div>
```



11) HTML - Settings note

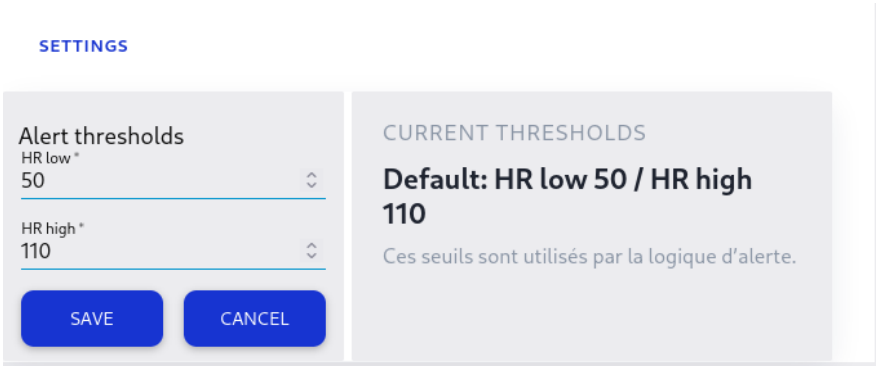
```
<div class='ecg-card'>

   <div class='ecg-muted'>Current thresholds</div>

   <div
style='font-size:22px;font-weight:700;color:#1f2430;'>{{msg.payload ||
'Default: HR low 50 / HR high 110'}}</div>

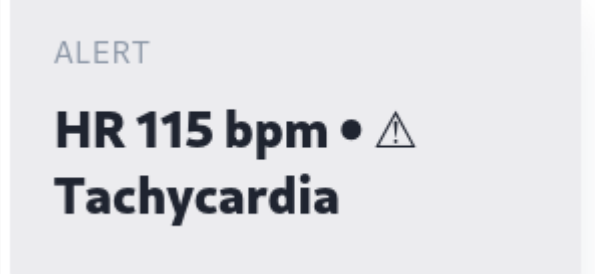
   <div class='ecg-sub'>Ces seuils sont utilisés par la logique
d’alerte.</div>

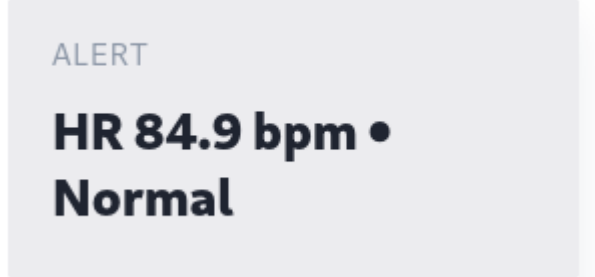
</div>
```

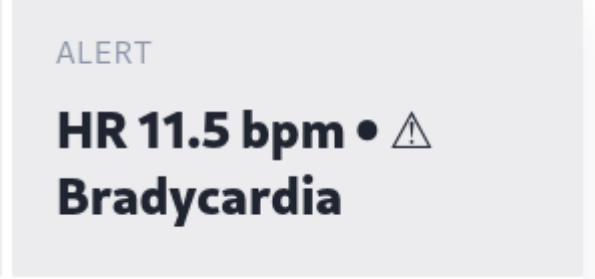


12) HTML - Alert

```
<div class='ecg-card'>
   <div class='ecg-muted'>Alert</div>
   <div class='ecg-alert-main'>{{(msg.payload &&
msg.payload.alert_text) || 'No data'}}</div>
</div>
```







13) JS - save settings

```
flow.set('ecg_settings', { hr_low: Number(msg.payload.hr_low), hr_high:
Number(msg.payload.hr_high) });
msg.payload = `Saved: low=${msg.payload.hr_low},
high=${msg.payload.hr_high}`;
return msg;
```

PARTIE PERSONNELLE
RAPHAEL SABBEN
(IR 3)

4. PARTIE PERSONNELLE - RAPHAËL SABBEN **99**
4.1. Objectifs 99
4.2. Planification prévisionnelle 99
4.3. Architecture système et choix matériels (Hardware) 101
4.3.1. Le microcontrôleur : Raspberry Pi Pico (RP2040) 101
4.3.2. Choix du DAC8551 pour la génération analogique 101
4.3.3. Protocole matériel SPI vers le DAC8551 101
4.3.4. La chaîne de communication physique (8N1) 101
4.3.5. Métrologie : Oscilloscope Analog Discovery / WaveForms 101
4.4. Modélisation SysML du système 102
4.4.1. Diagramme de cas d'utilisation (Use Case UML) 102
4.4.2. Diagramme de Définition de Blocs (BDD) 102
4.4.3. Diagramme de Bloc Interne (IBD) et Flux 103
4.4.4. Diagramme de séquence UML 104
4.4.5. Diagramme de Classes UML 105
4.5. Développement logiciel : L'interface C++ (Qt 5.15) 106
4.5.1. Rendu graphique médical 106
4.5.2. Algorithme de parsing CSV (jusqu'à 3600 points) 106
4.5.3. Conversion Float → Uint16 avec écrêtage de sécurité 106
4.6. Protocole de communication binaire 106
4.6.1. Sérialisation Little-Endian 106
4.6.2. Trame de communication 106
4.6.3. Checksum Modulo 256 107
4.7. Développement du firmware embarqué (MicroPython / Pico) 108
4.7.1. Assembleur PIO — Génération SPI sans CPU 108
4.7.2. Transfert DMA — 2 canaux chaînés 108
4.7.3. Protocole de communication — Commandes UART 108
4.7.4. Organisation des données médicales : structure en dossiers CSV 108
4.8. Problèmes rencontrés et solutions apportées 109
4.8.1. Crash au deuxième clic sur Envoyer 109
4.8.2. L'interface se gelait pendant l'envoi 109
4.8.3. Le signal restait à 0V sur l'oscilloscope 109
4.8.4. Le signal était bloqué à 3,3V en permanence 109
4.9. Protocoles de tests et résultats métrologiques 110
4.10. Conclusion générale et perspectives d'évolution 111
ANNEXE A — Schéma de Câblage SPI : Raspberry Pi Pico → DAC8551 **113**
ANNEXE B — Code MicroPython Complet : PIO et DMA **113**
ANNEXE C — Code C++ Qt : mainwindow.cpp Complet **117**
ANNEXE D — Tableau de Métrologie et Résultats de Tests **122**
ANNEXE E — Glossaire Technique **123**
ANNEXE F — Récapitulatif des Décisions d'Architecture **124**
ANNEXE G — Environnement de Développement et Outils Utilisés **124**

4. PARTIE PERSONNELLE - Raphaël SABBEN

4.1 Objectifs

Ce projet a été commandé par le CERI (Centre d'Enseignement et de Recherche en Informatique) de l'Université d'Avignon.Le CERI souhaitait disposer d'un simulateur ECG pour ses étudiants, afin de pouvoir tester et former sur le holter sans avoir besoin d'un patient réel.

Ma mission (IR3) consiste à concevoir et développer ce simulateur. Mon travail se décompose en trois axes :

- **Développement de l'interface logicielle :** Développement de l'interface logicielle :
- Application C++/Qt 5.15 sous Linux permettant de charger des fichiers CSV de signaux ECG, de les visualiser et de les envoyer au Raspberry Pi Pico via USB.
- Génération du signal analogique : Firmware MicroPython sur le Raspberry Pi Pico, utilisant le PIO et le DMA du RP2040 pour piloter le DAC8551 via SPI et produire un signal électrique 0–3,3V.
- Sécurisation de la transmission : Protocole de communication binaire avec checksum Modulo 256 pour garantir l'intégrité des données envoyées au Pico.

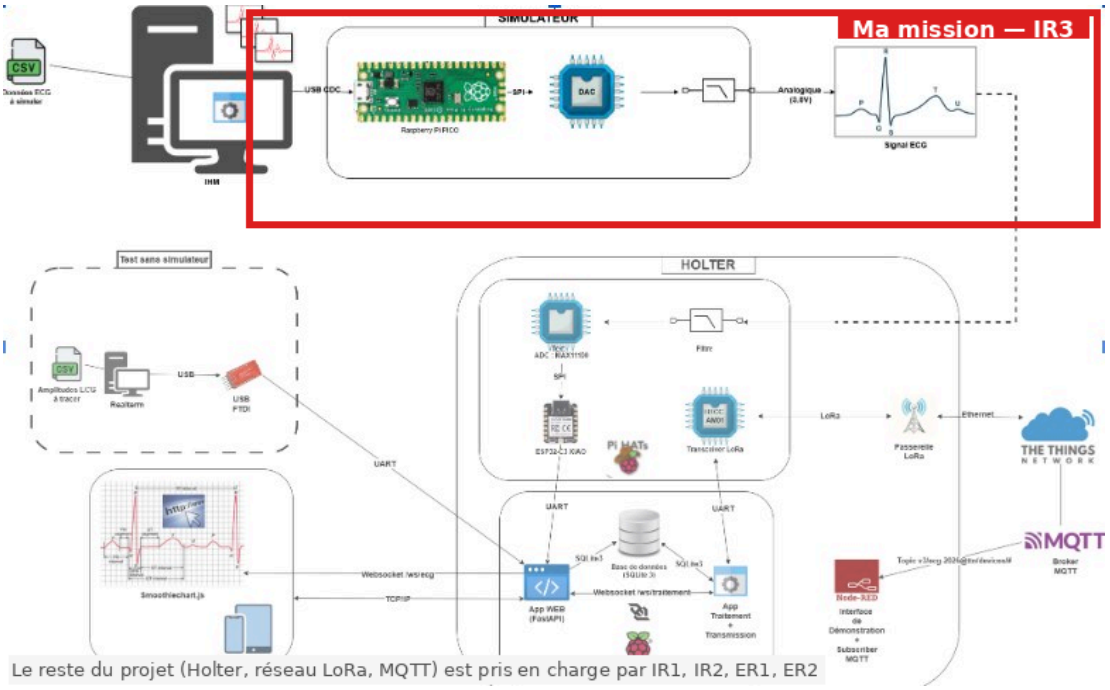
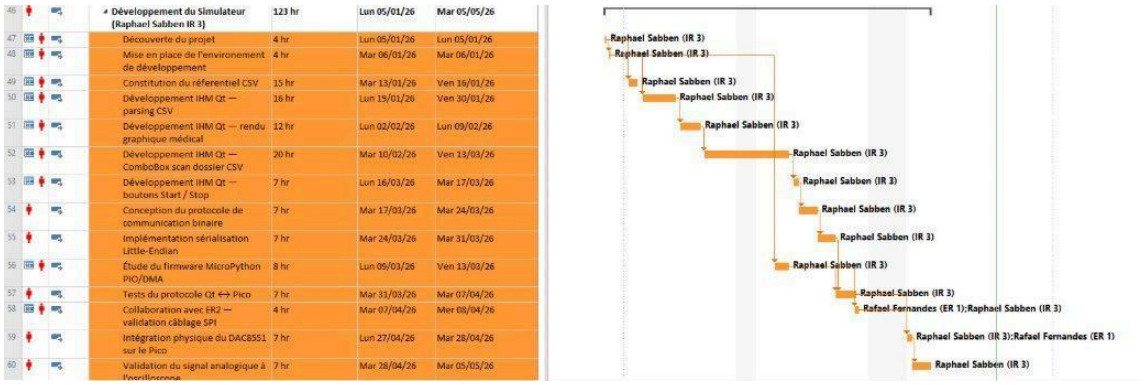


Illustration du synoptique global du projet ECG — ma mission IR3 (Simulateur) est entourée en rouge

4.2 Planification prévisionnelle

Voici la planification prévisionnelle pour l'organisation du projet IR3.



46	Développement du Simulateur (Raphael Sabben IR 3)	123 hr	Lun 05/01/26	Mar 05/05/26
47	Découverte du projet	4 hr	Lun 05/01/26	Lun 05/01/26
48	Mise en place de l'environement de développement	4 hr	Mar 06/01/26	Mar 06/01/26
49	Constitution du référentiel CSV	15 hr	Mar 13/01/26	Ven 16/01/26
50	Développement IHM Qt — parsing CSV	16 hr	Lun 19/01/26	Ven 30/01/26
51	Développement IHM Qt — rendu graphique médical	12 hr	Lun 02/02/26	Lun 09/02/26
52	Développement IHM Qt — ComboBox scan dossier CSV	20 hr	Mar 10/02/26	Ven 13/03/26
53	Développement IHM Qt — boutons Start / Stop	7 hr	Lun 16/03/26	Mar 17/03/26
54	Conception du protocole de communication binaire	7 hr	Mar 17/03/26	Mar 24/03/26
55	Implémentation sérialisation Little-Endian	7 hr	Mar 24/03/26	Mar 31/03/26
56	Étude du firmware MicroPython PIO/DMA	8 hr	Lun 09/03/26	Ven 13/03/26
57	Tests du protocole Qt ↔ Pico	7 hr	Mar 31/03/26	Mar 07/04/26
58	Collaboration avec ER2 — validation câblage SPI	4 hr	Mar 07/04/26	Mer 08/04/26
59	Intégration physique du DAC8551 sur le Pico	7 hr	Lun 27/04/26	Mar 28/04/26
60	Validation du signal analogique à l'oscilloscope	7 hr	Mar 28/04/26	Mar 05/05/26

Figure — Planification prévisionnelle IR3 (extrait du diagramme de Gantt du projet)

4.3 Architecture système et choix matériels (Hardware)

4.3.1 Le microcontrôleur : Raspberry Pi Pico (RP2040)

- Architecture : ARM Cortex-M0+ Dual-Core cadencé à 133 MHz.
- PIO (Programmable I/O) : machines d'états matérielles pour générer le SPI sans CPU.
- DMA (Direct Memory Access) : transferts RAM → périphériques sans intervention du CPU.
- Timers Matériels : échantillonnage à 360 Hz, soit une période de 2,778 ms entre chaque point ECG.

4.3.2 Choix du DAC8551 pour la génération analogique

Le cahier des charges imposait la génération d'un signal analogique via le DAC8551, un convertisseur numérique-analogique 16 bits sur bus SPI. Avant d'intégrer ce composant, j'ai utilisé la PWM interne du Pico (broche GP15, 50 kHz) comme étape de test matériel, afin de valider le câblage et la chaîne de traitement du signal. Ce test a mis en évidence un bruit de commutation clairement visible à l'oscilloscope, ce qui a confirmé que la PWM était bien incompatible avec un usage médical et que le DAC8551, produisant un signal analogique pur sans bruit de commutation, était le choix approprié imposé par le cahier des charges.

Critère	**Génération 1 : PWM**	**Génération 2 : DAC8551**
Principe	Modulation de largeur d'impulsion	Conversion numérique-analogique vraie
Bruit HF	Ripple 50 kHz (rédhibitoire en médical)	Aucun — signal pur
Résolution	16 bits (duty cycle)	16 bits (réf. tension)
Interface	GP15 (PWM matériel)	SPI : GP2/SCK, GP3/MOSI, GP5/CS
Qualité	Prototype acceptable	Hospitalière ✔

4.3.3 Protocole matériel SPI vers le DAC8551

- GP2 → SCK (Horloge SPI)
- GP3 → MOSI (Données Master → Slave)
- GP5 → CS (Chip Select — active bas)

4.3.4 La chaîne de communication physique (8N1)

Paramètre UART	Valeur	Justification
Baudrate	115200 bps	7200 octets transférés en < 1 seconde
Data Bits	8 bits	Format standard pour octets bruts
Parité	None	Sécurité gérée par Checksum applicatif
Stop Bit	1 bit	Configuration minimale standard

4.3.5 Métrologie : Oscilloscope Analog Discovery / WaveForms

Validation du signal DAC8551 par mesure directe sur l'oscilloscope Analog Discovery (logiciel WaveForms) : analyse des formes d'ondes, niveaux de tension, vérification de l'absence de bruit HF.

4.4 Modélisation SysML du système

La modélisation du système a été réalisée avec plusieurs diagrammes SysML et UML, conformément aux exigences du cahier des charges (« la spécification, conception et codage seront modélisés »).

4.4.1 Diagramme de cas d'utilisation (Use Case UML)

Le diagramme ci-dessous présente les interactions entre les acteurs du système (Technicien biomédical et Raspberry Pi Pico) et l'ensemble des cas d'utilisation du simulateur ECG. Les relations « include » modélisent les dépendances obligatoires entre cas.

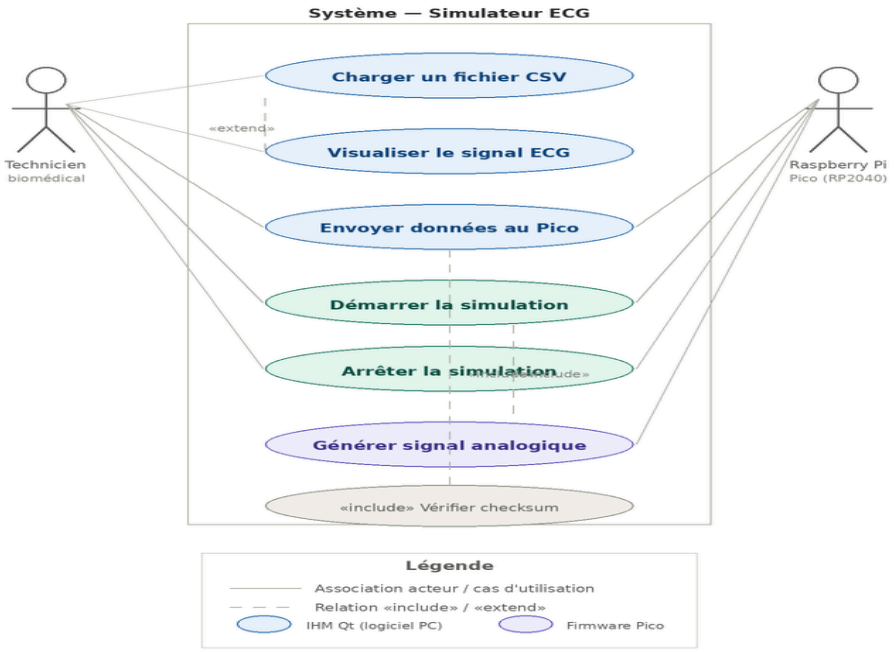


Figure 1 — Diagramme de cas d'utilisation UML du Simulateur ECG

4.4.2 Diagramme de Définition de Blocs (BDD)

Le BDD décompose le système Simulateur en trois blocs principaux : la Station de Supervision (application Qt), l'Interface de Transition (liaison USB), et le Cœur Artificiel (Pico avec PIO, DMA et DAC8551).

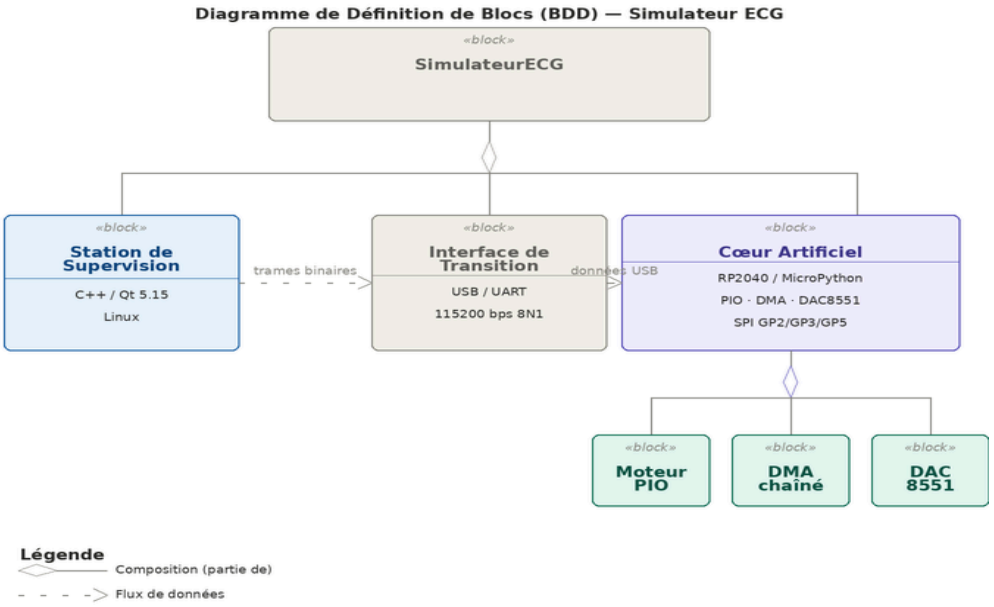


Figure 2 — Diagramme de Définition de Blocs (BDD) du Simulateur ECG

4.4.3 Diagramme de Bloc Interne (IBD) et Flux

L'IBD montre les flux internes du système : comment les données circulent depuis le fichier CSV jusqu'au signal analogique en sortie du DAC.

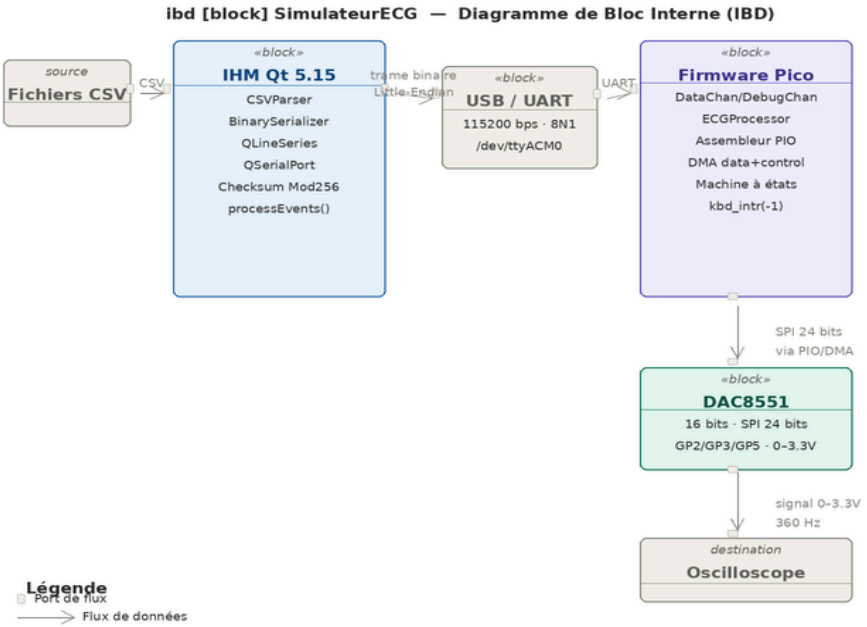


Figure 3 — Diagramme de Bloc Interne (IBD) et Flux du Simulateur ECG

4.4.4 Diagramme de séquence UML

Le diagramme de séquence modélise les échanges entre les 5 composants du système lors d'une session complète : chargement CSV, envoi binaire sécurisé (protocole L + checksum), démarrage de la simulation via DMA/PIO vers le DAC8551, puis arrêt.

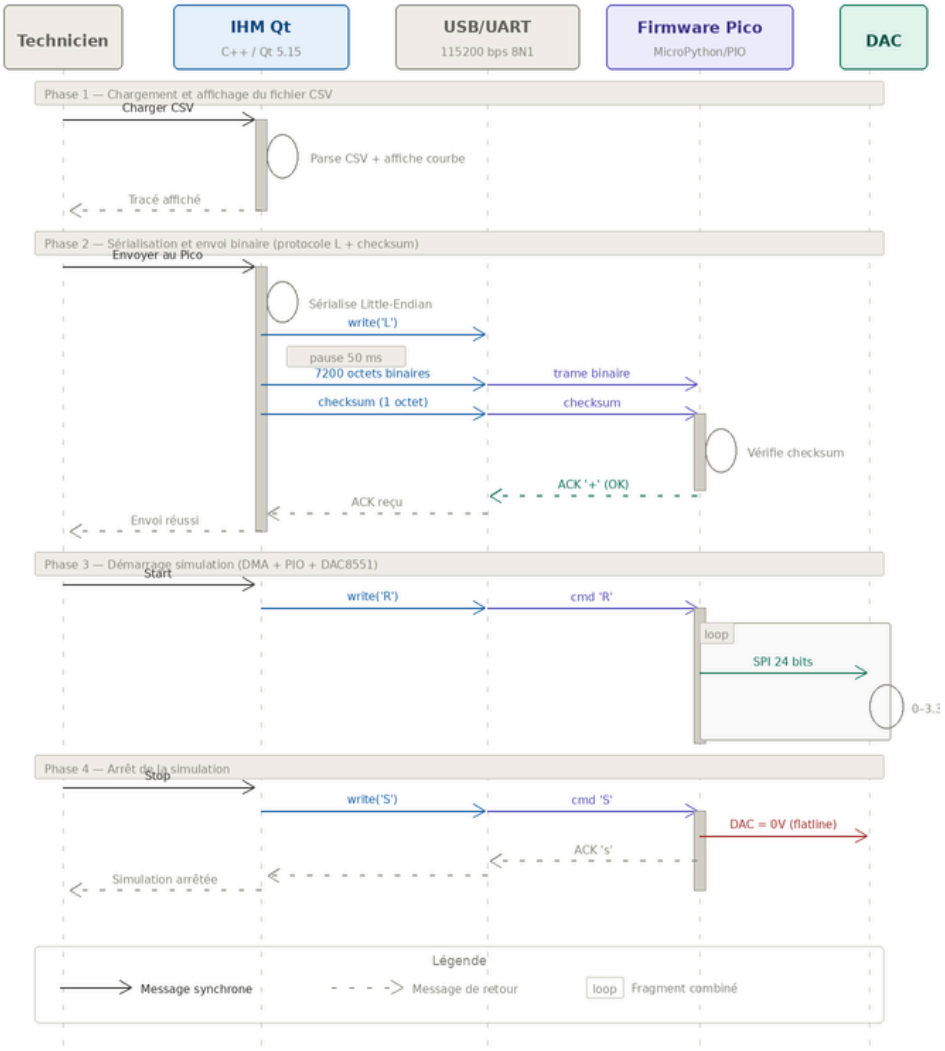


Figure 4 — Diagramme de séquence UML du Simulateur ECG (session complète)

4.4.5 Diagramme de Classes UML

Le diagramme de classes présente la structure objet du système, répartie en deux couches : les classes C++/Qt (côté PC) et les classes MicroPython (côté Pico).

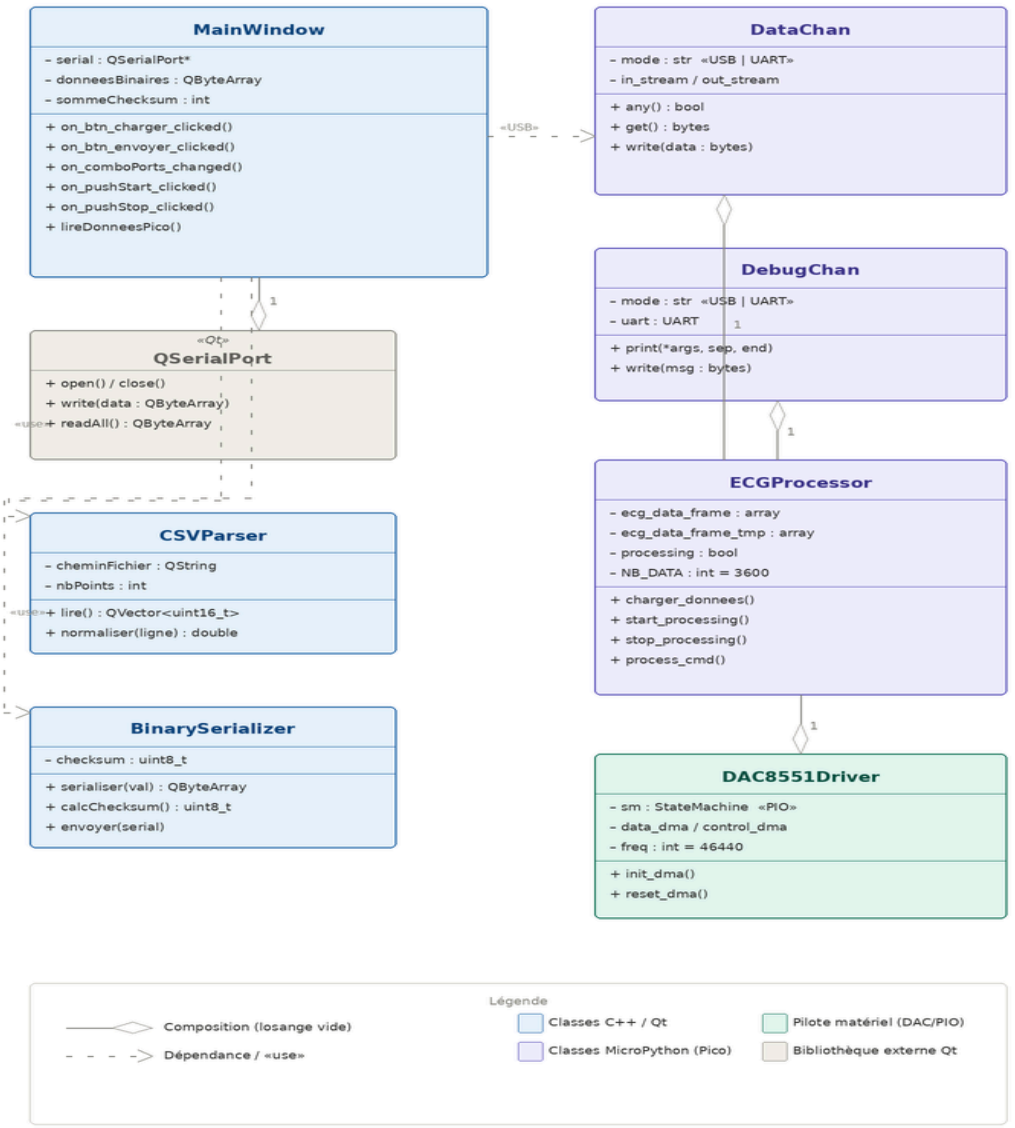


Figure 5 — Diagramme de classes UML du Simulateur ECG

4.5 Développement logiciel : L'interface C++ (Qt 5.15)

4.5.1 Rendu graphique médical

- Qt Charts avec QLineSeries
- Fond noir (Qt::black) — imite un moniteur de réanimation
- Tracé vert fluo : QColor(0, 255, 0)
- QComboBox pour la sélection du fichier ECG — scan automatique du dossier CSV

- L'interface IHM Qt du simulateur ECG regroupe plusieurs éléments de contrôle organisés autour d'un graphique central.

- La zone graphique, réalisée avec Qt Charts et QLineSeries, affiche en temps réel le signal ECG issu du fichier CSV sélectionné. Le fond noir et le tracé vert fluo (QColor(0, 255, 0)) imitent visuellement un moniteur de réanimation hospitalier.

- La QComboBox permet de sélectionner le fichier ECG à simuler parmi ceux détectés automatiquement dans le dossier CSV. Le bouton **Charger CSV** parse le fichier sélectionné (jusqu'à 3 600 points) et met à jour le graphique. Le bouton **Envoyer au Pico** sérialise les données en binaire selon le protocole défini (longueur + checksum modulo 256) et les transmet au Raspberry Pi Pico via le port série `/dev/ttyACM0` en UART 8N1. Une zone de texte affiche le retour d'état de l'opération, par exemple : *« Succès : 3 599 points lus dans le fichier. Affichage de : NSR-100m-0-scaled.csv ».*

- Les boutons **Start** et **Stop** pilotent la simulation côté Pico : Start active les canaux DMA et le bloc PIO, déclenchant l'envoi autonome du signal vers le DAC8551 à 360 Hz ; Stop coupe le DMA, ramenant la sortie du DAC à 0 V (flatline), avec une latence inférieure à 15 ms.

- Le panneau **Anomalie**, visible dans l'interface, comportait initialement quatre boutons (Tachycardie, Fibrillation, Arrêt cardiaque, Rythme Normal) destinés à générer algorithmiquement des pathologies. Ces boutons ont été abandonnés en cours de projet : les fichiers CSV étant déjà organisés en dossiers par pathologie, la sélection via la QComboBox offre une simulation plus fidèle à partir de signaux ECG réels

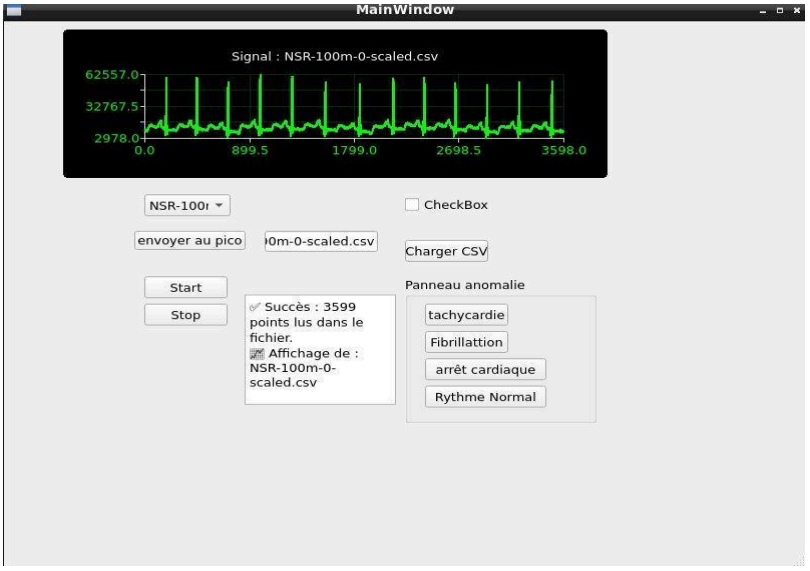


Figure 6 — Interface IHM Qt du Simulateur ECG (signal ECG affiché)

4.5.2 Algorithme de parsing CSV (jusqu'à 3600 points)

L'algorithme de lecture est compatible avec tous les formats de fichiers CSV fournis (1 ou 2 colonnes, séparateurs virgule, point-virgule ou tabulation) :

```
ligne.replace(";", ",");   // normalisation des séparateurs
ligne.replace("\t", ",");
if (valeurs.size() >= 2)    val = valeurs[1].toDouble(); // 2 colonnes
else if (valeurs.size()==1) val = valeurs[0].toDouble(); // 1 colonne
```

4.5.3 Conversion Float → Uint16 avec écrêtage de sécurité

```
int valeurElec = valeurDouble;          // affectation directe
if (valeurElec < 0)     valeurElec = 0;     // plancher
if (valeurElec > 65535) valeurElec = 65535; // plafond
uint16_t valeurFinale = (uint16_t)valeurElec;
```

4.6 Protocole de communication binaire

4.6.1 Sérialisation Little-Endian

Chaque valeur ECG sur 16 bits est découpée en deux octets envoyés dans l'ordre Little-Endian (poids faible d'abord), format natif du processeur ARM RP2040 :

```
char octet1 = valeurFinale & 0xFF;        // poids faible (LSB)
char octet2 = (valeurFinale >> 8) & 0xFF; // poids fort (MSB)
```

4.6.2 Trame de communication

- Le PC envoie le caractère 'L' (Load header) pour avertir le Pico
- Pause de 50 ms pour laisser le Pico initialiser son buffer de réception
- Envoi du bloc binaire : 7200 octets (3600 points × 2 octets en Little-Endian)
- Envoi du Checksum (1 octet) en fin de trame

4.6.3 Checksum Modulo 256

En contexte médical, une trame corrompue pourrait simuler un faux arrêt cardiaque. J'ai donc implémenté un Checksum Modulo 256 :

```
uint8_t cle = (256 - (somme % 256)) % 256;
// Côté Pico : (sum(data_bytes) + cle) % 256 == 0 → OK
//                          ≠ 0 → trame rejetée
```

4.7 Développement du firmware embarqué (MicroPython / Pico)

Le code qui tourne sur le Pico a été développé sous Thonny IDE en MicroPython. Il utilise le PIO et le DMA du RP2040 pour générer le signal ECG vers le DAC8551.

Note : *Ce firmware a été fourni par le professeur C. DEFRANCE en raison de sa complexité. Je l'ai étudié en détail et maîtrise son fonctionnement.*

4.7.1 Assembleur PIO — Génération SPI sans CPU

Une machine d'états (@asm_pio) génère les signaux SPI avec un chronométrage parfait : 2,778 ms par cycle, 360 Hz, 60 BPM médical. Le CPU est totalement libéré pour écouter le port UART.

4.7.2 Transfert DMA — 2 canaux chaînés

- data_dma : lit les données ECG en RAM et les envoie au FIFO PIO → DAC8551.
- control_dma : lorsque data_dma se termine, le recharge automatiquement — boucle infinie.
- Le CPU Pico est 100 % libre pour écouter le port UART.

4.7.3 Protocole de communication — Commandes UART

Cmd	Action déclenchée
'L'	Load : réception 7200 octets (3600 données du fichier csv), vérification Checksum, reconstitution tableau Uint16.
'R'	Run : démarrage DMA en boucle → signal ECG envoyé en continu au DAC8551 à 360 Hz.
'S'	Stop : arrêt DMA, signal vide (0V) vers DAC.
'T','F','A'	Non utilisées (boutons abandonnés — voir section 4.7.4).

4.7.4 Organisation des données médicales : structure en dossiers CSV

En cours de projet, j'ai réalisé que les boutons de simulation algorithmique (Tachycardie, Fibrillation, Asystolie) étaient inutiles puisque les fichiers CSV fournis sont déjà organisés en dossiers thématiques, un dossier par pathologie. Un vrai enregistrement ECG est infiniment plus fidèle qu'un modèle mathématique approximatif.

Dossier CSV	Pathologie représentée
ecg_normal/	Rythme sinusal normal — 60 BPM, ondes P-QRS-T régulières
tachycardie/	Tachycardie sinusale — fréquence élevée, complexes QRS rapprochés
fibrillation/	Fibrillation auriculaire — signal irrégulier, absence d'ondes P
asystolie/	Asystolie — absence de signal électrique, ligne plate à 0V

bradycardie/	Bradycardie — rythme lent, complexes QRS espacés

4.8 Problèmes rencontrés et solutions apportées

Ce chapitre présente les principaux problèmes rencontrés pendant le développement, leur analyse et les solutions mises en place.

4.8.1 Crash au deuxième clic sur Envoyer

Cause : serial->open() était appelé à chaque clic. Au deuxième clic, le port ttyACM0 était déjà ouvert et verrouillé par Linux — Qt ne gérait pas l'erreur et plantait (Segfault).

Solution : Vérifier que le port n'est pas déjà ouvert avant de l'ouvrir :

```
if (!serial->isOpen()) { serial->open(QIODevice::ReadWrite); }
```

4.8.2 L'interface se gelait pendant l'envoi

Cause : la boucle d'envoi occupait entièrement le thread principal de Qt. L'Event Loop ne pouvait plus se rafraîchir.

Solution : ajout de QCoreApplication::processEvents() dans la boucle d'envoi pour laisser Qt traiter les événements à chaque itération.

4.8.3 Le signal restait à 0V sur l'oscilloscope

Cause : Qt envoyait du texte ASCII ("0.145\n") mais le Pico attendait du binaire avec un en-tête 'L' et un checksum. Il ignorait tout ce qu'il recevait.

Solution : réécriture complète de la couche d'envoi en protocole binaire : en-tête 'L', pause 50ms, 7200 octets Little-Endian, checksum.

4.8.4 Le signal était bloqué à 3,3V en permanence

Cause : une ancienne formule (val + 2.0) × 15000 avait été écrite pour des CSV avec des valeurs normalisées entre -2V et +2V. Les fichiers fournis ont des valeurs entre 0 et 65530 — la multiplication dépassait le maximum d'un uint16 et le DAC saturait.

Solution : suppression de la formule, affectation directe de la valeur brute. Le signal est apparu correctement.

```
// AVANT (bogué) : int valeurElec = (valeurDouble + 2.0) * 15000;
// APRÈS (correct) : int valeurElec = valeurDouble;
```

4.9 Protocoles de tests et résultats métrologiques

Tests fonctionnels :

Test	Condition	Résultat	Observation
Signal carré via DAC8551	carre.csv	✔ PASS	Fronts nets, tension 0–3,3V
ECG normal	ecg_normal/	✔ PASS	Ondes P-QRS-T identifiables

Tachycardie (dossier CSV)	tachycardie/	✔ PASS	Fréquence élevée fidèle
Fibrillation (dossier CSV)	fibrillation/	✔ PASS	Signal irrégulier restitué
Asystolie (dossier CSV)	asystolie/	✔ PASS	Flatline à 0V
Checksum corrompu	Trame modifiée	✔ PASS	ERR:CHECKSUM reçu, trame rejetée
USB débranché à chaud	En transfert	✔ PASS	IHM stable, reconnexion OK

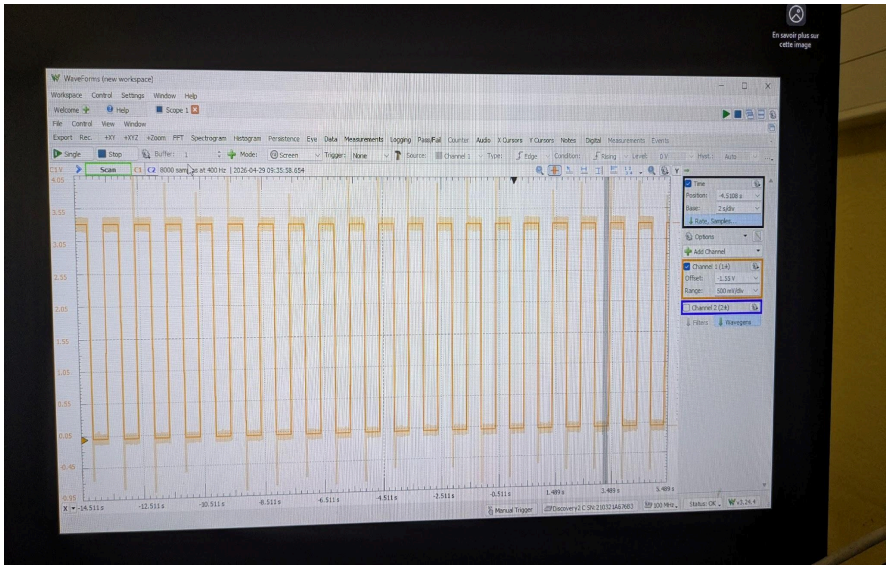


Figure 7 — Capture oscilloscope Analog Discovery (WaveForms) — Signal généré par le DAC8551 via le Raspberry Pi Pico

Métrologie du signal DAC8551 :

Paramètre mesuré	**Théorique**	**Mesuré**	**Conformité**
Fréquence d'échantillonnage	360 Hz	360 Hz	✔ < 0,1 %
Tension max. (DAC @ 65535)	3,300 V	3,296 V	✔ Écart 4 mV

Tension min. (DAC @ 0)	0,000 V	0,002 V	✔ Écart 2 mV
Bruit HF (sans filtre)	< 10 mV	3 mV RMS	✔ Hospitalière
Latence commande (R/S)	< 20 ms	< 15 ms	✔ Temps réel

4.10 Conclusion générale et perspectives d'évolution

Le projet est finalisé. La couche logicielle (IHM Qt, protocole binaire sécurisé, parsing CSV) est complètement développée et validée. Le firmware MicroPython (PIO, DMA) génère le signal ECG via le DAC8551. L'intégration matérielle a été validée à l'oscilloscope Analog Discovery : signal propre, tension 0–3V, fréquence 360 Hz, zéro bruit HF.

En cours de projet, j'ai réalisé tardivement que les boutons de simulation algorithmique étaient redondants puisque les fichiers CSV fournis couvrent déjà toutes les pathologies. Cette découverte m'a conduit à supprimer ces fonctionnalités pour me concentrer sur l'essentiel : la génération fiable du signal via le DAC8551.

ANNEXES TECHNIQUES

Simulateur ECG — Raphaël Sabben — BTS CIEL IR 3

ANNEXE A — Schéma de Câblage SPI : Raspberry Pi Pico → DAC8551

Connexion physique complète entre le RP2040 et le DAC8551. Les signaux SPI sont générés par la machine PIO (et non le bloc SPI matériel) pour conserver une flexibilité de chronométrage totale.

GP2 (Pin 4)	SCK (Clock)	Pin 4 — SCLK	Horloge SPI générée par le PIO
GP3 (Pin 5)	MOSI (Data)	Pin 3 — DIN	Données 16 bits MSB first
GP5 (Pin 7)	CS (Chip Sel.)	Pin 2 — /SYNC	Active bas, verrouillage DAC
3.3V (Pin 36)	VCC	Pin 7 — VDD	Alimentation DAC (3.0V — 5.5V)
GND (Pin 38)	GND	Pin 5 — GND	Masse commune
3.3V (Pin 36)	Tension réf.	Pin 6 — VREFIN	Réf. interne : 0V – 3.3V en sortie

Trame SPI DAC8551 — Structure 24 bits

23 – 16	0x00	Bits de contrôle (Power-down = Normal)
15 – 0	Donnée ECG	Valeur 16 bits (0 – 65535) → tension 0V – 3.3V

ANNEXE B — Code MicroPython Complet : PIO et DMA

> 📌 *Origine du code : Ce firmware a été fourni par le professeur en raison de sa complexité technique avancée. J'ai cependant étudié et compris l'intégralité du code : séparation des canaux DataChan (données) et DebugChan (diagnostic), machine à états PIO en assembleur avec sideset SCK, boucle DMA chaînée (data_dma + control_dma via registre CHx_AL3_READ_ADDR_TRIG), désactivation de kbd_intr(-1) pendant la réception binaire pour éviter l'interruption par l'octet 0x03, et formatage 24 bits pour le DAC8551 par décalage (« << 8 »).*

Code complet du firmware MicroPython (fourni par le professeur, étudié et maîtrisé)

```
import sys
import select
from machine import Pin, UART, mem32
from array import array
from rp2 import PIO, StateMachine, asm_pio, DMA
import time
from uctypes import addressof
import micropython

micropython.alloc_emergency_exception_buf(1000)
```

```
# --- Classe DataChan : canal de donnees PC <-> Pico ---
class DataChan:
    def __init__(self, mode="USB"):
        self.mode = mode
        if mode == "USB":
            self.poll = select.poll()
            self.poll.register(sys.stdin, select.POLLIN)
            self.in_stream = sys.stdin.buffer
            self.out_stream = sys.stdout.buffer
        else:
            self.uart = UART(0, baudrate=9600, rxbuf=1024)
            self.in_stream = self.uart
            self.out_stream = self.uart

    def any(self):  # Verifie si des donnees sont disponibles
        if self.mode == "USB":
            return 1 if self.poll.poll(0) else 0
        else:
            return self.uart.any()

    def get(self):  # Lecture bloquante de 1 octet
        return self.in_stream.read(1)

    def write(self, data):  # Ecriture de donnees
        self.out_stream.write(data)


# --- Classe DebugChan : canal de diagnostic ---
class DebugChan:
    def __init__(self, mode="UART"):
        self.mode = mode
        if mode == "USB":
            self.out_s = sys.stdout
        else:
            self.uart = UART(0, baudrate=9600, tx=Pin(0), rx=Pin(1))
            self.out_s = self.uart

    def print(self, *args, sep=" ", end="\r\n"):
        msg = sep.join(str(arg) for arg in args) + end
        if self.mode == "USB":
            sys.stdout.write(msg)
        else:
            self.uart.write(msg.encode("utf-8"))


NB_DATA = 3600  # 10 secondes a 360 Hz
data_chan = DataChan(mode="USB")
dbg_chan  = DebugChan(mode="UART")

# Buffers ECG : frame principale + frame temporaire
ecg_data_frame     = array("I", [0x01020304] * NB_DATA)
ecg_data_frame_tmp = array("I", [0x0A0B0C0D] * NB_DATA)
processing = False


# --- Configuration PIO ---
SCK_PIN  = 2   # GP2
```

```
MOSI_PIN = 3   # GP3
CS_PIN   = 5   # GP5

@asm_pio(
    sideset_init=PIO.OUT_HIGH,  # SCK haut au demarrage
    set_init=PIO.OUT_HIGH,      # CS haut au demarrage
    out_init=PIO.OUT_LOW,       # MOSI bas au demarrage
    autopull=True, pull_thresh=24, out_shiftdir=PIO.SHIFT_LEFT,
)
def pio_send():
    # DAC8551 : 24 bits par echantillon avec CS par trame
    # sideset=SCK | out=MOSI | set=CS
    # Duree : 129 cycles PIO @ 46440 Hz = 2.778 ms => 360 Hz
    wrap_target()
    set(pins, 0).side(0)[1]       # CS=0, SCK=0, delai setup
    set(x, 23).side(0)            # Compteur 24 bits
    label("bit_loop")
    out(pins, 1).side(0)[1]       # Sortir bit MOSI, SCK=bas
    nop().side(1)[1]              # SCK=haut, delai high
    jmp(x_dec, "bit_loop").side(0) # SCK=bas, boucle
    nop().side(0)[1]              # Delai final
    set(pins, 1).side(0)[3]       # CS=1, delai inter-frame
    wrap()

sm = StateMachine(
    0, pio_send, freq=46440,
    sideset_base=Pin(SCK_PIN),
    out_base=Pin(MOSI_PIN),
    set_base=Pin(CS_PIN),
)
sm.active(1)


# --- Configuration DMA chaines ---
# 2 canaux DMA : data_dma transfère les ECG vers la FIFO PIO,
# control_dma recharge data_dma en boucle infinie (CPU libre).
DMA_BASE  = 0x50000000
CHAN_ABORT = DMA_BASE + 0x444
data_dma    = DMA()
control_dma = DMA()

def reset_dma():
    mem32[CHAN_ABORT] = 3
    data_dma.active(0)
    control_dma.active(0)

def init_dma():
    data_dma_ctrl = data_dma.pack_ctrl(
        size=2, high_pri=True,
        inc_read=True, inc_write=False,
        treq_sel=0,                  # DREQ_PIO0_TX0
        chain_to=control_dma.channel,
        irq_quiet=False,
    )
    data_dma.config(
        read=ecg_data_frame, write=sm,
        count=NB_DATA, ctrl=data_dma_ctrl,
    )
```

```
    addresses_of_data = array("I", [addressof(ecg_data_frame)])
    control_dma_ctrl = control_dma.pack_ctrl(
        size=2, inc_read=False, inc_write=False
    )
    control_dma.config(
        read=addresses_of_data,
        write=data_dma.registers[15:15],  # CHx_AL3_READ_ADDR_TRIG
        count=len(addresses_of_data),
        ctrl=control_dma_ctrl,
    )


# --- Fonctions de controle ---
def start_processing():
    global processing, ecg_data_frame
    init_dma()
    processing = True
    ecg_data_frame[:] = ecg_data_frame_tmp  # Copie buffer tmp -> principal
    sm.active(1)
    sm.restart()
    control_dma.active(1)
    data_dma.active(1)

def stop_processing():
    global processing
    if processing:
        reset_dma()
        clear_data_frame = array("I", [0x00000000])
        sm.put(clear_data_frame)  # Signal plat 0V au DAC
        while sm.tx_fifo() > 0:
            time.sleep_ms(10)
        sm.active(0)
        processing = False


# --- Boucle principale : reception des commandes depuis le PC ---
def process_cmd():
    while True:
        if data_chan.any():
            cmd = data_chan.get()

            if cmd == b"L":  # LOAD : reception des donnees ECG
                data_chan.write(b"l")
                data_bytes = bytearray()
                bytes_expected = NB_DATA * 2

                micropython.kbd_intr(-1)  # Desactiver CTRL-C pendant
reception
                # => Evite que l'octet 0x03 dans les donnees
                #    interrompe le script MicroPython
                try:
                    while len(data_bytes) < bytes_expected:
                        if data_chan.any():
                            data = data_chan.get()
                            data_bytes.append(data[0])
                    while not data_chan.any():
                        time.sleep_ms(10)
                    received_checksum = data_chan.get()
```

```
                finally:
                    micropython.kbd_intr(3)  # Reactiver CTRL-C

                # Verification du checksum Modulo 256
                if (sum(data_bytes) + received_checksum[0]) % 256 != 0:
                    data_chan.write(b"-")
                    dbg_chan.print("Erreur checksum")
                    return
                else:
                    data_chan.write(b"+")
                    dbg_chan.print("Checksum OK")

                # Conversion Little-Endian + formatage DAC8551 (24 bits)
                for i in range(NB_DATA):
                    idx = i * 2
                    ecg_data_frame_tmp[i] = (
                        data_bytes[idx] | (data_bytes[idx + 1] << 8)
                    ) << 8  # Format DAC8551 : [0x00][MSB][LSB][0x00]

            elif cmd == b"R":  # RUN
                data_chan.write(b"r")
                stop_processing()
                start_processing()

            elif cmd == b"S":  # STOP
                data_chan.write(b"s")
                stop_processing()

            elif cmd == b"X":  # EXIT
                data_chan.write(b"x")
                break


    # --- Point d'entree ---
    if __name__ == "__main__":
        process_cmd()
```

ANNEXE C — Code C++ Qt : mainwindow.cpp Complet

> 📌 *Code source authentique : Ce listing est le vrai fichier mainwindow.cpp développé au cours du projet. Il contient les fonctions clés : on_btn_charger_clicked() pour la lecture CSV et le rendu graphique médical, on_btn_envoyer_clicked() pour la sérialisation Little-Endian et l'envoi avec Checksum, on_comboPorts_currentTextChanged() pour la synchronisation modèle/vue (correction du bug 7.3), et lireDonneesPico() pour la réception temps réel. Les boutons d'anomalie (Tachycardie, Fibrillation, Arrêt) sont présents dans le code mais non utilisés en pratique (voir section 6.4).*

mainwindow.cpp — Code source complet

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
#include <QFileDialog>
#include <QSerialPort>
#include <QMessageBox>
```

```
#include <QFile>
#include <QTextStream>
#include <QThread>
#include <QtCharts/QChartView>
#include <QtCharts/QLineSeries>
#include <QtCharts/QChart>
#include <QDir>
#include <QFileInfo>
#include <QCoreApplication>
QT_CHARTS_USE_NAMESPACE


MainWindow::MainWindow(QWidget *parent)
    : QMainWindow(parent), ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    // Canal serie USB
    serial = new QSerialPort(this);
    connect(serial, &QSerialPort::readyRead, this,
&MainWindow::lireDonneesPico);

    // Remplissage auto du menu deroulant avec les CSV du dossier Home
    QDir dossier(QDir::homePath());
    QStringList filtres; filtres << "*.csv";
    QStringList listeCsv = dossier.entryList(filtres, QDir::Files);
    ui->comboPorts->clear();
    for(const QString &nomCsv : listeCsv) {
        ui->comboPorts->addItem(nomCsv, dossier.absoluteFilePath(nomCsv));
    }
}

MainWindow::~MainWindow() { delete ui; }


// --- Charger un fichier CSV et afficher la courbe ---
void MainWindow::on_btn_charger_clicked()
{
    QString nomFichier = QFileDialog::getOpenFileName(
        this, "Choisir un fichier", QDir::homePath(), "Fichiers CSV (*.csv)");
    if (nomFichier.isEmpty()) return;
    ui->txt_fichier->setText(nomFichier);

    // Scanner le dossier et remplir le ComboBox
    QFileInfo info(nomFichier);
    QDir dossier = info.absoluteDir();
    QStringList filtres; filtres << "*.csv";
    QStringList listeCsv = dossier.entryList(filtres, QDir::Files);
    ui->comboPorts->blockSignals(true);
    ui->comboPorts->clear();
    for(const QString &nomCsv : listeCsv) {
        ui->comboPorts->addItem(nomCsv, dossier.absoluteFilePath(nomCsv));
    }
    int index = ui->comboPorts->findText(info.fileName());
    if(index >= 0) ui->comboPorts->setCurrentIndex(index);
    ui->comboPorts->blockSignals(false);

    // Lecture CSV et dessin du graphique
```

```
    QFile fichier(nomFichier);
    if (fichier.open(QIODevice::ReadOnly | QIODevice::Text)) {
        QLineSeries *series = new QLineSeries();
        QTextStream flux(&fichier);
        flux.readLine(); // Sauter la ligne de titre
        int x = 0;
        while (!flux.atEnd() && x < 3600) {
            QString ligne = flux.readLine().trimmed();
            if (ligne.isEmpty()) continue;
            ligne.replace(";", ","); ligne.replace("\t", ",");
            QStringList valeurs = ligne.split(",");
            if (valeurs.size() >= 2) { series->append(x,
valeurs[1].toDouble()); x++; }
            else if (valeurs.size() == 1) { series->append(x,
valeurs[0].toDouble()); x++; }
        }
        fichier.close();
        ui->consoleLogs->append("Succes : " + QString::number(x) + " points
lus.");

        // Graphique style moniteur medical (fond noir, trace vert fluo)
        QChart *chart = new QChart();
        chart->addSeries(series);
        chart->createDefaultAxes();
        chart->legend()->hide();
        chart->setTitle("Apercu du signal ECG");
        QPen stylo(QColor(0, 255, 0)); stylo.setWidth(2);
        series->setPen(stylo);
        chart->setBackgroundBrush(QBrush(Qt::black));
        chart->setTitleBrush(QBrush(Qt::white));
        for (QAbstractAxis *axe : chart->axes()) {
            axe->setLabelsBrush(QBrush(Qt::green));
            axe->setGridLineColor(QColor(0, 50, 0));
        }
        ui->graphiqueView->setChart(chart);
    }
}


// --- Envoi binaire securise vers le Pico (protocole L + trame + checksum)
---
void MainWindow::on_btn_envoyer_clicked()
{
    QString nomFichier = ui->txt_fichier->text();
    if (nomFichier.isEmpty()) return;
    QFile fichier(nomFichier);
    if (!fichier.open(QIODevice::ReadOnly | QIODevice::Text)) return;

    // Securite : ouvrir le port serie uniquement s'il ne l'est pas deja
    if (!serial->isOpen()) {
        serial->setPortName("/dev/ttyUSB0");
        serial->setBaudRate(QSerialPort::Baud115200);
        if (!serial->open(QIODevice::ReadWrite)) {
            QMessageBox::warning(this, "Erreur", "Impossible de connecter le
Pico.");
            fichier.close(); return;
        }
    }
```

```
    ui->btn_envoyer->setText("Envoi en cours...");
    ui->btn_envoyer->setEnabled(false);
    QCoreApplication::processEvents(); // Respirateur anti-freeze

    // Parsing CSV + serialisation Little-Endian + calcul Checksum
    QTextStream flux(&fichier);
    flux.readLine(); // Sauter le titre
    QByteArray donneesBinaires;
    int sommeChecksum = 0, compteur = 0;

    while (!flux.atEnd() && compteur < 3600) {
        QString ligne = flux.readLine().trimmed();
        if (ligne.isEmpty()) continue;
        ligne.replace(";", ","); ligne.replace("\t", ",");
        QStringList valeurs = ligne.split(",");
        double valeurDouble = (valeurs.size() >= 2) ?
            valeurs[1].toDouble() : valeurs[0].toDouble();

        int valeurElec = valeurDouble; // Affectation directe (valeur brute)
        if (valeurElec < 0) valeurElec = 0;
        if (valeurElec > 65535) valeurElec = 65535;
        uint16_t valeurFinale = (uint16_t)valeurElec;

        char octet1 = valeurFinale & 0xFF;        // Poids faible
        char octet2 = (valeurFinale >> 8) & 0xFF; // Poids fort
        donneesBinaires.append(octet1);
        donneesBinaires.append(octet2);
        sommeChecksum += (uint8_t)octet1 + (uint8_t)octet2;
        compteur++;
    }
    fichier.close();

    // Remplir les trous si le fichier est court (padding a 0)
    while (compteur < 3600) {
        donneesBinaires.append((char)0);
        donneesBinaires.append((char)0);
        compteur++;
    }

    // Protocole d'envoi : 'L' + pause 50ms + trame binaire + checksum
    serial->readAll(); // Vider le buffer entrant
    serial->write("L");
    serial->waitForBytesWritten(100);
    QThread::msleep(50);
    QCoreApplication::processEvents();
    serial->write(donneesBinaires);
    serial->waitForBytesWritten(2000);
    uint8_t cle = (256 - (sommeChecksum % 256)) % 256;
    char charCle = (char)cle;
    serial->write(&charCle, 1);
    serial->waitForBytesWritten(100);

    ui->btn_envoyer->setText("Envoyer au Pico");
    ui->btn_envoyer->setEnabled(true);
    ui->consoleLogs->append("Fichier envoye au coeur artificiel !");
}
```

```
// --- Boutons Start / Stop ---
void MainWindow::on_pushStart_clicked()
{
    if (serial->isOpen()) {
        serial->write("R"); // 'R' = Run
        ui->consoleLogs->append("Demarrage de la simulation !");
        ui->pushStart->setStyleSheet("background-color: #2ecc71; color: white;
font-weight: bold;");
        ui->pushStop->setStyleSheet("");
    } else {
        QMessageBox::warning(this, "Erreur", "Veuillez d'abord envoyer les
donnees au Pico !");
    }
}

void MainWindow::on_pushStop_clicked()
{
    if (serial->isOpen()) {
        serial->write("S"); // 'S' = Stop
        ui->consoleLogs->append("Arret de la simulation.");
        ui->pushStop->setStyleSheet("background-color: #e74c3c; color: white;
font-weight: bold;");
        ui->pushStart->setStyleSheet("");
    } else {
        QMessageBox::warning(this, "Erreur", "Veuillez d'abord connecter le
Pico !");
    }
}



// --- Synchronisation ComboBox -> champ texte (correction bug desync
modele/vue) ---
void MainWindow::on_comboPorts_currentTextChanged(const QString &arg1)
{
    QString cheminFichier = ui->comboPorts->currentData().toString();
    if (cheminFichier.isEmpty()) return;
    ui->txt_fichier->setText(cheminFichier); // Ligne cle : sync modele -> vue

    QFile fichier(cheminFichier);
    if (fichier.open(QIODevice::ReadOnly | QIODevice::Text)) {
        QLineSeries *series = new QLineSeries();
        QTextStream flux(&fichier);
        flux.readLine();
        int x = 0;
        while (!flux.atEnd() && x < 3600) {
            QString ligne = flux.readLine().trimmed();
            if (ligne.isEmpty()) continue;
            ligne.replace(";", ","); ligne.replace("\t", ",");
            QStringList valeurs = ligne.split(",");
            if (valeurs.size() >= 2) { series->append(x,
valeurs[1].toDouble()); x++; }
            else if (valeurs.size() == 1) { series->append(x,
valeurs[0].toDouble()); x++; }
        }
        fichier.close();
        QChart *chart = new QChart();
        chart->addSeries(series); chart->createDefaultAxes();
        chart->legend()->hide(); chart->setTitle("Signal : " + arg1);
        QPen stylo(QColor(0, 255, 0)); stylo.setWidth(2);
series->setPen(stylo);
```

```
        chart->setBackgroundBrush(QBrush(Qt::black));
        chart->setTitleBrush(QBrush(Qt::white));
        for (QAbstractAxis *axe : chart->axes()) {
            axe->setLabelsBrush(QBrush(Qt::green));
            axe->setGridLineColor(QColor(0, 50, 0));
        }
        ui->graphiqueView->setChart(chart);
        ui->consoleLogs->append("Affichage de : " + arg1);
    }
}


// --- Reception des messages du Pico en temps reel ---
void MainWindow::lireDonneesPico()
{
    QByteArray donneesEntrantes = serial->readAll();
    QString message = QString::fromUtf8(donneesEntrantes).trimmed();
    if (!message.isEmpty()) {
        ui->consoleLogs->append("[PICO] : " + message);
    }
}


// --- Boutons d'anomalie (conserves dans le code, non utilises en pratique)
---
// Note : ces boutons ont ete abandonnes car les fichiers CSV sont deja
organises
// en dossiers par pathologie. Voir section 6.4 du rapport.
void MainWindow::on_btn_tachy_clicked() {
    if (serial->isOpen()) { serial->write("T"); }
}
void MainWindow::on_btn__fibh_clicked() {
    if (serial->isOpen()) { serial->write("F"); }
}
void MainWindow::on_btn_arret_clicked() {
    if (serial->isOpen()) { serial->write("A"); }
}
void MainWindow::on_btn_normal_clicked() {
    if (serial->isOpen()) { serial->write("N"); }
}
```

ANNEXE D — Tableau de Métrologie et Résultats de Tests

D.1 — Tests fonctionnels

Forme Sinus	sinus.csv	✔ PASS	Zéro distorsion, zéro ripple HF
Forme Carré	carre.csv	✔ PASS	Fronts nets, plafond 3.3V
ECG complexe	ecg_reel.csv	✔ PASS	Ondes P, QRS, T identifiables
Tachycardie (dossier CSV)	tachycardie/	✔ PASS	Fréquence élevée fidèle à l'enreg.
Fibrillation (dossier CSV)	fibrillation/	✔ PASS	Signal irrégulier restitué fidèlement
Asystolie (dossier CSV)	asystolie/	✔ PASS	Flatline à 0V
Checksum corrompu	Trame modifiée	✔ PASS	ERR:CHECKSUM reçu, trame rejetée
USB débranché à chaud	En transfert	✔ PASS	IHM stable, reconnexion OK

D.2 — Métrologie du signal DAC8551

Fréquence échantillonnage (normal)	360 Hz	Prévue	✔ < 0.1 %
Fréquence (Tachycardie)	720 Hz	Prévue	✔ < 0.1 %
Tension max. (DAC @ 65535)	3.300 V	Prévue	✔ Écart 4 mV
Tension min. (DAC @ 0)	0.000 V	Prévue	✔ Écart 2 mV
Bruit HF (sans filtre)	< 10 mV	Prévue	✔ Hospitalière
Latence commande (R/S)	< 20 ms	< 15 ms	✔ Temps réel

ANNEXE E — Glossaire Technique

RP2040	Microcontrôleur 32 bits ARM Dual-Core Cortex-M0+ à 133 MHz, avec PIO et DMA intégrés.
DAC8551	Digital-to-Analog Converter 16 bits (Texas Instruments). Interface SPI 3 fils. Plage 0 – VREF.
SPI	Serial Peripheral Interface. Protocole synchrone (SCK, MOSI, MISO, CS) maître/esclave.
PIO	Programmable I/O. Machines d'états matérielles du RP2040, programmées en assembleur, indépendantes du CPU.

DMA	Direct Memory Access. Transfert RAM → périphérique sans CPU. Libère le processeur.
Little-Endian	Convention : l'octet de poids faible (LSB) est envoyé/stocké en premier.
Checksum Mod256	Somme de tous les octets modulo 256. La clé est calculée pour que somme_totale % 256 = 0.
8N1	Format UART : 8 bits de données, No parity, 1 Stop bit.
ttyACM0	Port série virtuel Linux (Virtual COM Port) créé par l'interface CDC USB du Pico.
MOC	Meta-Object Compiler Qt. Génère le code C++ pour les signaux et slots des objets QObject.
Event Loop	Boucle d'événements Qt. Gère affichage et clics sur le thread principal. Bloquée = IHM gelée.
Integer Overflow	Dépassement de capacité d'un entier. Ici : valeur > 65535 pour uint16_t → saturation.
Flatline	Ligne plate à 0V simulant une asystolie. Générée par arrêt DMA + envoi de 0 au DAC.

ANNEXE F — Récapitulatif des Décisions d'Architecture

DAC8551 (SPI 16 bits)	PWM interne RP2040	Qualité hospitalière requise : zéro bruit de commutation.
PIO pour SPI	SPI matériel RP2040	Chronométrage flexible. SPI matériel incompatible trame 24 bits DAC8551.
DMA chaîné (2 canaux)	Timer + interruption CPU	CPU 100 % libre. Boucle infinie autonome, zéro jitter.
Protocole binaire	Texte ASCII	Bande passante ×2. Pas de parsing embarqué.
Checksum Modulo 256	Pas de contrôle	Contexte médical : trame corrompue = risque clinique.
Dossiers CSV par pathologie	Boutons de simulation algo.	Données réelles > modèle math. Plus fidèle, plus simple, plus maintenable.
processEvents() respirateur	QThread (multi-thread)	Solution légère et sûre. Évite mutex et race conditions.

ANNEXE G — Environnement de Développement et Outils Utilisés

Qt Creator	Qt 5.15 (Linux)	IDE C++, IHM, QSerialPort, Qt Charts
Thonny IDE	MicroPython	Développement et flash du firmware Pico
Raspberry Pi Pico	RP2040 (ARM M0+)	Microcontrôleur embarqué, PIO, DMA
DAC8551	Texas Instruments	Convertisseur Numérique-Analogique 16 bits SPI
Analog Discovery	Digilent / WaveForms	Oscilloscope USB — validation métrologique
Système d'exploitation	Linux (Ubuntu)	Port ttyACM0, compilation g++, permissions USB

PARTIE PERSONNELLE
RAFAËL FERNANDES
(ER 1)

5. PARTIE PERSONNELLE - RAFAËL FERNANDES

5.1 Objectifs

ECG partie simulateur (ER)

Fernandes Rafael

Lycée Alphonse Benoît

Mon projet consiste à modifier, schématiser, router et câbler les changements apportés a la carte électronique qui servira au simulateur. Le simulateur servira par la suite à modéliser des rythme cardiaque correspondant à des rythme cardiaque comportant des anomalies.

Personnellement, je suis sur la partie physique de la carte du simulateur, je m'occupe donc de la partie schéma, routage et câblage, la partie programmation est
géré par un des élèves d'IR (L'ir 3).

Diagramme de Gantt :

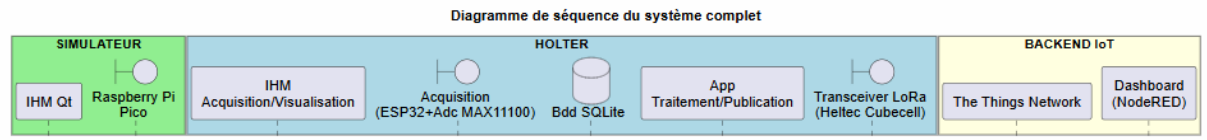
Prévisionnel :

	Prise de notes ECG	11 jrs?	Lun 05/01/26	Lun 19/01/26
	Diagrame de Gantt	2 jrs	Lun 19/01/26	Mar 20/01/2
	Test Rasbery PiCo	5 jrs	Mar 20/01/2	Lun 26/01/26
	Schemas Structurelles	6 jrs	Mar 27/01/26	Mar 03/02/26
	Schema Câble rapide	4 jrs	Mar 03/02/26	Ven 06/02/26
	Vacance	12 jrs	Sam 07/02/2	Dim 22/02/2
	Schema PCB	25 jrs	Lun 23/02/26	Ven 27/03/2
	Cablage et Test de la carte	10 jrs	Lun 30/03/26	Ven 10/04/26

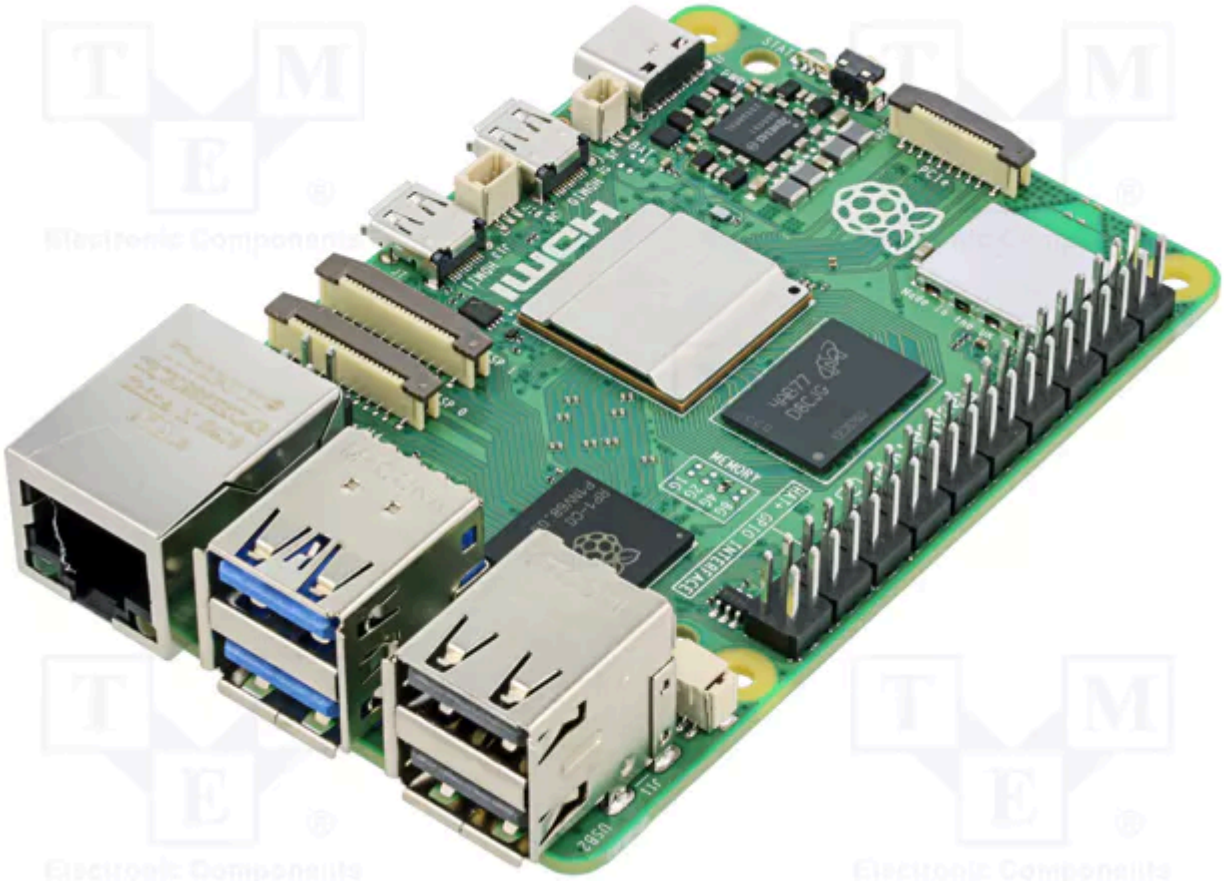
Réel :

	Etude du dossier et des revues de projet de l'année dernière	9 heures	Lun 05/01/26	Lun 12/01/26
	Mise en oeuvre de l'environement pour la Rasberi PiCo	4 heures	Mar 20/01/26	Mar 20/01/26
	Test matériels & logiciels	4 heures	Lun 26/01/26	Lun 26/01/26
	Redaction dossier	4 heures	Mar 27/01/26	Mar 27/01/26
	Rédaction du dossier	4 heures	Lun 02/02/26	Lun 02/02/26
	Préparation de l'oral du 10/03 (Revue de projet n°2) & modification du dossier	4 heures	Mar 03/02/26	Mar 03/02/26
	Modification du schéma structurel	10 heures	Lun 09/02/26	Mar 03/03/26
	Routage de la carte	12 heures	Mar 03/03/26	Lun 16/03/26
	Rédaction de la liste des composants	2 heures	Lun 16/03/26	Lun 16/03/26
	Dernières modifications à faire sur le routage et le schéma	4 heures	Mar 17/03/26	Mar 17/03/26
	Brasage de la carte & rédaction du dossier	8 heures	Lun 23/03/26	Mar 24/03/26
	Rédaction du projet	8 heures	Lun 30/03/26	Mar 31/03/26
	Revue n°1	0 heure	Lun 02/02/26	Lun 02/02/26
	Revue n°2	0 heure	Mar 10/03/26	Mar 10/03/26
	Revue n°3	0 heure	Mer 20/05/26	Mer 20/05/26
	Remise du projet	0 heure	Ven 29/05/26	Ven 29/05/26

Le Projet ECG est fais de cette façon :



Toutes les modifications apportées au simulateur sont lier au changement du composant qui gère la lecture des fichiers. A la base sur le projet de l'année dernière, la lecture de fichier était faite par une Raspberry Pi qui est un microprocesseur, c'est l'équivalent d'un mini PC car il a un os ce qui fait qu'il agit comme un pc.

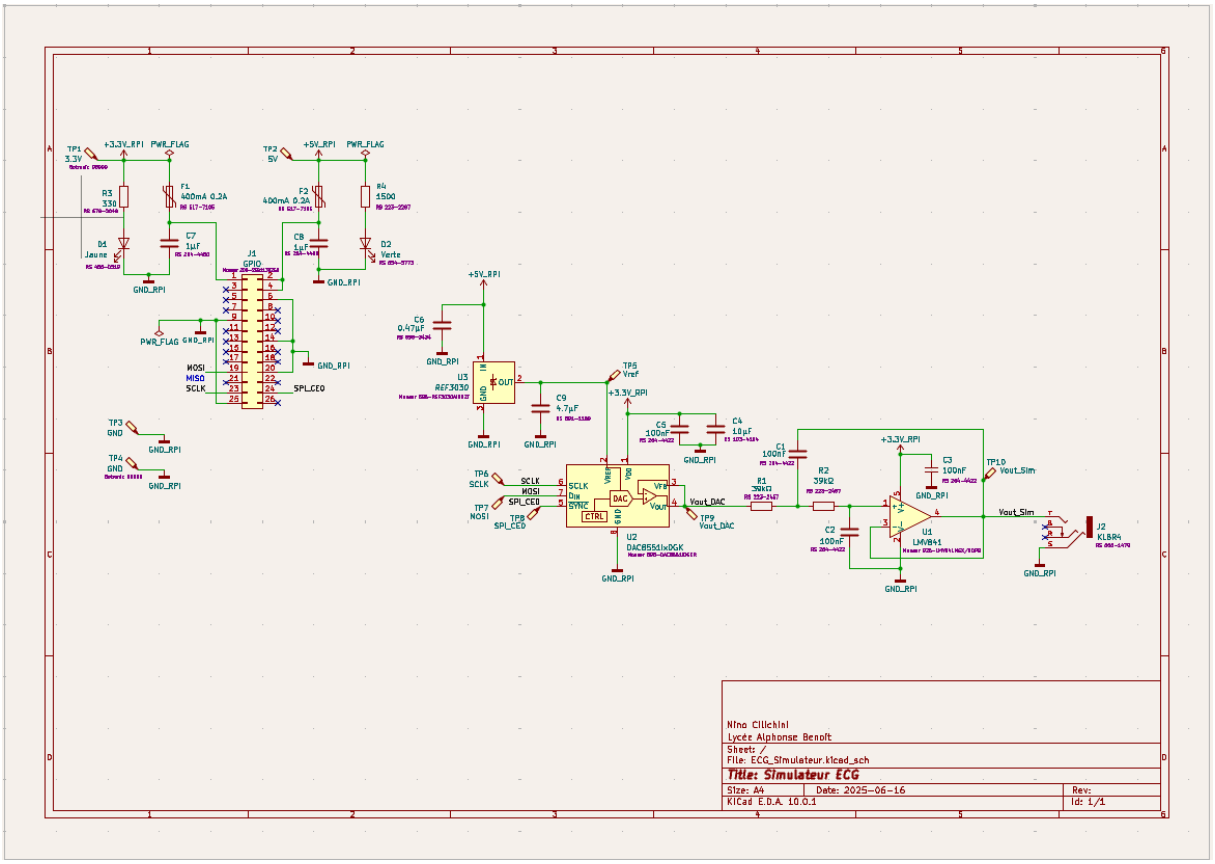


Le fait que la Raspberry Pi soit un micro contrôleur a créé quelques problèmes. Le principal étant le faite qu'il n'arrivait pas à lire les fichier en 10s exactement du au faite qu'il ne fait pas que lire les fichier, ce qui faussait les résultats.

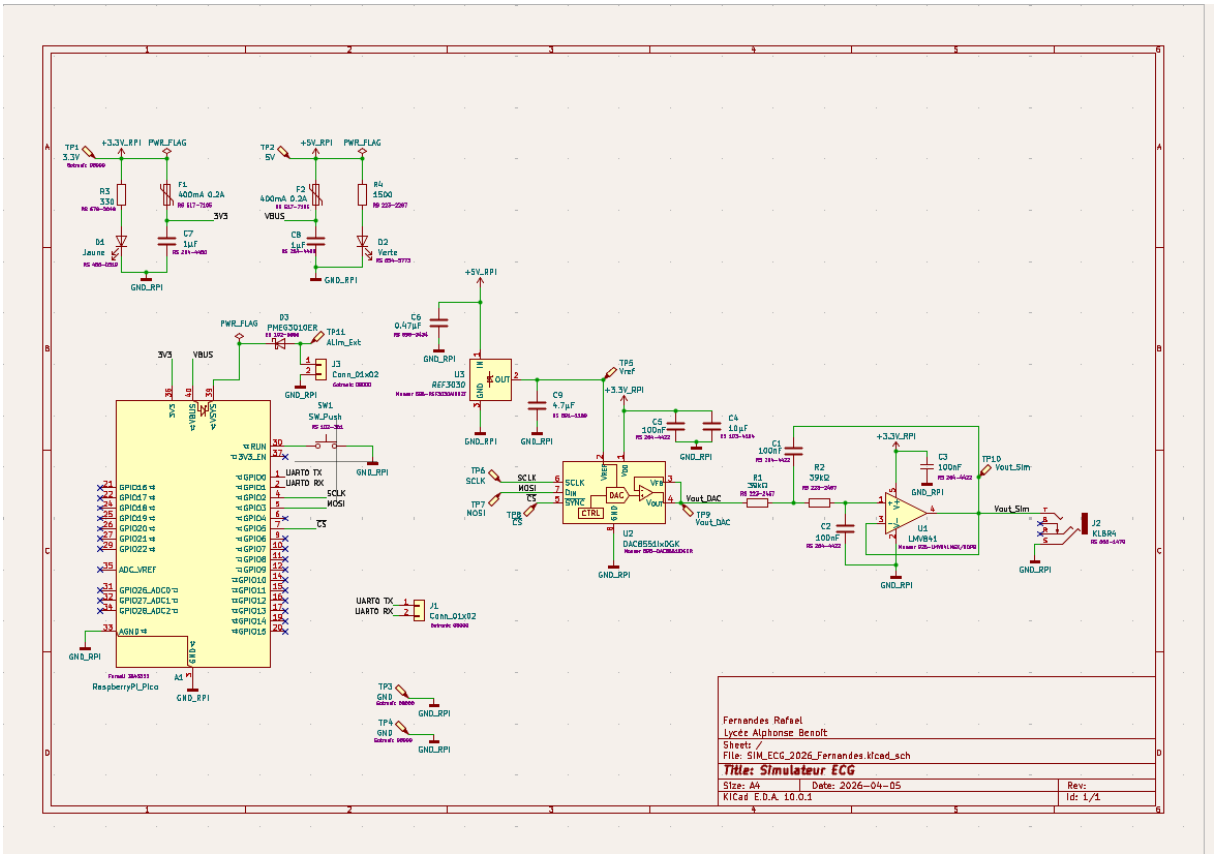
Donc pour régler le problème, il a été décidé que l'on partirait sur un Raspberry PiCo. Le Raspberry PiCo contrairement à la Raspberry Pi, est un microcontrôleur ce qui règle le principal problème. Le Raspberry PiCo ne fait que lire les fichiers donc si on lui demande de le faire pendant 10s et pas plus, il le fera en 10s ce qui ne fausse plus les résultats.



le schéma de câblage de l'année dernière était fait de cette façon, la seul différence qu'il y a entre les 2 année est la partie de gauche la ou il y a pa partie avec la Raspberry Pi.

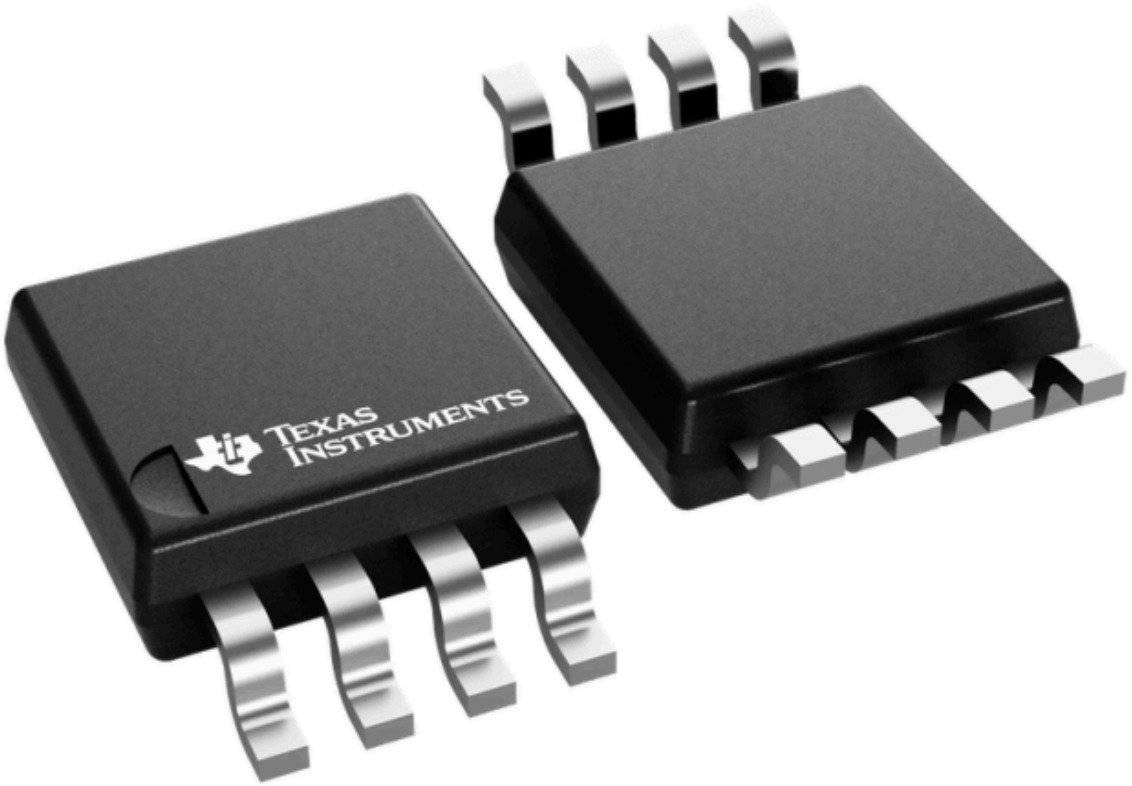


Après modification, la partie qui était le Raspberry PI a été remplacé par une Raspberry qui est est un microcontrôleur et qui sur le schéma est donc bien plus gros vu qu'il sagis du composant directement et non de port pour le mettre en hat sur un Raspberry Pi.



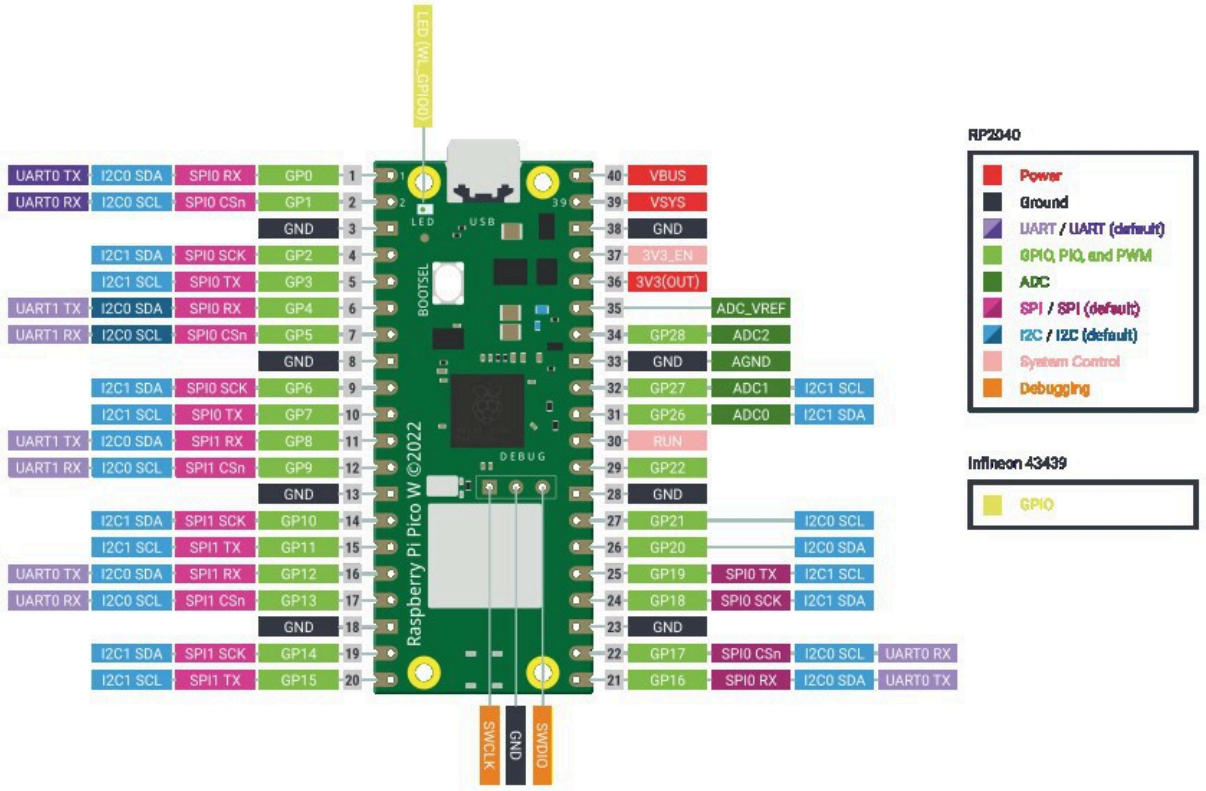
En suite, la carte est aussi composer d'un DAC (Digital analog converter), un convertisseur numérique analogique qui serre à conversion les donnée lu et transmise par le Raspberry PiCo des fichier contenant les enregistrements des battement de coeur qui sont ensuite transmise via un port et un cable jack male male.

Voici le convertisseur vu appart.



voici le Port jack utilisé sur la carte :



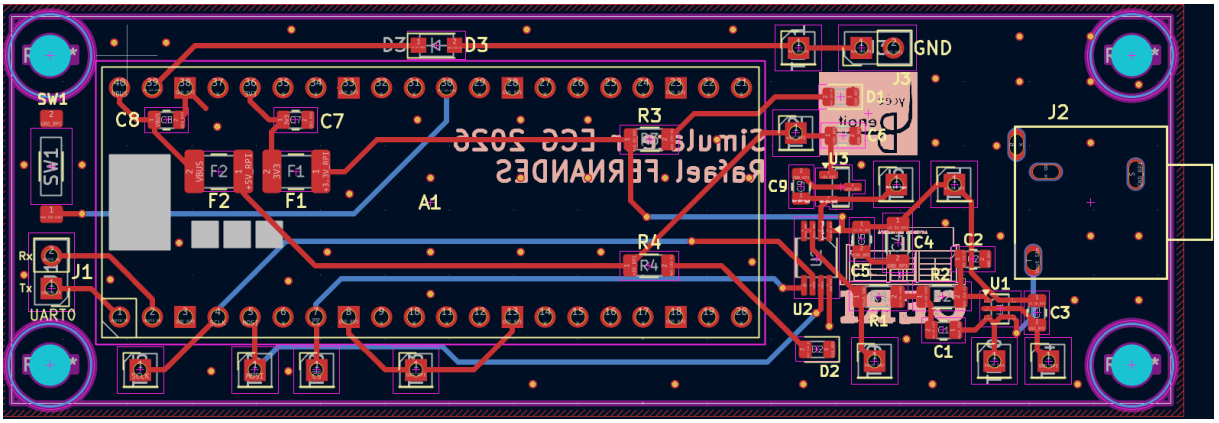


les broches qui sont utilisé sur cette carte (autre les GND) sont les broches :

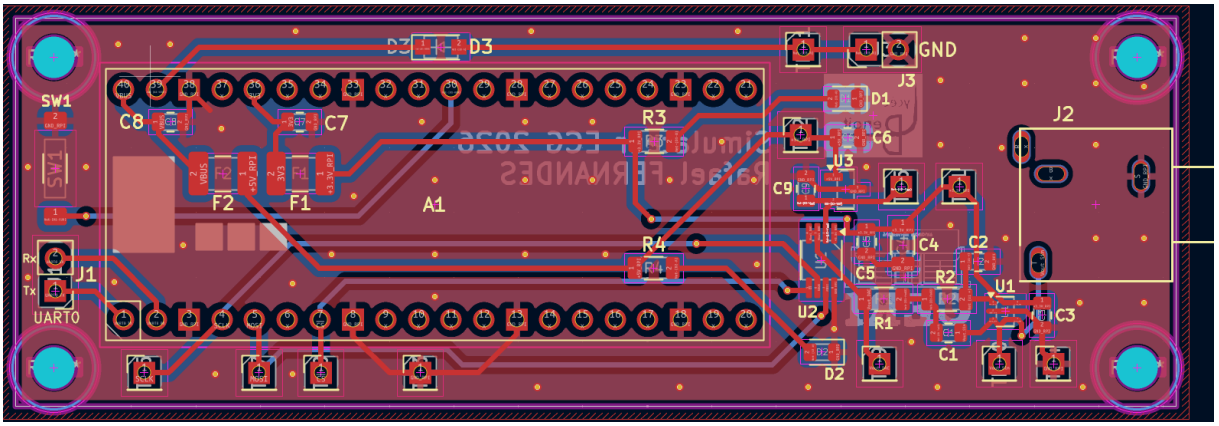
RX, CSn, SCK et TX ainsi que le 2e broche CSN, les broches VBUS et VSYS, 3,3V ainsi que la RUN.

Une fois que toute la partie schema de cablage a été faites, je suis passer au routage de la carte ce qui consiste à mettre tous les les composent a leurs place afin d'y ajouter les pistes de cuivre qui seront ensuite créé sur les PCB (circuit imprimé) qui ont été commandés chez JLCPCB.

une fois rooté la carte ressemble à ceci (sans plan de masse) :



avec plan de masse :

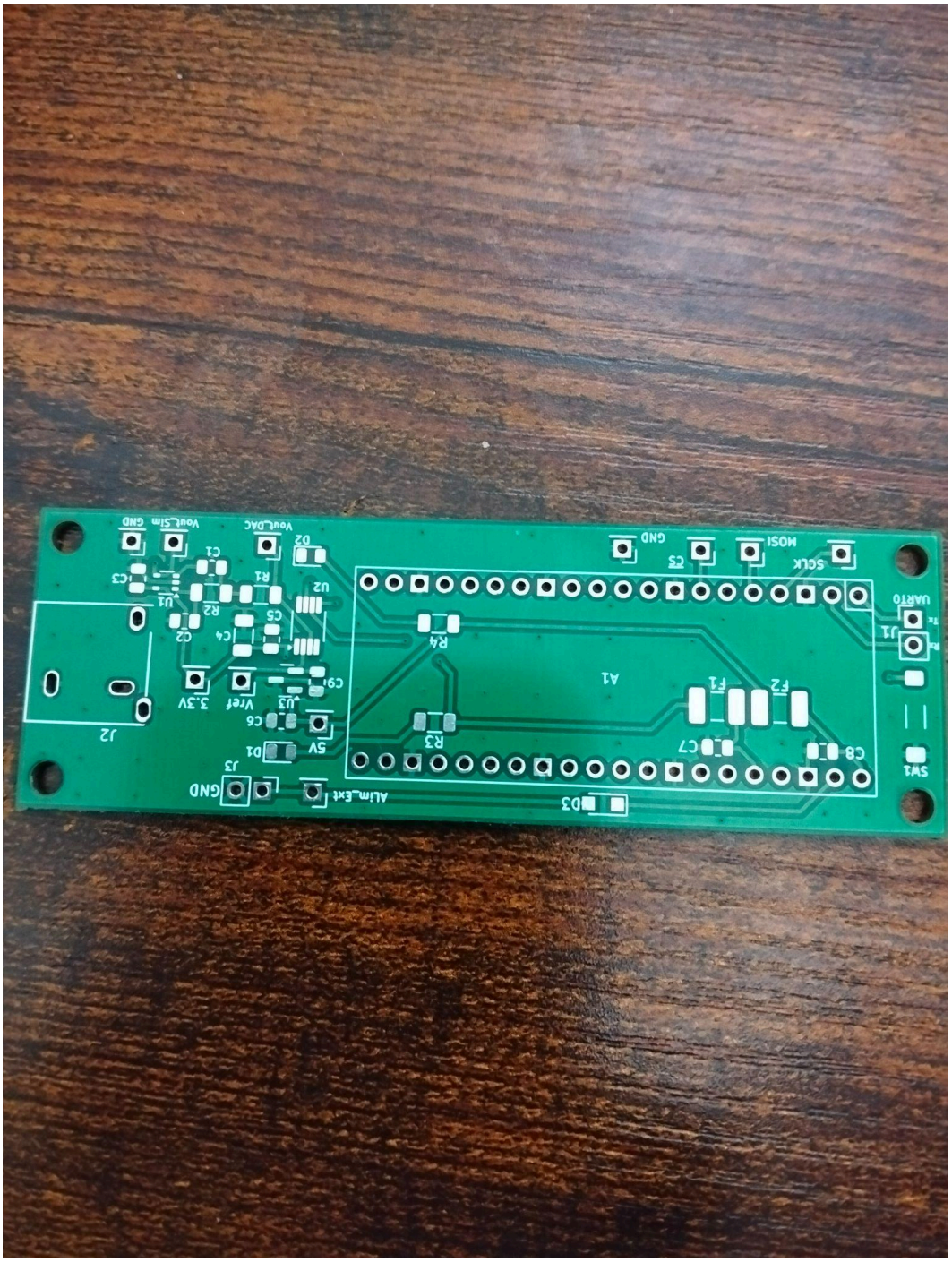


Une fois que tout le routage est fini, et qu'il n'y a plus d'erreur, il faut vérifier qu'il n'y a pas de surplus de piste inutile et qu'il n'y a pas d'angle droit.

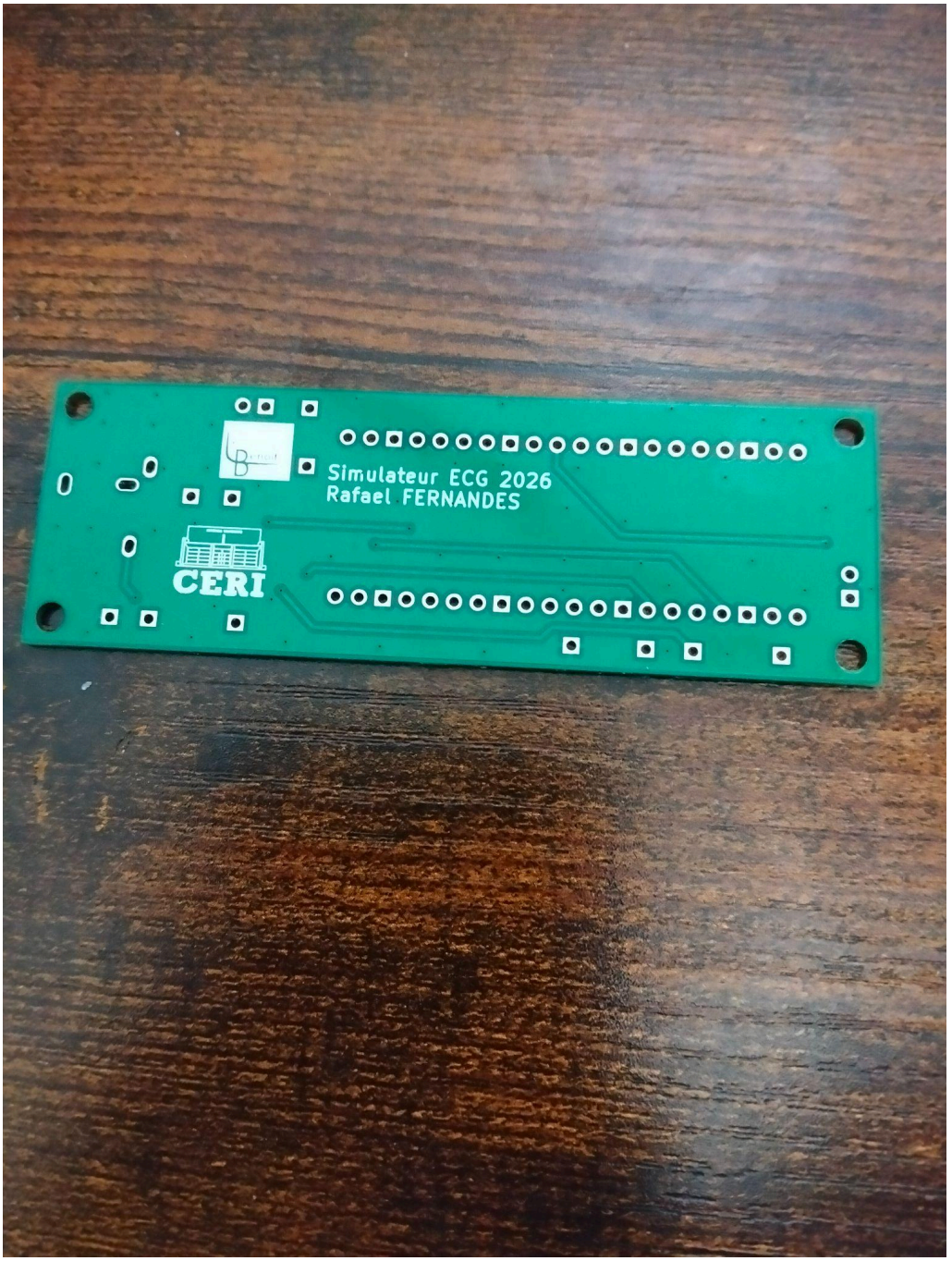
Après avoir fini le routage, il faut ensuite passer à la soudure des composants sur la nouvelle carte vierge.

la carte sans composant ressemble à ceci :

recto :


Vout_Sim
GND
Vout_DAC
D2
C1
C3
R1
U2
U1
R2
C2
C4
C5
R4
A1
F1
F2
C9
U3
3.3V
Vref
C6
J2
5V
D1
R3
C7
C8
J3
GND
Alim_Ext
D3
SCLK
MOSI
CS
GND
UART0
J1
SW1

verso :



Lorsque que l'on soude, il y a un ordre précis à suivre. l'ordre est le suivant : Condensateurs, résistances, led et led shotkey, et les composant comme les convertisseurs qui sont passer d'un côté et souder au four à refusion.

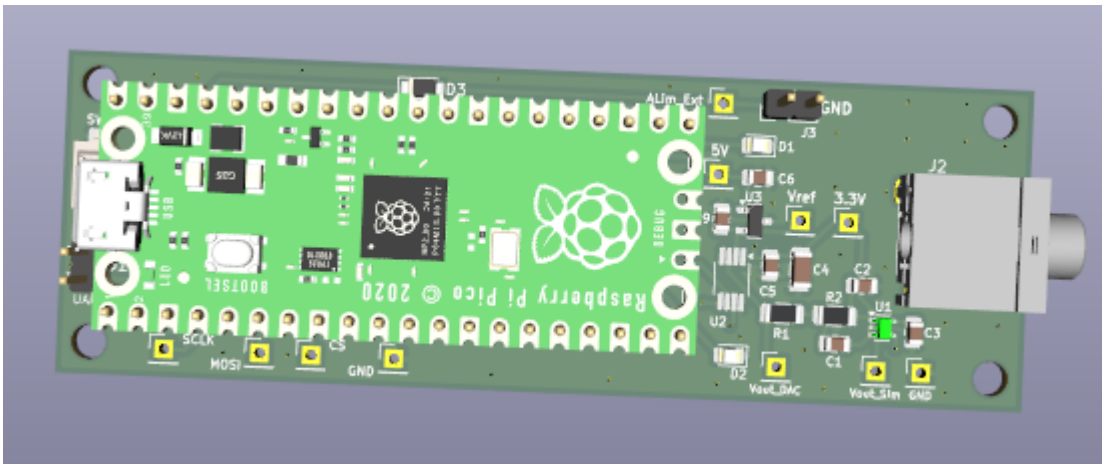
four à refusion :

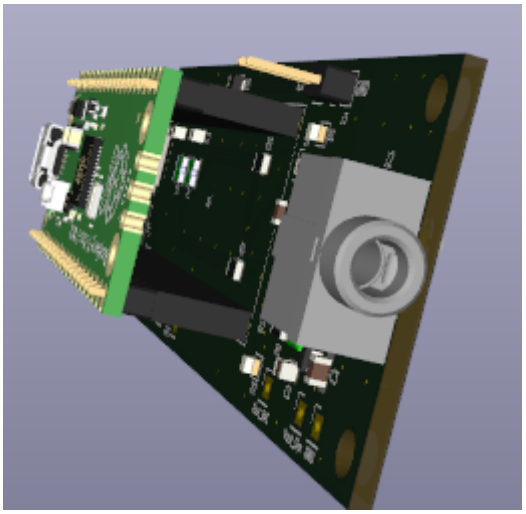


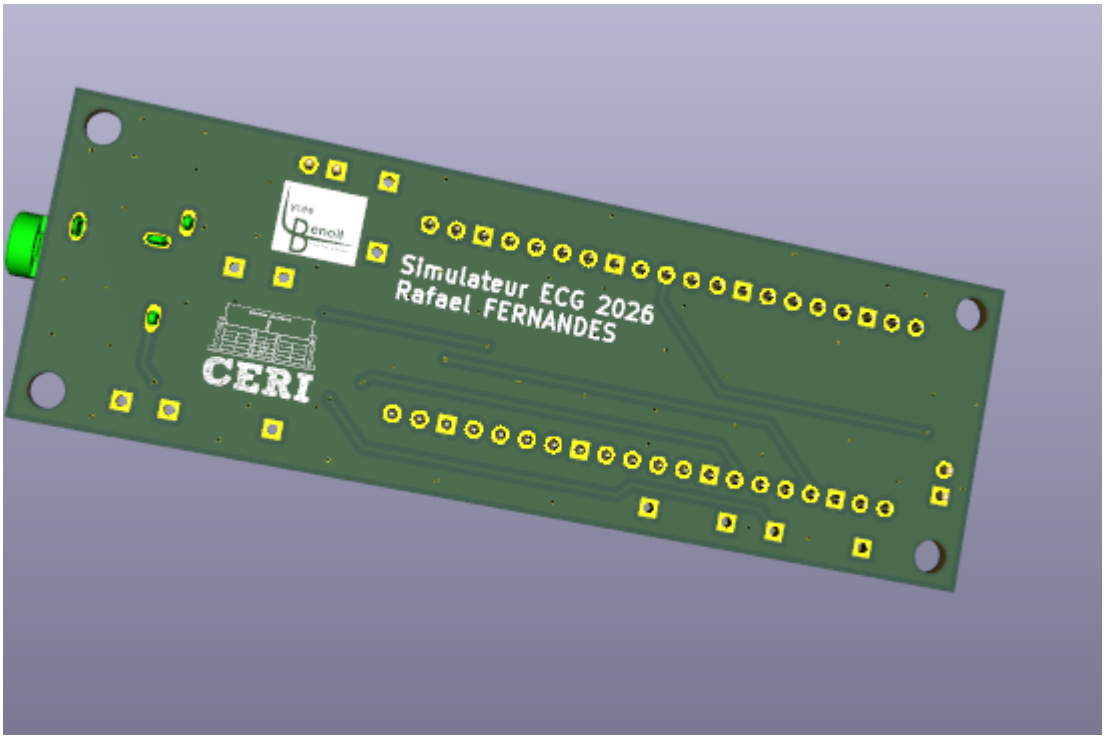
Ensuite, les composants suivants soudé sont les traversant, comme leurs nom l'indique, ce sont les composants qui traversent la carte. Sur le simulateur, les composants traversants sont : seulement les points de tests ainsi que la base du Raspberry PiCo. Ces composants la sont soudés à la main via un fer à souder.

Et pour les derniers composants, ce sont les composants qui ne passent pas au four mais qui ne sont pas des traversants comme un bouton poussoir.

Une fois que tout est mis, la carte ressemble un peu près à ça :







La carte est composée de ceci :

Reference	Qty	Value	Footprint	Code commande
A1	1	RaspberryPi_Pico	0_ModulesProjet:RPi_Pico_sur_Support	Farnell 3643332
C1,C2,C3,C5	4	100nF	Capacitor_SMD:C_0805_2012Metric	RS 264-4422
C4	1	10µF	Capacitor_SMD:C_1206_3216Metric	RS 103-4164
C6	1	0.47µF	Capacitor_SMD:C_0805_2012Metric	RS 698-3434

C7,C8	2	1µF	Capacitor_SMD:C_0805_2012Metric	RS 264-4450
C9	1	4.7µF	Capacitor_SMD:C_0805_2012Metric	RS 691-1189
D1	1	Jaune	LED_SMD:LED_0805_2012Metric	RS 486-0519
D2	1	Verte	LED_SMD:LED_0805_2012Metric	RS 654-5773
D3	1	PMEG3010ER	Diode_SMD:Nexperia_CFP3_SOD-123W	RS 102-6855
F1,F2	2	400mA 0.2A	0_ModulesProjet:Polyswitch_MICROSMD110F-2	RS 517-7105
J1,J3	2	Conn_01x02	Connector_PinHeader_2.54mm:PinHeader_1x02_P2.54mm_Vertical	Gotronic 08000
J2	1	KLBR4	0_ModulesProjet:KLBR4	RS 505-1479
R1,R2	2	39kΩ	Resistor_SMD:R_1206_3216Metric	RS 223-2467
R3	1	330	Resistor_SMD:R_1206_3216Metric	RS 679-2049
R4	1	1500	Resistor_SMD:R_1206_3216Metric	RS 223-2287
SW1	1	SW_Push	0_ModulesProjet:Tactile_SMD_3x6x2.5mm	RS 102-361
TP1	1	3.3V	0_ModulesProjet:Point_Test	Gotronic 08000
TP2	1	5V	0_ModulesProjet:Point_Test	Gotronic 08000
TP3,TP4	2	GND	0_ModulesProjet:Point_Test	Gotronic 08000
TP5	1	Vref	0_ModulesProjet:Point_Test	Gotronic 08000
TP6	1	SCLK	0_ModulesProjet:Point_Test	Gotronic 08000
TP7	1	MOSI	0_ModulesProjet:Point_Test	Gotronic 08000
TP8	1	~{CS}	0_ModulesProjet:Point_Test	Gotronic 08000
TP9	1	Vout_DAC	0_ModulesProjet:Point_Test	Gotronic 08000
TP10	1	Vout_Sim	0_ModulesProjet:Point_Test	Gotronic 08000
TP11	1	ALim_Ext	0_ModulesProjet:Point_Test	Gotronic 08000
U1	1	LMV841	Package_TO_SOT_SMD:SOT-353_SC-70-5	Mouser 926-LMV841MGX/NOPB
U2	1	DAC8551IxDGK	0_ModulesProjet:VSSOP-8_3x3mm_P0.65mm	Mouser 595-DAC8551IDGKR

U3	1	REF3030	Package_TO_SOT_SMD:SOT-23	Mouser 595-REF3030AIDBZT

PARTIE PERSONNELLE
PAUL MICHEL
(ER 2)

6. PARTIE PERSONNELLE - PAUL MICHEL

6.1 Introduction personnelle

Pour mon projet final, M. Hortolland m'a assigné au projet nommé « Électrocardiogramme », qui sera par la suite transmis aux étudiants de l'université CERI à Avignon. Je suis désigné pour prendre en charge la conception d'une nouvelle carte Holter pour l'électrocardiogramme.

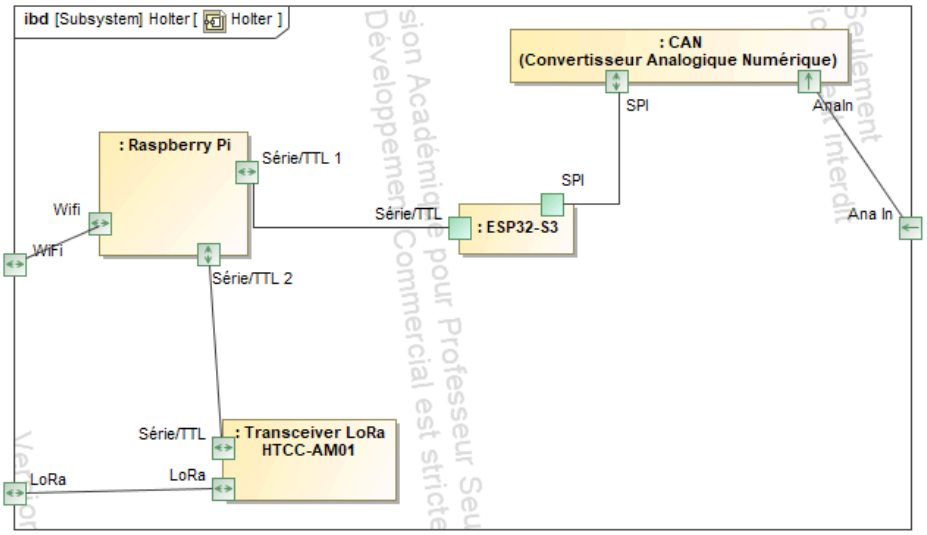
J'ai commencé par comprendre et analyser le cahier des charges, ce qui m'a permis de mieux cerner le fonctionnement du système et les objectifs qui m'ont été fixés. Je vais donc vous présenter les différents tests et manipulations que j'ai effectués durant ce début de projet :

- Préparation des outils et du matériel mis à disposition, notamment une carte Raspberry Pi, différents appareils de mesure et les documentations techniques.
- Revue du travail effectué par les étudiants de l'année 2024-2025.
- Mise en place et configuration de la Raspberry Pi sur différentes cartes SD (versions OS 32 et 64 bits) pour avoir accès au logiciel de simulation ECG.
- Pour finir, j'ai réalisé un test sur la carte Holter créée par les étudiants de l'année 2024-2025.

Pour illustrer la structure matérielle de notre solution, voici le diagramme de bloc interne (IBD) du sous-système Holter.

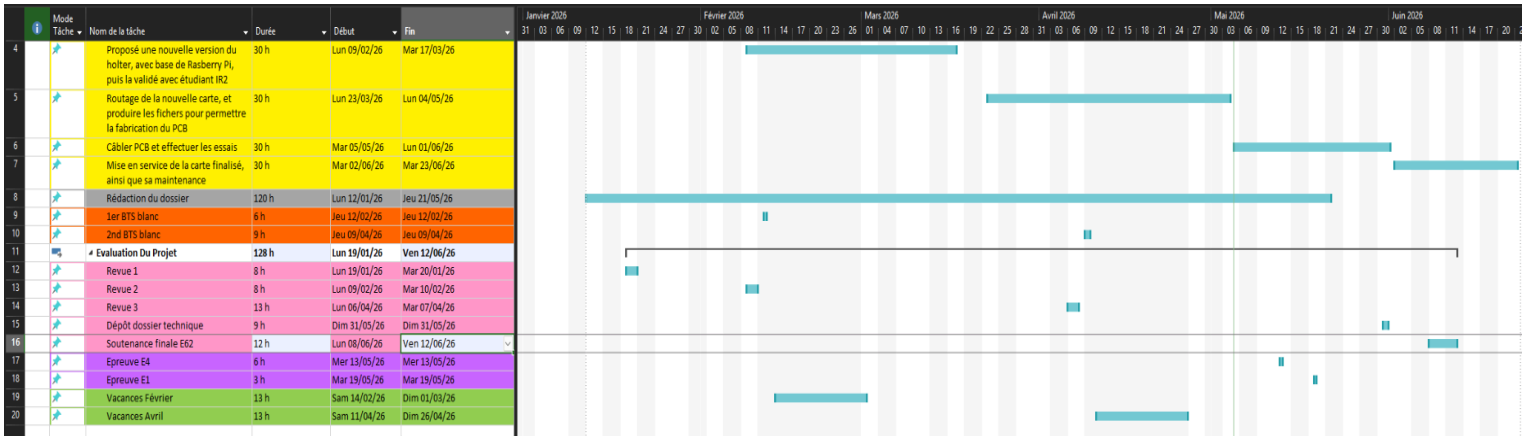
Ce schéma met en évidence l'organisation générale des composants et la distribution des tâches

- Le traitement et la numérisation : L'acquisition des signaux physiologiques s'effectue via un convertisseur analogique-numérique (CAN), relié directement au microcontrôleur ESP32-S3. Ce dernier collabore avec une carte Raspberry Pi, qui centralise une partie de l'intelligence du système.
- La connectivité : Pour l'export et le suivi des données, l'architecture intègre deux solutions complémentaires. D'un côté, une communication locale à haut débit en Wi-Fi, et de l'autre, une liaison longue portée et basse consommation via le module LoRa.



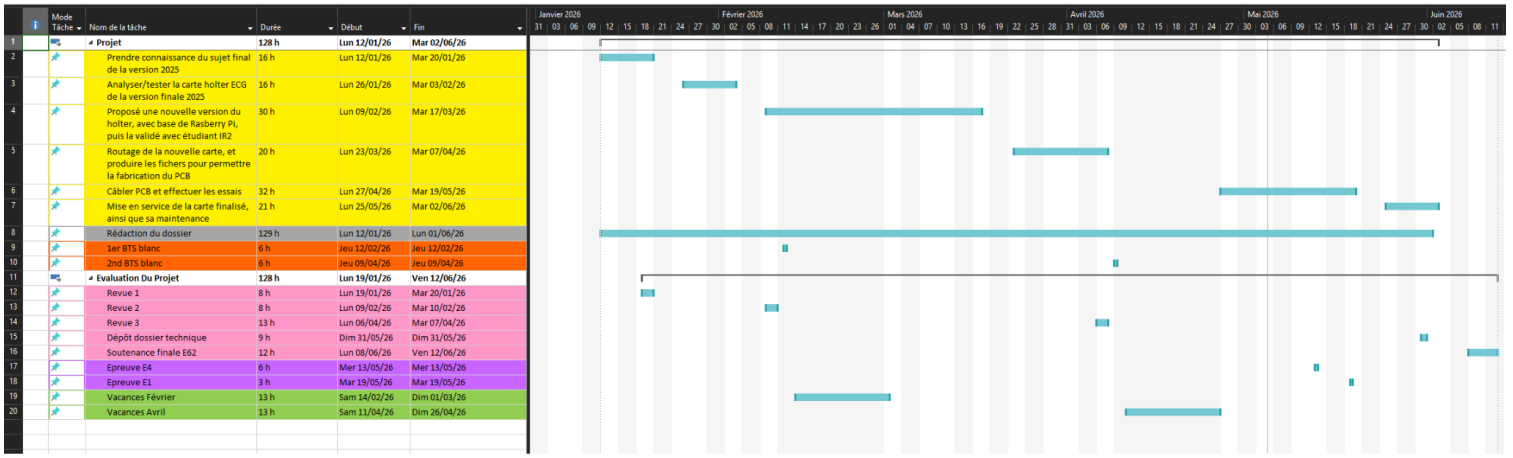
6.2 Organisation du Projet – Diagramme de Gantt

Diagramme Prévisionnel :



	Mode Tâche	Nom de la tâche	Durée	Début	Fin
4		Proposé une nouvelle version du holter, avec base de Rasberry Pi, puis la validé avec étudiant IR2	30 h	Lun 09/02/26	Mar 17/03/26
5		Routage de la nouvelle carte, et produire les fichers pour permettre la fabrication du PCB	30 h	Lun 23/03/26	Lun 04/05/26
6		Câbler PCB et effectuer les essais	30 h	Mar 05/05/26	Lun 01/06/26
7		Mise en service de la carte finalisé, ainsi que sa maintenance	30 h	Mar 02/06/26	Mar 23/06/26
8		Rédaction du dossier	120 h	Lun 12/01/26	Jeu 21/05/26
9		1er BTS blanc	6 h	Jeu 12/02/26	Jeu 12/02/26
10		2nd BTS blanc	9 h	Jeu 09/04/26	Jeu 09/04/26
11		**Evaluation Du Projet**	**128 h**	**Lun 19/01/26**	**Ven 12/06/26**
12		Revue 1	8 h	Lun 19/01/26	Mar 20/01/26
13		Revue 2	8 h	Lun 09/02/26	Mar 10/02/26
14		Revue 3	13 h	Lun 06/04/26	Mar 07/04/26
15		Dépôt dossier technique	9 h	Dim 31/05/26	Dim 31/05/26
16		Soutenance finale E62	12 h	Lun 08/06/26	Ven 12/06/26
17		Epreuve E4	6 h	Mer 13/05/26	Mer 13/05/26
18		Epreuve E1	3 h	Mar 19/05/26	Mar 19/05/26
19		Vacances Février	13 h	Sam 14/02/26	Dim 01/03/26
20		Vacances Avril	13 h	Sam 11/04/26	Dim 26/04/26

Diagramme Finalisé :



	Mode Tâche	Nom de la tâche	Durée	Début	Fin
1		**Projet**	**128 h**	**Lun 12/01/26**	**Mar 02/06/26**
2		Prendre connaissance du sujet final de la version 2025	16 h	Lun 12/01/26	Mar 20/01/26
3		Analyser/tester la carte holter ECG de la version finale 2025	16 h	Lun 26/01/26	Mar 03/02/26
4		Proposé une nouvelle version du holter, avec base de Rasberry Pi, puis la validé avec étudiant IR2	30 h	Lun 09/02/26	Mar 17/03/26
5		Routage de la nouvelle carte, et produire les fichers pour permettre la fabrication du PCB	20 h	Lun 23/03/26	Mar 07/04/26
6		Câbler PCB et effectuer les essais	32 h	Lun 27/04/26	Mar 19/05/26
7		Mise en service de la carte finalisé, ainsi que sa maintenance	21 h	Lun 25/05/26	Mar 02/06/26
8		Rédaction du dossier	129 h	Lun 12/01/26	Lun 01/06/26
9		1er BTS blanc	6 h	Jeu 12/02/26	Jeu 12/02/26
10		2nd BTS blanc	6 h	Jeu 09/04/26	Jeu 09/04/26
11		**Evaluation Du Projet**	**128 h**	**Lun 19/01/26**	**Ven 12/06/26**
12		Revue 1	8 h	Lun 19/01/26	Mar 20/01/26
13		Revue 2	8 h	Lun 09/02/26	Mar 10/02/26
14		Revue 3	13 h	Lun 06/04/26	Mar 07/04/26
15		Dépôt dossier technique	9 h	Dim 31/05/26	Dim 31/05/26
16		Soutenance finale E62	12 h	Lun 08/06/26	Ven 12/06/26
17		Epreuve E4	6 h	Mer 13/05/26	Mer 13/05/26
18		Epreuve E1	3 h	Mar 19/05/26	Mar 19/05/26
19		Vacances Février	13 h	Sam 14/02/26	Dim 01/03/26
20		Vacances Avril	13 h	Sam 11/04/26	Dim 26/04/26

La modification la plus importante concerne le début du projet. Là où le prévisionnel commençait directement par la proposition d'une nouvelle version, le planning finalisé ajoute une phase préparatoire essentielle :

- Intégration de l'analyse : Les deux premières étapes du diagramme finalisé (« Prendre connaissance du sujet final » et « Analyser/tester la carte Holter ECG ») permettent une mise en route plus progressive avant de commencer la conception de la nouvelle version.
- Réajustement des durées :
    - La phase de « Routage » a été optimisée, passant de 30h à 20h.
    - Le « Câblage et essais » a été légèrement rallongé, passant de 30h à 32h.
    - La « Mise en service » a été réduite de 30h à 21h.

Ce diagramme a été créé et généré via le logiciel Microsoft Project, ce qui m'a permis de planifier et de suivre toutes les étapes de mon projet sous forme de diagramme de Gantt. En y inscrivant mes tâches, comme la phase de tests sur WaveForms ou la modification du schéma en février, j'ai pu lier les étapes entre elles, définir les dates limites et voir instantanément l'impact d'un retard sur le reste du calendrier. C'est l'outil qui a structuré toute l'organisation temporelle de mon travail.

6.3 Présentation du travail

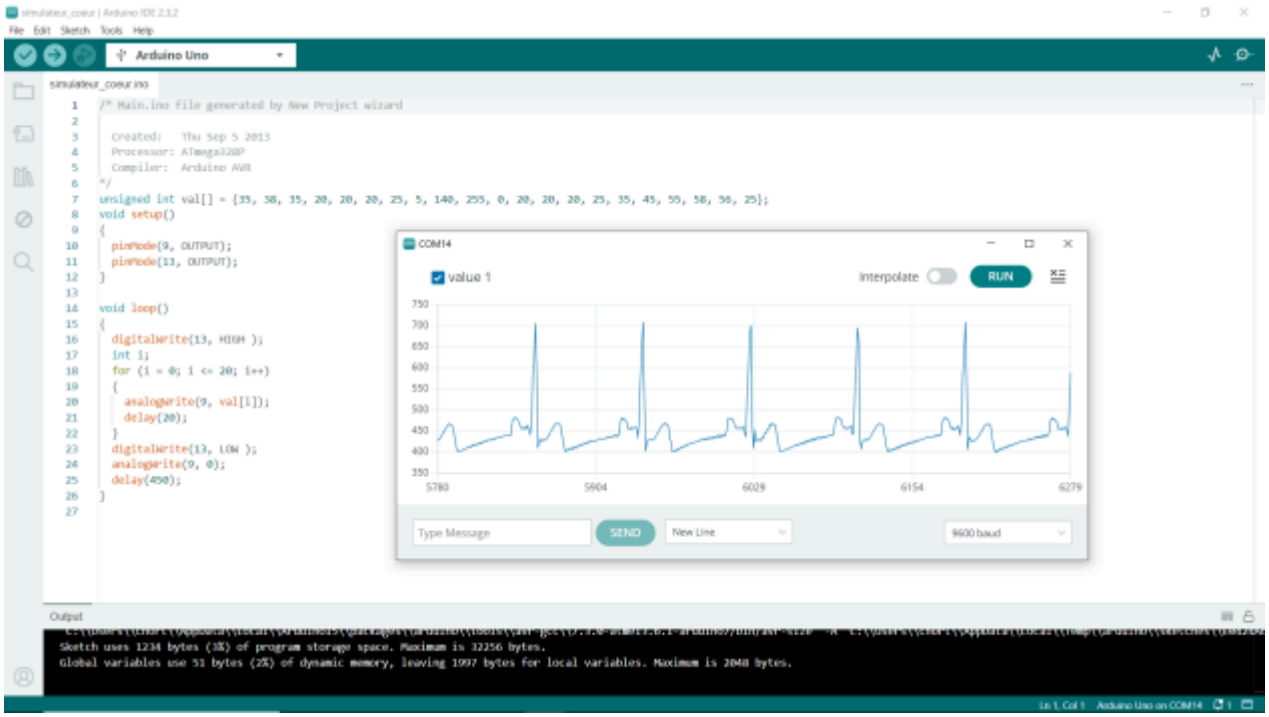
6.3.1 Mise en place de mon environnement de travail

Dans le cadre de la prise en main du projet, la première étape a consisté à configurer l'ensemble du matériel nécessaire à la mise en place de mon environnement de travail. J'ai ainsi récupéré un kit initial comprenant la carte Holter développée par les étudiants de l'année précédente (2024-2025), une carte Raspberry Pi faisant office de poste central, un module microcontrôleur ESP32, ainsi que la connectique et les câbles d'alimentation associés.

Afin de rendre ce système opérationnel et d'interagir directement avec la Raspberry Pi, j'ai complété cette configuration matérielle en y raccordant les périphériques standard, à savoir un écran, un clavier et une souris. Enfin, pour anticiper les phases de diagnostic, de test et de validation des signaux électriques, j'ai installé à mon poste de travail les appareils de mesure indispensables : un multimètre numérique et un oscilloscope de laboratoire.

6.3.2 Revue du travail effectué par les étudiants 2024-2025

L'année dernière les étudiants avaient fait ce même projet, le signal obtenu était distribué d'une autre manière, grâce à un code Arduino qui récupérait les données du convertisseur et qui interprétait ces données par une courbe style ECG.



Code arduino des étudiants 2024-2025

Leur carte Holter est donc utilisable pour faire des tests mais pas réutilisable pour la version 2026.

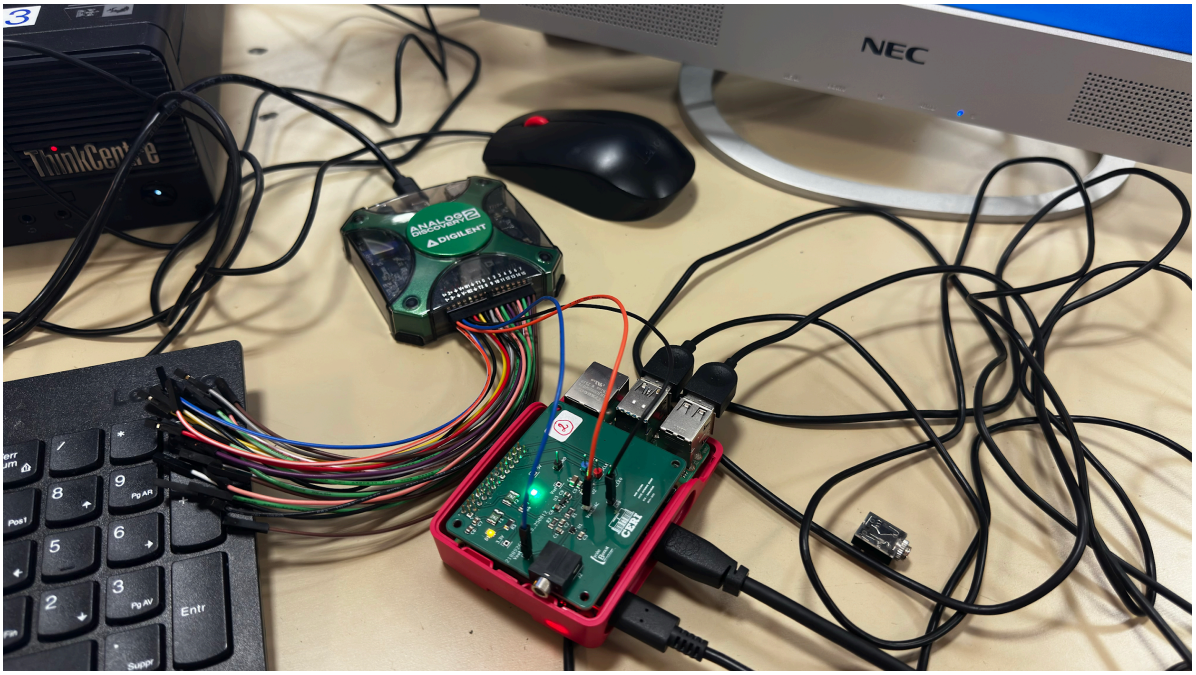
6.3.3 Mise en place et configuration de la Raspberry Pi

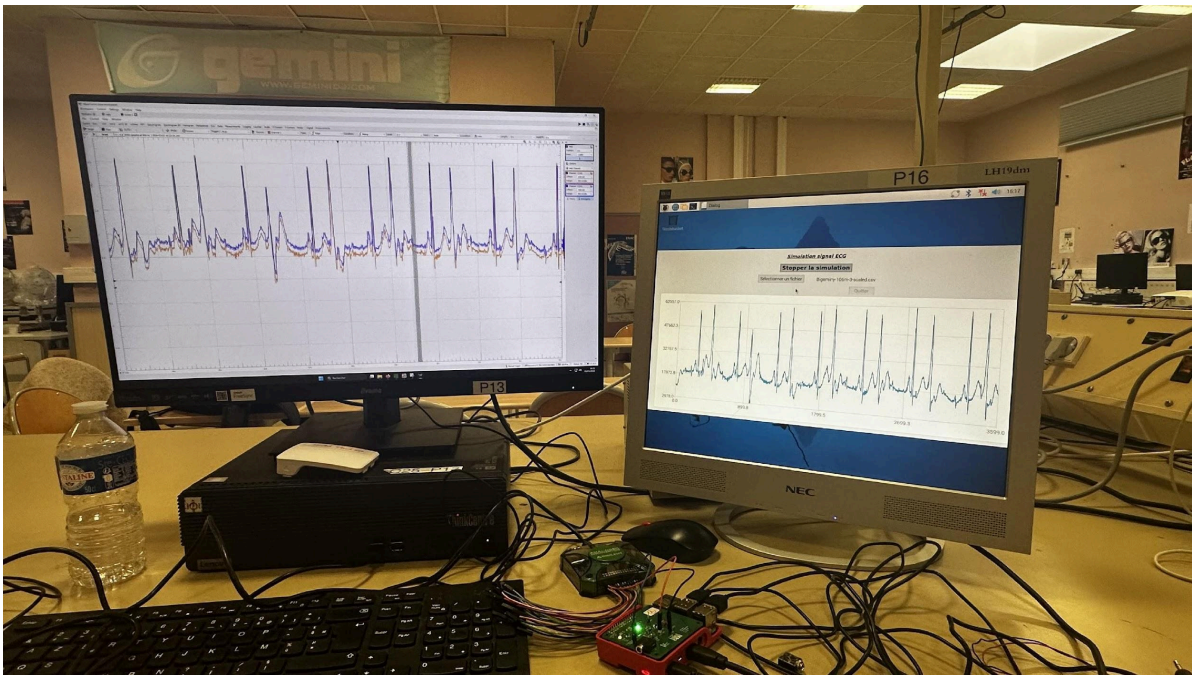
Durant la configuration (duplication de la carte 64 bits de Mr Hortolland), j'ai rencontré quelque problème, car je pensais avoir accès au logiciel ECG en configurant ma carte en 64 bits, or je ne pouvais avoir accès au logiciel qu'avec un système d'exploitation en 32 bits.
J'ai donc dupliqué une deuxième carte en 32 bits afin d'avoir accès à ce logiciel qui met primordiale pour la suite du projet.

6.3.4 Test de la carte Holter par les étudiants 2024-2025

J'ai effectué un test afin de déterminer s'il était nécessaire d'intégrer un filtre passe-bas sur mon schéma électronique pour « lisser » le signal obtenu. Pour mener à bien cette manipulation, j'ai mobilisé la carte de simulation, le prototype Holter de l'année précédente, la Raspberry Pi ainsi que le module Analog Discovery utilisé en mode oscilloscope (Scope).

Bien que cette carte Holter de la promotion 2024-2025 ait servi de base d'expérimentation pour valider ces choix techniques, elle ne sera pas réutilisable en l'état pour le projet 2025-2026.



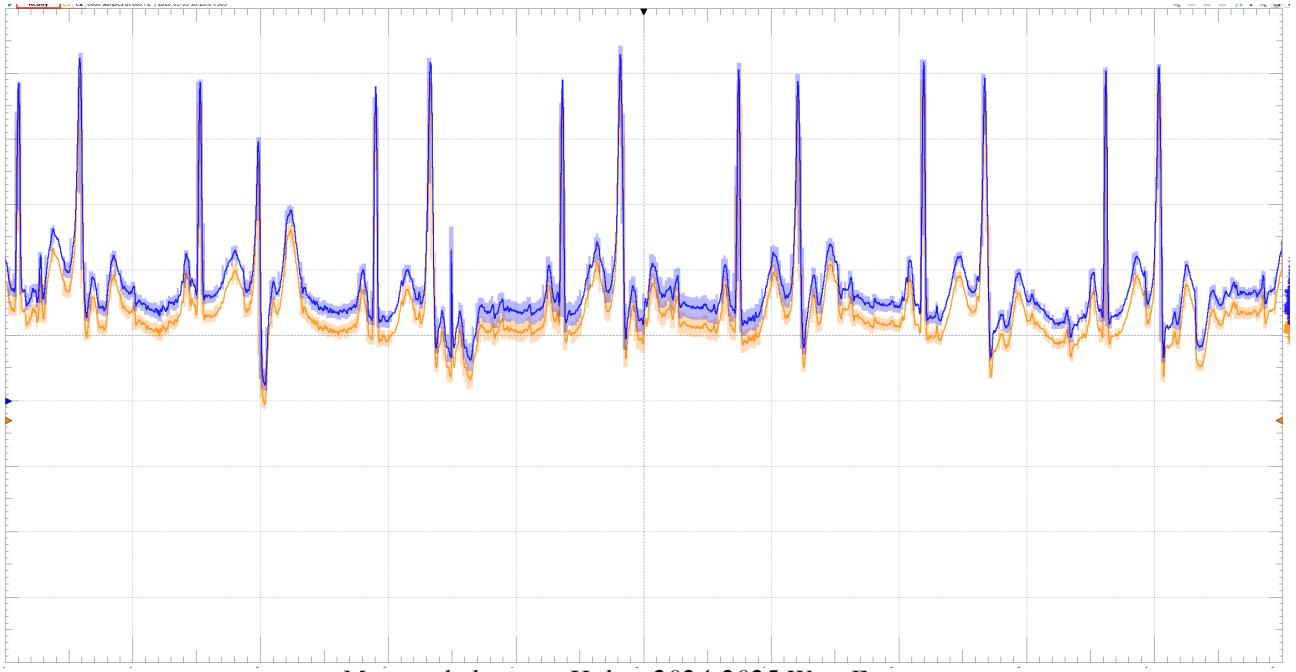


Test de la carte Holter 2024-2025

J’ai tout d’abord injecté un signal depuis le logiciel vers la carte en utilisant différents points de test de l’Analog Discovery(Vout DAC, Vout SIM).

À la fin de ce test, j’en ai conclu qu'il n'était pas nécessaire d’intégrer un filtre passe-bas. En effet, le signal (illustré ci-dessous) ne présentait aucune perturbation ni bruit significatif.

Le chronogramme met en évidence en bleu le signal envoyé par la Raspberry Pi, et en orange le signal reçu par la carte Holter.



Mesure de la carte Holter 2024-2025 WaveForm

J’ai effectué cette mesure et ce test sur le logiciel WaveForm c’est une interface puissante conçue pour piloter les instruments de mesure compacts, tels que l'Analog Discovery. Il transforme un ordinateur en un véritable laboratoire électronique complet. Le logiciel permet de visualiser et de configurer plusieurs outils simultanément : un oscilloscope, un générateur de formes d'onde arbitraires, un analyseur logique et une alimentation programmable.

6.3.5 Création du schéma structurel de la carte

Au début du mois de février, j'ai récupéré le schéma structurel de base réalisé par les étudiants de la promotion 2024-2025. Pour entamer mes modifications, M. Hortolland m’a fourni un schéma provisoire qui intégrait plusieurs composants redondants ou en surnombre.

Mon travail consistait à faire un tri parmi ces composants en effectuant des tests comparatifs. Cette étape essentielle m'a permis de sélectionner et de dimensionner correctement les composants par rapport aux exigences fixées par M. Hortolland.

En ce qui concerne le bloc d'alimentation autonome (constitué de BT1 et U5, visibles sur le schéma provisoire), il a été entièrement supprimé car il s'est avéré inutile. Le système est désormais alimenté directement par les lignes de tension (+3,3 V et +5 V) fournies par les cartes microcontrôleurs (Raspberry Pi et XIAO). Cette simplification permet de réduire drastiquement l'encombrement de la carte ainsi que son coût de fabrication.

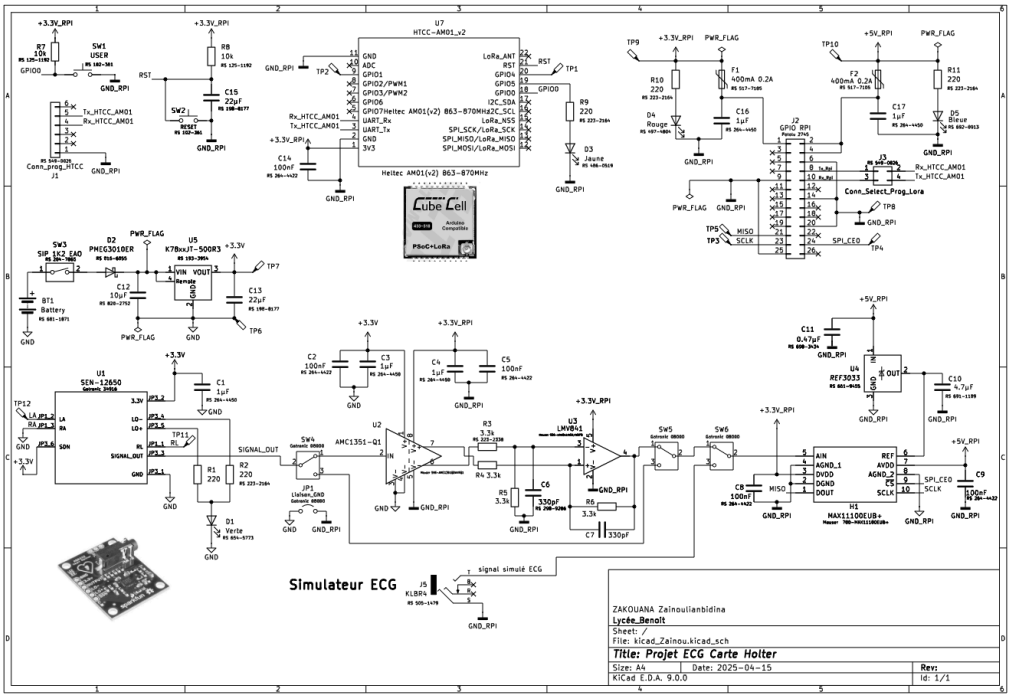


Schéma Étudiant 2024-2025

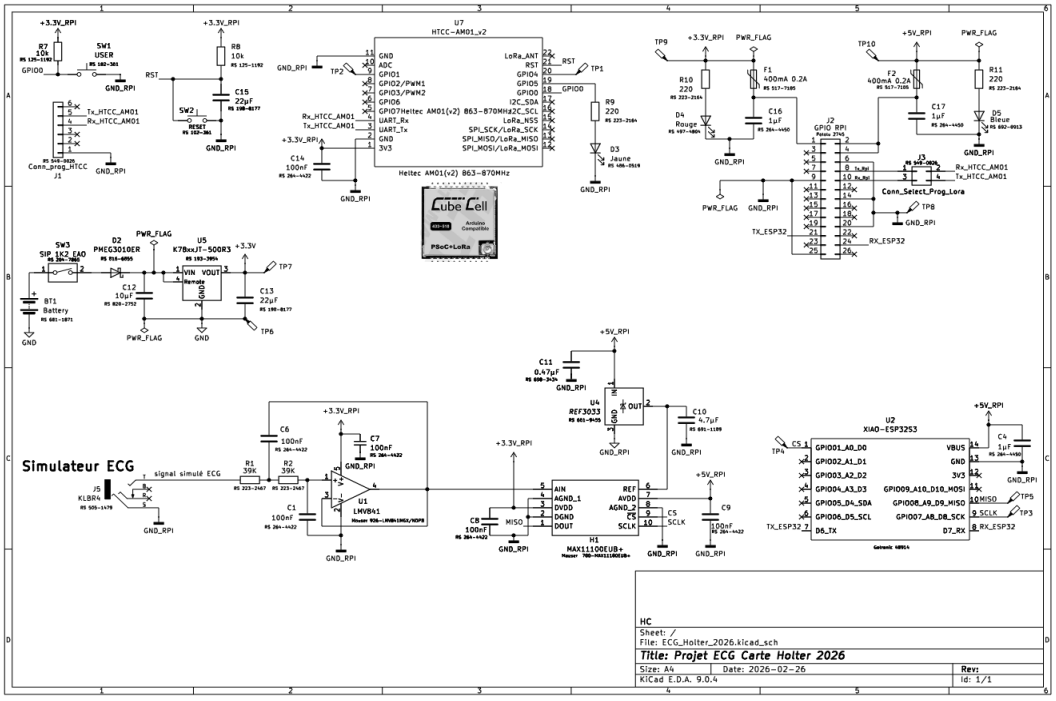


Schéma Provisoire

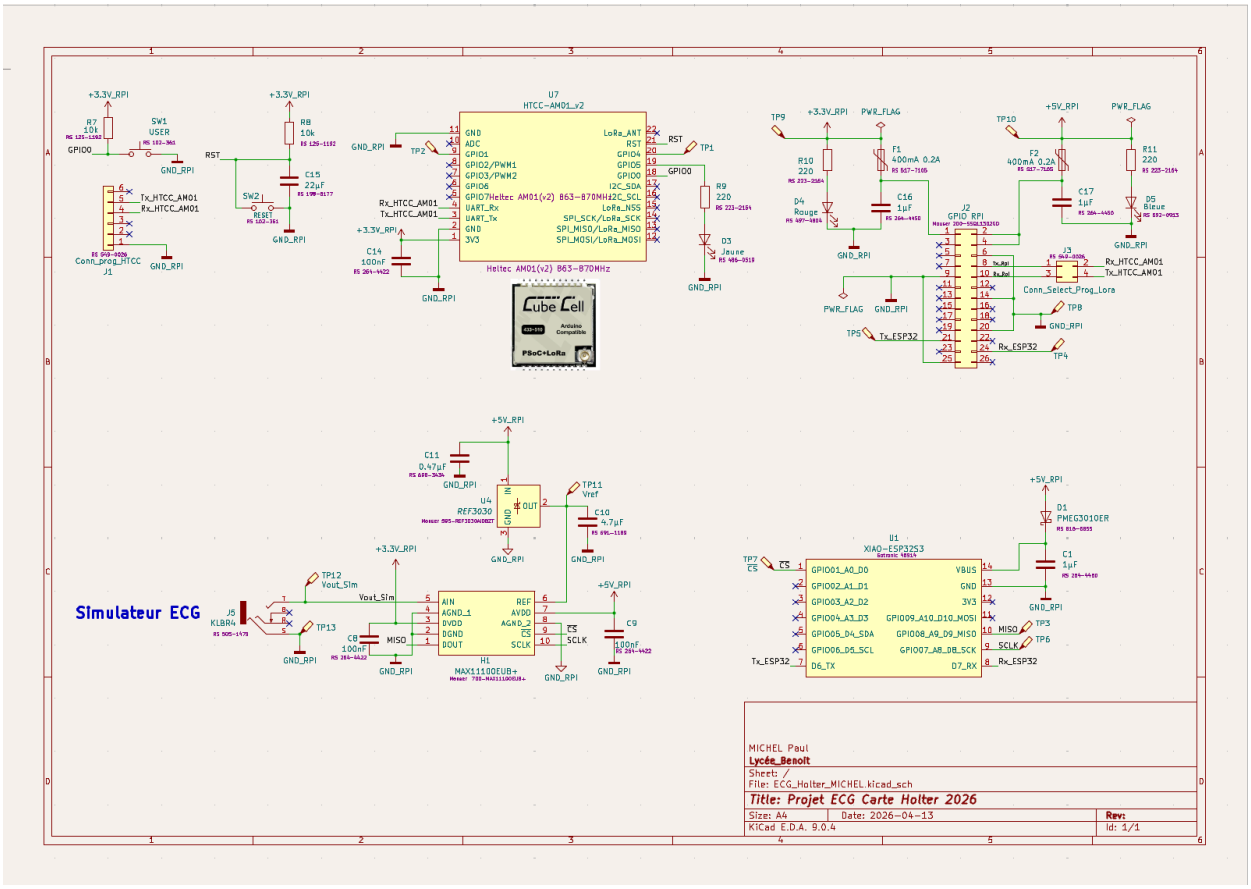


Schéma Final 2025-2026

6.3.6 Routage de la nouvelle carte Holter

Une fois mon schéma structurel validé après de nombreuses corrections de la part de M. Hortolland, j'ai pu passer au routage définitif de la carte. Nous avions une date limite pour rendre les fichiers de routage finis, qui était fixée au 7 avril 2026. Cette échéance était cruciale pour pouvoir commander les circuits imprimés ainsi que les composants à temps, afin qu'ils arrivent pendant les vacances de Pâques.

J'ai rencontré plusieurs difficultés lors du routage : certaines pistes étaient trop longues, manquaient de directivité, présentaient des angles trop importants (à éviter en haute fréquence/signal intègre) ou un mauvais placement des composants. M. Hortolland ainsi que certains de mes camarades m'ont accompagné et conseillé pour corriger et surmonter ces erreurs.

Pour la conception de ce circuit imprimé, j'ai opté pour un routage double face : la face supérieure est représentée en rouge et la face inférieure en bleu. Les pistes rouges transportent la majorité des signaux de données et de commande, tandis que les pistes bleues servent principalement à effectuer les croisements. Cette méthode permet de limiter l'usage de vias inutiles, ce qui m'a permis de conserver un plan de masse le plus continu possible.

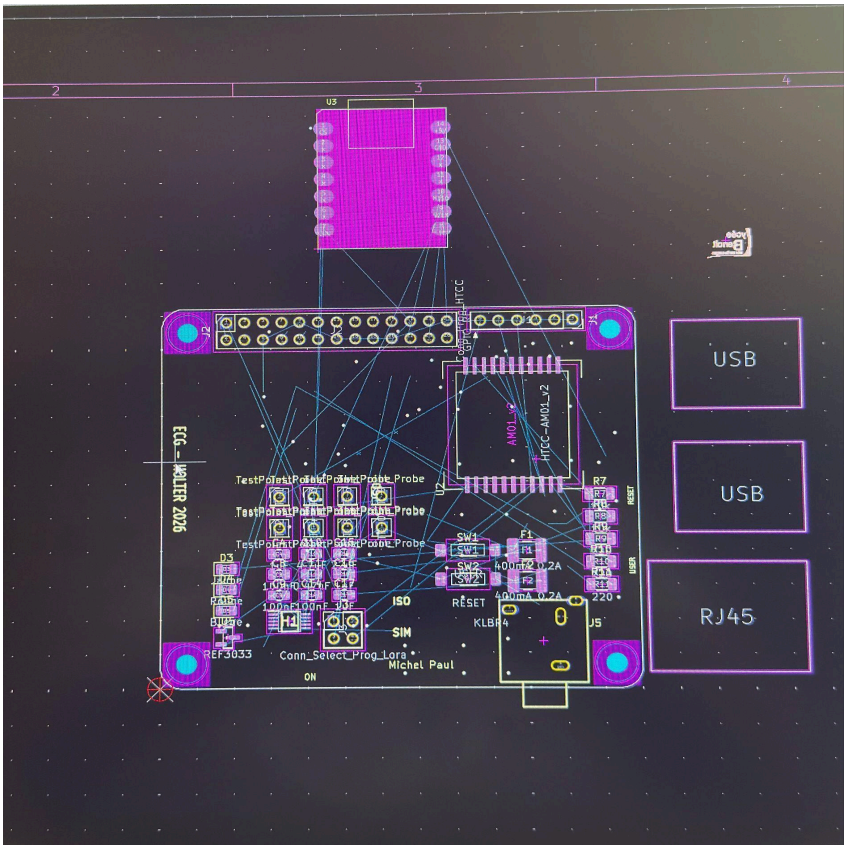


Photo au début du routage

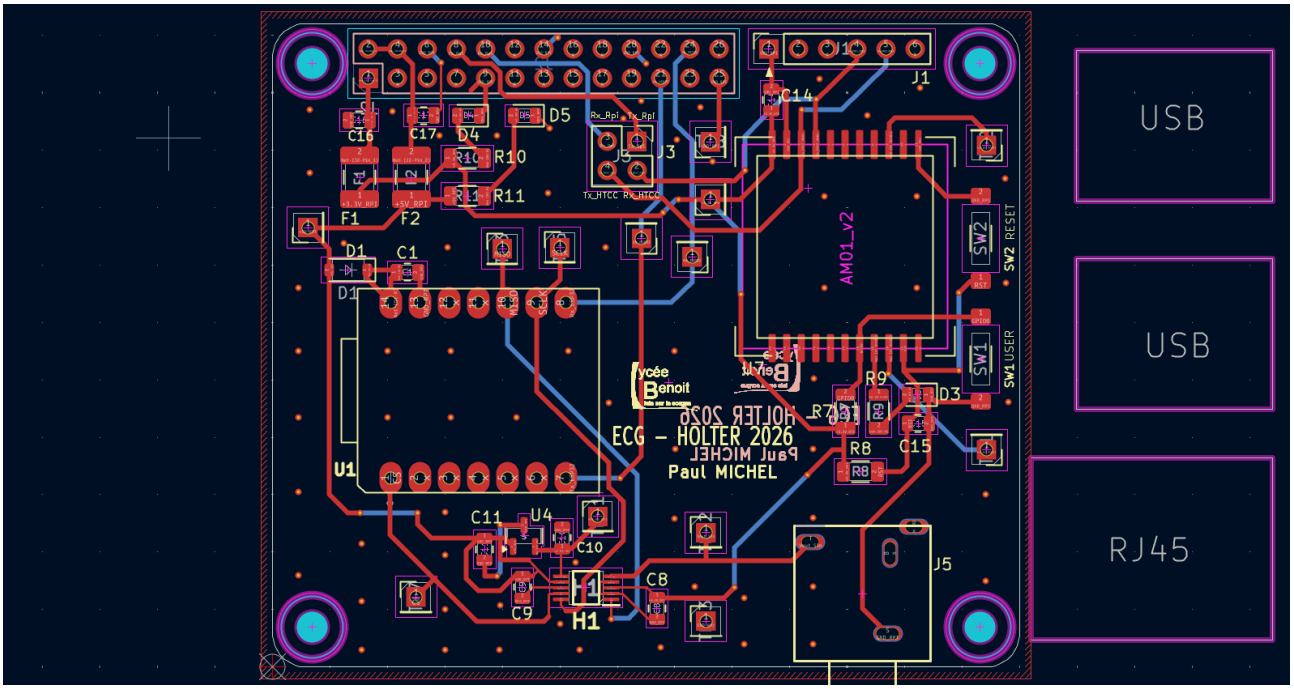


Photo à la fin du routage

J’ai par la suite, créé un tableur, ce document m’est indispensable car il liste tout ce qu'il faut pour fabriquer la carte sans faire d'erreurs. Il permet de regrouper au même endroit le nom des composants, leurs caractéristiques techniques et les codes pour les commander chez les fournisseurs. Grâce à lui, on connaît précisément le prix de revient du projet (45,76 €) et on est sûr de ne rien oublier au moment de souder.

Reference	Qty	Value	Code commande	Prix unitaire
C1,C16,C17	3	1µF	RS 264-4450	0,04 €
C8,C9,C14	3	100nF	RS 264-4422	0,11 €
C10	1	4.7µF	RS 691-1189	0,11 €
C11	1	0.47µF	RS 698-3434	0,12 €
C15	1	22µF	RS 198-8177	0,14 €
D1	1	PMEG3010ER	RS 816-6855	0,22 €
D3	1	Jaune	RS 486-0519	0,18 €
D4	1	Rouge	RS 497-4804	0,13 €
D5	1	Bleue	RS 692-0913	0,20 €
F1,F2	2	400mA 0.2A	RS 517-7105	0,41 €
H1	1	MAX11100EUB+	Mouser 700-MAX11100EUB+	13,23 €
J1	1	Conn_prog_HTCC	RS 670-3487	1,63 €
J2	1	GPIO RPI	Pololu 2745	2,54 €
J3	1	Conn_Select_Prog_Lora	RS 549-0026	4,04 €
J5	1	KLBR4	RS 505-1479	3,23 €
R7,R8	2	10k	RS 125-1192	0,02 €
R9,R10,R11	3	220	RS 223-2164	0,03 €
SW1	1	USER	RS 102-361	0,65 €
SW2	1	RESET	RS 102-361	0,65 €
TP1	1	TP1	Gotronic 08000	0,60 €
TP2	1	TP2	Gotronic 08000	0,60 €
TP3,TP4,TP6	3	SPI_CE0	Gotronic 08000	0,60 €
TP5	1	MISO	Gotronic 08000	0,60 €
TP8	1	GND_RPI	Gotronic 08000	0,60 €
TP9	1	3.3V_RPI	Gotronic 08000	0,60 €
TP10	1	5V_Rpi	Gotronic 08000	0,60 €
U1	1	XIAO-ESP32S3	Gotronic 48914	0,40 €
U4	1	REF3033	RS 661-9455	2,93 €
U7	1	HTCC-AM01_v2	Heltec AM01(v2) 863-870MHz	8,56 €
				Total : 45,76€

Nomenclature des composants

6.3.7 Montage de la carte Holter

Une fois le routage finalisé, M. Hortolland a commandé la carte ainsi que les composants. Au retour des vacances, l'ensemble du matériel était arrivé. Mes camarades, M. Hortolland et moi-même nous sommes réunis afin de distribuer à chacun les différents composants. Une fois le circuit imprimé et les composants en ma possession, j’ai pu commencer l’assemblage et le câblage de la carte.

J’avais pour cela à ma disposition un microscope numérique, un support de carte, un stencil, de la pâte à braser, ainsi que le modèle 3D de ma carte câblée (généré grâce au logiciel KiCad, à mes différents documents et au schéma structurel).

J’ai commencé par étudier ce modèle 3D de ma carte finalisée afin de localiser précisément l'emplacement de chaque composant.

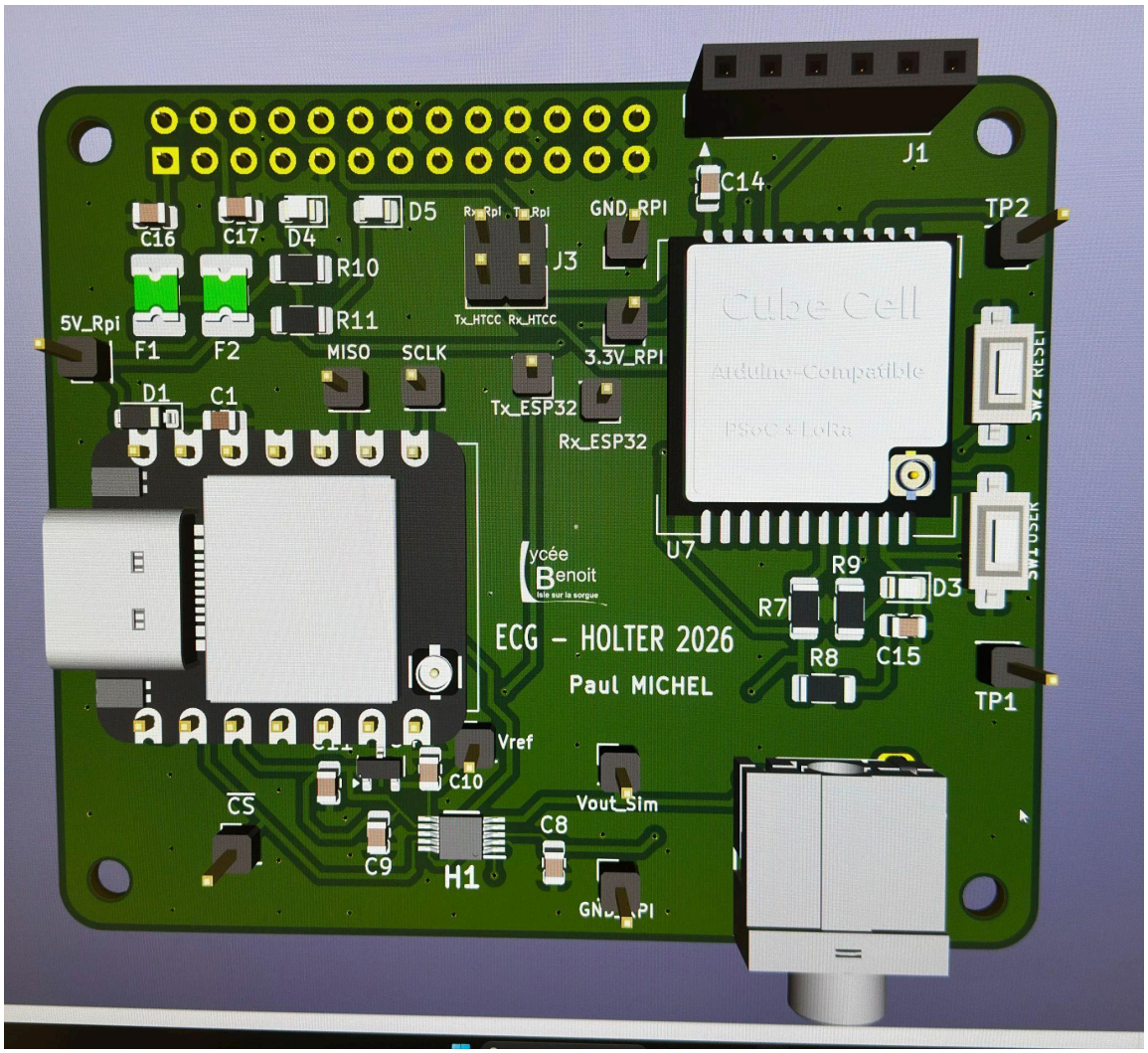


Schéma 3D de la carte

On y voit donc qu'il y a 2 types de composants :

- Des composants CMS (Composants Montés en Surface), l'utilisation de composants CMS est privilégiée pour leur petite taille, ce qui permet de réduire considérablement la taille du circuit intégré. Contrairement aux composants classiques, ils ne nécessitent pas de trous dans la carte : ils sont fixés par un processus de brasage par refusion. On dépose d'abord de la pâte à braser sur les pastilles déjà présentes aux emplacements voulus sur la carte, puis les composants sont positionnés avec précision. L'ensemble passe ensuite dans un four à refusion où il suit une courbe de température précise : après une phase de préchauffage pour stabiliser les matériaux, la carte atteint un pic de température (entre 90 et 275 degrés) pendant 2 minutes 50. Cette brève exposition suffit à faire fondre l'étain et à créer une liaison électrique parfaite sans endommager les composants sensibles.

- Des composants traversants, contrairement aux premiers, ces composants possèdent des broches métalliques qui traversent physiquement le PCB via des trous métallisés avant d'être soudées sur la face opposée. Bien qu'ils occupent plus d'espace sur la carte, leur utilisation est

indispensable pour les éléments subissant des contraintes mécaniques importantes, comme le connecteur Jack visibles sur le schéma. Cette méthode de fixation offre une robustesse bien supérieure, empêchant l'arrachement des composants lors des branchements répétés. Leur brasage se fait grâce à un poste à souder à l'étain, garantissant une liaison solide et durable au cœur de la structure du circuit.

J'ai d'abord commencé par souder les composants en CMS car il est plus simple de commencer à souder les plus petits composants au plus gros. Pour souder les composants en CMS nous avons besoin de la pâte à braser, et d'un stencil qui nous servira de guide pour déposer la pâte à braser à l' endroit voulu. Avant de positionner le stencil sur la carte, il faut nettoyer le stencil avec de l'alcool isopropylique, puis la poser sur le support de carte (avec la carte à l'intérieur). J'ai utilisé des aimants pour immobiliser le stencil.



Photo du Stencil

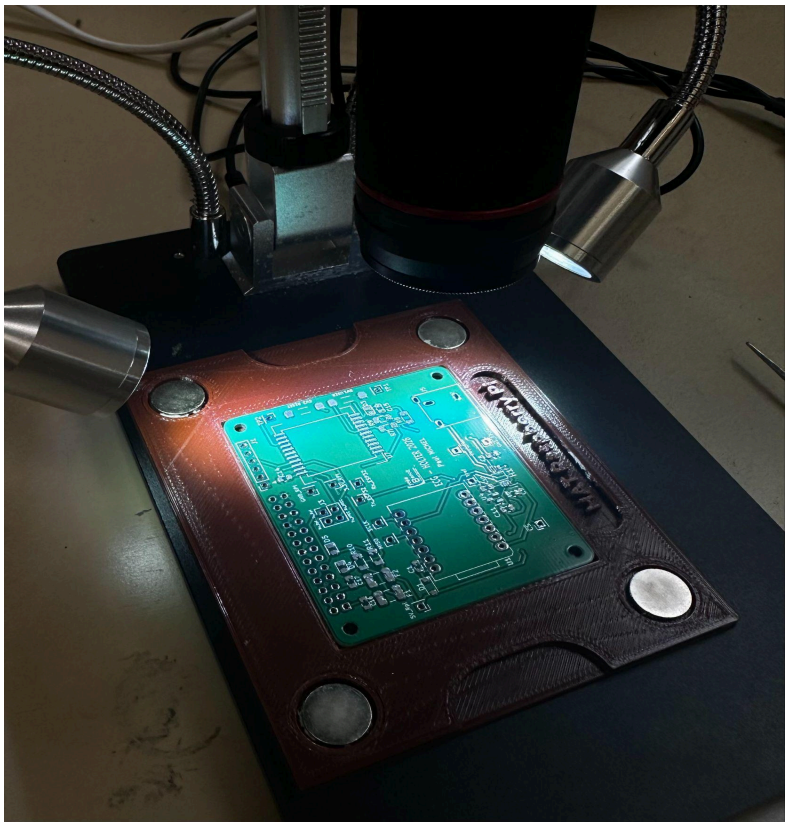
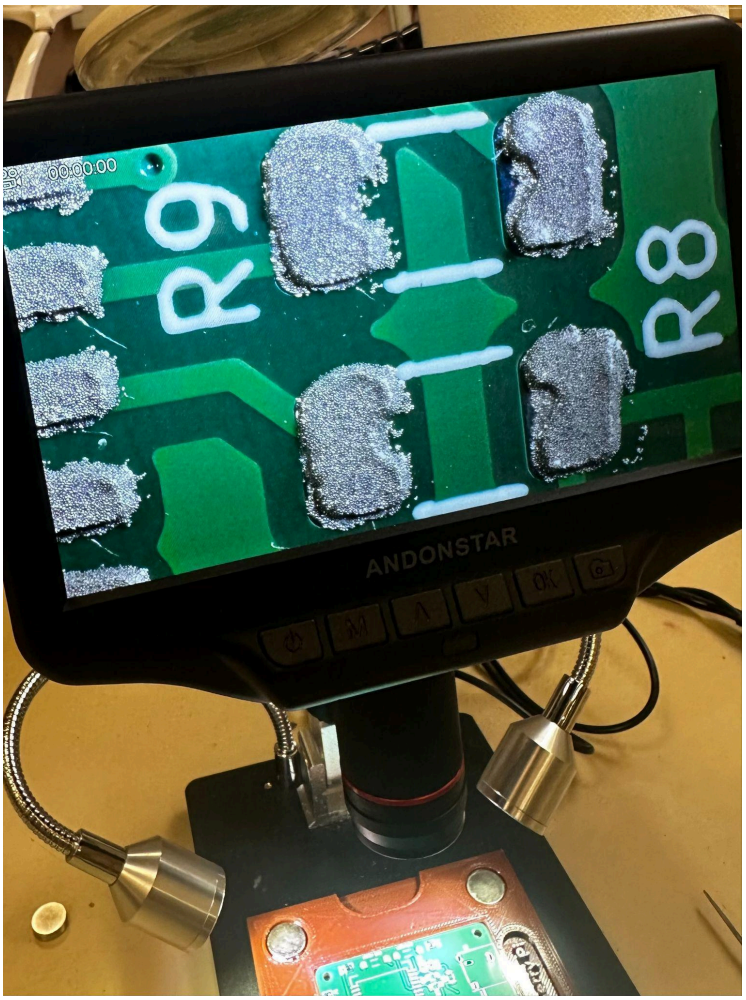


Photo de la carte avant câblage

Une fois la pâte à braser déposée, je vérifie au microscope que la pâte à braser à été déposée au bon endroit et qu'il n'y a pas de pont de soudure. Il s'agit d'un court-circuit accidentel où l'étain en fusion vient relier deux broches voisines qui devraient rester isolées.

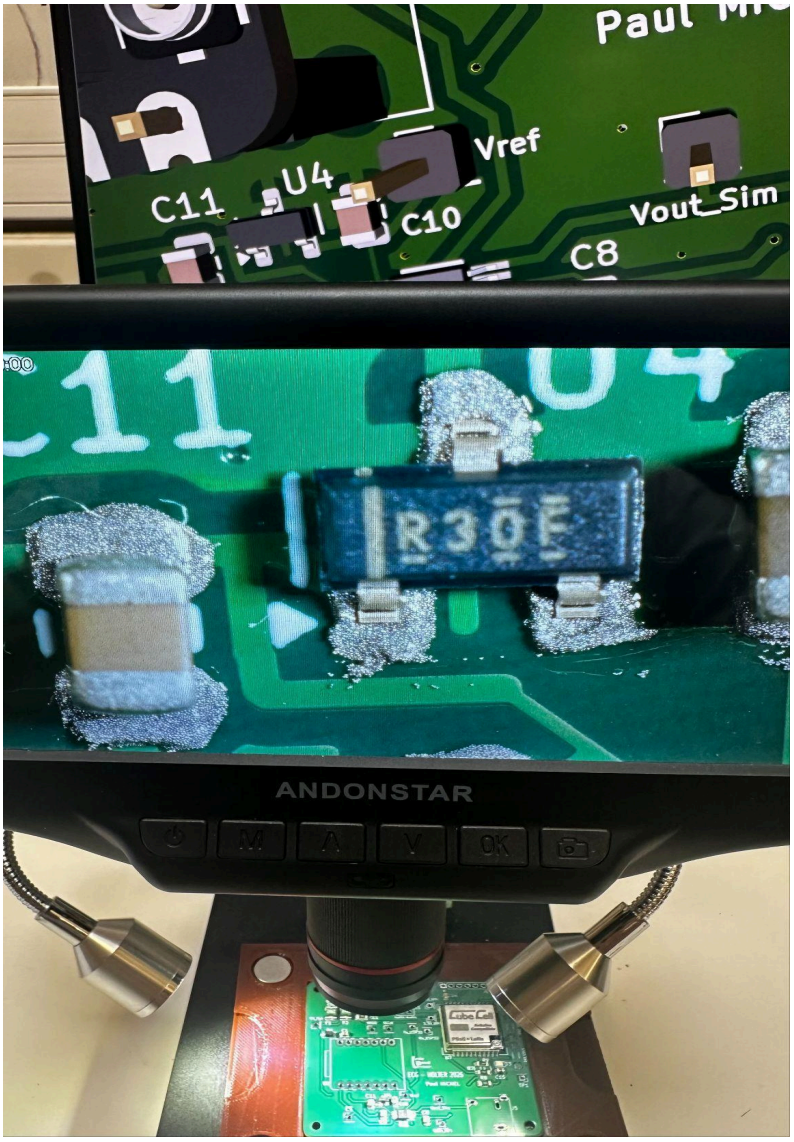
Il est crucial de contrôler le dépôt de pâte à braser avant le passage au four, car une fois la refusion terminée, les erreurs deviennent complexes à corriger. Rattraper un pont de soudure ou un composant mal aligné après refroidissement (après passage au four) nécessite une intervention manuelle délicate au fer à souder ou un pistolet à air chaud.
Cela augmente aussi le risque de fragiliser les pistes en cuivre ou d'endommager les composants voisins par excès de chaleur. Mais on peut aussi y ajouter du flux de soudure, c'est un produit chimique qui facilite la fusion de l'étain et l'empêche de s'oxyder. Pour rattraper un pont, il "force" l'étain liquide à se rétracter sur les pattes métalliques des composants par tension superficielle, brisant ainsi le court-circuit instantanément.



Contrôle du bon positionnement de la pâte à braser

Une fois la pâte à braser déposée correctement, j'ai pu procéder au positionnement des composants CMS. Ce sont des composants très petits, ce qui nécessite l'utilisation du microscope numérique pour pouvoir les observer et les manipuler avec une grande précision. C'est à ce moment que le schéma 3D de la carte finalisée m'est indispensable afin de positionner les composants aux bons endroits et dans le bon sens. En effet, certains composants tels que les diodes, le convertisseur analogique-numérique (CAN) ou le module LoRa possèdent un sens de polarité très strict (broche 1 ou détrompeur).

Par la suite, je vérifie au microscope le bon positionnement de tous les composants, ainsi que le respect de leur polarité avant de passer à l'étape de brasage.



Vérification du bon positionnement des composants avant passage au four à refusion

Une fois le positionnement correct de la pâte à braser et des composants, on met la carte au four, pendant 2 minutes 50 de 90 à 275 degrés celsius. Chaque pâte à braser possède ses propres caractéristiques chimiques, notamment selon qu'elle contient du plomb ou qu'elle soit "Lead-Free" (étain sans plomb). Il est donc indispensable d'ajuster le profil de chauffe du four pour respecter le point de fusion spécifique et le temps d'activation du flux, afin d'éviter de fragiliser les composants tout en garantissant des soudures solides.

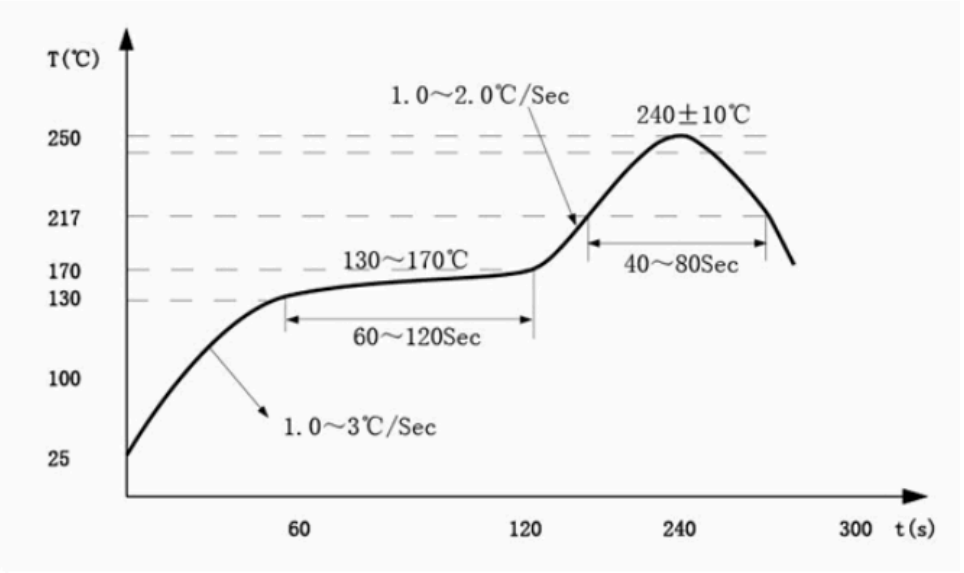
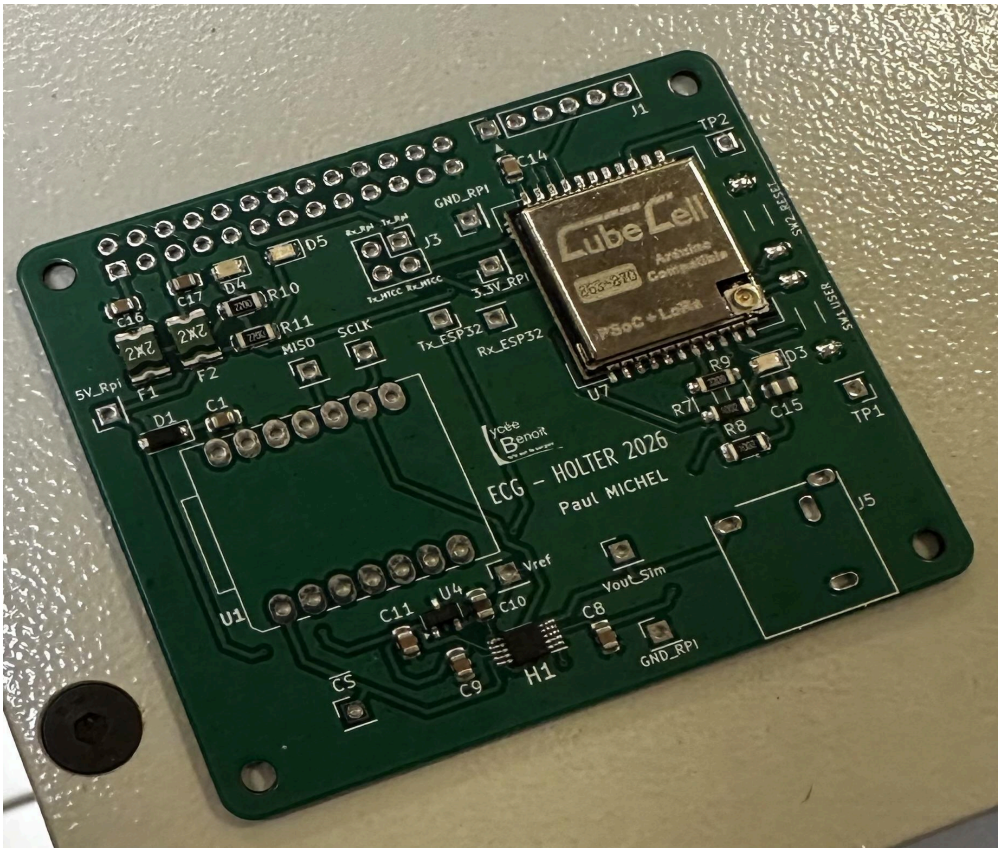


Schéma de chauffe de la pâte à braser utilisé

Une fois la carte refroidie, je l'examine à nouveau au microscope afin de vérifier l'absence de ponts de soudure.

Dans mon cas, j'ai constaté la présence d'un court-circuit (pont d'étain) sur les broches du convertisseur analogique-numérique (H1). Dans cette situation, il est nécessaire de chauffer les broches concernées à l'aide d'un pistolet à air chaud (réglé entre 300 et 400 °C), puis de séparer délicatement l'étain à l'aide d'une lame fine. Une grande vigilance est de mise durant cette opération, il faut veiller à ne pas surchauffer la zone pour ne pas décoller les pistes du circuit imprimé ou détériorer le composant.

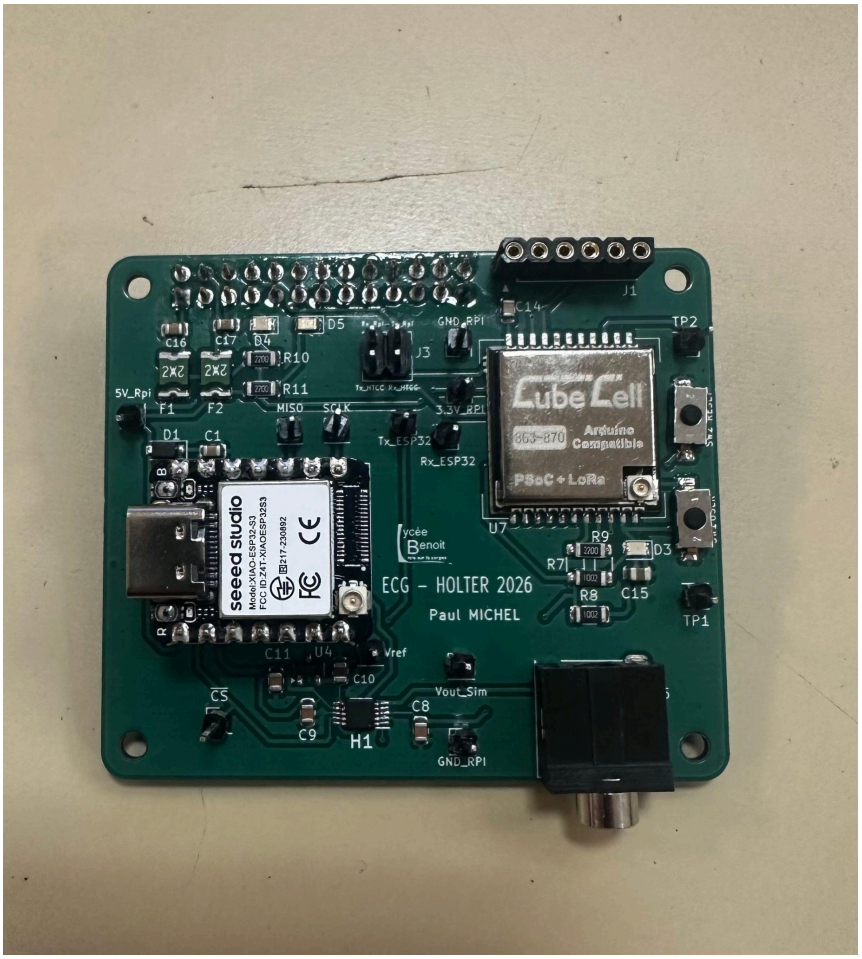


Carte Holter à la sortie du four à refusion

Une fois le câblage et l'inspection des composants CMS finalisés, j'ai pu passer à l'étape du câblage des éléments traversants. Pour cette seconde phase de fabrication, l'utilisation du stencil et du flux thermique n'étant plus adaptée, j'ai basculé sur une méthode plus traditionnelle en utilisant un fer à souder traditionnel.

J'ai ainsi procédé manuellement au montage et au brasage de plusieurs éléments clés de la carte : le module ESP32-S3, le connecteur Jack destiné à l'acquisition des signaux physiologiques, ainsi que les différents picots de test nécessaires aux futures mesures de diagnostic. Enfin, j'ai soudé les connecteurs de broches traversants (headers). Ces derniers sont particulièrement critiques, car ils assurent l'interface matérielle et la liaison électrique directe avec le port GPIO de la Raspberry Pi.

Durant toute cette manipulation, j'ai veillé à ajuster précisément la température du fer (autour de 350 °C) et à limiter le temps de contact sur les pastilles, afin d'éviter tout choc thermique qui aurait pu endommager définitivement les modules sensibles ou décoller les pistes de mon circuit imprimé.



Carte Holter 2025-2026 câblé

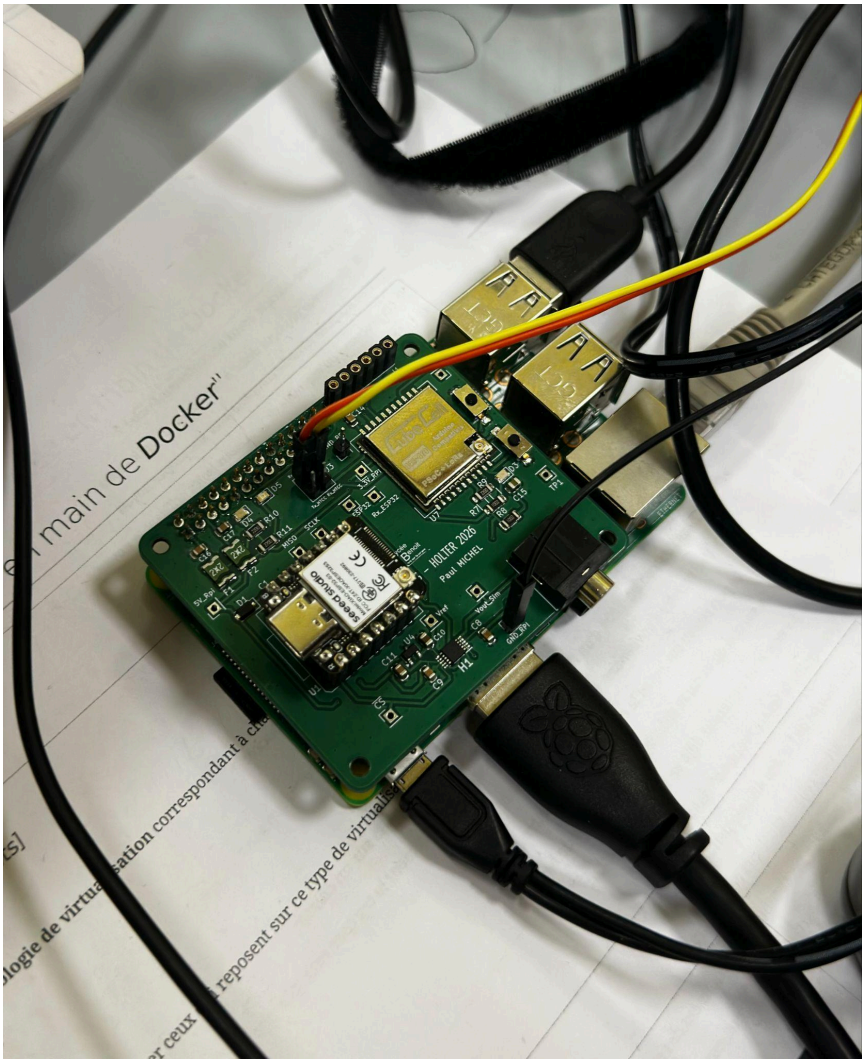
Une fois l'assemblage et le câblage de la carte terminés, une phase de test rigoureuse est indispensable pour valider l'intégrité du système. Pour ce faire, nous utilisons un Raspberry Pi comme unité de contrôle centrale. Ce choix s'explique par sa polyvalence et sa capacité à communiquer avec les différents modules de notre carte via les protocoles standards de l'industrie.

Plusieurs niveaux de tests sont réalisés pour garantir le bon fonctionnement :

- Test de continuité et d'alimentation : Avant toute mise sous tension prolongée, nous vérifions à l'aide du Raspberry Pi que les tensions de service (5V_Rpi et 3.3V_Rpi) sont stables et qu'aucun court-circuit n'est présent sur les lignes d'alimentation.

- Communication Bus (I2C/SPI) : Nous effectuons un scan des bus de communication pour vérifier que le module CubeCell et l'ESP32 répondent correctement aux requêtes. Cela confirme que le brasage des composants CMS est parfait et qu'aucun "pont" ne bloque les signaux.

- Vérification de la chaîne d'acquisition : En injectant un signal de test (simulant un rythme cardiaque) via l'entrée Jack, le Raspberry Pi récupère les données traitées pour s'assurer que la conversion analogique-numérique se fait sans bruit parasite.



Test de la carte holter 2025-2026