

BTS CIEL

Épreuve E6.2 – Session 2025

Dossier de revue n°1

Projet ECG

L'électrocardiogramme



- Mathias BOUFFLET (IR1)
- Johann AMADORI (IR3)
- Thomas PELAT (IR2)

- Nino CILICHINI (ER1)
- Zainou ZAKOUANA (ER2)

Table des matières

Table des matières.....	2
Partie commune.....	3
1. Présentation Générale du Projet.....	4
1.1 Description du projet.....	4
1.2 Contexte.....	4
1.3 Expression du besoin.....	5
1.4 Présentation du client.....	5
2. Synoptique d'architecture matérielle.....	7
2.1 Diagrammes des cas d'utilisation	9
3. Spécification de la base de données.....	10
3.1 Comprendre un ECG.....	11
Partie Personnelle Mathias BOUFFLET (IR 1).....	12-41
Partie Personnelle Thomas PELAT (IR 2).....	42-60
Partie Personnelle Johann AMADORI (IR 3).....	61-93
ANNEXES SIMULATEUR.....	94-116
Partie Personnelle Zainou ZAKOUANA (ER 2).....	117-143
Partie Personnelle Nino CILICHINI (ER 1).....	144-166
ANNEXES.....	167-175

Partie commune

1. Présentation Générale du Projet

Le projet **ECG – Électrocardiogramme** s'inscrit dans le cadre du BTS CIEL (Cybersécurité, Informatique et Électronique) du Lycée Alphonse Benoît à L'Isle-sur-la-Sorgue. Il a pour objectif de concevoir un **holter connecté**, capable d'enregistrer et de surveiller en continu l'activité cardiaque d'un patient, tout en intégrant des fonctionnalités avancées telles que l'alerte en cas de dépassement de seuils critiques. Ce projet est mené en collaboration avec le **CERI (Centre d'Enseignement et de Recherche en Informatique) d'Avignon**, qui souhaite exploiter cette solution pour des applications pédagogiques et médicales.

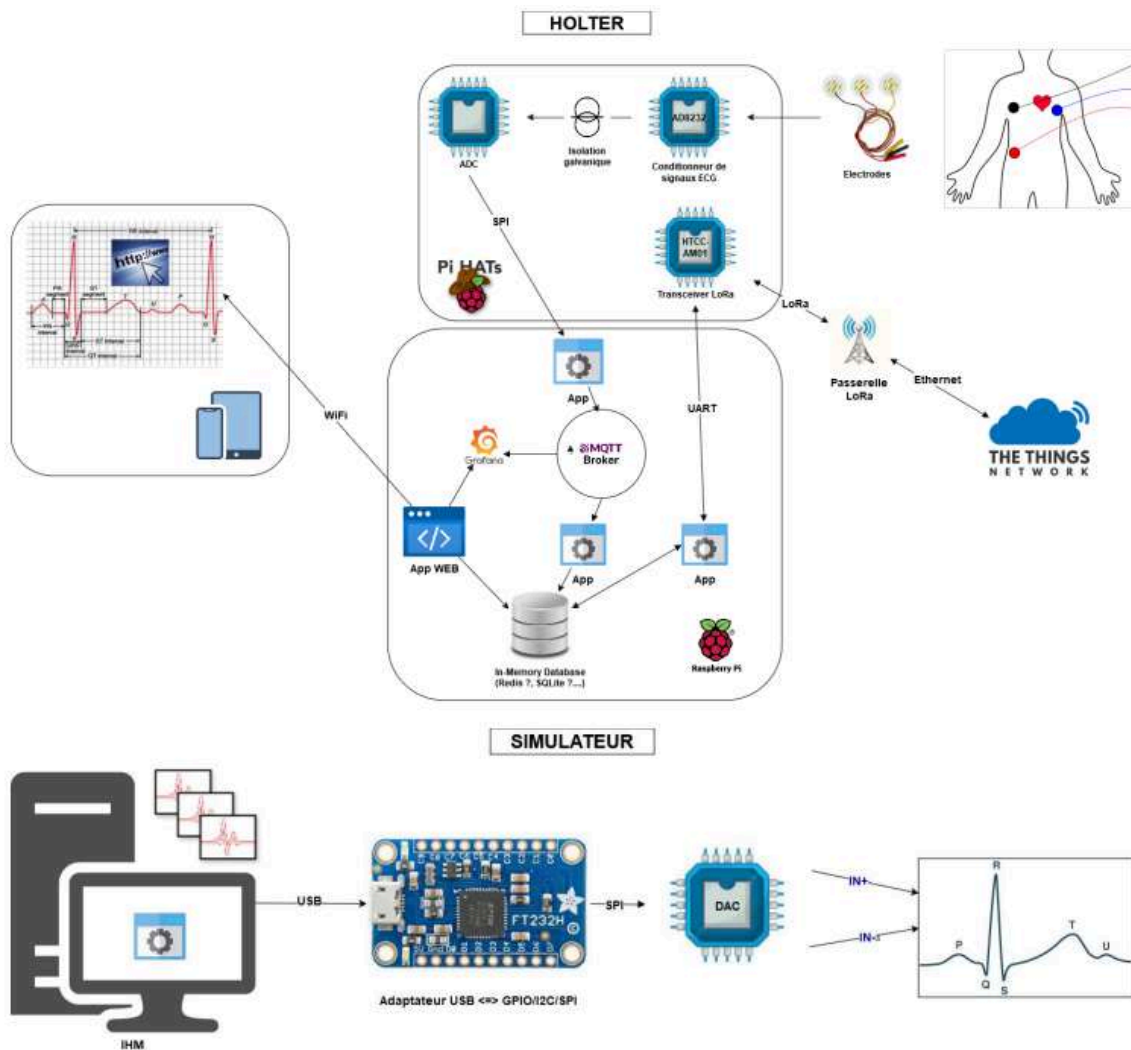
1.1 Description du Projet

L'électrocardiogramme est un outil fondamental en cardiologie, permettant de suivre l'activité électrique du cœur et de détecter d'éventuelles anomalies. Cependant, les solutions existantes sont souvent coûteuses et nécessitent un suivi en milieu médical.

Notre projet vise à développer un **prototype de holter ECG connecté**, permettant une surveillance continue et accessible des signaux cardiaques. Deux éléments clés composent cette solution :

1. **Un holter ECG connecté**, capable d'enregistrer les signaux cardiaques, de les transmettre via **LoRa** et de les stocker localement sur une **base de données MQTT**.
2. **Un simulateur de signaux ECG**, permettant d'imiter l'activité cardiaque pour tester et entraîner le système sans avoir besoin d'un patient réel.

L'ensemble repose sur une architecture intégrant un **Raspberry Pi**, des **capteurs ECG**, ainsi qu'une plateforme de supervision via un **dashboard web sécurisé** permettant la visualisation et l'analyse des données en temps réel.



1.2 Contexte

Aujourd'hui, les solutions d'ECG connectées existent, mais elles sont souvent limitées en accessibilité et en analyse en temps réel. De plus, elles nécessitent généralement un environnement médical ou une expertise avancée pour être exploitées efficacement.

Ce projet s'inscrit donc dans un contexte d'innovation **IoT et télémédecine**, avec un fort enjeu en matière de **cybersécurité**, de traitement des données et de transmission sécurisée via des protocoles comme MQTT et LoRa. Le développement est réalisé dans un cadre pédagogique, avec une équipe de **5 étudiants (Boufflet Mathias, Pelat Thomas, Amadori Johann, Cilichini Nino et Zenou)**, spécialisés en **informatique et électronique**, sous la supervision de **C.DEFRANCE, F.ESCURET et C.HORTOLLAND**.

Le projet répond aux attentes du **CERI** et pourra, à terme, être utilisé dans des formations en ingénierie biomédicale ou pour des études en **intelligence artificielle appliquée à la santé**.

1.3 Expression du besoin

Dans le cadre de l'unité d'enseignement portant sur l'Internet des Objets (IoT), le CERI souhaite améliorer l'architecture d'un système de surveillance cardiaque basé sur un électrocardiogramme (ECG). Actuellement, il existe des solutions permettant l'enregistrement des signaux ECG, mais elles présentent certaines limites, notamment en matière d'analyse en temps réel et d'accessibilité des données.

Le besoin exprimé par le CERI se décline en plusieurs objectifs :

- **Améliorer l'intégration du système** : L'objectif est de proposer une solution clé en main, facilement exploitable et réutilisable dans un contexte pédagogique et clinique.
- **Apporter plus de flexibilité** : La mise en place d'un simulateur de signaux cardiaques permettra d'éviter la nécessité de disposer d'un patient pour tester et analyser les relevés ECG.
- **Faciliter l'exploitation des données** : Le système devra être compatible avec des modules d'intelligence artificielle (IA) afin d'automatiser la détection d'anomalies comme les arythmies cardiaques.
- **Favoriser la connectivité et la supervision à distance** : L'utilisation du protocole LoRa et d'une infrastructure cloud permettra d'assurer la transmission des données ECG en temps réel, tout en offrant un accès sécurisé aux informations enregistrées.

L'objectif final du projet est donc de concevoir un **holter connecté**, capable de détecter et signaler des anomalies du rythme cardiaque en temps réel, tout en proposant un environnement d'analyse et de stockage des données optimisé. Ce système devra être à la fois fiable, économique et évolutif afin d'être exploité pour la formation, la recherche et les applications médicales.

1.4 Présentation du client

Le projet est développé en partenariat avec le **CERI** (*Centre d'Enseignement et de Recherche en Informatique*) de l'**Université d'Avignon**, un centre spécialisé dans les technologies de l'information et de la communication.

Le **CERI** est un acteur majeur dans la formation et la recherche en informatique, avec un fort accent sur l'innovation et les nouvelles technologies. Il propose des formations allant du **BUT Informatique** jusqu'au **Master**, notamment dans le domaine des **Systèmes Informatiques Communicants (SICOM)**, qui se concentre sur les réseaux, l'IoT, l'intelligence artificielle et la cybersécurité.

Dans le cadre de ses activités pédagogiques et de recherche, le CERI développe des projets technologiques en collaboration avec des étudiants et des entreprises. L'objectif est d'offrir aux étudiants une approche **concrète et appliquée** de l'ingénierie informatique, tout en répondant à des **besoins industriels et académiques réels**.

Le CERI a exprimé le besoin de développer un **holter connecté** afin d'améliorer l'enseignement et la recherche sur les dispositifs médicaux intelligents. Ce projet s'inscrit dans une démarche de **transfert technologique**, visant à concevoir un prototype fonctionnel

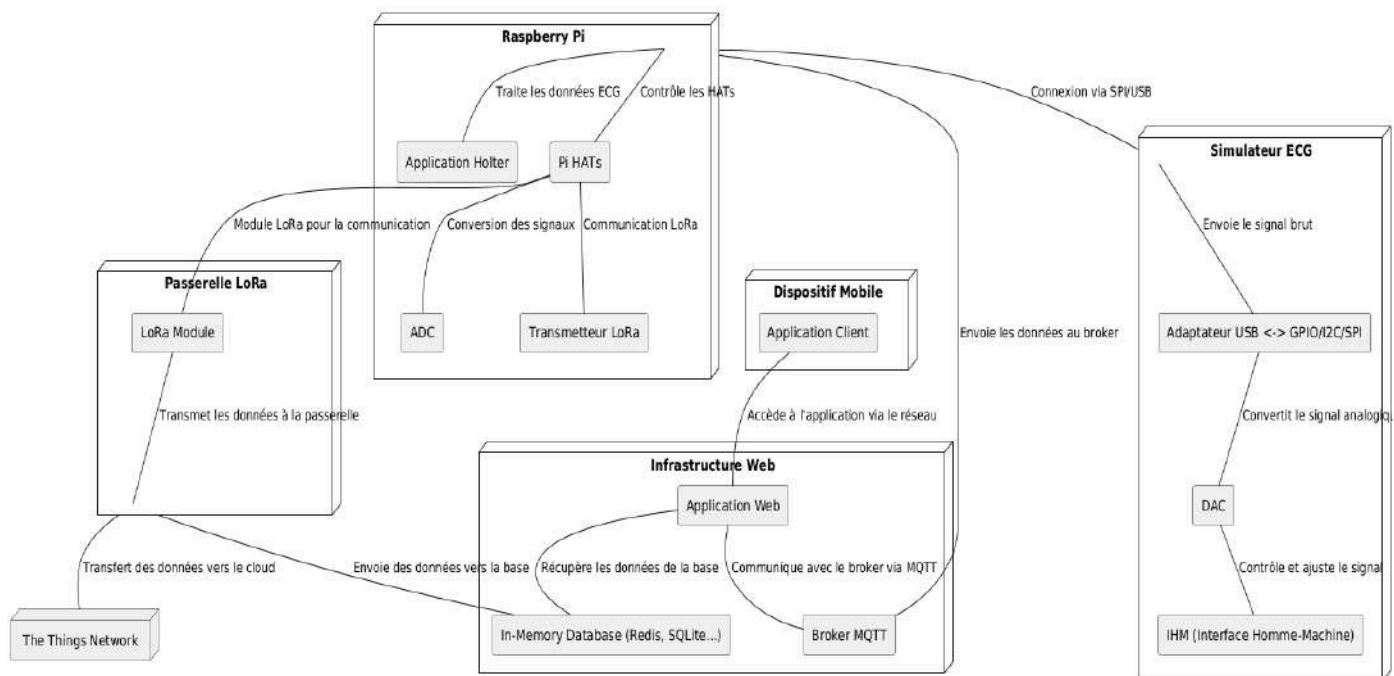
qui pourra être utilisé dans d'autres unités d'enseignement, notamment celles liées à **l'intelligence artificielle et à l'IoT**.

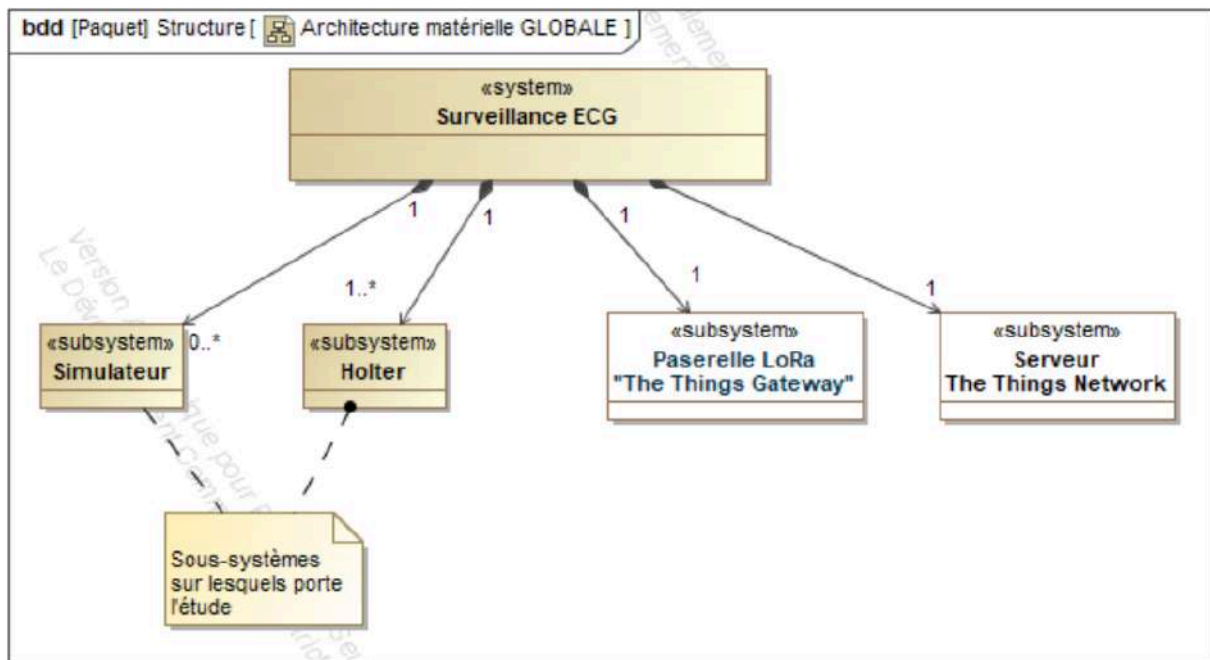
Les principaux contacts pour ce projet au sein du CERI sont :

- **Marc SILANUS** (Réfèrent technique) – marc.silanus@univ-avignon.fr
- **Philippe GOZLAN** (Responsable pédagogique) – philippe.gozlan@univ-avignon.fr

L'implication du CERI garantit un **encadrement technique et pédagogique** tout au long du développement du projet, avec un accès à des ressources matérielles et logicielles adaptées.

2. synoptique d'architecture matérielle



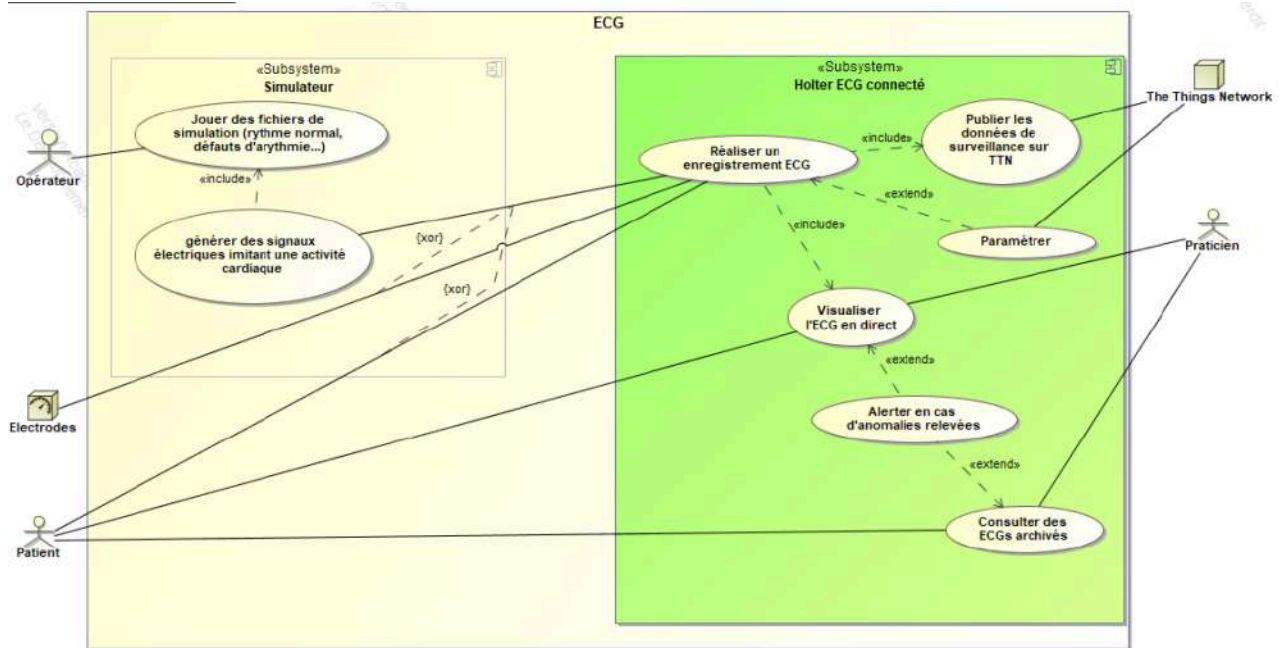


-**Simulateur ECG** : Le simulateur ECG utilise un système de sélection de fichier CSV depuis l'application IHM. Les données numériques sont envoyées au convertisseur DAC8551, qui contrôle et ajuste le signal. Ensuite, le signal est transmis via SPI pour se connecter à la Raspberry Pi (le Holter), où il est traité.

-**Le Holter** récupère le signal du simulateur et l'envoie à un convertisseur MAX111000. Une application convertit ensuite le signal analogique en signal numérique. Cette valeur est ensuite convertie en format JSON et envoyée au broker MQTT via un abonnement à un topic. Le broker est relié à Grafana pour afficher les informations dans une iframe intégrée à une application web. Par ailleurs, une autre application récupère les informations du broker et les stocke dans une base de données

-**La passerelle LoRa** est utilisée pour envoyer les données sur le cloud The Things Network. Pour cela, une application récupère les données depuis la base de données et les transmet via le protocole UART à un transceiver LoRa. Celui-ci utilise la technologie LoRa pour envoyer les données à la passerelle, qui les transfère ensuite vers le cloud.

2.1 Diagrammes des cas d'utilisation

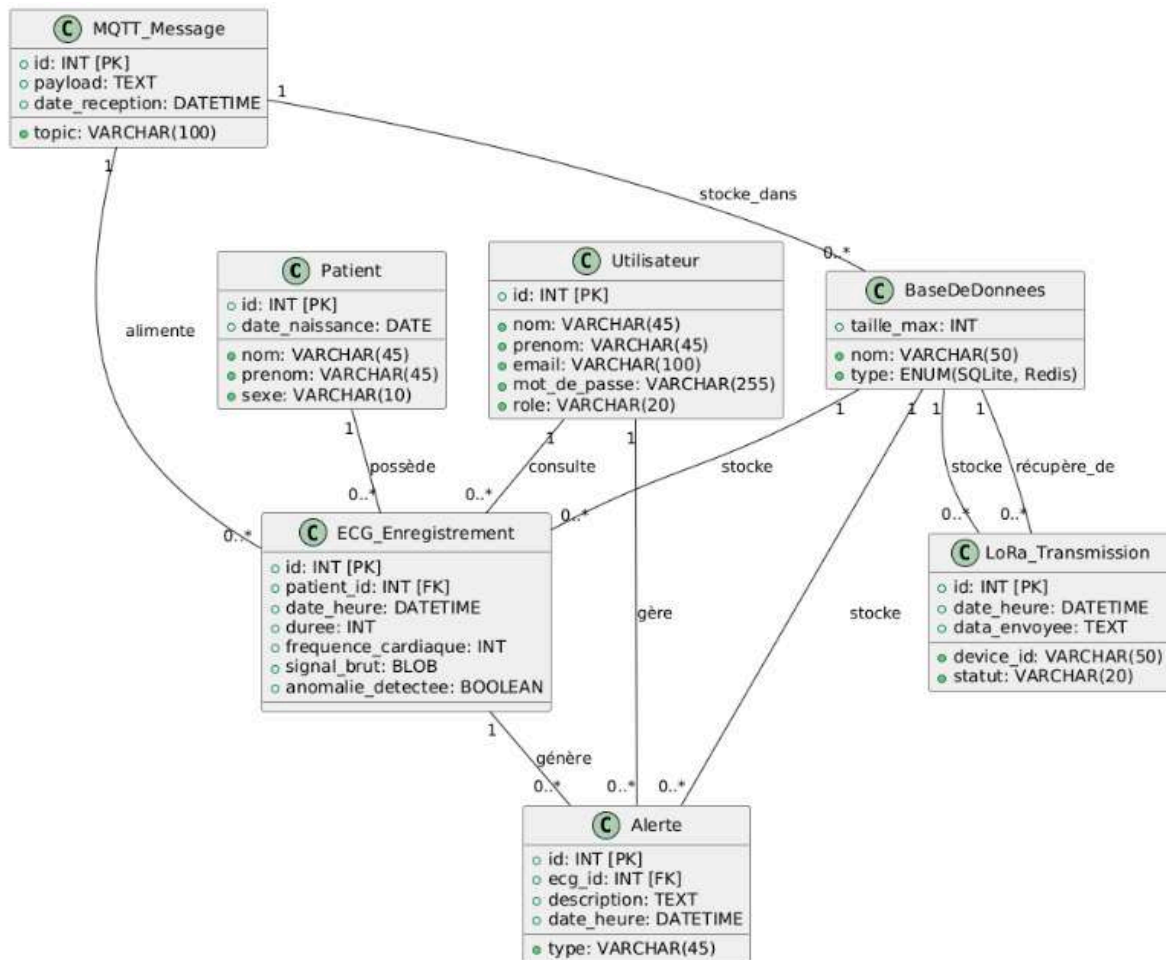


Afin de répondre à l'ensemble des besoins évoqués précédemment, le projet sera divisé en deux parties : **le Holter ECG connecté et le simulateur**.

- **Le simulateur** disposera de sa propre interface IHM permettant de contrôler les pathologies cardiaques simulées. Son pilotage sera entièrement manuel, ce qui signifie que les pathologies seront déclenchées par l'utilisateur et non de manière automatisée, cette automatisation étant jugée inutile pour cet usage.
- **Le Holter ECG connecté** fonctionnera via une application **headless** (exécutée en arrière-plan, sans interface graphique). Son rôle sera d'enregistrer les signaux ECG correspondant aux pathologies transmises par le simulateur. En cas de détection d'anomalie, une alerte sera générée et archivée dans la base de données (BDD). Par ailleurs, le signal capté sera publié sur le **cloud TTN (The Things Network)** pour permettre une surveillance à distance.

Ce système garantit à la fois un contrôle précis des simulations et une surveillance efficace des données ECG en temps réel.

3. Spécification de la base de données



Nous aurons besoin d'une base de données pour stocker et gérer les informations des patients ainsi que leurs mesures. Le diagramme ci-dessus montre comment les données seront générées et transmises. Nous y stockerons l'ID du patient, la date et l'heure de la mesure, la fréquence cardiaque, le signal brut, ainsi que toute anomalie éventuelle détectée.

3.1 Comprendre un ECG

Comprendre l'objectif d'un ECG de base

L'électrocardiogramme (ECG) est un examen médical permettant d'enregistrer l'activité électrique du cœur. Son objectif principal est de détecter d'éventuelles anomalies cardiaques, telles que des arythmies, des infarctus du myocarde ou d'autres pathologies affectant la conduction électrique du cœur. L'ECG est souvent utilisé en consultation médicale pour surveiller la santé cardiaque des patients, notamment ceux présentant des antécédents cardiovasculaires. Grâce à l'enregistrement des signaux électriques du cœur, cet outil permet aux médecins d'obtenir un diagnostic rapide et précis, facilitant ainsi une prise en charge adaptée.

L'électrocardiographie

L'électrocardiographie est la technique qui permet d'enregistrer et d'analyser l'activité électrique du cœur. Cet enregistrement est réalisé à l'aide d'électrodes placées sur la peau du patient, généralement au niveau des bras, des jambes et du thorax. L'ECG enregistre les variations de potentiel électrique générées par les contractions cardiaques et les traduit sous forme de tracé graphique. Il existe plusieurs types d'ECG : l'ECG de repos, l'ECG d'effort et l'ECG ambulateur (Holter). Cette technique est largement utilisée en cardiologie pour évaluer le fonctionnement du cœur et identifier d'éventuelles anomalies.

Comment lire un ECG

La lecture d'un ECG repose sur l'interprétation du tracé obtenu, qui se divise en plusieurs ondes représentant les différentes phases du cycle cardiaque :

- **Onde P** : correspond à la dépolarisation des oreillettes.
- **Complexe QRS** : reflète la dépolarisation des ventricules et précède leur contraction.
- **Onde T** : représente la repolarisation des ventricules.

L'analyse d'un ECG repose sur plusieurs paramètres : la fréquence cardiaque, la régularité du rythme, l'amplitude et la durée des ondes. Un tracé normal présente une fréquence cardiaque comprise entre 60 et 100 battements par minute avec un rythme régulier. Toute anomalie dans la forme ou la durée des ondes peut indiquer une pathologie cardiaque nécessitant un examen approfondi.

Partie Personnelle
Boufflet Mathias
(IR 1)

1. Étudiant n°1 – Acquisition, Visualisation et Intégration des Signaux ECG

1.1 Présentation du système d'acquisition ECG

Le système mis en place simule, traite et transmet des signaux cardiaques pour une analyse en temps réel, en s'appuyant sur une architecture électronique et logicielle adaptée aux environnements IoT.

Composants principaux :

- **Simulateur ECG** : lit un fichier CSV pour reproduire différents rythmes cardiaques (dont des pathologies).
- **DAC8551** : convertit les données numériques du fichier CSV en signaux analogiques.
- **Conditionneur AD8232** : filtre et amplifie les signaux pour une lecture plus précise.
- **Isolation galvanique** : assure la sécurité électrique du circuit.
- **MAX111000** : convertit les signaux analogiques en données numériques exploitables.
- **Broker MQTT local** : centralise et diffuse les données ECG en temps réel.
- **Interface Grafana** : affiche les courbes ECG en temps réel via un iframe.
- **Base de données SQLite** : stocke les signaux ECG pour analyse et archivage.
- **Transmetteur LoRa HTCC AM01** : envoie les données vers une passerelle LoRa, puis vers le cloud The Things Network.

Fonctionnement global :

L'application client sélectionne un fichier CSV, dont les données sont converties en signal analogique par le DAC8551, puis traitées par le conditionneur AD8232. Ce signal est ensuite numérisé par le MAX111000, transformé en JSON, et transmis via MQTT. Les données sont stockées dans une base SQLite et affichées dans Grafana. Enfin, elles sont également transmises via LoRa vers un serveur distant pour un suivi à distance.



1. Introduction au protocole MQTT

MQTT (Message Queuing Telemetry Transport) est un protocole léger, idéal pour les systèmes embarqués et les environnements IoT. Il utilise un modèle **publish/subscribe** via un **broker centralisé**, ce qui permet une communication asynchrone, peu gourmande en bande passante et en énergie. Cela le rend particulièrement adapté à notre projet ECG.

2. Fonctionnement du protocole MQTT

MQTT repose sur trois éléments clés :

- **Broker** : serveur central qui gère, filtre et distribue les messages.
- **Publishers** : envoient des messages sur un topic.
- **Subscribers** : reçoivent les messages des topics auxquels ils sont abonnés.

Le protocole propose trois niveaux de **Qualité de Service (QoS)** :

- **QoS 0** : envoi sans garantie.
- **QoS 1** : envoi au moins une fois.
- **QoS 2** : envoi exactement une seule fois.

3. Intégration de MQTT dans le projet ECG

Dans notre système, MQTT permet de transmettre les données ECG en temps réel.

3.1 Installation du broker

Nous avons choisi **Mosquitto**, un broker léger et compatible avec le Raspberry Pi. Son installation se fait simplement via le gestionnaire de paquets.

```
sudo apt update
sudo apt install -y mosquitto mosquitto-clients
```

sudo apt update → Mise à jour des paquets disponibles et Vérifié les dernières versions des logiciels disponibles sur les dépôts. Permet d'éviter d'installer une version obsolète de Mosquitto.

sudo apt install -y mosquitto mosquitto-clients → **mosquitto** : Installe le broker MQTT, qui sert d'intermédiaire entre les appareils qui publient et ceux qui s'abonnent aux messages.

mosquitto-clients : Installe des outils en ligne de commande permettant de tester facilement MQTT (**mosquitto_pub** et **mosquitto_sub**).

-y : Accepte automatiquement l'installation sans demander de confirmation.

Ensuite, on active et démarre le service Mosquitto :

```
sudo systemctl enable mosquitto  
sudo systemctl start mosquitto
```

sudo systemctl enable mosquitto → **systemctl enable** : Configure Mosquitto pour qu'il **se lance automatiquement** à chaque démarrage du système. Sans cette commande, Mosquitto ne démarrera pas automatiquement après un redémarrage.

```
sudo systemctl restart mosquitto
```

il est préférable de redémarrer le service pour s'assurer qu'il prend bien en compte toutes les configurations

Pour configurer Mosquitto afin qu'il utilise un port spécifique, désactive l'accès anonyme et ajoute une authentification par mot de passe, voici la commande :

```
sudo nano /etc/mosquitto/conf.d/custom.conf
```

Cette commande ouvre l'éditeur nano pour créer ou modifier le fichier de configuration /etc/mosquitto/conf.d/custom.conf.

Dans nano on vas pouvoir mettre ces lignes

```
listener 1883  
allow_anonymous false  
password_file /etc/mosquitto/passwd
```

1.2.1 Configurer les topics dans Mosquitto

Quand je parle de configurer les topics, cela signifie qu'il faut s'abonner à un topic. Dans notre projet, nous allons nous **abonner** au topic "lab/ecg".

Voici la commande à exécuter dans le terminal :

```
mosquitto_sub -t "lab/ecg"
```

La commande `mosquitto_sub -t "lab/ecg"` permet de s'abonner au topic MQTT "lab/ecg".

- **mosquitto_sub** : Client Mosquitto utilisé pour souscrire à un topic MQTT.
- **-t "lab/ecg"** : Spécifie le topic auquel on souhaite s'abonner.

Lorsqu'un message est publié sur ce topic, il s'affiche immédiatement dans la console. Cette commande est utile pour surveiller en temps réel les données transmises via MQTT.

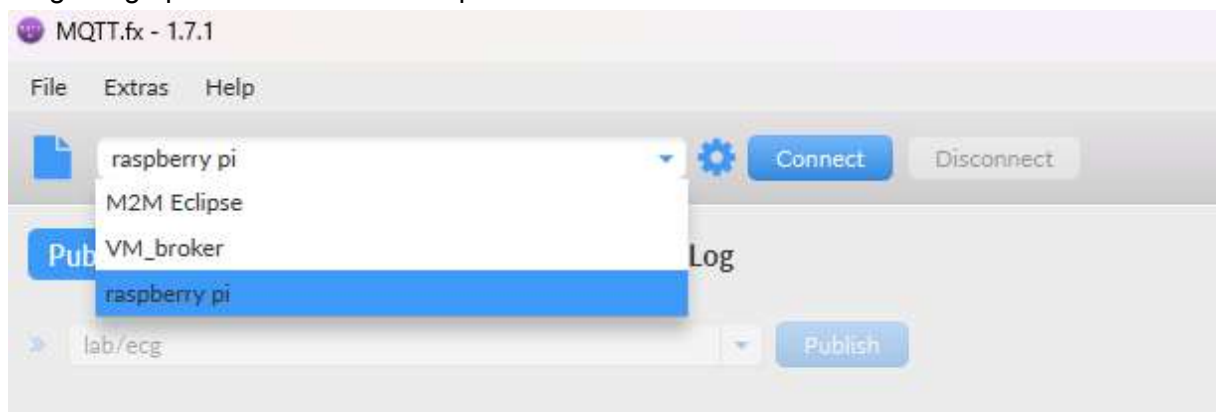
Maintenant on va voir comment écouter le topics avec la commande subscribe

```
mosquitto_sub -t "lab/ecg"
```

Voici la commande à exécuter dans le terminal `mosquitto_sub -t "lab/ecg"`

Pour tout simplement voir si la messagerie est bien activé nous pouvons envoyer des message via l'application MQTT.fx (Version 1.7.1 sur l'ordi actuelle)

Une fois MQTT.fx démarré nous arrivons sur une interface blanche et il suffit de cliqué sur l'engrenage pour crée un nouveau profils



Une nouvelle page vas se lancé et vas falloir choisir le profil name ici :

→Profile name : raspberry pi

→Profile type : MQTT Broker

→Adresse IP du Broker : 192.168.4.86

→Le port du broker 1883

→Le reste on touche pas on laisse les réglages par défaut

Puis on clique sur apply pour appliquer les réglages mis !

The screenshot shows the 'MQTT Broker Profile Settings' window. At the top, 'Profile Name' is 'raspberry pi' and 'Profile Type' is 'MQTT Broker'. Below this is a blue button labeled 'MQTT Broker Profile Settings'. The 'Broker Address' is '192.168.4.86' and 'Broker Port' is '1883'. 'Client ID' is 'MQTT_FX_Client' with a 'Generate' button next to it. Below these fields are tabs for 'General', 'User Credentials', 'SSL/TLS', 'Proxy', and 'LWT'. The 'General' tab is active, showing 'Connection Timeout' (30), 'Keep Alive Interval' (60), 'Clean Session' (checked), 'Auto Reconnect' (unchecked), 'Max Inflight' (10), and 'MQTT Version' (Use Default, 3.1.1). At the bottom are buttons for 'Clear Publish History' and 'Clear Subscription History'.

Une fois la fenêtre fermée, cliquez sur le bouton **Connect** pour vérifier la connexion au broker. Pour ce test, un message au format JSON est envoyé en cliquant sur **Publish**.

The screenshot shows the MQTT interface with tabs for 'Publish', 'Subscribe', 'Scripts', 'Broker Status', and 'Log'. The 'Publish' tab is active. A text input field contains 'lab/ecg' and a 'Publish' button is next to it. Below the input field, a JSON message is displayed in a code editor:

```
{
  "timestamp": "2025-03-28T12:00:00Z",
  "ecg_value": 0.45,
  "bpm": 72
}
```

Côté console, on observe que le message a bien été reçu sur le topic `lab/ecg`. Lorsqu'un client est abonné à un topic MQTT, il reçoit automatiquement les messages publiés en temps réel, ce qui permet une transmission continue et efficace des données, comme pour un signal ECG.

```
pi@rpi-cl2-04:~ $ mosquitto_sub -t "lab/ecg"
{
  "timestamp": "2025-03-28T12:00:00Z",
  "ecg_value": 0.45,
  "bpm": 72
}
```

Pourquoi avoir choisi MQTT pour le projet ECG ?

MQTT (Message Queuing Telemetry Transport) est un protocole léger, souvent utilisé dans les systèmes embarqués et les projets IoT. Dans notre projet ECG, il permet de transmettre les données sous format **JSON**, ce qui facilite leur traitement et leur affichage dans un outil comme **Grafana**.

Avantages de MQTT :

- **Léger et optimisé** : adapté aux appareils à faibles ressources comme le Raspberry Pi, il consomme peu de bande passante.
- **Faible latence** : grâce au modèle publish/subscribe, les données sont transmises quasi instantanément.
- **Fiabilité** : prise en charge de trois niveaux de QoS pour garantir la bonne réception des messages.
- **Architecture évolutive** : permet d'ajouter facilement de nouveaux capteurs ou services.
- **Compatibilité avec Grafana** : pour une visualisation en temps réel des courbes ECG.

Inconvénients :

- **Sécurité limitée par défaut** : nécessite la mise en place de TLS/SSL et d'une authentification.
- **Dépendance au broker** : en cas de panne du broker Mosquitto, la communication est interrompue.
- **Gestion limitée des gros volumes** : MQTT est optimisé pour les messages courts et fréquents.

- **Configuration initiale** : nécessite une bonne maîtrise pour optimiser le QoS et la transmission.
-

Fonctionnement global de l'application ECG

1. Acquisition du signal

- Un capteur ECG mesure l'activité électrique du cœur.
- Le signal analogique est converti en données numériques via le convertisseur MAX11100.

2. Traitement des données

- Les données sont reçues en hexadécimal via SPI.
- Conversion en millivolts puis calcul des BPM en détectant les pics R.

3. Transmission via MQTT

- Les valeurs ECG et BPM sont publiées en JSON sur le topic [lab/ecg](#).

4. Affichage et exploitation

- Les données sont affichées en temps réel sur la Raspberry Pi.
 - Elles peuvent être visualisées dans Grafana ou enregistrées pour une analyse ultérieure.
-

Détails techniques

- **MAX11100** : convertit une tension de 0 à 3,3V en une valeur sur 16 bits, transmise en hexadécimal.
 - **Conversion** : les données sont traduites en millivolts selon la tension de référence.
 - **Détection des BPM** : un algorithme identifie les pics R et calcule l'intervalle pour déduire la fréquence cardiaque.
-

Conclusion

MQTT est un choix pertinent pour notre projet ECG grâce à sa légèreté, sa rapidité et sa compatibilité avec des outils comme Grafana. Il permet une acquisition fiable et en temps réel des données cardiaques. Toutefois, des mesures de sécurité et de robustesse doivent être mises en place pour garantir un fonctionnement stable et sécurisé..

Présentation du code du projet ECG (28/03/2025)

Le code actuelle se presente sous la forme de plusieurs fichier

Headers :

- appworker.h
- serialsender.h
- spidev.h

Sources:

- appworker.cpp
- main.cpp
- serialsender.cpp
- spidev.cpp

appworker.h

```
1  #ifndef APPWORKER_H
2  #define APPWORKER_H
3
4  #include <QTimer>
5  #include <QObject>
6
7  #include "serialsender.h"
8  #include "spidev.h"
9
10 class AppWorker : public QObject
11 {
12     Q_OBJECT
13 public:
14     explicit AppWorker(QObject *parent = nullptr);
15
16 private :
17     SerialSender * _monitor;
18     SpiDev * _max11100;
19     QTimer * _timer;
20
21     bool _quit = false;
22
23 public slots:
24     void demarrer();
25     void quitter();
26     void onTimerTimeOut();
27
28 signals:
29     void termine();
30 };
31
32 #endif // APPWORKER_H
33 |
```

Lit les valeurs SPI du MAX11100.

Convertit les valeurs en millivolts.

Envoie les données sur le port série.

Il gère la communication avec l'ADC MAX11100 via SPI.

Il récupère les données ECG à intervalles réguliers grâce à un **QTimer**.

Il envoie ces données sur un port série pour affichage ou transmission.

Il dispose de mécanismes pour quitter proprement l'application.

serialsender.h

```
1  #ifndef SERIALSENDER_H
2  #define SERIALSENDER_H
3
4  #include <QObject>
5  #include <QSerialPort>
6
7  class SerialSender : public QObject
8  {
9      Q_OBJECT
10 public:
11     explicit SerialSender(QString serialPortName="/dev/serial0", QObject *parent = nullptr);
12     void send(QString data);
13
14 private:
15     QSerialPort * _serial;
16 };
17
18 #endif // SERIALSENDER_H
19
```

En résumé

La classe **SpiDev** est une abstraction pour faciliter l'interaction avec un périphérique SPI dans un système embarqué ou sur un ordinateur utilisant un périphérique SPI (comme un Raspberry Pi). Elle permet d'ouvrir le périphérique SPI, de configurer des paramètres comme la vitesse et le mode de communication, puis d'effectuer des lectures et écritures via SPI.

Cette classe pourrait être utilisée pour communiquer avec des composants électroniques comme des capteurs, des mémoires ou des périphériques qui utilisent l'interface SPI.

appworker.cpp

```
1  #include <unistd.h>
2
3  #include <iostream>
4
5  #include <QThread>
6  #include <QSerialPortInfo>
7
8  #include "spidev.h"
9  #include "appworker.h"
10
11 using namespace std;
12
13 AppWorker::AppWorker(QObject *parent) : QObject(parent)
14 {
15     // Instancier l'objet moniteur série
16     _monitor = new SerialSender("/dev/serial0");
17
18     // Instancier le périphérique SPI associé à l'ADC MAX11100
19     spiDev_cfg_t spiConfig;
20     const char spiDevice[] = "/dev/spidev0.0";
21     spiConfig.mode=0;
22     spiConfig.speed=1000000; // SCLK = 1MHz
23     spiConfig.delay=0;
24     spiConfig.bits_per_word=8;
25
26     _max11100 = new SpiDev(spiDevice, &spiConfig);
27 }
28
29 void AppWorker::demarrer()
30 {
31     // SI périphérique SPI initialisé correctement ALORS
32     if (_max11100->begin())
33     {
34         // Lancer l'échantillonnage de la tension d'entrée de l'ADC toutes les 10ms
35         _timer = new QTimer();
36         _timer->setInterval(10);
37         connect(_timer, &QTimer::timeout, this, &AppWorker::onTimerTimeOut);
38         _timer->start();
39     }
40 }
41
42 void AppWorker::quitter()
43 {
44     delete _max11100;
45     delete _monitor;
46 }
47
48 void AppWorker::onTimerTimeOut()
49 {
50     uint8_t txBuffer[3] = {0, 0, 0};
51     uint8_t rxBuffer[3];
```

```

52
53 // Acquérir tension depuis l'ADC MAX11100
54 memset(rxBuffer, 0, sizeof(txBuffer)/sizeof(uint8_t));
55
56 _max11100->xfer(
57     txBuffer, sizeof(txBuffer)/sizeof(uint8_t)
58     , rxBuffer, sizeof(txBuffer)/sizeof(uint8_t)
59 );
60
61 // Concaténer les 2 dernier octets de la trame (image de la tension
62 // d'entrée de l'ADC) en un entier sur 16 bits
63 uint16_t tensionNumerique = (rxBuffer[ 1 ] << 8 | rxBuffer[ 2 ]);
64
65 // Multiplier cet entier de 16 bits par le quantum de l'ADC
66 // pour obtenir la tension en Volt
67 double tensionAnalogique = (tensionNumerique * 3.3) / 65536;
68
69 // Convertir la tension en Volt en chaîne de caractères (avec
70 // saut de ligne à la fin)
71 QString tensionChaine = QString::number(tensionAnalogique, 'f', 6);
72 tensionChaine.append('\n');
73
74 // Envoyer sur liaison série
75 _monitor->send(tensionChaine);
76 }
77

```

En résumé

Ce code est un processus de lecture périodique de la tension d'un ADC (ici le MAX11100) via SPI. À chaque intervalle de 10 ms :

Les données de l'ADC sont lues.

Elles sont converties en tension analogique.

La valeur est ensuite envoyée via le port série (/dev/serial0).

Ce système semble être utilisé pour surveiller une tension (par exemple, d'un capteur) et l'envoyer en temps réel via une liaison série.

Main.cpp

```

1  #include <QCoreApplication>
2  #include <QThread>
3
4  #include "appworker.h"
5
6  int main(int argc, char *argv[])
7  {
8      QCoreApplication a(argc, argv);
9
10     AppWorker app;
11     app.demarrer();
12
13     return a.exec();
14 }
15

```

En résumé

Le programme crée une application Qt qui utilise `QCoreApplication` pour gérer l'exécution de l'application. Un objet `AppWorker` est créé et sa méthode `demarrer()` est appelée pour initialiser l'échantillonnage des données via SPI. Ensuite, la boucle d'événements Qt (`a.exec()`) démarre pour permettre à l'application de fonctionner de manière continue, en gérant les événements comme les minuteries, les entrées/sorties série, etc.

Serialsender.cpp

```
1  #include <QDebug>
2
3  #include "serialsender.h"
4
5  SerialSender::SerialSender(QString serialPortName, QObject *parent) : QObject(parent)
6  {
7      // Crée un objet associé au port série dont le nom
8      // a été fourni lors de l'appel au constructeur
9      _serial = new QSerialPort();
10
11     // Configure le port série
12     _serial->setPortName(serialPortName);
13     _serial->setBaudRate(QSerialPort::Baud115200);
14     _serial->setDataBits(QSerialPort::Data8);
15     _serial->setStopBits(QSerialPort::OneStop);
16     _serial->setParity(QSerialPort::NoParity);
17     _serial->setFlowControl(QSerialPort::NoFlowControl);
18
19     // Ouvre le port série en sortie uniquement
20     _serial->open(QIODevice::WriteOnly);
21 }
22
23 void SerialSender::send(QString data)
24 {
25     _serial->write(data.toUtf8());
26 }
27
28 |
```

En résumé

La classe `SerialSender` gère la communication série avec un périphérique via un port série spécifique. Elle configure le port série avec des paramètres comme le nom du port, le débit en bauds, les bits de données, les bits de stop, la parité et le contrôle de flux. Ensuite, elle permet d'envoyer des données via le port série à l'aide de la méthode `send`.

spidev.cpp

Le code source de la bibliothèque `Spidev` provient de M. Defrance, que j'ai récupéré directement sur le dépôt Framagit

liens du depot : <https://framagit.org/codebase-bts-ciel/rasp-io-libs>

```

1  #include <stdio.h>
2  #include <stdint.h>
3  #include <fcntl.h>
4  #include <stdlib.h>
5  #include <unistd.h>
6  #include <errno.h>
7  #include <sys/ioctl.h>
8  #include <string.h>
9
10 #include <linux/spi/spidev.h>
11
12 #include "spidev.h"
13
14 SpiDev::SpiDev(const char * devName){
15     _devName = NULL;
16     if (devName != NULL ){
17         _devName = (char *)malloc(strlen(devName)+1);
18         if (_devName != NULL)
19             strcpy(_devName,devName);
20     }
21     _isOpen = false;
22 }
23
24 SpiDev::SpiDev(const char * devName, spidev_cfg_t * cfg){
25     _devName = NULL;
26     if (devName != NULL ){
27         _devName = (char *)malloc(strlen(devName)+1);
28         if (_devName != NULL)
29             strcpy(_devName, devName);
30     }
31     if (cfg != NULL){
32         memcpy(&_cfg, cfg, sizeof(spidev_cfg_t));
33     }
34     else {
35         _cfg.mode = 0;
36         _cfg.speed = 1000000;
37         _cfg.bits_per_word = 8;
38         _cfg.delay = 0;
39     }
40     _isOpen = false;
41 }
42
43
44 SpiDev::~SpiDev(){
45     if (_devName != NULL ) {
46         free(_devName);
47         _devName = NULL;
48     }
49     if (_isOpen)
50         close(_fd);

```

(extrait du code source de spidev.cpp)

Ce code fournit une interface pour configurer et interagir avec un périphérique SPI sous Linux, en permettant de définir les paramètres SPI (mode, vitesse, bits) et de transférer des données entre le système et le périphérique via des appels système SPI.

1.4 Création d'un Dashboard Grafana pour la visualisation des ECG



Présentation de grafana : **Grafana** est une plateforme open-source de visualisation et d'analyse de données, principalement utilisée pour la création de tableaux de bord interactifs permettant de suivre des métriques provenant de diverses sources de données en temps réel. Elle est particulièrement populaire pour le monitoring des systèmes, la visualisation des données d'application et la surveillance des performances des infrastructures. Grafana permet de récupérer et de visualiser des données provenant de plusieurs sources telles que des bases de données, des systèmes de monitoring, des services cloud, ou des flux de données en temps réel.

Fonctionnalités principales de Grafana :

1. **Visualisation des Données** : Grafana permet de créer des graphiques interactifs à partir de diverses sources de données comme les bases de données SQL, NoSQL, et les systèmes de monitoring. Elle offre une grande flexibilité pour personnaliser les visualisations selon les besoins de l'utilisateur.
2. **Tableaux de Bord Personnalisables** : L'utilisateur peut créer des tableaux de bord en ajoutant des graphiques, des diagrammes et d'autres visualisations afin d'obtenir une vue d'ensemble complète des données surveillées.
3. **Alertes et Notifications** : Grafana permet de configurer des alertes pour être notifié lorsque certaines conditions sont remplies (par exemple, lorsqu'une métrique dépasse un seuil défini).
4. **Support de multiples Sources de Données** : Grafana prend en charge une large gamme de sources de données, telles que MySQL, PostgreSQL, Prometheus, InfluxDB, ou encore MQTT, ce qui la rend adaptée à divers scénarios de collecte de données.

1.4 Installation de grafana

Installer les dépendances nécessaires Grafana nécessite certaines dépendances. Tu peux les installer avec cette commande :

```
sudo apt install -y software-properties-common
```

Ajouter le dépôt de Grafana Pour installer Grafana, tu dois d'abord ajouter le dépôt officiel. Exécute cette commande pour l'ajouter :

```
sudo add-apt-repository "deb https://packages.grafana.com/oss/deb stable main"
```

Ajouter la clé GPG pour vérifier l'intégrité des paquets (Ajouter une clé GPG est une mesure de sécurité essentielle pour s'assurer que les paquets logiciels installés sont authentiques et n'ont pas été modifiés. Cela protège le système contre les logiciels malveillants, les modifications non autorisées et garantit que le logiciel provient bien de la source prévue.)

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys 3A8F3CC8F7E1E3D0
```

Installer Grafana Maintenant, installe Grafana en exécutant cette commande :

```
sudo apt install grafana
```

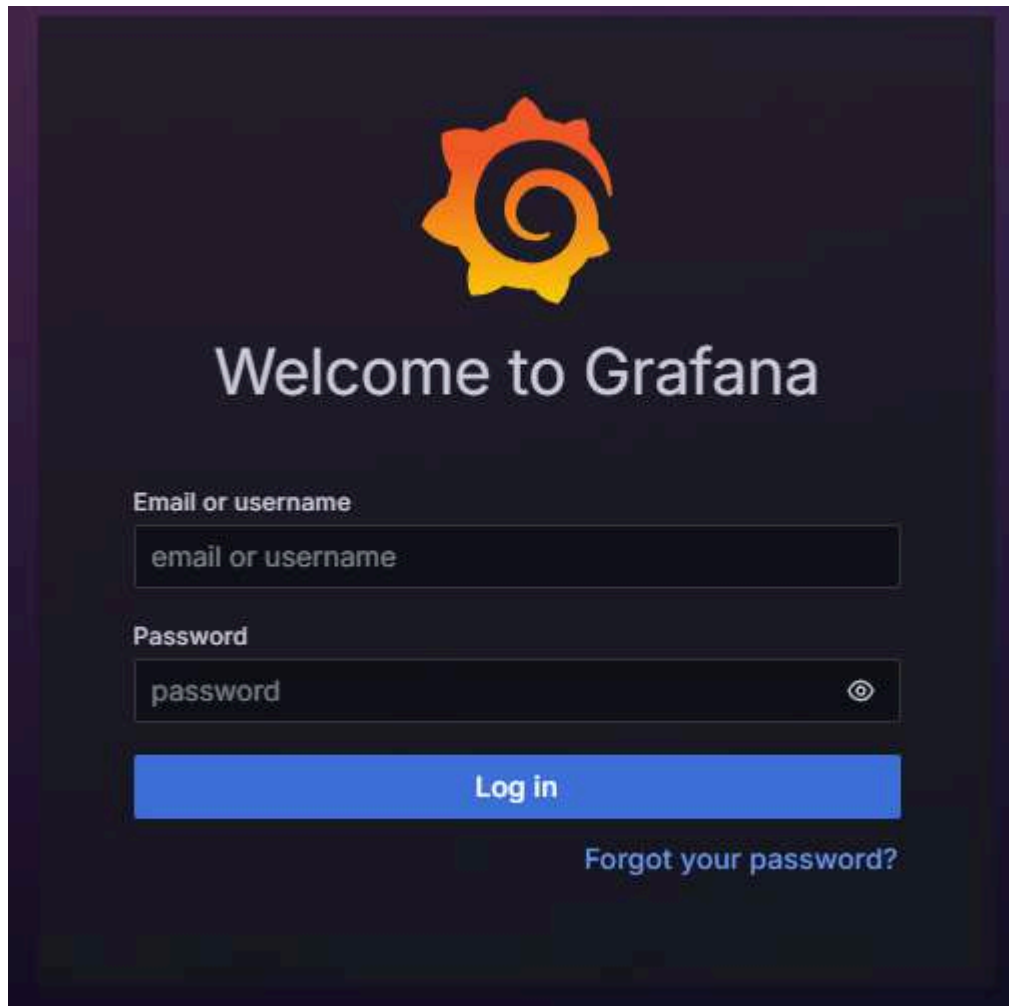
Activer et démarrer le service Grafana Une fois l'installation terminée, tu peux démarrer Grafana et le configurer pour qu'il démarre automatiquement au démarrage de ton Raspberry Pi avec ces commandes :

```
sudo systemctl enable grafana-server  
sudo systemctl start grafana-server
```

Vérifier l'installation Pour vérifier que Grafana fonctionne correctement, tu peux accéder à l'interface web de Grafana depuis ton navigateur. L'interface est disponible à l'adresse suivante :

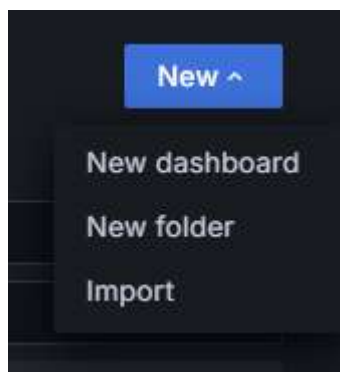
<http://192.168.4.86:3000>

Nous arrivons directement sur la page de login Grafana !

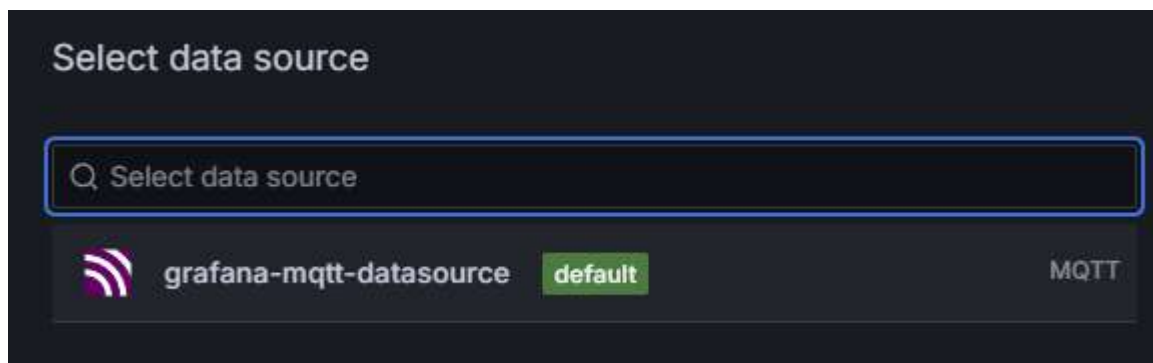


Maintenant nous allons voir comment créer un graphique (Dashboard)

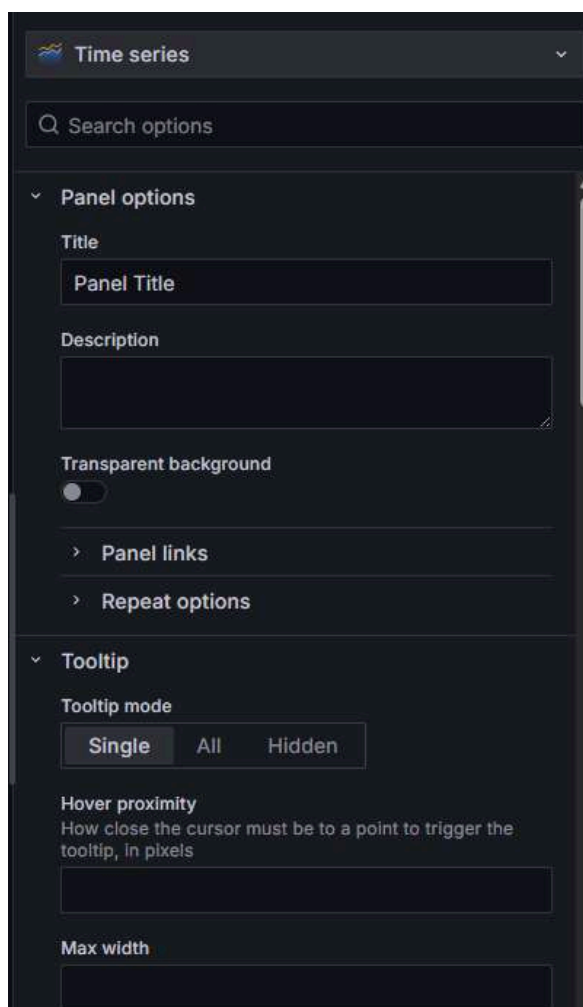
Du coup une fois arrivé dans le menu de Grafana après s'être connecté avec les identifiants admin/admin, il suffit d'aller dans le panel "Dashboard" et cliquer sur "New" et "New dashboard".



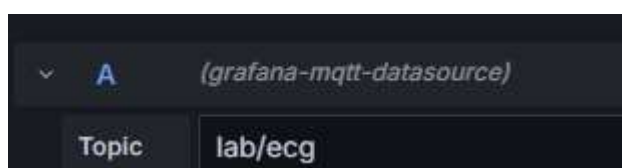
on continue en cliquant sur + Add visualization et on selectionne la data source "grafana-mqtt-datasource"



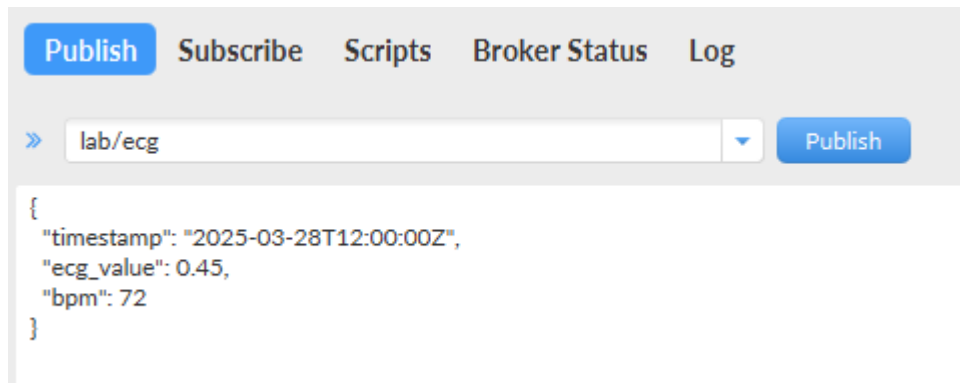
on ajoute les informations comme le nom du dashboard sur la droite de l'interface



sans oublié de mettre sont topic



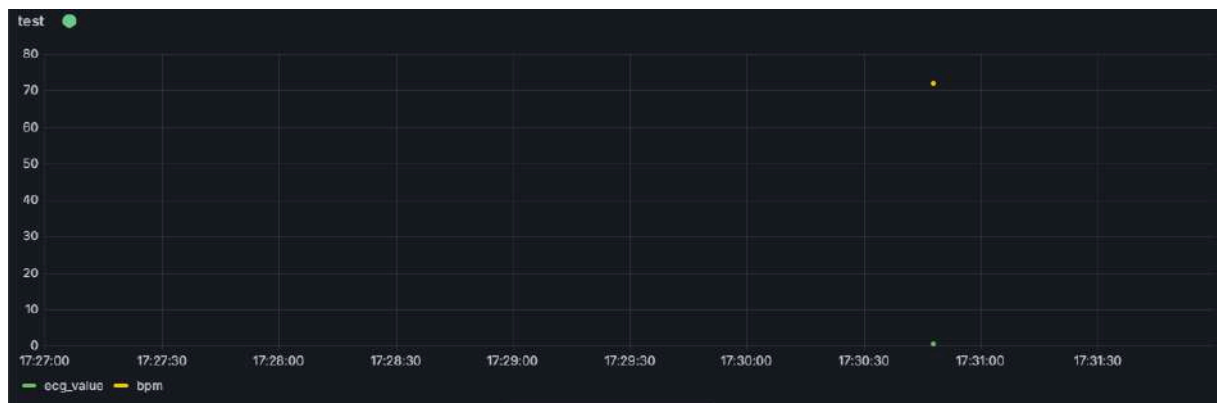
Nous allons effectuer un test afin de vérifier le bon fonctionnement de l'envoi des données et nous assurer que Grafana récupère bien les données depuis le broker MQTT



The image shows a web-based MQTT client interface. At the top, there are navigation buttons: 'Publish' (highlighted in blue), 'Subscribe', 'Scripts', 'Broker Status', and 'Log'. Below these, there is a text input field containing 'lab/ecg' and a 'Publish' button. Underneath the input field, a JSON payload is displayed:

```
{
  "timestamp": "2025-03-28T12:00:00Z",
  "ecg_value": 0.45,
  "bpm": 72
}
```

L'envoi des données au format JSON est effectué, et nous pouvons confirmer que les données sont bien transmises, car notre dashboard Grafana affiche correctement les données envoyées manuellement

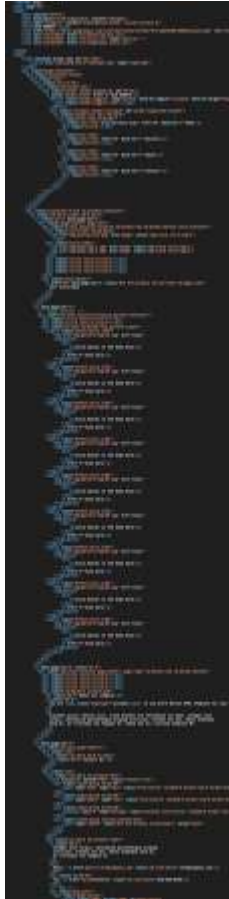


Profitions d'être dans Grafana pour récupérer le lien d'intégration afin de l'insérer dans une iframe, qui se présentera sous la forme suivante :

```
<iframe
src="http://192.168.4.86:3000/d/ce9z0112zxatca/test-dash?orgId=1&from=2025-01-14T13:50:19.056Z&to=2025-01-14T13:55:19.056Z&timezone=browser&viewPanel=panel-1&theme=dark" width="800" height="600" frameborder="0"></iframe>
```

1.4 Test d'intégration Iframe

Comme vu précédemment, nous avons récupéré notre lien de partage Grafana afin de l'intégrer sous forme d'iframe sur notre site. Pour le moment, il s'agit seulement d'une phase de test, donc le site est uniquement en HTML et CSS



(Exemple du site internet en html vu de loin)

Une fois notre iframe correctement intégrée au code, il faut vérifier que nous avons bien accès à notre dashboard depuis notre site en local.

```
<div class="container-fluid tm-content-container">
  <ul class="cd-hero-slider mb-0 py-5">
    <li class="px-3" data-page-no="1">
      <div class="page-width-1 page-left">
        <div class="d-flex position-relative tm-border-top tm-border-bottom intro-container">
          <div class="intro-left tm-bg-dark">
            <iframe src="http://192.168.4.86:3000/d/ce0e01122xatca/test_dash?orgId=1&from=2025-01-14T13:50:19.056Z&to=2025-01-14T13:55:19.056Z&timezone-browser&viewPanel=panel-1" />
          </div>
          <div class="intro-right">
```

Malheureusement, j'ai rencontré un problème et j'ai dû modifier certains fichiers de **grafana.ini** pour activer l'authentification anonyme. Pour cela, c'est très simple : il suffit d'exécuter la commande suivante pour ouvrir l'éditeur nano et modifier le fichier **grafana.ini**

```
sudo nano /etc/grafana/grafana.ini
```

Ensuite, nous allons éditer ces trois lignes ou les ajouter, selon les installations, pour activer l'authentification anonyme dans le fichier `grafana.ini`."

Les lignes à modifier ou ajouter sont :

```
[auth.anonymous]
enabled = true
org_name = Main Org.
org_role = Viewer
```

De mon côté, je n'ai pas eu d'autres problèmes, mais j'ai vu sur des forums comme Reddit que certaines personnes se plaignent que cela ne fonctionne toujours pas !

1.4 Test de l'Iframe + Convertisseur

Nous entrons maintenant dans une nouvelle phase de test, celle de tester le convertisseur analogique-numérique pour obtenir des données traitables. J'ai assemblé plusieurs programmes qui me servent à :

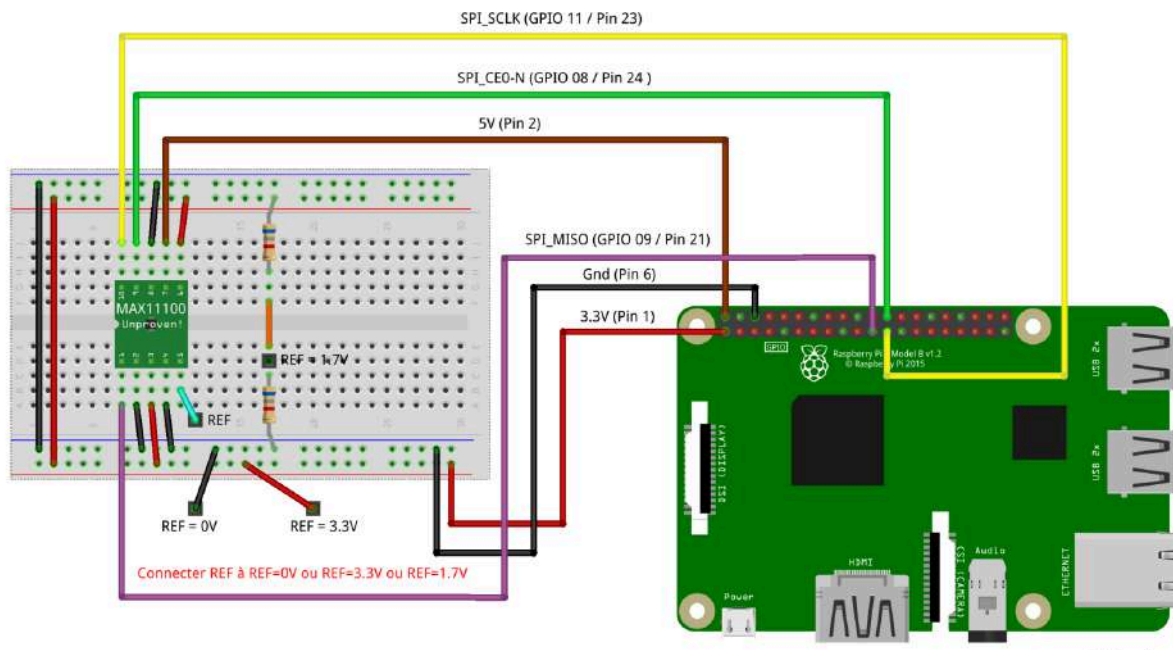
- Récupérer le signal et le convertir en numérique
- Convertir cette donnée numérique en donnée traitable
- Convertir la donnée traitable en BPM
- Mettre la donnée BPM au format JSON
- Envoyer l'info au format JSON dans mon broker

Ensuite, Grafana s'occupe du reste pour afficher les données.

```
1  #include <stdio.h>
2  #include <stdint.h>
3  #include <stdlib.h>
4  #include <string.h>
5  #include <unistd.h>
6  #include <vector>
7  #include <mosquitto.h>
8  #include <QDebug>
9  #include "spidev.h"
10 #include <unistd.h> // Pour usleep()
11
12 #define SPT_DEVICE "/dev/spidev0.0"
13 #define SAMPLE_RATE 33 // Fréquence d'échantillonnage réduite (10x plus lent)
14 #define THRESHOLD 1000 // Seuil de détection des pics R
15 #define MQTT_HOST "192.168.4.80"
16 #define MQTT_PORT 1883
17 #define MQTT_TOPIC "Lab/temperature"
18
19 std::vector<std::chrono::steady_clock::time_point> peakTimes;
20
21 // B fonction pour envoyer les BPM via MQTT avec LOGS
22 void sendMQTTData(int bpm) {
23     qDebug() << "B Initialisation MQTT...";
24     mosquitto_lib_init();
25     struct mosquitto *mosq = mosquitto_new(NULL, true, NULL);
26
27     if (!mosq) {
28         qDebug() << "X Erreur : Impossible de créer une instance Mosquitto !";
29         return;
30     }
31
32     qDebug() << "B Tentative de connexion au broker MQTT : " << MQTT_HOST << "!" << MQTT_PORT;
33     if (mosquitto_connect(mosq, MQTT_HOST, MQTT_PORT, 1) != MOSQ_ERR_SUCCESS) {
34         qDebug() << "X Erreur : Échec de la connexion au broker MQTT !";
35         mosquitto_destroy(mosq);
36         return;
37     }
38
39     // Créer le message JSON pour envoyer les BPM
40     char payload[100];
```

(Extrait du programme qui était juste un test)

Une fois que le programme est bien lancé et que notre montage du convertisseur est en place, pour tester la transmission des données et vérifier que la conversion se fait correctement, nous allons brancher nos tensions de référence sur le REF 1.7V et le REF 3.3V. Cela est temporaire, car ces tests sont réalisés pour valider le fonctionnement.



(Le montage ci-dessus)

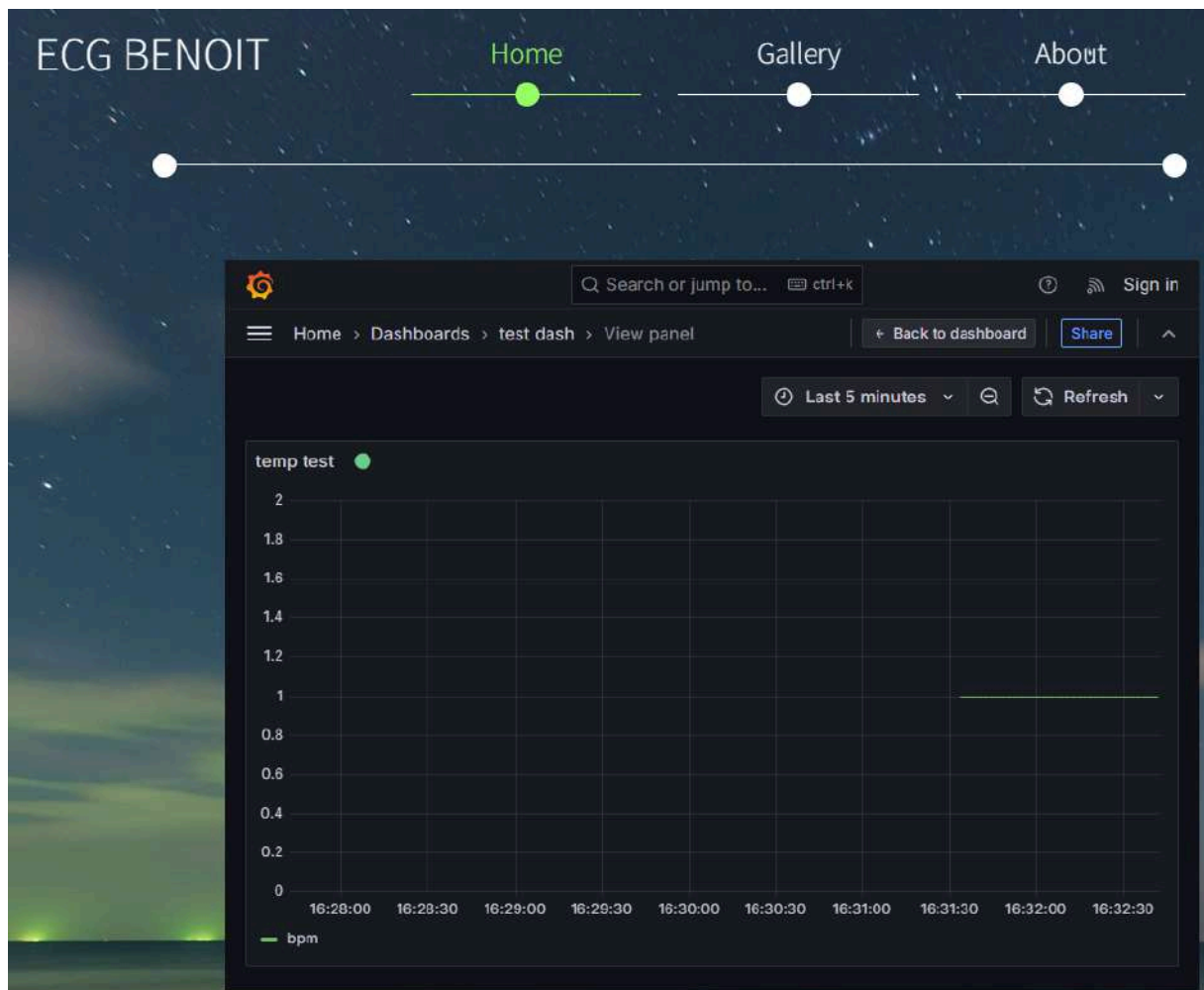
Nous pouvons exécuter notre programme pour vérifier son bon fonctionnement. Une fois le programme exécuté, nous constatons qu'il se passe bien quelque chose, et que des valeurs sont calculées !

```
ecg2 X
[ ] Durée entre les pics : 38686 ms
[ ] BPM calculé : 1
[ ] Durée entre les pics : 38686 ms
[ ] BPM calculé : 1
[ ] Durée entre les pics : 38686 ms
[ ] BPM calculé : 1
[ ] Durée entre les pics : 38686 ms
[ ] BPM calculé : 1
[ ] Durée entre les pics : 38686 ms
[ ] BPM calculé : 1
[ ] Durée entre les pics : 38686 ms
[ ] BPM calculé : 1
[ ] Durée entre les pics : 38686 ms
[ ] BPM calculé : 1
```

Mais maintenant, il faut vérifier si les informations qui nous intéressent sont bien envoyées au broker. Pour cela, nous allons nous abonner à notre topic et nous assurer que nous retrouvons bien notre valeur dans ce format précis, afin que Grafana puisse l'interpréter correctement.

```
"bpm": 1}
"bpm": 1}
"bpm": 1}
"bpm": 1}
"bpm": 1}
```

Donc, si le broker reçoit ces informations, je n'ai aucun souci à me faire pour le dashboard sur mon site, qui devrait afficher plusieurs valeurs comme celle-ci :



Pour de futures évolutions avec Grafana, je prévois d'utiliser Grafana 8.x avec l'API de streaming, afin d'obtenir des visualisations en temps réel plus fluides. En effet, le système actuel manque de fluidité.

1.4 Test du simulateur + holter

Maintenant que nous avons de bonnes bases de code, nous pouvons tester les deux appareils connectés entre eux (le simulateur vers le holter). Nous allons vérifier si le simulateur transmet bien les données vers mon convertisseur.

Étant donné que je vais devoir afficher les valeurs de manière précise, je vais ne pas utiliser Grafana, car j'ai besoin de quelque chose qui fonctionne à 100%. Nous allons utiliser le port série pour envoyer les données au logiciel **BetterSerialPlotter**.**

BetterSerialPlotter est un outil utilisé pour visualiser et tracer des données transmises via une communication série, généralement depuis des microcontrôleurs ou d'autres dispositifs qui envoient des données par interface série (comme le UART). Il permet aux utilisateurs de tracer des données en temps réel sous forme de graphiques interactifs et conviviaux, ce qui est utile pour le débogage, l'analyse ou la visualisation des données provenant de capteurs, systèmes de contrôle ou autres sorties de systèmes embarqués.

Les fonctionnalités principales incluent :

- Visualisation des données en temps réel.
- Traçage de plusieurs canaux ou variables simultanément.
- Personnalisation de l'apparence des graphiques.
- Exportation des données ou graphiques pour une analyse ultérieure.

BetterSerialPlotter est souvent préféré par rapport aux outils de traçage série de base en raison de sa flexibilité, facilité d'utilisation et fonctionnalités avancées pour la visualisation des données série

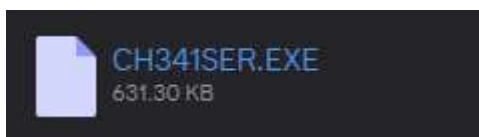
Du coup, je vais devoir utiliser un convertisseur USB <-> série pour établir la communication entre le Raspberry Pi et le convertisseur.



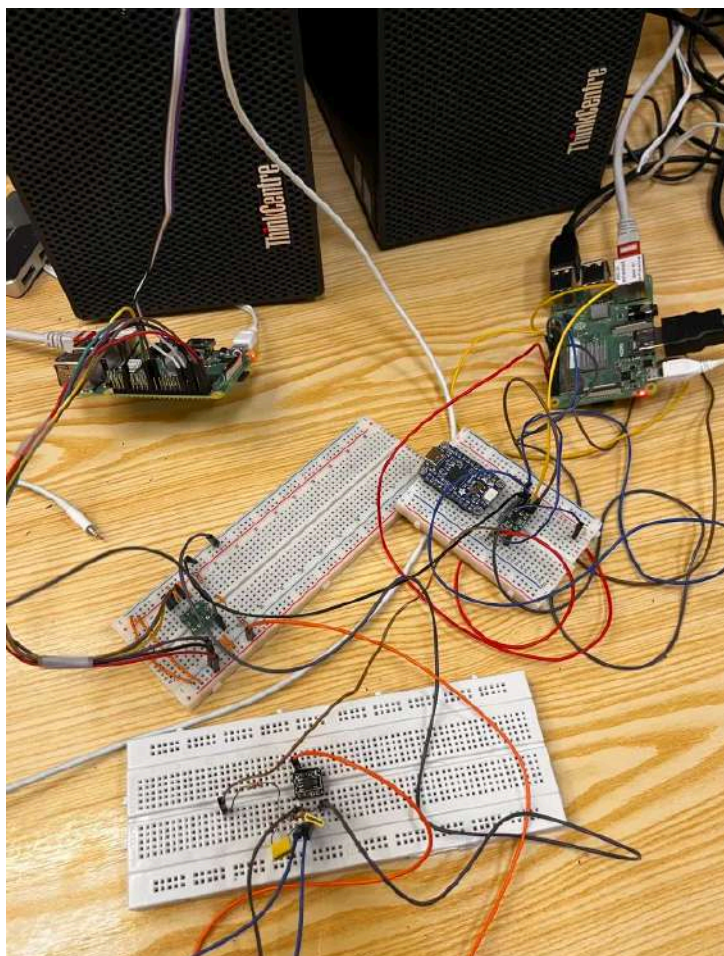
Il faudra donc câbler le convertisseur de cette manière :

- TRX convertisseur → DRX Raspberry
- DRX convertisseur → TRX Raspberry
- GND convertisseur → GND du circuit

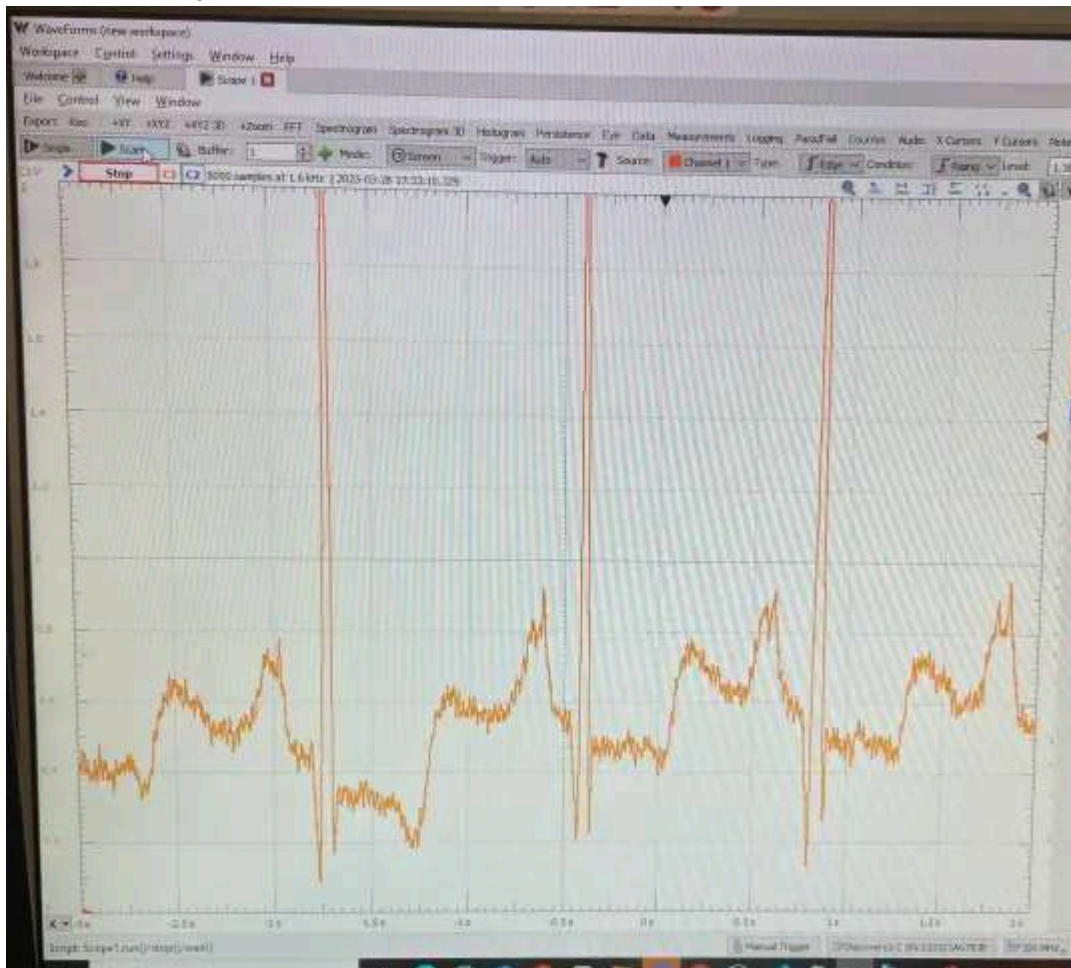
Pour éviter les problèmes de connectivité entre Windows et l'appareil, il est nécessaire de télécharger un vieux pilote, car j'avais rencontré un souci lors des premiers tests avec M. Defrance, où le message indiquait que le port n'était pas ouvert lorsque j'exécutais mon programme.



Maintenant que nous avons un port USB → série fonctionnel et un montage complet du Holter ainsi que du simulateur, nous pouvons tester le programme du simulateur (qui est fonctionnel, car nous avons effectué des tests avec des waveforms au préalable). À présent, nous voulons vérifier si nous recevons correctement les valeurs.



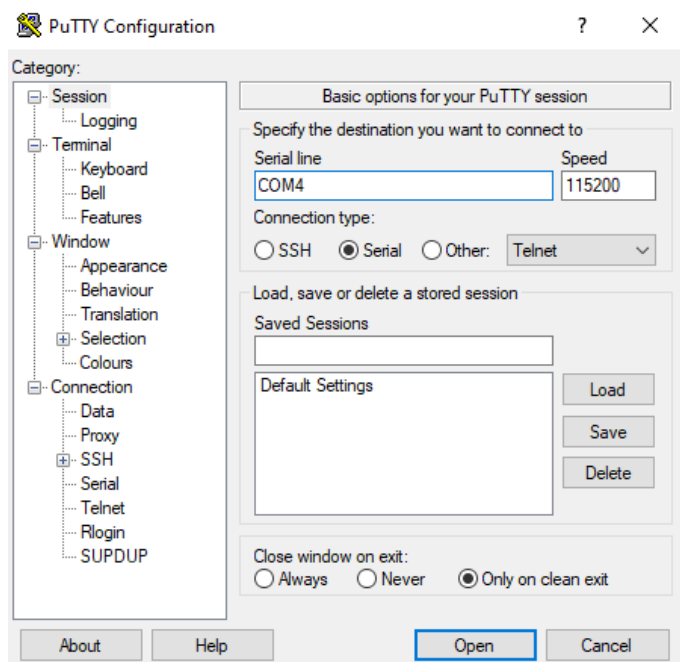
Johann, lorsqu'il exécute son programme, peut percevoir cela sur le waveform. Donc, si tout se passe bien, je devrais obtenir les mêmes résultats sur betterserialplotter !



Dans un premier temps, nous allons tester notre programme à vide en l'exécutant une première fois.

```
app-max11100-monitor X
17:40:07: Starting /home/pi/devQt/ecg-main/qt/build-app-max11100-monitor-Desktop-Debug/app-max11100-monitor...
```

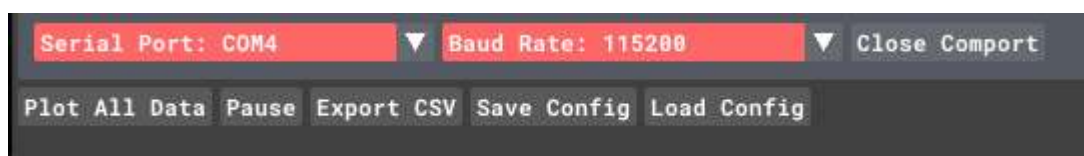
Voilà, notre programme est exécuté et nous allons maintenant démarrer PuTTY pour voir si nous avons bien des valeurs fluctuantes autour de 0 !

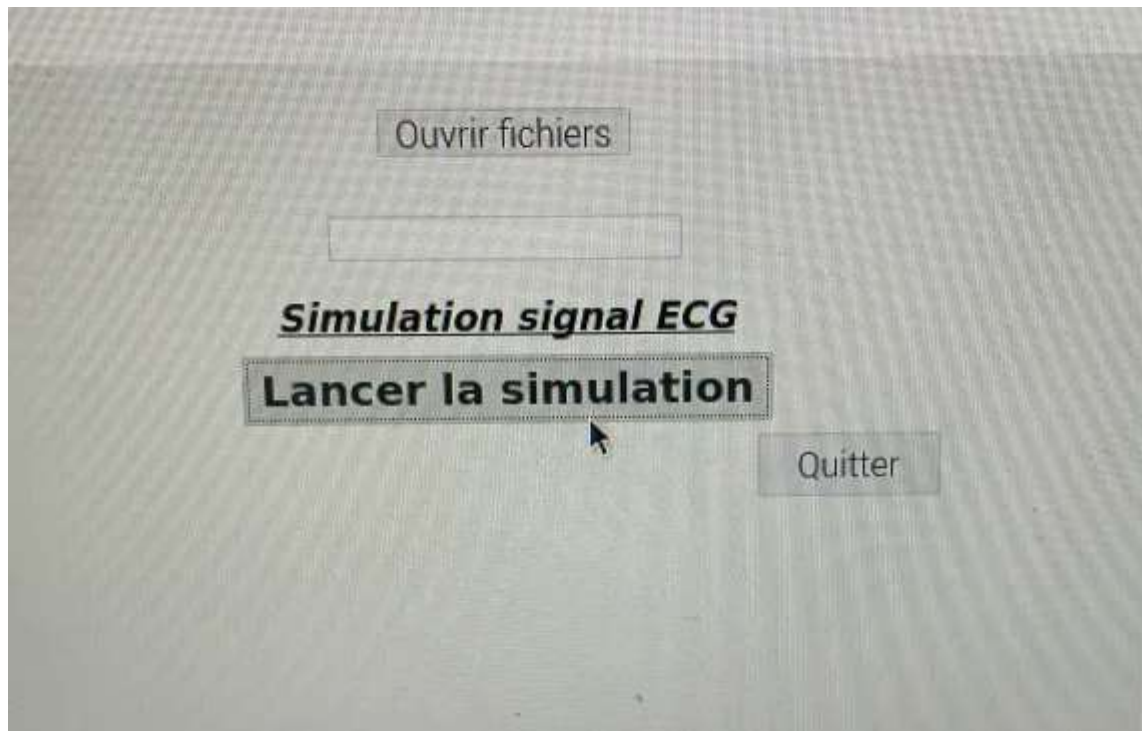


Voici la configuration à mettre pour démarrer putty !

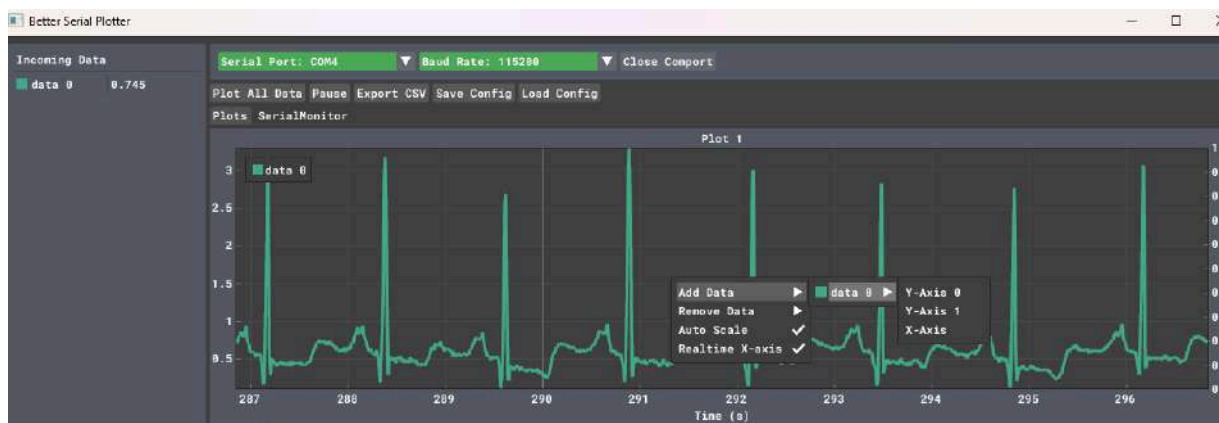


et la je peut confirmé que le programme est bien fonctionnel ! maintenant nous pouvons fermer putty et lancé betterserialplotter et mettre la meme configuration que ici !





Johann peut alors lancé la simulation et moi je vais lancé le programme a nouveau



et je confirme bien que je reçois les valeurs que johann m'a envoyé

Fiche d'audit de sécurité

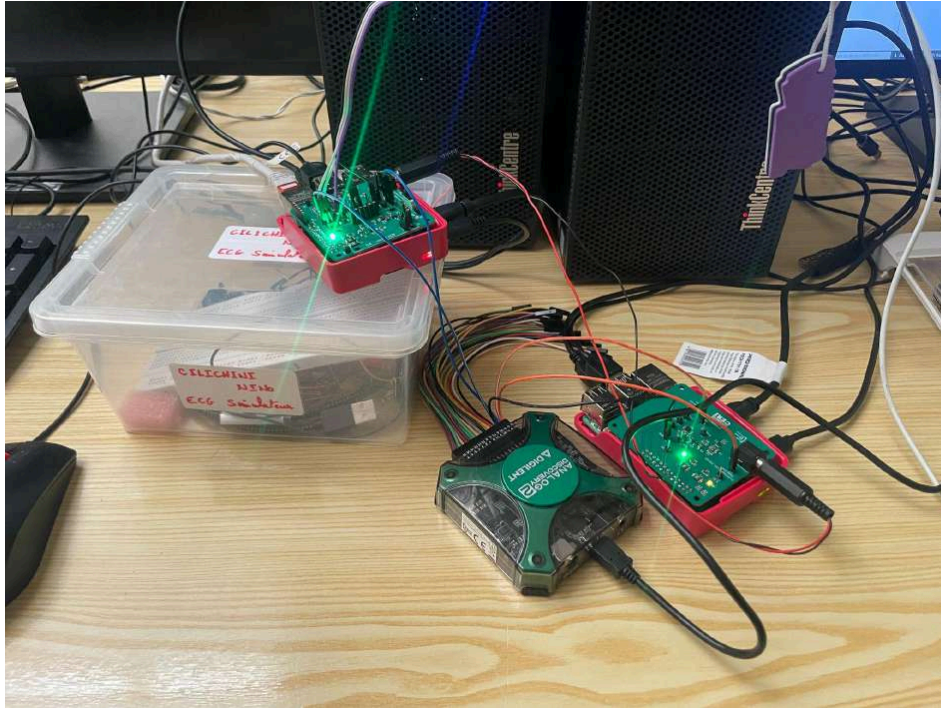
Fiche d'audit de sécurité.		
Projet	Auteur	Date
Projet ECG 2025		
Partie de projet		
Désignation	Risque	Solution
Accès WEB	Intrusion, injection de requête SQL, lecture de trame	Passer le site en HTTPS Sécuriser les formulaires Sécuriser les droits des fichiers sur le serveur Mot de passe complexe
Transmission des données MQTT	Interception des données ECG sensibles	Utilisation de TLS (SSL) pour MQTT ?
Accès au Raspberry Pi	Accès non autorisé à la machine	Mots de passe forts, SSH avec clé ,Mettre en place firewall, désactiver accès <u>root</u>
Logiciel (firmware/applications)	Vulnérabilité dans le code	Analyse statique, tests de pénétration
Interface utilisateur (IHM)	Injection de commandes malveillantes	Implémenter validation stricte
Synthèse		
Le projet ECG connecté présente une architecture fonctionnelle bien structurée, intégrant la simulation, la conversion et la transmission de signaux cardiaques via MQTT et LoRa. Cependant, plusieurs failles de sécurité ont été identifiées : absence de chiffrement sur MQTT, accès SSH insuffisamment sécurisé, et données non chiffrées localement. Il est recommandé de renforcer la sécurité réseau (TLS, firewall), sécuriser les accès (clé SSH), et chiffrer les données sensibles pour garantir la confidentialité et l'intégrité du système.		

Conclusion

À l'heure actuelle, le projet est devenu globalement cohérent et fonctionnel, même s'il n'est malheureusement pas encore totalement finalisé. En effet, plusieurs imprévus ont entraîné une perte de temps considérable, ce qui a freiné l'avancement de certaines parties essentielles.

Il me reste une semaine avant l'oral pour finaliser le développement du serveur web. L'objectif est de créer un site en PHP comportant un bouton permettant d'envoyer un message au format JSON vers le broker, afin de lancer la capture de l'ECG et d'enregistrer les données dans la base de données.

La partie sécurité, quant à elle, n'a pas pu être intégrée faute de temps. Nous aurions pu, par exemple, ajouter une page de connexion afin de restreindre l'accès au site et le rendre plus sécurisé. Enfin, le système de lecture en temps réel via Grafana (streaming live) n'a pas encore été mis en place, ce qui risque de saturer l'iframe sur le site en l'absence de flux direct.



Bilan de projet : Dans ce projet, j'ai développé un programme capable de convertir un signal analogique en signal numérique, puis de le transformer en une donnée exploitable en volts (avec une tension de référence de 3,3 V), grâce au convertisseur MAX11100. Ce programme permet également d'envoyer la donnée convertie sur un topic prédéfini d'un broker MQTT.

Les données récupérées sont cohérentes avec celles envoyées par le simulateur, ce qui valide le bon fonctionnement de l'ensemble du processus de conversion et de transmission.

J'ai rencontré de nombreuses difficultés, notamment lors de l'installation et de la configuration de Grafana, ainsi que dans la phase de conversion des données brutes en valeurs exploitables. Malgré ces obstacles, les résultats obtenus sont encourageants.

J'espère encore avoir le temps nécessaire pour finaliser le développement du serveur web, afin de proposer une solution complète, presque aboutie, d'un point de vue fonctionnel et technique.

Partie Personnelle
Thomas Pelat
(IR 2)

Sommaire :

1. Introduction

- 1.1 Contexte du projet
- 1.2. Objectifs
- 1.3. Technologies utilisées

2. Recherche et Analyse

- 2.1. Solutions de bases de données "In Memory" (SQLite, Redis...)
- 2.2. Broker MQTT et communication des données ECG

3. Conception et Développement

- 3.1. Application de stockage des données ECG en base de données

4. Diagramme de classe / séquence

5. Fiche recette

1. Introduction

1.1. Contexte du projet

Dans le cadre du développement d'un holter connecté au sein du CERI (Centre d'Enseignement et de Recherche en Informatique) de l'Université d'Avignon .

Le projet consiste à développer un système de supervision d'un holter ECG, un dispositif médical permettant l'enregistrement continu de l'activité cardiaque. Ce système repose sur la technologie LoRa pour la transmission des données et un broker MQTT pour la communication entre les différents composants.

1.2. Objectifs

L'objectif principal est de concevoir une architecture complète incluant :

- La collecte des données ECG via MQTT
- Le stockage optimisé des données en base de données

- Une gestion de version et une documentation claire pour faciliter l'utilisation et l'évolution du système

1.3. Technologies utilisées

Le projet repose sur plusieurs technologies clés :

- **MQTT** : protocole léger de messagerie pour l'envoi des données ECG
- **Bases de données "In Memory" (SQLite)** : pour le stockage rapide des données
- **Qt (C++)** : pour le développement des applications console
- **Développement web** : pour la supervision et l'administration du système
- **Framagit** : pour la gestion de version et la collaboration

2. Recherche et Analyse

2.1 Solutions de bases de données "In Memory" (SQLite, Redis...)

Tout d'abord j'ai choisi une base de donnée en Sqlite plutôt que Redis par le fait qu'elle est plus simple à intégrer, à déployer et qu'elle nécessite aucune connexion internet.

Une base de données SQLite "In Memory" est une base qui stocke toutes ses données directement en mémoire vive au lieu d'un fichier sur disque.

Cela signifie que :

Toutes les opérations de lecture et d'écriture sont extrêmement rapides car elles ne passent pas par un disque dur ou un SSD.

Les données sont temporairement stockées : elles disparaissent dès que l'application est fermée ou que la connexion à la base est interrompue. Elles sont ensuite envoyées et stockées sur un cloud.

On active ce mode en utilisant le nom spécial "memory" lors de la connexion à SQLite.

```

3 Database::Database() {
4     db = QSqlDatabase::addDatabase("QSQLITE");
5     db.setDatabaseName(":memory:");
6     if (!db.open()) {
7         qFatal("Impossible d'ouvrir la base de données SQLite.");
8     }

```

Les avantages d'une base Sqlite in memory :

- Les accès en lecture/écriture sont bien plus rapides qu'avec un disque dur ou un SSD.
- Idéal pour du traitement en temps réel ou des calculs rapides sur de gros volumes de données.
- Pas besoin de créer un fichier de base de données sur le disque.
- Fonctionne comme une base SQLite classique, avec le même SQL et les mêmes commandes.
- Permet de stocker des données intermédiaires sans les écrire sur le disque.
- Utile pour des tests unitaires, des données temporaires ou du traitement en batch.
- Puisque les données disparaissent à la fermeture, il n'y a pas besoin de gérer leur suppression.

Inconvénients d'une base Sqlite in memory :

- Une fois l'application fermée, toutes les données sont perdues.
- Impossible d'utiliser SQLite In Memory pour du stockage persistant.
- La taille de la base est limitée par la mémoire vive disponible.
- Si la RAM est saturée, l'application peut ralentir ou crasher.
- Comme SQLite est une base locale (non client-serveur), elle ne permet pas d'être partagée entre plusieurs programmes en mode In Memory.

2.2. Broker MQTT et communication des données ECG

1. Qu'est-ce que MQTT ?

MQTT (Message Queuing Telemetry Transport) est un protocole de communication léger conçu pour les applications IoT et M2M (Machine to Machine). Il est idéal pour envoyer et recevoir des données ECG en raison de sa faible consommation de bande passante et de son efficacité énergétique.



2. Architecture de MQTT

Le protocole fonctionne selon un modèle "publish/subscribe", où les appareils communiquent via un broker MQTT central :

- Publisher (Émetteur) : envoie des messages à un topic spécifique.
- Broker MQTT : serveur qui gère la distribution des messages.
- Subscriber (Abonné) : reçoit les messages du topic auquel il est abonné.

Exemple d'architecture pour les données ECG :

- ➔ Le dispositif de mesure ECG publie les données sur le topic `ecg/data`.
- ➔ Le broker MQTT reçoit et redistribue ces données aux abonnés.
- ➔ Le système de supervision (ex. un serveur ou une application) s'abonne à `ecg/data` pour récupérer et afficher les mesures.

3. Pourquoi utiliser MQTT pour les données ECG ?

Faible consommation réseau → Adapté aux appareils IoT et aux connexions instables.
Asynchrone et temps réel → Permet d'envoyer les données ECG en continu.

4. Intégration avec le projet

Liaison entre l'ECG et le stockage en base de données :

Les données sont reçues via MQTT et stockées dans SQLite In Memory pour un accès rapide.

5. Installation du broker MQTT

Le broker Mosquitto a été choisi pour sa légèreté et sa compatibilité avec les systèmes embarqués comme le Raspberry Pi. L'installation s'effectue comme suit :

```
sudo apt update  
sudo apt install -y mosquitto mosquitto-clients
```

`sudo apt update` → Mise à jour des paquets disponibles et Vérifié les dernières versions des logiciels disponibles sur les dépôts. Permet d'éviter d'installer une version obsolète de Mosquitto.

`sudo apt install -y mosquitto mosquitto-clients` → `mosquitto` : Installe le broker MQTT, qui sert d'intermédiaire entre les appareils qui publient et ceux qui s'abonnent aux messages.

`mosquitto-clients` : Installe des outils en ligne de commande permettant de tester facilement MQTT (`mosquitto_pub` et `mosquitto_sub`).

`-y` : Accepte automatiquement l'installation sans demander de confirmation.

Ensuite, on active et démarre le service Mosquitto :

```
sudo systemctl enable mosquitto  
sudo systemctl start mosquitto
```

`sudo systemctl enable mosquitto` → `systemctl enable` : Configure Mosquitto pour qu'il se lance automatiquement à chaque démarrage du système. Sans cette commande, Mosquitto ne démarrera pas automatiquement après un redémarrage.

```
sudo systemctl restart mosquitto
```

Il est préférable de redémarrer le service pour s'assurer qu'il prend bien en compte toutes les configurations

```
jdoo@VM-E6-2025:~$ systemctl status mosquitto.service
● mosquitto.service - Mosquitto MQTT Broker
   Loaded: loaded (/lib/systemd/system/mosquitto.service; enabled; preset: enabled)
   Active: active (running) since Fri 2025-03-28 13:08:05 CET; 3h 7min ago
     Docs: man:mosquitto.conf(5)
           man:mosquitto(8)
  Process: 710 ExecStartPre=/bin/mkdir -m 740 -p /var/log/mosquitto (code=exited, status=0/SUCCESS)
  Process: 714 ExecStartPre=/bin/chown mosquitto /var/log/mosquitto (code=exited, status=0/SUCCESS)
  Process: 720 ExecStartPre=/bin/mkdir -m 740 -p /run/mosquitto (code=exited, status=0/SUCCESS)
  Process: 723 ExecStartPre=/bin/chown mosquitto /run/mosquitto (code=exited, status=0/SUCCESS)
 Main PID: 726 (mosquitto)
    Tasks: 1 (limit: 9453)
   Memory: 2.0M
      CPU: 9.157s
  CGroup: /system.slice/mosquitto.service
          └─726 /usr/sbin/mosquitto -c /etc/mosquitto/mosquitto.conf

Warning: some journal files were not opened due to insufficient permissions.
```

Pour vérifier que mosquitto est bien en route, on tape la commande “systemctl status mosquitto.service” puis on vérifie que “active” soit écrit.

3. Conception et Développement

3.1. Application de stockage des données ECG en base de données

Après avoir installé le broker Mqtt sur une machine virtuelle Debian 12, nous devons ensuite créer notre base de donnée sur Qt creator, nous avons choisi la version Qt5 pour une meilleure compatibilité avec la raspberry pi 3.

Pour coupler Qt creator et le broker Mqtt, je dois installé des extensions comme “libqt5mqtt5-dev” pour cela il faut taper la commande : **sudo zypper install libqt5mqtt5-dev**

Nous pouvons donc commencer à créer notre base de données.

Le dossier projet.pro qui est notre base de donnée sera composé :

Des fichiers d'en-tête (.h) sont utilisés pour déclarer les classes, les fonctions, et les variables qu'on souhaite rendre accessibles à d'autres fichiers source dans votre projet. Ils servent à l'organisation du code.

Un fichier .cpp est un fichier source qui contient le code d'implémentation du programme en C++. Il sert à définir comment les différentes parties du programme vont fonctionner.

QT -= gui

QT += mqt

QT += sql

CONFIG += c++17 console

CONFIG -= app_bundle

You can make your code fail to compile if it uses deprecated APIs.

In order to do so, uncomment the following line.

#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000

disables all the APIs deprecated before Qt 6.0.0

*SOURCES += *

*app.cpp *

*centrenotifications.cpp *

*connexionbdd.cpp *

*ecgdao.cpp *

*main.cpp *

logger.cpp

Default rules for deployment.

qnx: target.path = /tmp/\${TARGET}/bin

else: unix:!android: target.path = /opt/\${TARGET}/bin!

isEmpty(target.path): INSTALLS += target

*HEADERS += *

*app.h *

*centrenotifications.h *

*connexionbdd.h *

*ecgdao.h *

logger.h

On remarque donc sur cette image la composition du dossier “appjournalisation.pro”.

Différents codes Qt:



```
1  #include <QDebug>
2  #include <QSqlQuery>
3
4  #include "connexionbdd.h"
5
6  // Initialisation des attributs static
7  ConnexionBDD * ConnexionBDD::_singleton = nullptr;
8  int ConnexionBDD::_nbRefs = 0;
9
10 ConnexionBDD * ConnexionBDD::getInstance()
11 {
12     if (_singleton == nullptr) {
13         _singleton = new ConnexionBDD();
14     }
15
16     _nbRefs++;
17
18     return _singleton;
19 }
20
21 bool ConnexionBDD::execute(QSqlQuery &query)
22 {
23     return query.exec();
24 }
25
26 void ConnexionBDD::close()
27 {
28     _nbRefs--;
29     if(_nbRefs == 0) {
30         delete _singleton;
31         _singleton = nullptr;
32     }
33 }
34
35 ConnexionBDD::ConnexionBDD()
36 {
37     _connexionBdd = QSqlDatabase::addDatabase("QSQLITE");
38     _connexionBdd.setDatabaseName(":memory:");
39
40     if(_connexionBdd.open()) {
41         qDebug() << "BDD ouverte";
42     } else {
43         qDebug() << "Erreur ouverture BDD";
44     }
45 }
46
47 ConnexionBDD::~ConnexionBDD()
48 {
49     _connexionBdd.close();
50 }
51
```

```
< > ecgdao.cpp # <Select Symbol>

1 #include <QSqlQuery>
2 #include <QStringList>
3 #include <QVariant>
4 #include <QSqlError>
5 #include <QDebug>
6
7 #include "ecgdao.h"
8
9 EcgDAO::EcgDAO()
10 {
11     _bdd = ConnexionBDD::getInstance();
12 }
13
14 /**
15  * @brief EcgDAO::init
16  * Initialize la BDD en créant les tables adéquates.
17  * @return
18  */
19 bool EcgDAO::init()
20 {
21     bool ret = false;
22     QSqlQuery req;
23
24     req.prepare("CREATE TABLE Infos ("
25         " id INTEGER PRIMARY KEY AUTOINCREMENT,"
26         " patient TEXT NOT NULL,"
27         " age INTEGER NOT NULL,"
28         " sexe TEXT NOT NULL,"
29         " date INTEGER NOT NULL);"
30     );
31
32     if(_bdd->execute(req)) {
33         ret = true;
34     } else {
35         qDebug() << req.lastError().text();
36         return ret;
37     }
38
39     req.prepare("CREATE TABLE releves ("
40         " id INTEGER PRIMARY KEY AUTOINCREMENT,"
41         " valeur INTEGER NOT NULL,"
42         " fkInfos INTEGER REFERENCES Infos (id);"
43     );
44
45     if(_bdd->execute(req)) {
46         ret = true;
47     } else {
48         qDebug() << req.lastError().text();
49     }
50
51     _startTime = QDateTime::currentDateTime();
52
53     return ret;
54 }
55
```

```

1  #include "QThread"
2
3  #include "logger.h"
4
5  Logger::Logger(QObject *parent)
6  : QObject(parent)
7  {
8      _centreNotifs = CentreNotifications::getInstance();
9
10     //connect(_centreNotifs, &CentreNotifications::sigReady, this, &Logger::onsigCentreNotificationsReady);
11 }
12
13 void Logger::~Logger()
14 {
15 }
16
17 void Logger::start(QString patient, int age, Genre genre)
18 {
19     // Création des tables de la BDD in memory
20     _ecg.init();
21
22     _startTime = QDateTime::currentDateTime();
23     _patient = patient;
24
25     _ecg.setInfos(patient, age, static_cast<char>(genre), _startTime.toSecsSinceEpoch());
26
27     // Mise en écoute des données parvenant du holter/simulateur
28     _centreNotifs->subscribe(CentreNotifications::DATA);
29     connect(_centreNotifs, &CentreNotifications::sigData, this, &Logger::onSigData);
30 }
31
32 void Logger::stop()
33 {
34     disconnect(_centreNotifs, &CentreNotifications::sigData, this, &Logger::onSigData);
35 }
36
37 void Logger::save()
38 {
39     QString filePath = _startTime.toString("dd-MM-yyyy-")
40         + "-"
41         + _patient.simplified().replace(" ", "-")
42         + ".sqlite"
43         ;
44
45     _ecg.save(filePath);
46 }
47
48 void Logger::onsigCentreNotificationsReady()
49 {
50 }
51
52 void Logger::onSigData(uint16_t data)
53 {
54     qDebug() << "<- " << data;
55     _ecg.setData(data);
56 }
57
58

```

```

centrenotifications.cpp
<Select Symbol>

1  #include <QDebug>
2
3  #include "centrenotifications.h"
4
5  // Initialisation des attributs static
6  CentreNotifications * CentreNotifications::_singleton = nullptr;
7  int CentreNotifications::_nbRefs = 8;
8
9  CentreNotifications * CentreNotifications::getInstance()
10 {
11     if (_singleton == nullptr) {
12         _singleton = new CentreNotifications();
13     }
14
15     _nbRefs++;
16
17     return _singleton;
18 }
19
20 CentreNotifications::CentreNotifications(QObject *parent)
21 : QObject(parent)
22 {
23     _client = new QMqttClient();
24
25     connect(_client, &QMqttClient::connected, this, &CentreNotifications::onClientConnected);
26     connect(_client, &QMqttClient::messageReceived, this, &CentreNotifications::onMqttMsgReceived);
27
28     _client->setHostname("localhost");
29     _client->setPort(1883);
30     _client->setClientId("App-Journalisation-ECG");
31
32     _client->connectToHost();
33 }
34
35 void CentreNotifications::subscribe(notification_t notif)
36 {
37     if (_client->state() == QMqttClient::Connected) {
38         switch (notif) {
39             case DATA:
40                 _client->subscribe(TOPIC_DONNEE);
41                 break;
42             case CONTROL:
43                 _client->subscribe(TOPIC_CONTROLE);
44             default:
45                 break;
46         }
47     } else {
48         _pendingSubscriptions.append(notif);
49     }
50 }
51
52
53
54 void CentreNotifications::close()
55 {
56     _nbRefs--;
57     if (_nbRefs == 8) {
58         delete _singleton;
59         _singleton = nullptr;
60     }
61 }
62
63 void CentreNotifications::onClientConnected()
64 {
65     qDebug() << "Connecté au broker";
66
67     emit sigReady();
68
69     // Traite les éventuelles souscriptions en attente
70     // (souscriptions faites avant la connexion effective au broker)
71     while (!_pendingSubscriptions.isEmpty()) {
72         notification_t n = _pendingSubscriptions.takeFirst();
73         subscribe(n);
74     }
75 }
76
77 void CentreNotifications::onMqttMsgReceived(const QByteArray &message, const QMqttTopicName &topic)
78 {
79     qDebug() << QString(message) << " reçu sur " << topic.name();
80     if (topic.name().contains(TOPIC_DONNEE)) {
81         emit sigData(message.toInt());
82     } else if (topic.name().contains(TOPIC_CONTROLE)) {
83         emit sigCtrl(message);
84     }
85 }

```

Script Python :

```
import time # Module standard de Python
import csv # Module standard de Python
import json
import paho.mqtt.client as mqtt # Module à installer :
# * soit via apt install python3-paho-mqtt
# * soit via pip dans environnement virtuel
# (voir https://www.raspberrypi.com/news/using-python-with-virtual-environments-the-magpi-148/)

# Paramètres du broker MQTT
broker_address = "localhost"
broker_port = 1883
broker_ctrl_topic = "ECG/ctrl"
broker_data_topic = "ECG/data"

def publish_ecg_data() :
    global client
    # Chargement des données
    data = []
    with open('NSR-100m-0-scaled.csv', newline='') as csv_file :
        lines = csv.reader(csv_file)
        for l in lines :
            data.append(l[ 0 ])

    for d in data :
        client.publish(broker_data_topic, d)
        time.sleep(0.003)

if __name__ == "__main__" :
```

```

client = mqtt.Client()
client.connect(broker_address, broker_port)

start = {
    "running" : True,
    "description" : {
        "nom" : "John Doe",
        "age" : 45,
        "genre" : 'M'
    }
}

client.publish(broker_ctrl_topic, json.dumps(start));

publish_ecg_data()

stop = {
    "running" : False,
    "description" : None
}

client.publish(broker_ctrl_topic, json.dumps(stop));

```

Ce script python sert à simuler la partie holter connecté et envoie les 3600 données grâce au fichier .csv qui est implémenté dedans.

Diagramme de classe :

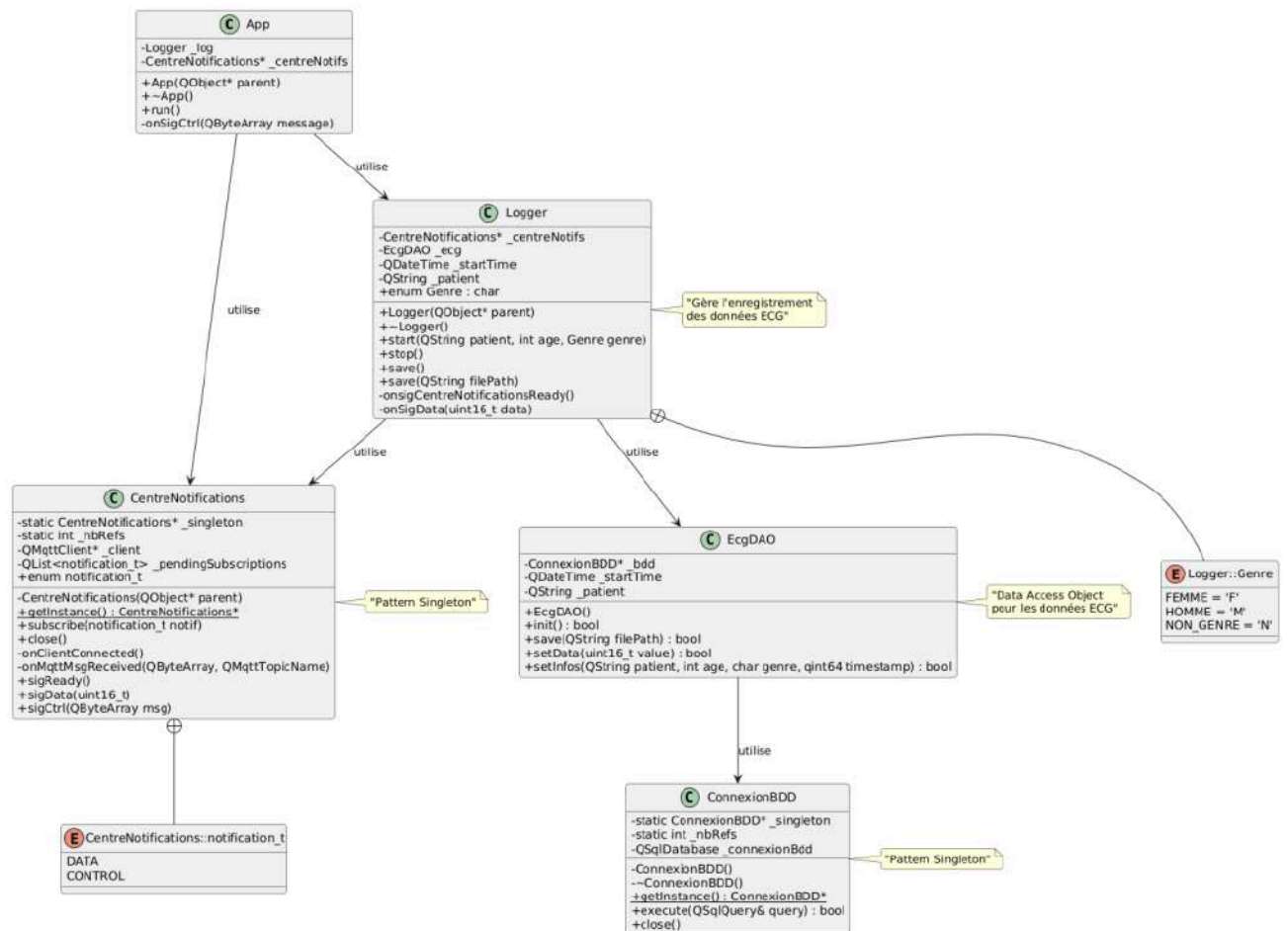
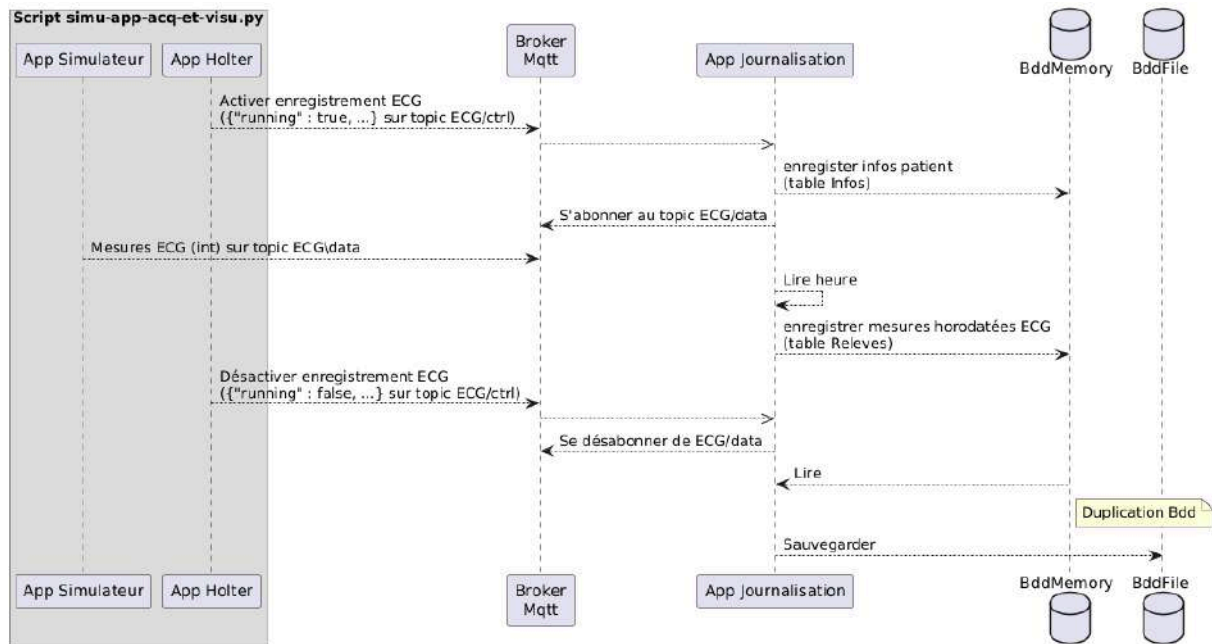


Diagramme de classe :



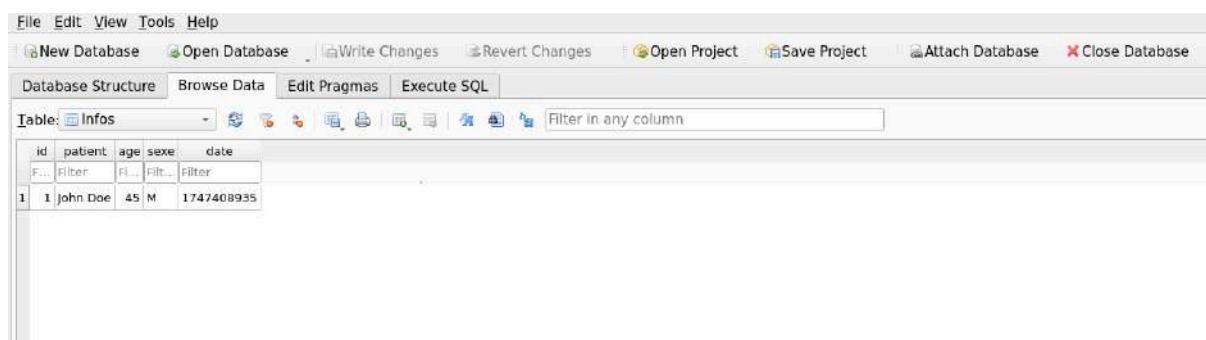
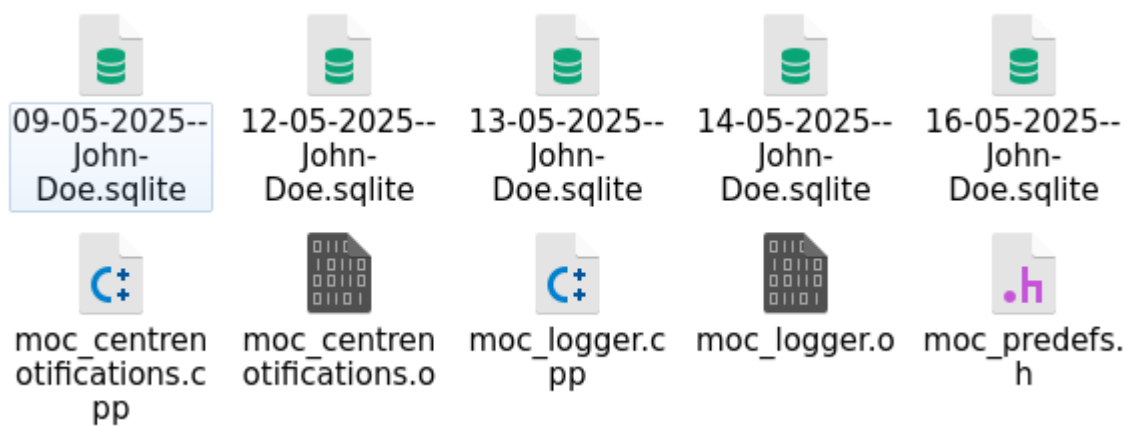
Résultat du programme Qt :

```

jdoe@VM-E6-2025: ~/projet/build-app-journalisation-Desktop-Debug$ ./app-journalisation
BDD ouverte
Connecté au broker
{"running": true, "description": {"nom": "John Doe", "age": 45, "genre": "M"}} reçu sur "ECG/ctrl"
10889 reçu sur "ECG/data" non/simu-app-acquisition-et-visualisation.py, line 45, in module
<- 10889
10552 reçu sur "ECG/data" non/simu-app-acquisition-et-visualisation.py, line 28, in publish_ecg_data
<- 10552
10215 reçu sur "ECG/data"
<- 10215
10047 reçu sur "ECG/data" non/simu-app-acquisition-et-visualisation.py
<- 10047
10384 reçu sur "ECG/data"
<- 10384
10384 reçu sur "ECG/data"
<- 10384
10552 reçu sur "ECG/data"
<- 10552
10047 reçu sur "ECG/data"
<- 10047
9710 reçu sur "ECG/data"
<- 9710
9374 reçu sur "ECG/data"
<- 9374
9879 reçu sur "ECG/data"
<- 9879
9879 reçu sur "ECG/data"
<- 9879
9879 reçu sur "ECG/data"
<- 9879
9206 reçu sur "ECG/data"
<- 9206
9374 reçu sur "ECG/data"

```

Remontées des informations dans la base de données :



4. Fiche recette

FICHE RECETTE, QUALIFICATION DU SYSTEME			
Nom du système :	Sqlite	Code de la campagne de test :	
Technicien 1 : Technicien 2 : Technicien 3 : Technicien 4 :	Thomas Pelat	Date :	17/05/25
IDENTIFICATION DU SCENARIO			
Identification du scénario de recette			
Titre :	Base de données Sqlite		
Objectif du scénario :	Remontés des informations dans une base de données Sqlite inmemory à l'aide d'un broker mqtt		
CONDITIONS INITIALES NECESSAIRES POUR EFFECTUER LA RECETTE			
Installation broker mqtt QT5 Machine virtuelle			
EXECUTION DU TEST			
Description courte :	Exécution des scripts qt et python et observer une création de fichier dans la base de données		
Résultats attendus :	Les informations du patient sont remontées dans la base de données		
BILAN			
REMARQUES			
CONCLUSION			
VALIDE		NON VALIDE	

Partie Personnelle
Johann AMADORI
(IR 3)

Sommaire

1.Introduction

- 1.1 Contexte et objectifs du projet
- 1.2 Présentation du simulateur ECG

2.Technologies utilisées

- 2.1 Bus SPI (Serial Peripheral Interface)
- 2.2 Le Raspberry Pi
- 2.3 Le DAC8551

3.Installation et configuration

- 3.1 Mise en place de l'environnement Qt
- 3.2 Problèmes rencontrés avec le FT232H
- 3.3 Solution adoptée (connexion directe Raspberry Pi – DAC)

4.Mise en œuvre

- 4.1 Organisation logicielle du simulateur
- 4.2 Fonctionnement du système (architecture, threads, SPI)
- 4.3 Génération de signaux ECG à partir de fichiers médicaux

5.Préparation des données ECG

- 5.1 Format d'origine (.mat)
- 5.2 Script Python de conversion en .csv

6.Tests et intégration matérielle

- 6.1 Vérification du signal à l'oscilloscope (Analog Discovery)
- 6.2 Cartes d'adaptation développées par les étudiants ER
- 6.3 Montage final du système avec Holter

7.Conclusion

8.Annexes simulateur

Introduction

Contexte et objectifs du simulateur

Dans le cadre d'un projet réalisé pour le CERI (Centre d'Enseignement et de Recherche en Informatique) de l'Université d'Avignon, j'ai conçu un simulateur de signaux ECG destiné à accompagner le développement d'un holter connecté. Ce simulateur a pour but de générer des signaux électrocardiographiques réalistes, afin de tester et valider des dispositifs médicaux dans un environnement maîtrisé, sans avoir à recourir à des patients réels.

Ce projet vise plusieurs objectifs concrets :

- Permettre la simulation de profils ECG variés dans des conditions sécurisées et reproductibles ;
- Offrir un support pédagogique pour former les étudiants du CERI à l'acquisition et au traitement des signaux biomédicaux ;
- Tester les communications électroniques et l'interfaçage de composants, notamment dans des travaux de recherche en IoT et IA biomédicale ;
- Fournir une interface Qt intuitive permettant de charger et d'envoyer des signaux ECG à un système embarqué, en l'occurrence un Raspberry Pi.

Ce simulateur s'inscrit donc dans une démarche à la fois pédagogique et technique, et apporte une solution concrète pour accompagner les projets du CERI dans le domaine de la santé connectée.

Présentation du simulateur ECG

Le simulateur ECG est un système complet qui permet la génération fidèle de signaux ECG à partir de relevés médicaux réels. Il peut interagir avec un holter ou d'autres dispositifs de mesure, en simulant les impulsions électriques du cœur.

Architecture technique

L'architecture repose sur les éléments suivants :

- Un Raspberry Pi, qui assure la génération continue du signal ECG ;
- Un DAC8551, convertisseur numérique-analogique 16 bits, qui transforme les valeurs numériques en tension analogique ;
- Une communication SPI entre le Raspberry Pi et le DAC pour garantir une transmission rapide et précise ;
- Une application Qt (C++) qui permet à l'utilisateur de sélectionner les fichiers ECG, démarrer/stopper la simulation, et suivre son état via une interface graphique.

Initialement, une communication via le module FT232H (USB-SPI) avait été envisagée, mais elle s'est révélée trop lente (latence bloquée à 15 ms), rendant impossible l'atteinte des 360 Hz nécessaires (1 point toutes les ~3 ms). J'ai donc opté pour une connexion SPI native sur le Raspberry Pi, bien plus réactive.

Fonctionnalités principales

Le simulateur permet :

- La lecture de fichiers ECG au format `.csv`, issus de bases de données médicales réelles ;
 - La génération analogique fidèle des signaux ECG ;
 - Le contrôle en temps réel du signal via l'application Qt ;
 - La simulation de scénarios cliniques spécifiques (ex : anomalies cardiaques) pour tester des dispositifs dans différentes conditions.
-

Ce simulateur représente une plateforme d'expérimentation efficace, utile pour les étudiants, les enseignants, et les chercheurs du CERI souhaitant explorer les problématiques liées à l'embarquée biomédicale, à l'IoT médical et à l'intégration de signaux dans des environnements critiques.

Explication des Technologies Utilisées

Le bus SPI (Serial Peripheral Interface)

Le bus SPI (Serial Peripheral Interface) est un protocole de communication série utilisé pour l'échange rapide de données entre un microcontrôleur et divers périphériques comme les capteurs, les écrans ou les mémoires flash. Il permet une communication bidirectionnelle efficace en utilisant un ensemble de fils dédiés pour la transmission et la réception des données. Ce protocole est largement adopté dans les systèmes embarqués en raison de sa simplicité et de sa rapidité d'exécution.

Protocole de communication série synchrone

Le protocole SPI est un type de communication série synchrone, ce qui signifie que l'échange de données est cadencé par un signal d'horloge partagé entre les composants. Contrairement aux protocoles asynchrones, où l'horloge n'est pas partagée, le SPI garantit une transmission précise et coordonnée des informations. Grâce à cette synchronisation, le SPI peut fonctionner à des vitesses élevées tout en réduisant les erreurs de transmission.

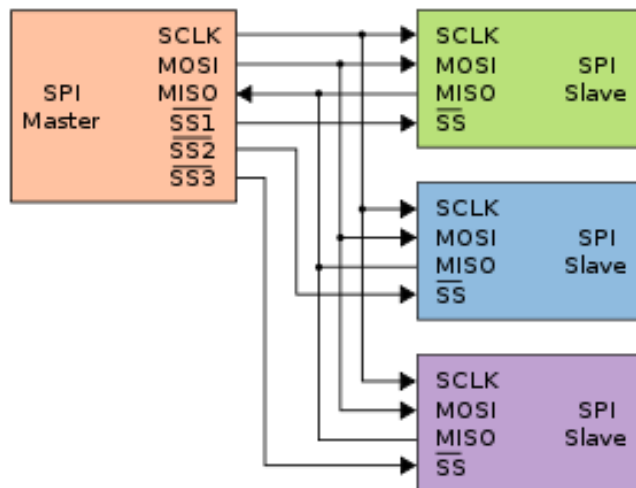
Le bus SPI repose sur quatre lignes principales :

- **MOSI (Master Out Slave In)** : ligne sur laquelle le maître envoie des données au périphérique esclave.
- **MISO (Master In Slave Out)** : ligne sur laquelle l'esclave transmet les données au maître.
- **SCLK (Serial Clock)** : signal d'horloge généré par le maître pour synchroniser la communication.
- **CS (Chip Select)** : signal utilisé pour sélectionner l'esclave actif dans le cas où plusieurs périphériques sont connectés au même bus SPI.

Grâce à ces lignes, la communication se fait en mode full-duplex, ce qui permet l'envoi et la réception de données simultanément.

Mode de fonctionnement maître-esclave

Le protocole SPI fonctionne sur un principe maître-esclave, où un dispositif maître (généralement un microcontrôleur) contrôle la communication et un ou plusieurs esclaves exécutent les instructions reçues. Le maître génère l'horloge et sélectionne quel esclave doit communiquer via la ligne CS. Ce mode de fonctionnement permet de gérer plusieurs périphériques sur un même bus tout en maintenant une communication rapide et efficace.



Le Raspberry Pi

Rôle dans le projet et interfacement avec les périphériques

Dans ce projet, le Raspberry Pi joue un rôle central en tant que contrôleur principal du simulateur ECG. Il est responsable de la gestion des données ECG, de leur transmission vers un convertisseur numérique-analogique (DAC8551) et de la génération d'un signal analogique réaliste destiné au holter ECG.

Le Raspberry Pi reçoit les données ECG numériques depuis une application Qt exécutée sur un PC. Ces données sont ensuite envoyées au DAC8551 via le bus SPI, qui les convertit en un signal de tension analogique correspondant à un véritable ECG. Grâce à son interface GPIO étendue et à sa compatibilité avec divers protocoles de communication (SPI, I2C, UART), le Raspberry Pi peut interagir facilement avec différents capteurs et modules.

Dans ce projet, il est connecté aux éléments suivants :

- **PC via USB ou réseau** : pour recevoir les fichiers ECG et les instructions de simulation.
- **DAC8551 via SPI** : pour transmettre les échantillons ECG sous forme numérique et générer un signal analogique.
- **Alimentation externe ou USB** : pour garantir une autonomie et une stabilité suffisantes.

L'utilisation du bus SPI permet une communication rapide et fiable entre le Raspberry Pi et le DAC, assurant ainsi une fréquence d'échantillonnage de 360 Hz (une mise à jour toutes les 3 ms), nécessaire pour une reproduction fidèle du signal ECG.

Avantages par rapport à l'utilisation d'un microcontrôleur dédié

Le choix du Raspberry Pi plutôt qu'un microcontrôleur dédié (comme un Arduino ou un STM32) repose sur plusieurs avantages clés :

- **Puissance de calcul et flexibilité**
Le Raspberry Pi dispose d'un processeur plus puissant que la plupart des microcontrôleurs, permettant une gestion fluide des fichiers ECG et une communication efficace avec le PC via un système d'exploitation (Linux). Un microcontrôleur aurait nécessité une gestion plus complexe de la mémoire et du traitement des données.

- **Facilité d'intégration logicielle**

Le Raspberry Pi peut exécuter des scripts en Python ou des applications Qt en C++, facilitant le développement et l'interface utilisateur. Avec un microcontrôleur, il aurait fallu écrire du code bas niveau en C et gérer manuellement les interruptions et les buffers SPI.

- **Compatibilité avec des outils de développement avancés**

Contrairement aux microcontrôleurs, le Raspberry Pi supporte un environnement de développement complet, avec accès à des bibliothèques et frameworks modernes comme Qt, spidev (SPI en Python), et des outils de gestion de version comme Git/Framagit.

- **Connectivité et mise à jour logicielle simplifiées**

Grâce à son système Linux embarqué, le Raspberry Pi peut recevoir des mises à jour à distance, enregistrer des logs et exécuter des scripts automatisés, alors qu'un microcontrôleur nécessiterait un flash manuel du firmware à chaque mise à jour.

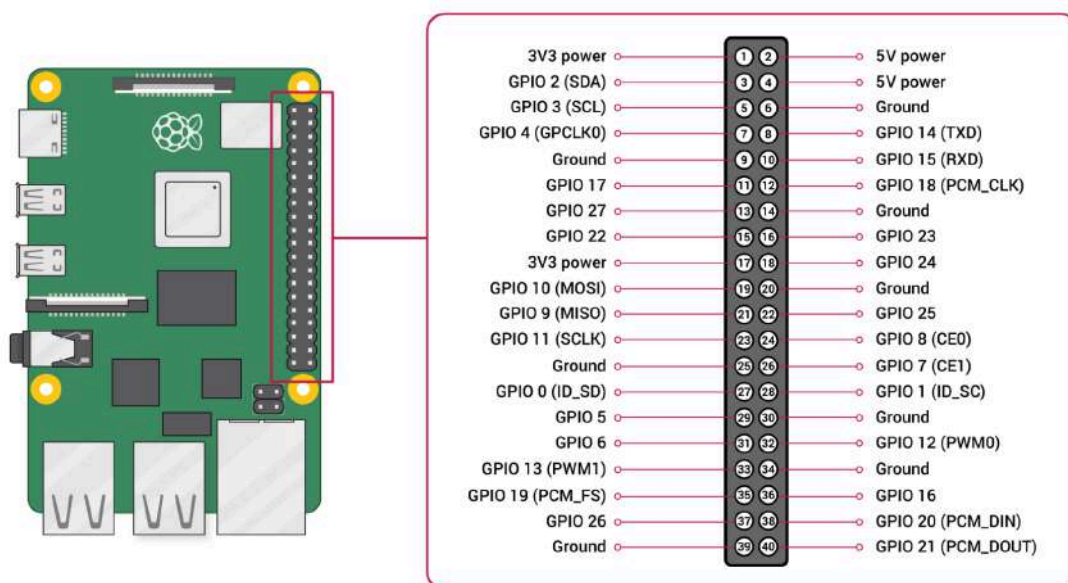
- **Meilleure gestion des fichiers ECG**

Le stockage et la gestion des fichiers ECG sont plus pratiques sur un Raspberry Pi, qui peut utiliser une carte microSD ou un disque externe, contrairement à un microcontrôleur qui dispose d'une mémoire interne limitée et qui nécessiterait des solutions supplémentaires pour stocker de grands ensembles de données.

Inconvénients par rapport à un microcontrôleur

- Consommation énergétique plus élevée.
- Délai de démarrage plus long (quelques secondes contre quelques millisecondes pour un microcontrôleur).
- Plus complexe à intégrer dans un circuit électronique compact.

En résumé, le Raspberry Pi a été retenu pour sa puissance de calcul, sa compatibilité avec des outils logiciels avancés et sa flexibilité pour interfacer facilement les différents éléments du simulateur ECG.



Le DAC8551

Convertisseur numérique-analogique 16 bits

Le DAC8551 est un convertisseur numérique-analogique (CNA) de 16 bits conçu pour convertir un signal numérique en une tension analogique avec une grande précision. Ce composant est particulièrement adapté aux applications nécessitant une conversion de haute résolution et une faible distorsion du signal, ce qui en fait un choix idéal pour la simulation de signaux ECG.

Avec une résolution de 16 bits, il peut générer 65 536 niveaux de tension distincts, permettant ainsi une restitution très fine des variations du signal ECG. Son architecture garantit une sortie analogique stable et sans fluctuations indésirables (glitch ultra-faible), ce qui est essentiel pour la fidélité du signal simulé.

Fonctionnement et caractéristiques principales

Le DAC8551 fonctionne selon le principe de la conversion numérique-analogique : il reçoit une valeur numérique en entrée et la convertit en une tension de sortie proportionnelle.

Principales caractéristiques du DAC8551 :

- **Résolution** : 16 bits (65 536 niveaux de tension)

- **Précision élevée** : Faible erreur de non-linéarité (INL) pour minimiser les distorsions
- **Faible bruit et dérive thermique** : Garantit la stabilité du signal analogique sur le long terme
- **Tension de sortie rail-to-rail** : Peut générer une tension comprise entre 0V et la tension d'alimentation
- **Interface de communication SPI** : Permet un contrôle rapide et précis du convertisseur
- **Temps de conversion rapide** : Essentiel pour atteindre la fréquence de 360 Hz requise pour la simulation ECG

Interface SPI pour recevoir les données numériques et générer une tension analogique

Le DAC8551 communique via le bus SPI (Serial Peripheral Interface), un protocole de transmission synchrone permettant un échange rapide et fiable des données entre le Raspberry Pi et le convertisseur.

Schéma de fonctionnement du SPI avec le DAC8551 :

1. **Le Raspberry Pi envoie des données numériques via SPI** en configurant les broches MOSI (Master Out, Slave In) et SCLK (Serial Clock).
2. **Le DAC8551 reçoit ces données et les interprète** pour ajuster sa sortie analogique en conséquence.
3. **La tension de sortie analogique évolue en fonction des données transmises**, reproduisant ainsi fidèlement le signal ECG préenregistré.

Le choix du SPI permet d'atteindre la fréquence d'échantillonnage nécessaire (360 Hz, soit une mise à jour toutes les 3 ms), garantissant une transition fluide entre les points du signal ECG.

Utilisation pour la génération de signaux ECG précis

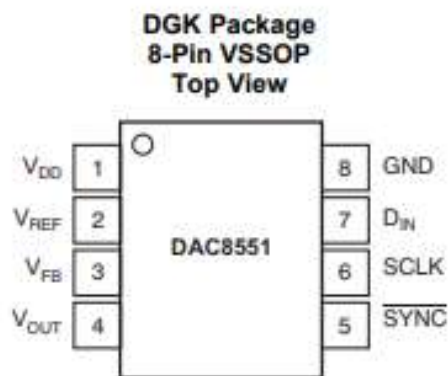
L'objectif principal du DAC8551 dans ce projet est de convertir les relevés ECG numériques en signaux analogiques exploitables par un holter ECG.

Processus de génération du signal ECG :

1. **Lecture des relevés ECG numériques** stockés sur le Raspberry Pi.

2. **Transmission des échantillons via SPI** au DAC8551.
3. **Conversion en une tension analogique correspondante**, formant une onde ECG réaliste.
4. **Sortie du signal vers le holter ECG**, permettant son enregistrement et son analyse comme s'il provenait d'un patient réel.

Grâce à sa précision élevée et sa faible latence, le DAC8551 permet de reproduire fidèlement les variations fines du signal ECG, garantissant une simulation fiable et représentative d'un véritable électrocardiogramme.



Pin Functions

PIN		TYPE	DESCRIPTION
NAME	NO.		
D _{IN}	7	I	Serial data input. Data is clocked into the 24-bit input shift register on each falling edge of the serial clock input. Schmitt-Trigger logic input.
GND	8	GND	Ground reference point for all circuitry on the part
SCLK	6	I	Serial clock input. Data can be transferred at rates up to 30-MHz Schmitt-Trigger logic input.
$\overline{\text{SYNC}}$	5	I	Level-triggered control input (active LOW). This is the frame synchronization signal for the input data. When $\overline{\text{SYNC}}$ goes LOW, it enables the input shift register and data is transferred in on the falling edges of the following clocks. The DAC is updated following the 24th clock (unless $\overline{\text{SYNC}}$ is taken HIGH before this edge, in which case the rising edge of $\overline{\text{SYNC}}$ acts as an interrupt and the write sequence is ignored by the DAC8551). Schmitt-Trigger logic input.
V _{DD}	1	PWR	Power supply input, 2.7 V to 5.5 V
V _{FB}	3	I	Feedback connection for the output amplifier. For voltage output operation, tie to V _{OUT} externally.
V _{OUT}	4	O	Analog output voltage from DAC. The output amplifier has rail-to-rail operation.
V _{REF}	2	I	Reference voltage input

Installation et Configuration

Mise en place du framework Qt/C++ et IDE QtCreator

Dans cette section, nous détaillons les étapes nécessaires pour configurer l'environnement de développement à l'aide du framework Qt et de l'IDE QtCreator, afin de pouvoir développer une application C++ capable de communiquer avec le module FT232H.

1. Installation de Qt et QtCreator

- Qt est un framework puissant permettant le développement d'applications graphiques et non graphiques. Il fournit une bibliothèque riche pour manipuler des interfaces utilisateurs et interagir avec du matériel, comme c'est le cas pour l'utilisation du FT232H.
- QtCreator, quant à lui, est un environnement de développement intégré (IDE) dédié au développement avec Qt. Il facilite la gestion de projets, la compilation et le débogage des applications.

2. L'installation de Qt et de QtCreator se fait généralement à travers les étapes suivantes :

- **Téléchargement de Qt** : Se rendre sur le site officiel de **Qt** <https://www.qt.io/download> et télécharger la dernière version stable compatible avec le système d'exploitation utilisé.
- **Installation de QtCreator** : Une fois Qt téléchargé, il est nécessaire d'installer QtCreator, un IDE spécialement conçu pour travailler avec Qt. Il peut être téléchargé à partir du même site. L'installation peut aussi se faire via les gestionnaires de paquets sous Linux, par exemple :

- **Ubuntu** : `sudo apt install qtcreator qt5-qmake qtbase5-dev`

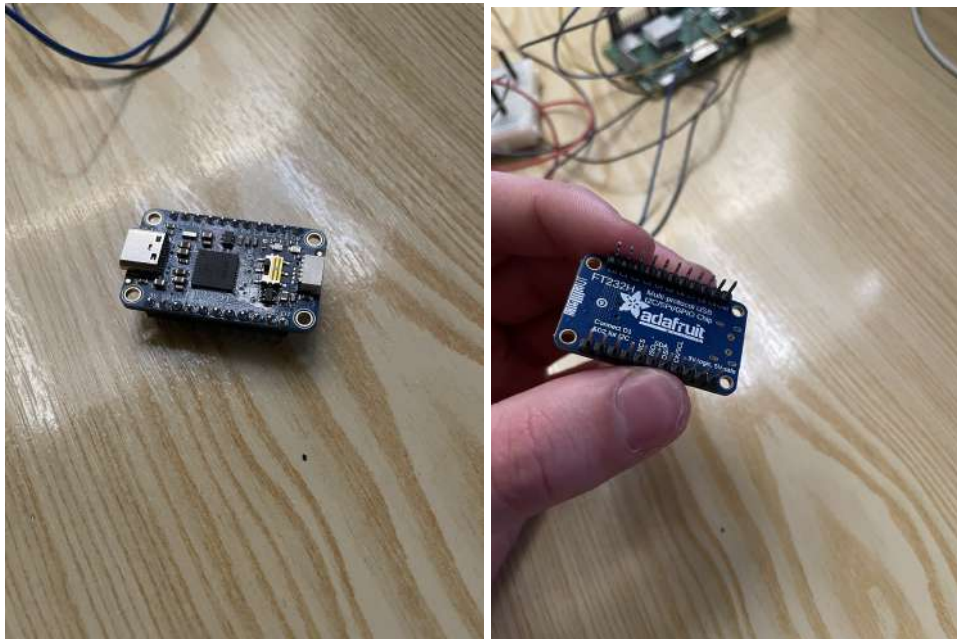
- **Fedora** : `sudo dnf install qtcreator qt5-qtbase-devel`

- **Configuration de QtCreator** : Une fois installé, QtCreator doit être configuré avec le kit de développement approprié (comme le compilateur C++ utilisé). L'IDE doit également être configuré pour utiliser les bibliothèques Qt installées.

3. Configuration des outils de développement sous QtCreator

- Il convient ensuite de configurer QtCreator pour qu'il puisse compiler des projets en C++ en utilisant les outils et bibliothèques Qt. Cela peut inclure la configuration des kits spécifiques pour l'architecture cible ou l'environnement de développement.

Problème avec le FT232H et Solution Adoptée



Le module FT232H, conçu par FTDI, permet de créer une interface USB vers divers protocoles comme GPIO, SPI et I2C.

Explication de l'approche avec le FT232H

L'objectif initial du projet était de convertir une interface USB en SPI à l'aide du module FT232H de FTDI, dans le but de piloter un DAC (Digital-to-Analog Converter). L'idée de départ était d'utiliser ce module comme interface USB pour communiquer avec un DAC via le protocole SPI. Cependant, lors de la mise en œuvre, un certain nombre de limitations ont été identifiées, qui ont rendu cette approche peu viable.

Limitations de la librairie FTDI (latence bloquée à 15 ms)

Le principal problème rencontré avec l'utilisation du FT232H a été la latence associée à la communication via l'interface USB. La librairie D2XX de FTDI, qui permet de contrôler le FT232H, impose une latence minimale d'environ 15 ms entre chaque opération de communication. Cette latence s'avère problématique pour des applications où des fréquences de communication plus élevées sont nécessaires.

Dans le cas de ce projet, où l'objectif était de générer une fréquence de 360 Hz pour piloter le DAC, chaque cycle de communication devait être réalisé en moins de 3 ms. Cela représente un écart trop important, étant donné que la latence minimale imposée par la librairie FTDI est déjà de 15 ms. Par conséquent, il devenait impossible d'atteindre la fréquence de communication requise.

Besoin d'une fréquence de 360 Hz (3 ms) non atteignable

Pour obtenir une fréquence de 360 Hz, il fallait que chaque cycle de communication (transfert de données via SPI) se réalise en moins de 3 ms. Malheureusement, la latence imposée par le FT232H, qui est d'environ 15 ms, rendait cette fréquence inatteignable. Le FT232H n'était donc pas adapté pour ce type de tâche en raison de la latence trop élevée pour une communication rapide et continue à la fréquence requise.

Face à cette limitation, la solution adoptée a été de supprimer le FT232H et de connecter directement le Raspberry Pi au DAC via le bus SPI natif du Raspberry Pi. Cette approche a permis de contourner les problèmes liés à la latence du FT232H et d'atteindre la performance requise pour le pilotage du DAC.

Suppression du FT232H et connexion directe du Raspberry Pi au DAC

Le Raspberry Pi est équipé de ports GPIO (General Purpose Input/Output), et il inclut également des interfaces de communication comme SPI et I2C. Le port SPI du Raspberry Pi a été utilisé pour connecter le Raspberry Pi directement au DAC, en évitant ainsi l'interface USB et le module FT232H.

Dans cette nouvelle approche, le Raspberry Pi a pris en charge directement la gestion du bus SPI, ce qui a permis de réduire considérablement la latence et d'obtenir des cycles de communication plus rapides, adaptés aux besoins du projet.

Avantages de cette approche

La connexion directe du Raspberry Pi au DAC présente plusieurs avantages :

- **Réduction de la latence** : En utilisant le bus SPI natif du Raspberry Pi, la latence est beaucoup plus faible que celle imposée par le FT232H. Cela permet de réduire le temps de réponse pour chaque cycle de communication, ce qui est crucial pour atteindre des fréquences élevées comme 360 Hz.
- **Contrôle direct** : Le Raspberry Pi dispose d'une meilleure gestion du temps réel via son bus SPI intégré, offrant ainsi un contrôle plus précis des signaux envoyés au DAC.
- **Performance accrue** : Grâce à cette connexion directe, la vitesse de communication entre le Raspberry Pi et le DAC a été significativement améliorée, permettant d'atteindre des fréquences de communication bien supérieures à celles possibles avec le FT232H.
- **Simplicité d'implémentation** : L'implémentation avec le Raspberry Pi est plus simple et plus directe, sans nécessiter de modules ou de pilotes supplémentaires comme c'était le cas avec le FT232H.

Performances et résultats obtenus

En passant au Raspberry Pi pour la gestion du bus SPI, il a été possible de communiquer avec le DAC à une fréquence de 360 Hz, bien en dessous de la latence de 15 ms imposée par le FT232H. Cette solution a permis de répondre aux exigences du projet en matière de fréquence de communication et de performances.

Les résultats obtenus avec cette approche ont été très satisfaisants, car le Raspberry Pi a permis de maintenir une communication stable et rapide avec le DAC, ce qui était indispensable pour le bon fonctionnement du système.

Conclusion

Le problème majeur rencontré avec le FT232H était la latence trop élevée imposée par la librairie FTDI, ce qui rendait impossible l'atteinte de la fréquence de communication nécessaire pour piloter le DAC. En optant pour une solution alternative consistant à connecter directement le Raspberry Pi au DAC via son interface SPI, il a été possible de surmonter cette limitation, d'atteindre la fréquence de 360 Hz requise et de bénéficier d'une solution plus simple, plus rapide et plus performante.

Mise en œuvre

Explication du Code et Schéma d'Architecture

Le système se compose d'une interface utilisateur ergonomique, d'un moteur de traitement asynchrone, et d'une couche de communication bas-niveau pour piloter le DAC à 360 Hz.

Architecture et Répartition des Composants

Le projet est organisé autour de **trois modules principaux** :

1. Interface Utilisateur – MainUI

L'interface est conçue avec Qt Designer et codée dans les fichiers `mainui.ui`, `mainui.cpp` et `mainui.h`. C'est ici que l'utilisateur peut :

- Ouvrir un fichier contenant des valeurs ECG
- Démarrer ou stopper la simulation
- Suivre l'état du signal (via un graphique `QChartView`)

L'interface envoie des signaux Qt vers la couche de traitement dès qu'une action est effectuée (par exemple `simuToggled(true)` quand on lance la simulation).

2. Contrôleur Applicatif – App

Le rôle de la classe `App` (fichier `app.h`, `app.cpp`) est de faire le lien entre l'interface et le traitement en arrière-plan. Elle :

- Instancie et gère l'objet `MainUI`

- Crée un thread de travail (**QThread**)
- Y affecte l'objet **AppWorker** pour traiter les données
- Connecte les signaux/slots entre tous les éléments

C'est donc le cerveau du projet, qui orchestre les différents modules.

3. Traitement en Arrière-plan – **AppWorker**

Le fichier **appworker.cpp** contient toute la logique métier qui va :

- Lire un fichier ECG ligne par ligne (données entières)
- Stocker ces valeurs dans un **QVector<int>**
- Piloter le DAC8551 via le bus SPI
- Générer un signal ECG à la bonne fréquence (360 Hz = 1 point toutes les 2,77 ms)

L'exécution de ce travail est faite dans un thread indépendant, pour ne pas bloquer l'interface utilisateur.

QThread::msleep(2); // Attente 2 ms entre chaque point

L'objet **AppWorker** écoute aussi les ordres de type **suspend()** ou **quit()** pour gérer l'état de la simulation.

Communication SPI – **SpiDev**

Pour piloter le convertisseur DAC8551, j'ai écrit une couche d'abstraction SPI dans les fichiers **spidev.cpp** / **spidev.h**. Cette classe utilise les appels système Linux (**ioctl**, **open**, **read**, **write**) pour envoyer des trames SPI formatées.

Le DAC reçoit des paquets de 3 octets :

- Le premier est toujours **0x00** (contrôle)

- Les deux suivants correspondent à la valeur ECG sur 16 bits

Le périphérique SPI utilisé est `/dev/spidev0.0` du Raspberry Pi.

Cycle de Fonctionnement

Voici un résumé du fonctionnement complet du simulateur :

1. L'utilisateur ouvre un fichier `.csv` contenant les valeurs ECG
 2. Le contrôleur (`App`) le transfère au module `AppWorker`, qui lit et stocke les données
 3. En cliquant sur "Lancer la simulation", un signal déclenche le traitement
 4. Le thread entre dans une boucle infinie et écrit chaque point vers le DAC toutes les 2 ms
 5. À tout moment, on peut stopper ou quitter, ce qui suspend ou termine proprement le thread
-

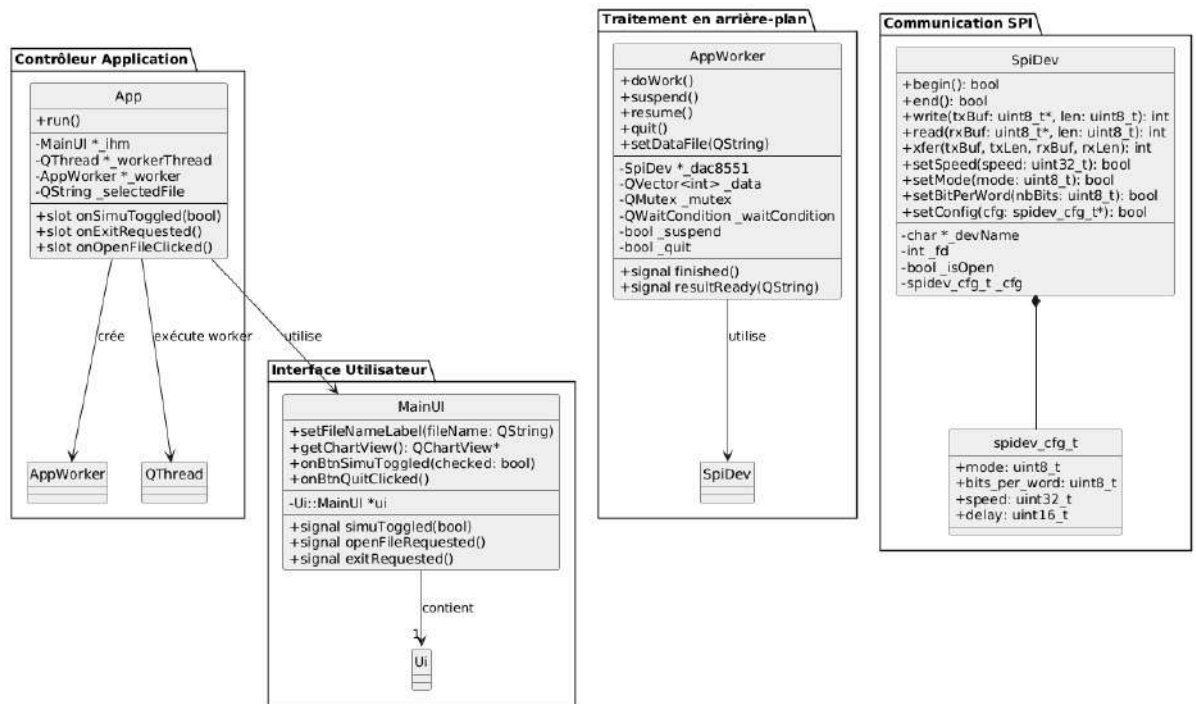
Aspects Techniques Intéressants

- **Multithreading avec Qt** : le traitement est découplé de l'IHM grâce à un `QThread` dédié
 - **Communication SPI bas-niveau** : manipulation directe des registres SPI du Pi
 - **Temps réel approximatif** : 360 Hz respecté grâce à `msleep(2)`
 - **Gestion de fichiers robuste** : prise en charge d'erreurs de lecture, lignes invalides
-

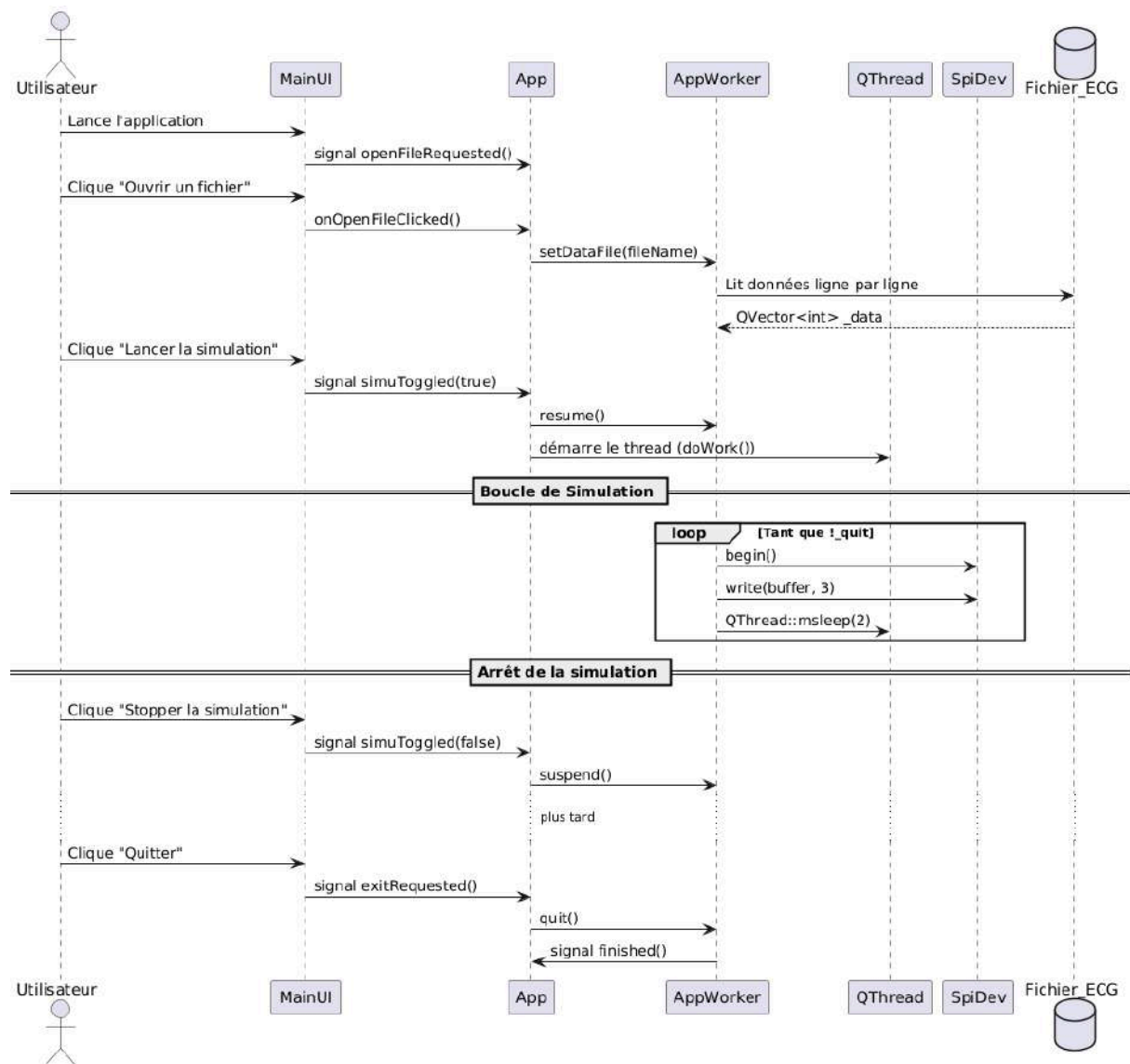
Conclusion

Ce projet m'a permis de combiner plusieurs compétences : développement en C++/Qt, communication matérielle via SPI, gestion multithread, et conception d'interface graphique. J'ai particulièrement apprécié le challenge de rendre le système réactif, fiable et temps-réel sur un petit système embarqué comme le Raspberry Pi.

- Schéma PlantUML du système diagramme de classes



- Schéma PlantUML du Diagramme de Séquence Complet



Préparation des données ECG

Conversion fichiers .mat en .csv:

Conversion des fichiers **.mat** en **.csv** pour l'exploitation des signaux ECG

Dans le cadre de mon projet, les signaux ECG que je souhaitais utiliser provenaient initialement de fichiers au format **.mat** (MATLAB). Afin de pouvoir les exploiter dans mon simulateur ECG, qui attend des fichiers texte simples, j'ai développé un script Python de conversion automatisée.

Ce script a deux objectifs :

1. Extraire les valeurs ECG brutes contenues dans les fichiers **.mat** et les enregistrer au format **.csv**
2. Générer une version mise à l'échelle des signaux, avec des valeurs comprises entre **0** et **65536**, compatibles avec la résolution 16 bits du convertisseur numérique-analogique DAC8551 utilisé dans mon simulateur

Concrètement, le script commence par me demander le chemin du dossier source contenant les fichiers **.mat**, ainsi que le dossier de destination pour les fichiers **.csv**. Il recrée ensuite toute l'arborescence des sous-dossiers si nécessaire. Pour chaque fichier **.mat**, il effectue les opérations suivantes :

- **Chargement des données ECG** via la bibliothèque **scipy.io**
- **Sauvegarde des valeurs originales** dans un fichier **.csv** brut
- **Recherche des valeurs min et max** dans le signal, avec une petite marge (5 %) pour éviter les saturations
- **Mise à l'échelle linéaire** des valeurs dans l'intervalle **[0, 65536]**, grâce à une fonction personnalisée (**map_range**)
- **Écriture d'un second fichier .csv** contenant ces valeurs normalisées, prêt à être injecté dans le DAC

L'approche est générique et automatisée, ce qui me permet de convertir rapidement une grande quantité de signaux ECG provenant de bases de données médicales, tout en les adaptant précisément au format attendu par le matériel.

Les fichiers proviennent d'un dossier : ECG signals (1000 fragments).zip

Description du dossier :

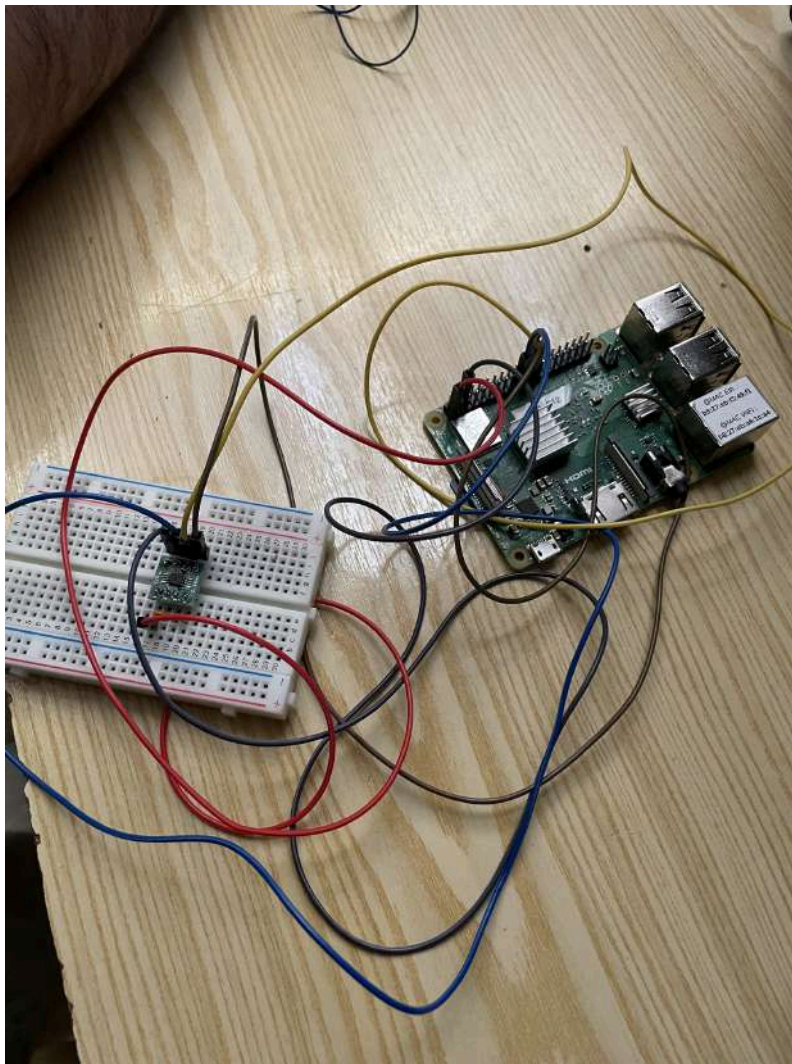
For research purposes, the ECG signals were obtained from the PhysioNet service (<http://www.physionet.org>) from the MIT-BIH Arrhythmia database. The created database with ECG signals is described below. 1) The ECG signals were from 45 patients: 19 female (age: 23-89) and 26 male (age: 32-89). 2) The ECG signals contained 17 classes: normal sinus rhythm, pacemaker rhythm, and 15 types of cardiac dysfunctions (for each of which at least 10 signal fragments were collected). 3) All ECG signals were recorded at a sampling frequency of 360 [Hz] and a gain of 200 [adu / mV]. 4) For the analysis, 1000, 10-second (3600 samples) fragments of the ECG signal (not overlapping) were randomly selected. 5) Only signals derived from one lead, the MLII, were used. 6) Data are in mat format (Matlab).

Tests et intégration matérielle

Au départ, nos branchements étaient réalisés manuellement : le Raspberry Pi était relié au DAC8551 via des fils Dupont, connectés directement aux broches SPI de la carte. Cette méthode fonctionnait, mais elle restait relativement fragile et peu pratique pour un usage répété ou pédagogique.

Par la suite, Nino (étudiant ER) a conçu une carte d'adaptation dédiée, bien plus simple et robuste. Cette carte vient se brancher directement sur les broches GPIO du Raspberry Pi, ce qui facilite grandement le câblage et limite les erreurs de connexion. Grâce à cette amélioration, le simulateur est devenu plus fiable, plus compact, et mieux adapté aux démonstrations ou aux manipulations.

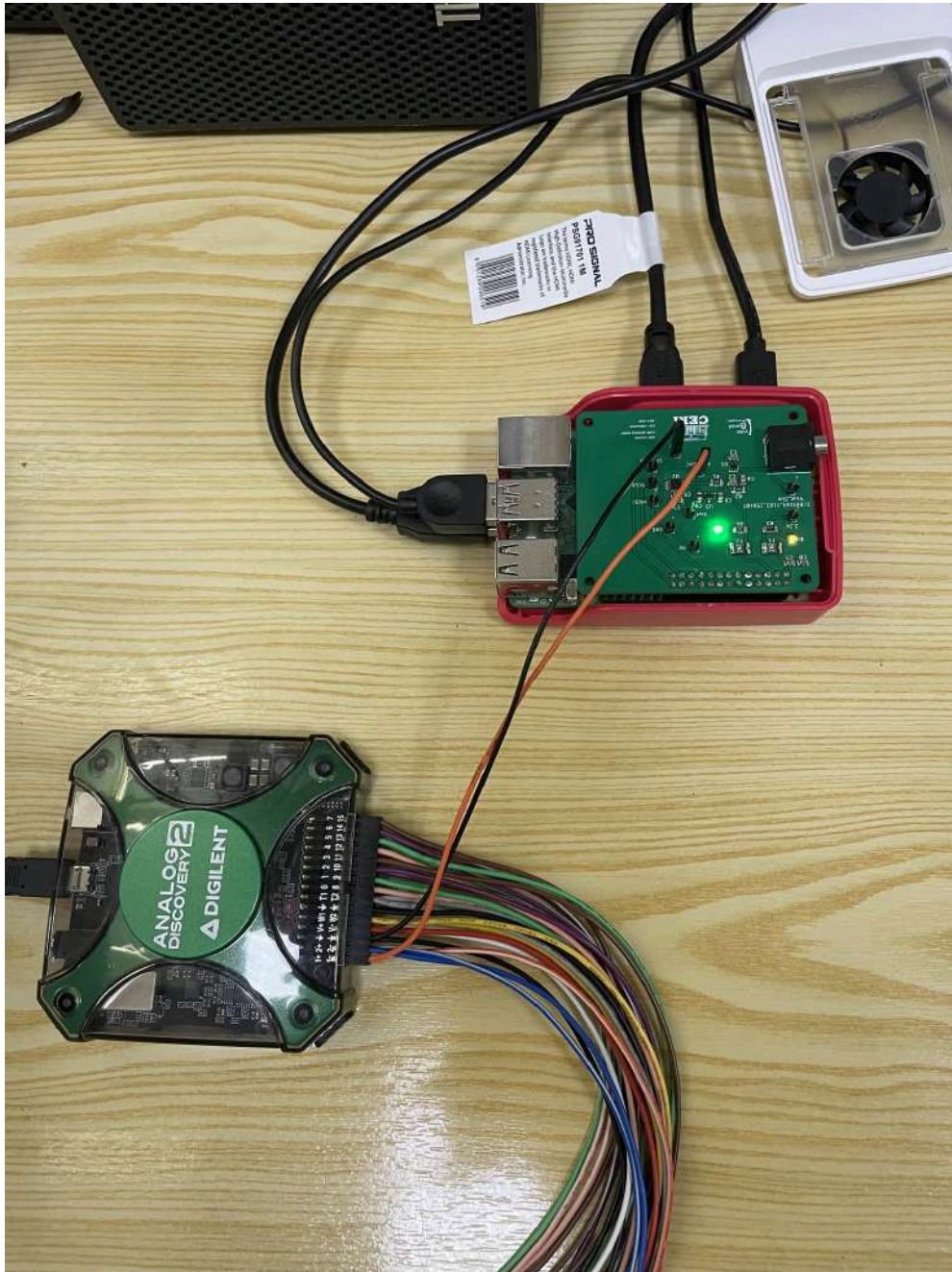
Ici juste le simulateur avant la carte de Nino:



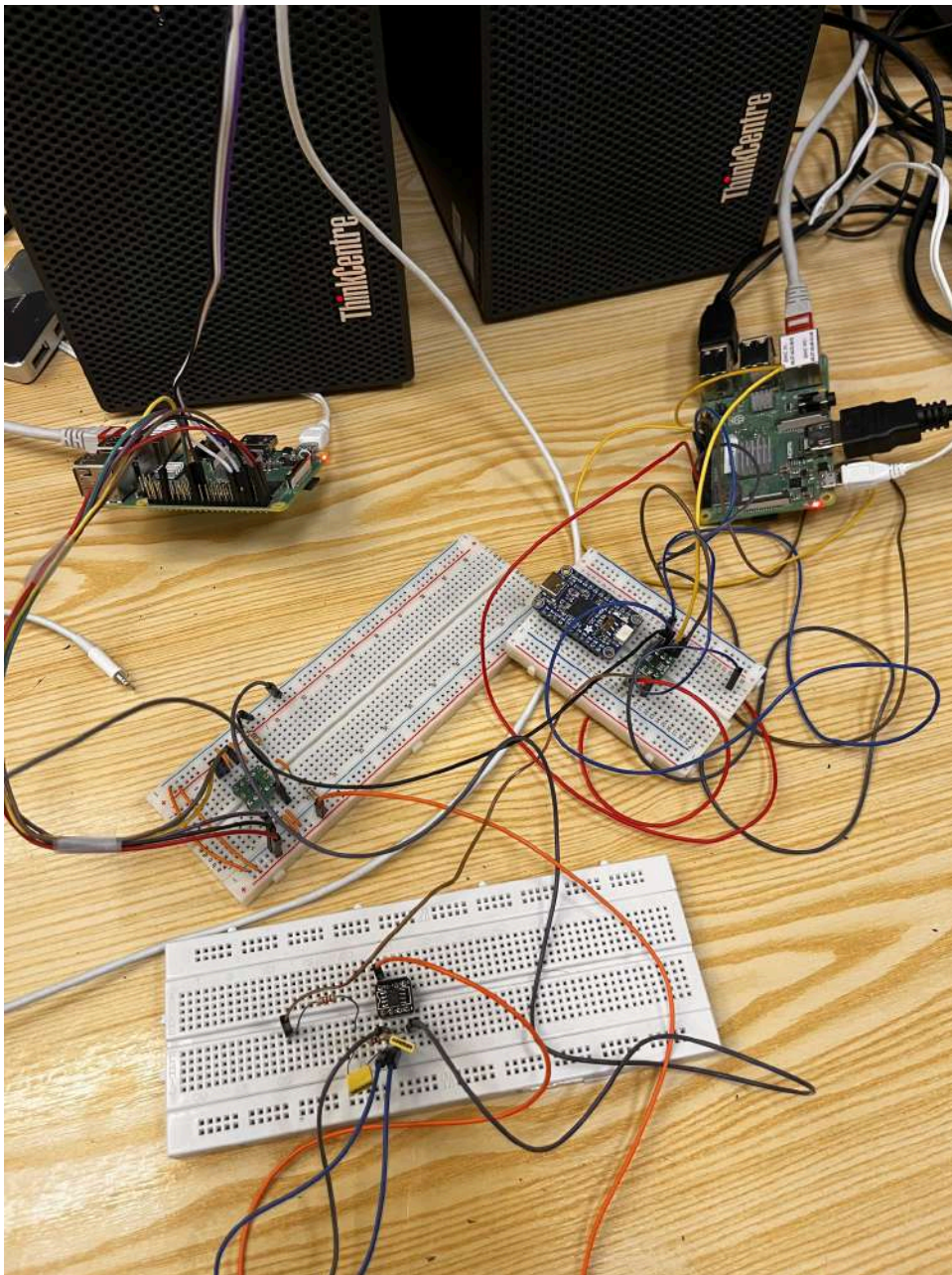
Ici juste le simulateur avec la carte de Nino et filtre intégré:



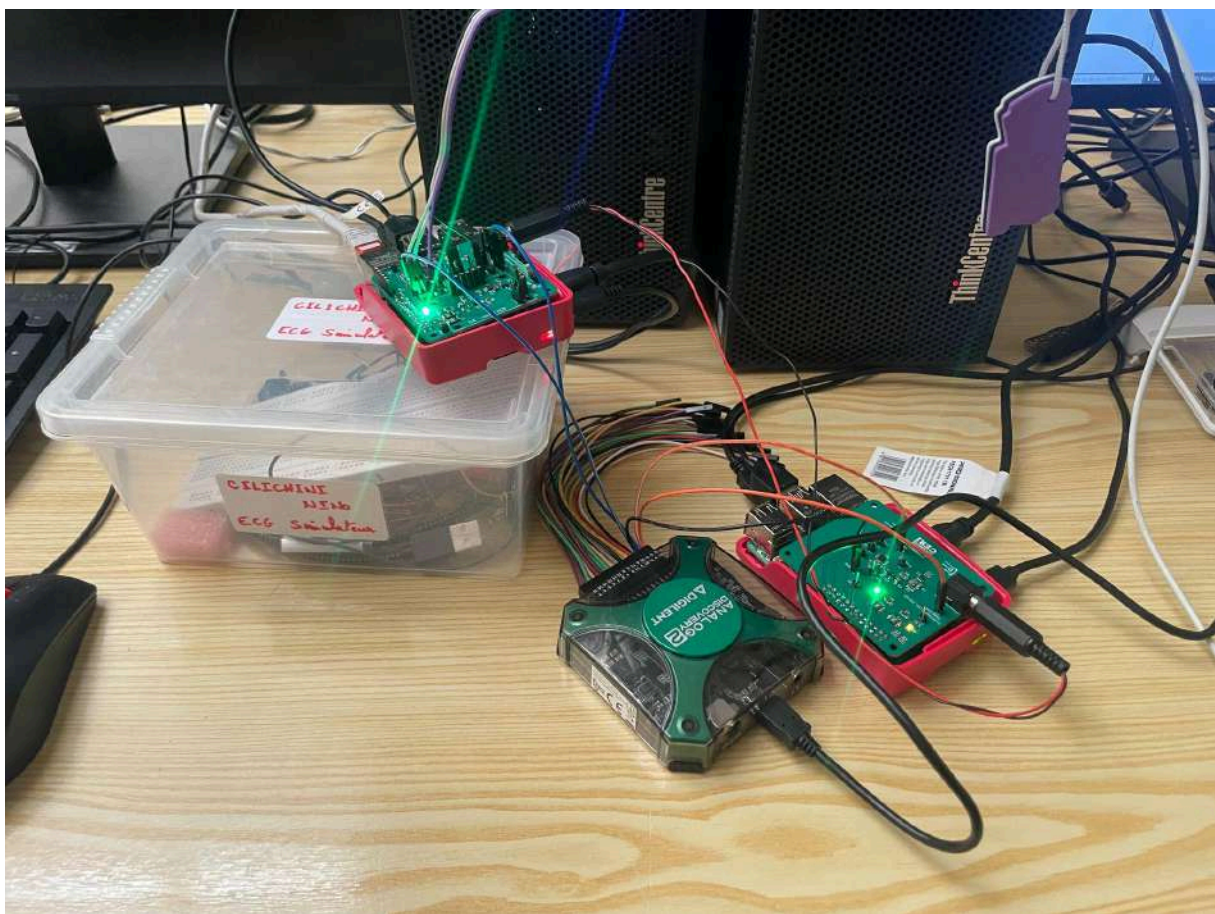
Voici le simulateur connecté à l'Analog Discovery, utilisé pour vérifier les signaux à l'aide d'un oscilloscope :



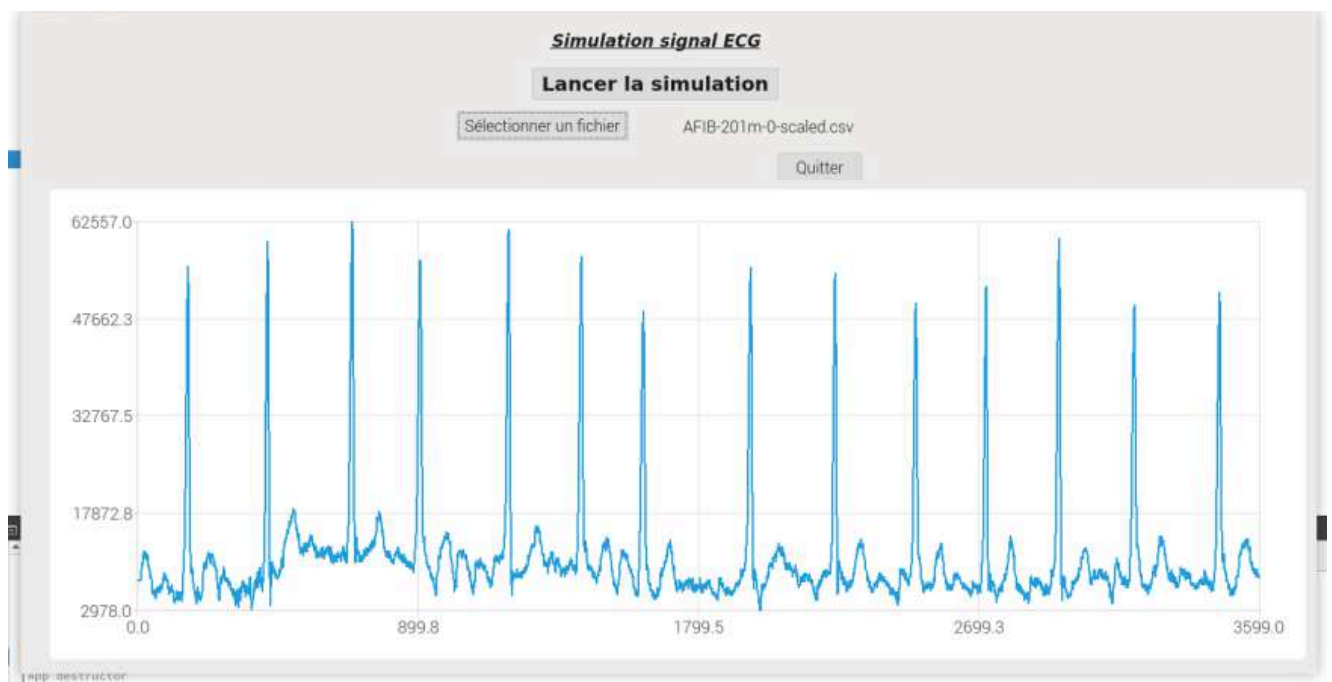
Voici le montage complet du projet avec le Holter mais sans les cartes développées par les étudiants ER.:



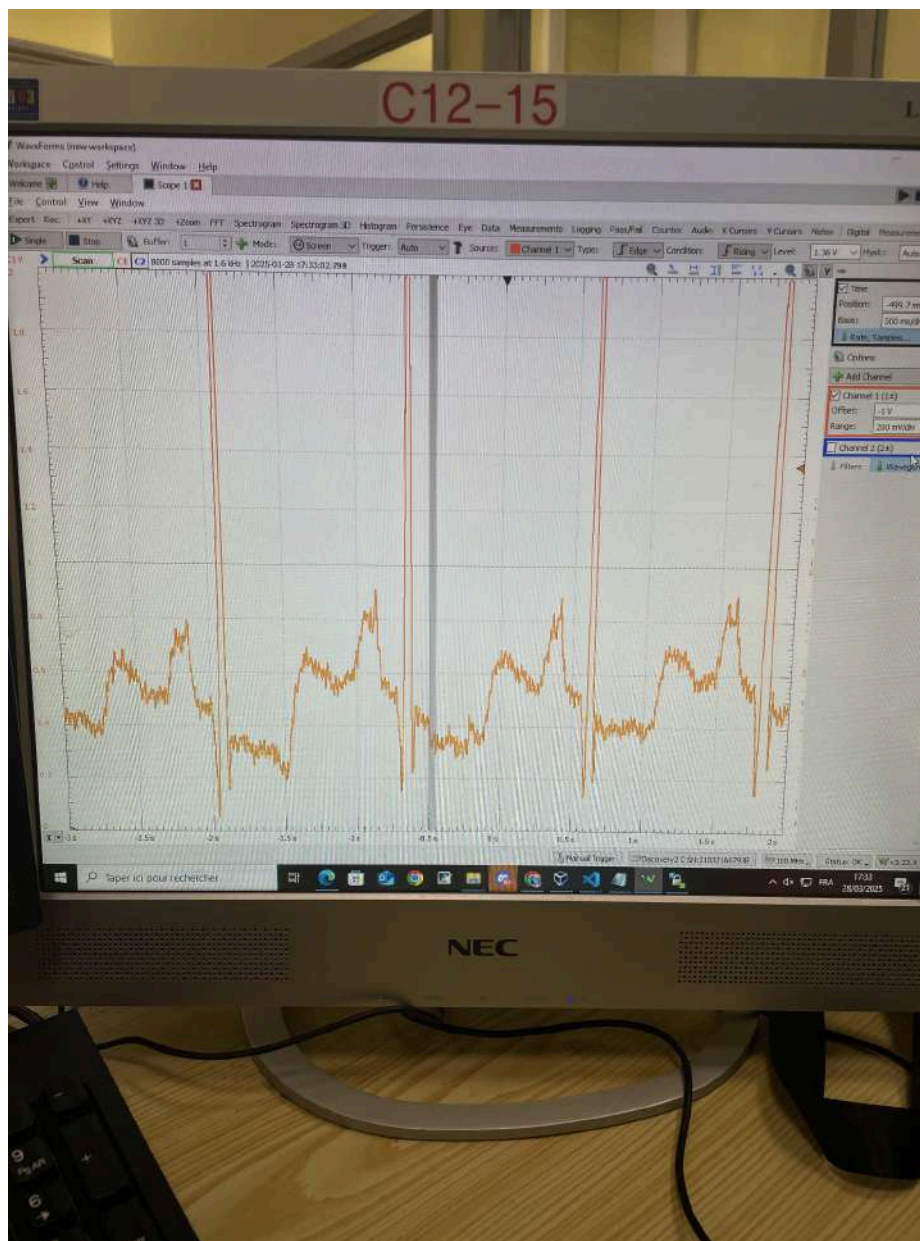
Voici le montage complet du projet, comprenant le Holter et les cartes développées par les étudiants ER.



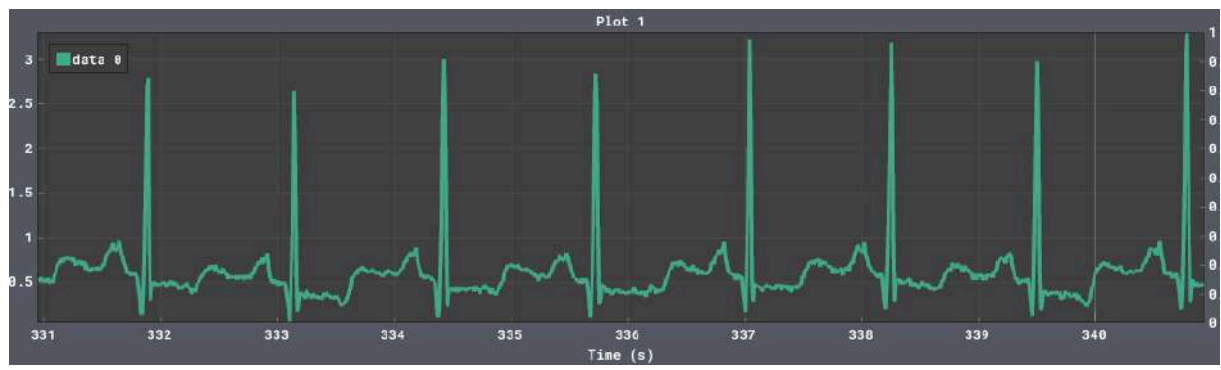
Voici l'interface utilisateur du simulateur ECG, avec un aperçu de la forme du signal qui va être généré:



Ci-dessous, le signal généré observé à l'oscilloscope (**attention : il ne s'agit pas du même signal que celui affiché dans l'interface précédente**) :



Et pour finir, voici le signal récupéré par le Holter :



Conclusion générale

Ce projet de simulateur ECG m'a permis de mettre en œuvre une solution complète mêlant à la fois développement logiciel, intégration matérielle et traitement de données biomédicales. Réalisé pour le CERI de l'Université d'Avignon, il s'inscrit dans une démarche concrète visant à accompagner la recherche et l'enseignement dans le domaine de la santé connectée.

Tout au long de ce projet, j'ai été confronté à des problématiques réelles : choix du matériel, limitations techniques (notamment liées au FT232H), communication SPI bas-niveau, gestion de fichiers médicaux complexes, et respect de contraintes de temps réel. Ces défis m'ont amené à faire des choix techniques justifiés, comme le remplacement du module USB-SPI par une communication directe via le bus SPI du Raspberry Pi, bien plus performant et stable.

Le développement de l'application Qt, l'organisation logicielle en thread, ainsi que la conversion automatique des fichiers `.mat` en `.csv` ont renforcé la robustesse et l'autonomie du simulateur. J'ai également eu l'opportunité de collaborer avec d'autres étudiants sur la conception de cartes d'adaptation matérielle, ce qui a contribué à fiabiliser l'ensemble du montage.

Ce projet m'a offert une expérience concrète et complète de l'intégration logicielle et matérielle dans un contexte biomédical. Il m'a permis de renforcer mes compétences en C++/Qt, en traitement du signal, en communication SPI, ainsi qu'en gestion de projet technique.

En définitive, le simulateur ECG développé constitue un outil fiable, pédagogique et évolutif, pouvant être utilisé à la fois pour des tests de dispositifs médicaux et pour la formation d'étudiants aux systèmes embarqués appliqués à la santé.

Annexes simulateur

Fiche d'audit de sécurité du simulateur:

Fiche d'audit de sécurité.		Réf. FASECG25
Projet	Auteur	Date
ECG	AMADORI Johann	28/05/2025
Partie de projet		
Simulateur ECG		
Désignation	Risque	Solution
Fichiers CSV	Fichier infecté ou malformé, plantage logiciel	Valider le contenu avant traitement, limiter le poids, filtrer les entrées
Accès physique	Vol, sabotage, branchement d'un périphérique USB malveillant	Boîtier verrouillé, démarrage protégé par mot de passe, surveillance physique
Port SPI	Manipulation électrique, bruit, injection de données	Vérification CRC, protection électrique, lignes dédiées
Interface Qt (GUI)	Mauvaise gestion des erreurs utilisateur, crash	Validation des entrées, messages d'erreur clairs, gestion des exceptions
Synthèse		
Le simulateur ECG est exposé à des risques réseau, physiques et logiciels. Il est essentiel de contrôler l'accès au système, de valider toutes les entrées (CSV, utilisateur), et de sécuriser les communications SPI. Des contre-mesures adaptées comme le chiffrement, la validation côté logiciel et des protections matérielles doivent être mises en place.		

Code python pour conversion .mat vers .csv:

```
import scipy.io
import numpy as np
import os, glob
import re

# Fonction de mise à l'échelle de valeurs
def map_range(x, in_min, in_max, out_min, out_max):
    return (x - in_min) * (out_max - out_min) // (in_max - in_min) + out_min

if __name__ == "__main__":
    print("Saisir le chemin du dossier contenant les fichiers .mat à convertir en .csv :")
    inputDir = input()
    print("Saisir le chemin du dossier de destination des fichiers .csv :")
    outputDir = input()

    # Création du dossier de destination si inexistant
    if not os.path.exists(outputDir):
        os.makedirs(outputDir)

    srcSubfolders = [name for name in os.listdir(inputDir) if os.path.isdir(os.path.join(inputDir, name))]
    destSubfolders = [name.split()[1] for name in os.listdir(inputDir) if
os.path.isdir(os.path.join(inputDir, name))]
    subfolders = list(zip(srcSubfolders, destSubfolders))
```

```

# Création de l'arborescence des sous-dossiers dans le dossier de destination
for subfolder in destSubfolders:

    subfolderPath = os.path.join(outputDir, subfolder)
    if not os.path.exists(subfolderPath):
        os.makedirs(subfolderPath)


# Parcours des sous-dossiers
patternSrcFile = r'(\d+m) \((\d+)\)\.mat'
for srcSubfolder, destSubfolder in subfolders:

    srcSubfolderPath = os.path.join(inputDir, srcSubfolder)
    destSubfolderPath = os.path.join(outputDir, destSubfolder)


    # Parcours des fichiers .mat dans le sous-dossier puis conversion en .csv et sauvegarde dans le
    # sous-dossier de destination
    for filename in glob.glob(os.path.join(srcSubfolderPath, '*.mat')) :
        with open(filename, 'r') as f:

            # Définition des noms des fichiers de destination
            patternDestFile = rf'{destSubfolder}-\1-\2.csv'
            patternScaledDestFile = rf'{destSubfolder}-\1-\2-scaled.csv'

            newFilename = os.path.join(destSubfolderPath, re.sub(patternSrcFile, patternDestFile,
os.path.basename(filename)))

            newScaledFilename = os.path.join(destSubfolderPath, re.sub(patternSrcFile,
patternScaledDestFile, os.path.basename(filename)))

            print(f"newFilename: {newFilename}")


    # Chargement du fichier .mat
    mat = scipy.io.loadmat(filename)

```

```

# Sauvegarde des valeurs brutes en .csv
np.savetxt(newFilename, mat['val'], delimiter="\n", fmt='%d')

# Normalisation des valeurs entre 0 et 65536

# . Recherche des valeurs min et max
minVal = np.min(mat['val'])
maxVal = np.max(mat['val'])
# . Ajout de 5% de marge
margin = 0.05 * (maxVal - minVal)
minVal -= margin
maxVal += margin
# . Mise à l'échelle des valeurs
scaledVal = np.array([map_range(x, minVal, maxVal, 0, 65536) for x in mat['val']])

# Sauvegarde des valeurs mises à l'échelle
np.savetxt(newScaledFilename, scaledVal, delimiter="\n", fmt='%d')

print("Conversion terminée.")

```

Voici le programme complet développé sous QtCreator pour assurer le fonctionnement du simulateur ECG:

app-simulateur.pro:

```
QT += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

QT += charts

CONFIG += c++17

# You can make your code fail to compile if it uses deprecated APIs.
# In order to do so, uncomment the following line.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000 # disables all the APIs deprecated before Qt 6.0.0

SOURCES += \
    app.cpp \
    appworker.cpp \
    main.cpp \
    mainui.cpp \
    spidev.cpp

HEADERS += \
    app.h \
    appworker.h \
    mainui.h \
    spidev.h

FORMS += \
    mainui.ui

# Default rules for deployment.
qnx: target.path = /tmp/${TARGET}/bin
```

```
else: unix:!android: target.path = /opt/${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

app.cpp:

```
// === app.cpp ===

#include <QCoreApplication>
#include <QDebug>
#include <QFileDialog>
#include <QMessageBox>

#include "app.h"

App::App(QObject *parent)
    : QObject{parent}
{
    _ihm = new MainUI();

    connect(_ihm, &MainUI::simuToggled, this, &App::onSimuToggled);
    connect(_ihm, &MainUI::exitRequested, this, &App::onExitRequested);
    connect(_ihm, &MainUI::openFileRequested, this, &App::onOpenFileClicked);

    _worker = new AppWorker();
    _workerThread = new QThread();
    _worker->moveToThread(_workerThread);

    connect(_workerThread, &QThread::started, _worker, &AppWorker::doWork);
    connect(_workerThread, &QThread::finished, [this]() {
        qDebug() << "Thread terminé";
        _workerThread->quit();
    });
    connect(_worker, &AppWorker::finished, [this]() {
        qDebug() << "Worker terminé";
        _workerThread->quit();
    });
    connect(_worker, &AppWorker::finished, _worker, &AppWorker::deleteLater);
    connect(_worker, &AppWorker::resultReady, [](const QString &result) {
        qDebug() << "Résultat reçu:" << result;
    });
}
```

```

App::~App()
{
    qDebug() << "App destructor";
}

void App::run()
{
    _ihm->show();

    if (!_selectedFile.isEmpty()) {
        _worker->setDataFile(_selectedFile);
    }

    _workerThread->start();
}

void App::onSimuToggled(bool checked)
{
    if (checked) {
        if (_selectedFile.isEmpty()) {
            QMessageBox::warning(_ihm, "Erreur", "Aucun fichier sélectionné.");
            _ihm->setFileNameLabel("Aucun fichier");
            return;
        }
        _worker->setDataFile(_selectedFile);
        _worker->resume();
    } else {
        _worker->suspend();
    }
}

void App::onExitRequested()
{
    _worker->quit();
    QApplication::quit();
}

void App::onOpenFileClicked()
{
    QString fileName = QFileDialog::getOpenFileName(
        _ihm,
        tr("Ouvrir un fichier CSV"),
        "/home/pi/devQt/ecg/qt/app-simulateur/ECG signals (1000 fragments)",
        tr("Fichiers CSV (*.csv)")
    );
}

```

```

if (!fileName.isEmpty()) {

    QFile file(fileName);

    if (!file.open(QIODevice::ReadOnly | QIODevice::Text)) {

        QMessageBox::warning(_ihm, "Erreur", "Impossible d'ouvrir le fichier.");

        _ihm->setFileNameLabel("Aucun fichier sélectionné.");

        return;

    }

    if (file.size() == 0) {

        QMessageBox::warning(_ihm, "Erreur", "Le fichier est vide.");

        _ihm->setFileNameLabel("Aucun fichier sélectionné.");

        return;

    }

    file.close();

    _selectedFile = fileName;

    _ihm->setFileNameLabel(QFileInfo(fileName).fileName());

    QLineSeries *series =new QLineSeries;

    int x = 0;

    if (file.open(QIODevice::ReadOnly | QIODevice::Text)) {

        QTextStream in(&file);

        while (!in.atEnd()){

            QString line =in.readLine().trimmed();

            bool ok;

            int value = line.toInt(&ok);

            if (ok) {

                series->append(x++, value);

            }

        }

        file.close();

    }

    QChart *chart =new QChart();

    chart->legend()->hide();

    chart->addSeries(series);

    chart->createDefaultAxes();

    _ihm->getChartView()->setChart(chart);

}

}

```

app.h:

```
// app.h

#ifndef APP_H
#define APP_H

#include <QObject>
#include <QThread>
#include "mainui.h"
#include "appworker.h"

class App : public QObject
{
    Q_OBJECT
public:
    explicit App(QObject *parent = nullptr);
    ~App();
    void run();

private:
    MainUI *_ihm;
    QThread *_workerThread;
    AppWorker *_worker;
    QString _selectedFile;

private slots:
    void onSimuToggled(bool checked);
    void onExitRequested();
    void onOpenFileClicked();
};

#endif // APP_H
```

appworker.cpp:

```
// === appworker.cpp ===

#include <QThread>
#include <QVector>
#include <QFile>
#include <QDebug>

#include "appworker.h"

AppWorker::AppWorker(QObject *parent) : QObject{parent}
{
    _suspend = true;
    _quit = false;

    spiDev_cfg_t spiCfg = {1, 8, 100000, 0};
    _dac8551 = new SpiDev("/dev/spidev0.0", &spiCfg);
}

AppWorker::~AppWorker()
{
    qDebug() << "AppWorker destructor";
    if (_dac8551) delete _dac8551;
}

void AppWorker::suspend()
{
    QMutexLocker locker(&_mutex);
    _suspend = true;
}

void AppWorker::resume()
{
    QMutexLocker locker(&_mutex);
    _suspend = false;
    _waitCondition.wakeAll();
}

void AppWorker::quit()
{
    QMutexLocker locker(&_mutex);
    _quit = true;
}
```

```

        _suspend = false;

        _waitCondition.wakeAll();
    }

void AppWorker::setDataFile(QString fileName)
{
    _data.clear();

    QFile file(fileName);

    if (!file.open(QIODevice::ReadOnly | QIODevice::Text)) {
        qWarning() << "Erreur ouverture fichier:" << fileName;
        return;
    }

    QTextStream stream(&file);

    int lineNumber = 0;

    while (!stream.atEnd()) {
        QString line = stream.readLine().trimmed();

        ++lineNumber;

        if (line.isEmpty()) continue;

        bool ok;
        int value = line.toInt(&ok);

        if (ok) {
            _data.append(value);
        } else {
            qWarning() << "Ligne invalide (" << lineNumber << "):" << line;
            _data.append(-1);
        }
    }

    file.close();

    qDebug() << "Fichier chargé avec" << _data.size() << "valeurs.";
}

void AppWorker::doWork()
{
    uint8_t spiBuffer[3];

    int idx = 0;

    if (_dac8551->begin()) {
        for (;;) {
            QMutexLocker locker(&_mutex);

            if (_quit) break;

```

```

while (!_suspend) _waitCondition.wait(&_mutex);

if (_data.isEmpty()) {
    qWarning() << "Aucune donnée chargée, attente...";
    QThread::msleep(100);
    continue;
}

spiBuffer[0] = 0x00;
spiBuffer[1] = (uint8_t)(_data[idx] >> 8) & 0x00ff);
spiBuffer[2] = (uint8_t)(_data[idx] & 0x00ff);

_dac8551->write(spiBuffer, 3);
QThread::msleep(2);

idx = (idx + 1) % _data.size();
}

emit finished();
} else {
    qWarning() << "Erreur initialisation DAC8551";
}
}
}

```

appworker.h:

```
// === appworker.h ===

#ifndef APPWORKER_H
#define APPWORKER_H

#include <QObject>
#include <QTimer>
#include <QMutex>
#include <QWaitCondition>
#include <QVector>

#include "spidev.h"

class AppWorker : public QObject
{
    Q_OBJECT
public:
    explicit AppWorker(QObject *parent = nullptr);
    ~AppWorker();
    void doWork();
    void suspend();
    void resume();
    void quit();
    void setDataFile(QString fileName);

private:
    SpiDev * _dac8551;
    bool _quit;
    bool _suspend;
    QMutex _mutex;
    QWaitCondition _waitCondition;
    QVector<int> _data;

signals:
    void finished();
    void resultReady(const QString &result);
};

#endif // APPWORKER_H
```

main.cpp:

```
#include <QApplication>

#include "app.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    App app;
    app.run();

    return a.exec();
}
```

mainui.cpp:

```
// === mainui.cpp ===

#include <QMessageBox>
#include <QFileInfo>
#include "mainui.h"
#include "ui_mainui.h"

MainUI::MainUI(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::MainUI)
{
    setWindowFlags(Qt::Window | Qt::FramelessWindowHint);
    ui->setupUi(this);
    ui->fileLabel->setText("Aucun fichier sélectionné.");

    connect(ui->btnSimu, &QPushButton::toggled, this, &MainUI::onBtnSimuToggled);
    connect(ui->btnQuit, &QPushButton::clicked, this, &MainUI::onBtnQuitClicked);
    connect(ui->openFileButton, &QPushButton::clicked, this, &MainUI::openFileRequested);
}

MainUI::~MainUI()
```

```

{
    delete ui;
}

void MainUI::onBtnSimuToggled(bool checked)
{
    if (checked) {
        ui->btnSimu->setText("Stopper la simulation");
        ui->btnQuit->setEnabled(false);
    } else {
        ui->btnSimu->setText("Lancer la simulation");
        ui->btnQuit->setEnabled(true);
    }

    emit simuToggled(checked);
}

void MainUI::onBtnQuitClicked()
{
    emit exitRequested();
}

void MainUI::setFileNameLabel(const QString &fileName)
{
    if (fileName.isEmpty()) {
        ui->fileLabel->setText("Aucun fichier sélectionné.");
    } else {
        ui->fileLabel->setText(QFileInfo(fileName).fileName());
    }
}

QChartView* MainUI::getChartView() const{
    return ui->chartView;
}

```

mainui.h:

```
// === mainui.h ===

#ifndef MAINUI_H
#define MAINUI_H

#include <QDialog>
#include <QCloseEvent>
#include <QtCharts/QChartView>
#include <QtCharts/QLineSeries>
QT_CHARTS_USE_NAMESPACE

namespace Ui {
class MainUI;
}

class MainUI : public QDialog
{
    Q_OBJECT

public:
    explicit MainUI(QWidget *parent = nullptr);
    ~MainUI();
    void setFileNameLabel(const QString &fileName);
    QChartView* getChartView() const;

protected:
    void closeEvent(QCloseEvent *event) override {
        event->ignore();
    }

private:
    Ui::MainUI *ui;

private slots:
    void onBtnSimuToggled(bool checked);
    void onBtnQuitClicked();

signals:
    void simuToggled(bool checked);
    void openFileRequested();
    void exitRequested();
```

```
};
```

```
#endif // MAINUI_H
```

spidev.cpp:

```
#include <QDebug>
```

```
#include <stdio.h>
```

```
#include <stdint.h>
```

```
#include <fcntl.h>
```

```
#include <stdlib.h>
```

```
#include <unistd.h>
```

```
#include <errno.h>
```

```
#include <sys/ioctl.h>
```

```
#include <string.h>
```

```
#include <linux/spi/spidev.h>
```

```
#include "spidev.h"
```

```
SpiDev::SpiDev(const char * devName){
```

```
    _devName = NULL;
```

```
    if (devName != NULL ){
```

```
        _devName = (char *)malloc(strlen(devName)+1);
```

```
        if (_devName != NULL)
```

```
            strcpy(_devName,devName);
```

```
    }
```

```
    _isOpen = false;
```

```
}
```

```
SpiDev::SpiDev(const char * devName, spidev_cfg_t * cfg){
```

```
    _devName = NULL;
```

```
    if (devName != NULL ){
```

```
        _devName = (char *)malloc(strlen(devName)+1);
```

```
        if (_devName != NULL)
```

```
            strcpy(_devName, devName);
```

```
    }
```

```
    if (cfg != NULL){
```

```
        memcpy(&_cfg, cfg, sizeof(spidev_cfg_t));
```

```
    }
```

```

else {

    _cfg.mode = 0;

    _cfg.speed = 1000000;

    _cfg.bits_per_word = 8;

    _cfg.delay = 0;

}

_isOpen = false;
}

SpiDev::~SpiDev() {

    qDebug() << "SpiDev destructor";

    if (_devName != nullptr) {

        free(_devName);

        _devName = nullptr;

    }

    if (_isOpen)

        close(_fd);

}

bool SpiDev::setBitPerWord(uint8_t nbBits) {

    /* Set bits per word*/

    if (ioctl(_fd, SPI_IOC_WR_BITS_PER_WORD, &nbBits) < 0) {

        return false;

    }

    if (ioctl(_fd, SPI_IOC_RD_BITS_PER_WORD, &nbBits) < 0) {

        return false;

    }

    _cfg.bits_per_word = nbBits;

    return true;

}

bool SpiDev::setSpeed(uint32_t speed) {

    /* Set SPI speed*/

    if (ioctl(_fd, SPI_IOC_WR_MAX_SPEED_HZ, &speed) < 0) {

        return false;

    }

    if (ioctl(_fd, SPI_IOC_RD_MAX_SPEED_HZ, &speed) < 0) {

        return false;

    }

    _cfg.speed = speed;

```

```

        return true;
    }

bool SpiDev::setMode(uint8_t mode){
    /* Set SPI_POL and SPI_PHA */
    if (ioctl(_fd, SPI_IOC_WR_MODE, &mode) < 0) {
        return false;
    }

    if (ioctl(_fd, SPI_IOC_RD_MODE, &mode) < 0) {
        return false;
    }

    _cfg.mode = mode;

    return true;
}

int SpiDev::xfer(uint8_t *txBuf, uint8_t txLen, uint8_t *rxBuf, uint8_t rxLen){
    struct spi_ioc_transfer spi_message[1];

    memset(spi_message, 0, sizeof(spi_message));

    spi_message[0].rx_buf = (unsigned long)rxBuf;
    spi_message[0].tx_buf = (unsigned long)txBuf;
    spi_message[0].len = txLen;
    return ioctl(_fd, SPI_IOC_MESSAGE(1), spi_message);
}

int SpiDev::write(uint8_t *txBuf, uint8_t txLen){
    struct spi_ioc_transfer spi_message[1];

    memset(spi_message, 0, sizeof(spi_message));

    spi_message[0].tx_buf = (unsigned long)txBuf;
    spi_message[0].len = txLen;

    return ioctl(_fd, SPI_IOC_MESSAGE(1), spi_message);
}

int SpiDev::read(uint8_t *rxBuf, uint8_t rxLen){
    struct spi_ioc_transfer spi_message[1];

    memset(spi_message, 0, sizeof(spi_message));

```

```

spi_message[0].rx_buf = (unsigned long)rxBuf;

spi_message[0].len = rxLen;

return ioctl(_fd, SPI_IOC_MESSAGE(1), spi_message);
}

bool SpiDev::begin(){
    /* open spidev device */

    if(_isOpen == true )

        return true;

    if(_devName == NULL)

        return false;

    _fd = open(_devName, O_RDWR);

    if(_fd < 0) {

        return false;

    }

    /* Set SPI_POL and SPI_PHA */

    if (ioctl(_fd, SPI_IOC_WR_MODE, &_cfg.mode) < 0) {

        close(_fd);

        return false;

    }

    if (ioctl(_fd, SPI_IOC_RD_MODE, &_cfg.mode) < 0) {

        close(_fd);

        return false;

    }

    /* Set bits per word*/

    if (ioctl(_fd, SPI_IOC_WR_BITS_PER_WORD, &_cfg.bits_per_word) < 0) {

        close(_fd);

        return false;

    }

    if (ioctl(_fd, SPI_IOC_RD_BITS_PER_WORD, &_cfg.bits_per_word) < 0) {

        close(_fd);

        return false;

    }

    /* Set SPI speed*/

    if (ioctl(_fd, SPI_IOC_WR_MAX_SPEED_HZ, &_cfg.speed) < 0) {

        close(_fd);

        return false;

    }

    if (ioctl(_fd, SPI_IOC_RD_MAX_SPEED_HZ, &_cfg.speed) < 0) {

        close(_fd);

        return false;

    }

```

```

    }

    _isOpen = true;

    return true;
}

bool SpiDev::setConfig(spidev_cfg_t *cfg){
    if (cfg != NULL){
        memcpy(&_cfg.cfg,sizeof(spidev_cfg_t));

        if (_isOpen){
            /* Set SPI_POL and SPI_PHA */

            if (ioctl(_fd, SPI_IOC_WR_MODE, &_cfg.mode) < 0) {

                close(_fd);

                return false;

            }

            if (ioctl(_fd, SPI_IOC_RD_MODE, &_cfg.mode) < 0) {

                close(_fd);

                return false;

            }

            /* Set bits per word*/

            if (ioctl(_fd, SPI_IOC_WR_BITS_PER_WORD, &_cfg.bits_per_word) < 0) {

                close(_fd);

                return false;

            }

            if (ioctl(_fd, SPI_IOC_RD_BITS_PER_WORD, &_cfg.bits_per_word) < 0) {

                close(_fd);

                return false;

            }

            /* Set SPI speed*/

            if (ioctl(_fd, SPI_IOC_WR_MAX_SPEED_HZ, &_cfg.speed) < 0) {

                close(_fd);

                return false;

            }

            if (ioctl(_fd, SPI_IOC_RD_MAX_SPEED_HZ, &_cfg.speed) < 0) {

                close(_fd);

                return false;

            }

            return true;

        }

    }

    return false;
}

```

spidev.h:

```
#ifndef _SPIDEV_H
#define _SPIDEV_H

typedef struct {
    uint8_t mode;
    uint8_t bits_per_word;
    uint32_t speed;
    uint16_t delay;
} spidev_cfg_t;

class SpiDev {
private:
    char * _devName;
    int _fd;
    spidev_cfg_t _cfg;
    bool _isOpen;
public:
    SpiDev(const char * devName);
    SpiDev(const char * devName, spidev_cfg_t * cfg);
    ~SpiDev();
    bool begin();
    bool end();
    int read(uint8_t *rxBuf, uint8_t rxLen);
    int write(uint8_t *txBuf, uint8_t txLen);
    int xfer(uint8_t *txBuf, uint8_t txLen, uint8_t *rxBuf, uint8_t rxLen);
    bool setSpeed(uint32_t speed);
    bool setMode(uint8_t mode);
    bool setBitPerWord(uint8_t nbBits);
    bool setConfig(spidev_cfg_t *cfg);
};

#endif
```

Partie Personnelle

Nino CILICHINI

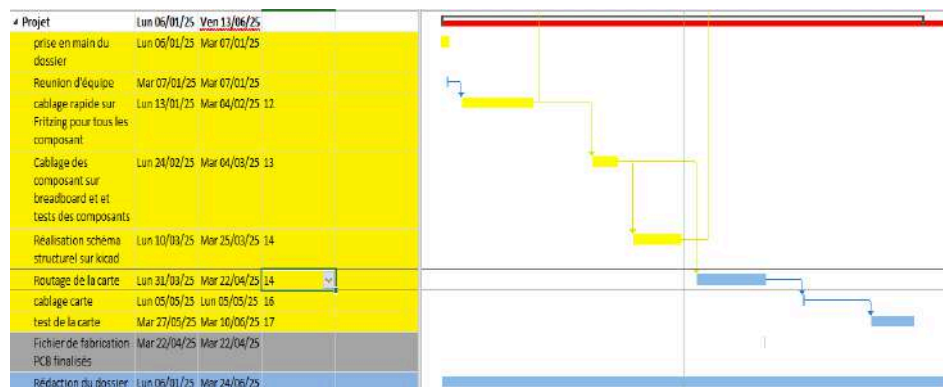
(ER 1)

Introduction

Dans le cadre de ce projet, ma mission consiste à concevoir la partie électronique du **Holter ECG connecté**, en développant une carte électronique (hat Raspberry Pi) permettant la capture de l'ECG sur un patient. Cette carte intégrera les composants nécessaires à l'acquisition des signaux cardiaques, à savoir un capteur de pouls, un amplificateur isolé pour garantir la sécurité du patient, et un convertisseur analogique-numérique pour la communication des données via le bus **SPI**. En collaboration étroite avec les étudiants en charge de la transmission des mesures en LoRa et du simulateur d'ECG, je serai responsable de la conception et du routage de la carte, de la production des fichiers nécessaires à la fabrication du PCB, ainsi que des essais de câblage et de validation. Une attention particulière sera portée à la documentation de chaque phase du projet, à travers des schémas de câblage rapides. Cette étape fondamentale permettra de poser les bases du dispositif, en assurant à la fois la précision des mesures et la sécurité du patient.

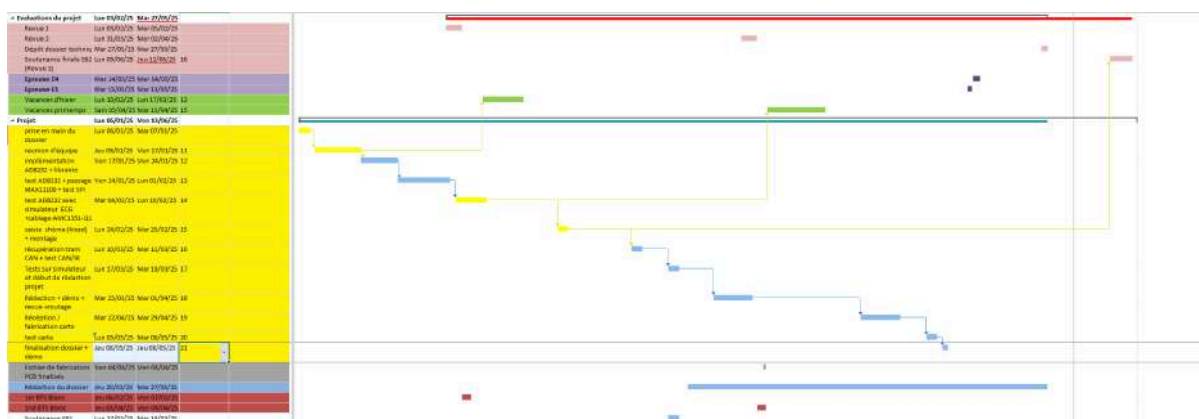
Avant de commencer ce projet j'ai réalisé une planification prévisionnelle du projet pour estimer mon temps de travail.

planification prévisionnel



Planification réelle

N'ayant pas encore d'expérience en gestion de projet, je ne savais pas toujours à quoi m'attendre pour certaines tâches. Cela a entraîné des écarts entre le planning prévisionnel et la réalité, notamment sur le choix des composants, le routage de la carte et la validation, qui m'ont pris plus de temps que prévu. Malgré cela, j'ai su m'adapter pour respecter les grandes étapes du projet



Présentation des éléments du système

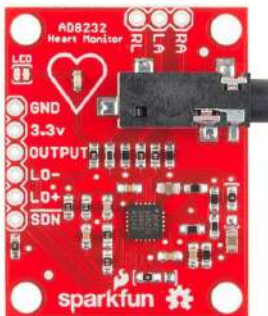
Une Raspberry pi 4:

Le **Raspberry Pi 4** servira de plateforme principale pour le traitement des données ECG capturées. Il agit comme le cerveau de notre système, permettant d'exécuter les programmes nécessaires à la gestion des signaux et à la communication des données. Il sera utilisé avec les bibliothèques **bcm2835** pour l'interface avec les différents composants électroniques. La Raspberry pi permet également de gérer la communication via le bus **SPI** et **UART** pour envoyer les données de l'**ECG** à d'autres modules, et de transmettre les informations nécessaires au suivi du patient.



Capteur de pouls Sparkfun SEN 12650:

La carte **SparkFun AD8232** permet de mesurer l'activité électrique du cœur en détectant les faibles signaux produits lors des battements cardiaques. Pour cela, elle utilise trois électrodes placées sur la peau : deux pour capter la différence de potentiel électrique entre deux points du corps, et une troisième servant de référence. Ces signaux, très faibles et sensibles aux interférences, sont envoyés à la puce AD8232 qui se charge de les amplifier et de les filtrer pour éliminer les bruits parasites (comme les interférences musculaires ou les mouvements). Le signal obtenu, appelé électrocardiogramme (ECG), est ensuite disponible en sortie sous forme analogique. Ce signal peut être visualisé ou traité par un microcontrôleur, permettant ainsi de suivre le rythme cardiaque ou d'analyser les différentes phases d'un cycle cardiaque.



Amplificateur Isolé:

L'**amplificateur isolé** est un composant essentiel pour garantir la **sécurité du patient**. Il permet d'isoler électriquement les signaux provenant du capteur de pouls du circuit de traitement, réduisant ainsi le risque de perturbation ou de choc électrique pour l'utilisateur. Ce composant amplifie également les signaux ECG faibles pour les rendre plus lisibles et exploitables par le Raspberry Pi.



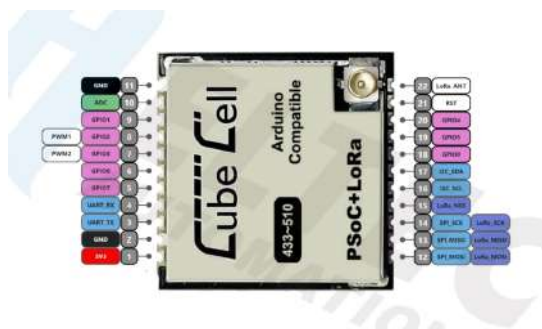
Convertisseur Analogique-Numérique (ADC) 16 bits:

Le **convertisseur analogique-numérique (ADC)** est utilisé pour convertir les signaux analogiques provenant du capteur de pouls en valeurs numériques compréhensibles par le Raspberry Pi. En utilisant un **convertisseur ADC avec bus SPI**, nous assurons une communication rapide et fiable des données entre le capteur et le Raspberry Pi, permettant de traiter les signaux ECG en temps réel.



AM01 Module (V2) 868MHz:

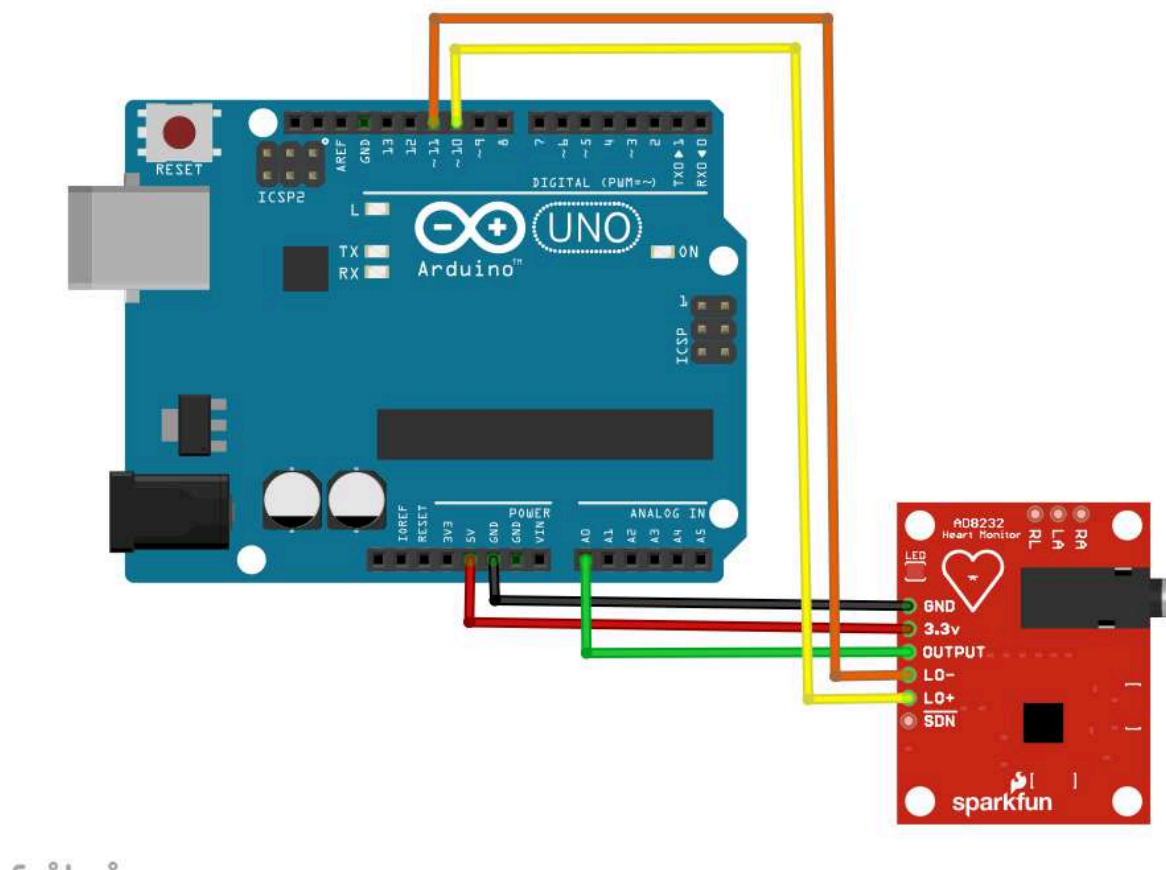
Le module AM01 (V2) est un module **LoRa** conçu pour fonctionner sur la bande de fréquence **868 MHz**, largement utilisée en Europe et dans d'autres régions pour les communications longue portée. Il permet des transmissions sans fil entre dispositifs avec une portée pouvant atteindre plusieurs kilomètres selon les conditions. De plus, il est équipé d'un **connecteur U.FL**, facilitant l'utilisation d'une antenne externe pour optimiser la réception et l'émission du signal.



Mise en oeuvre des différents composants avec Arduino

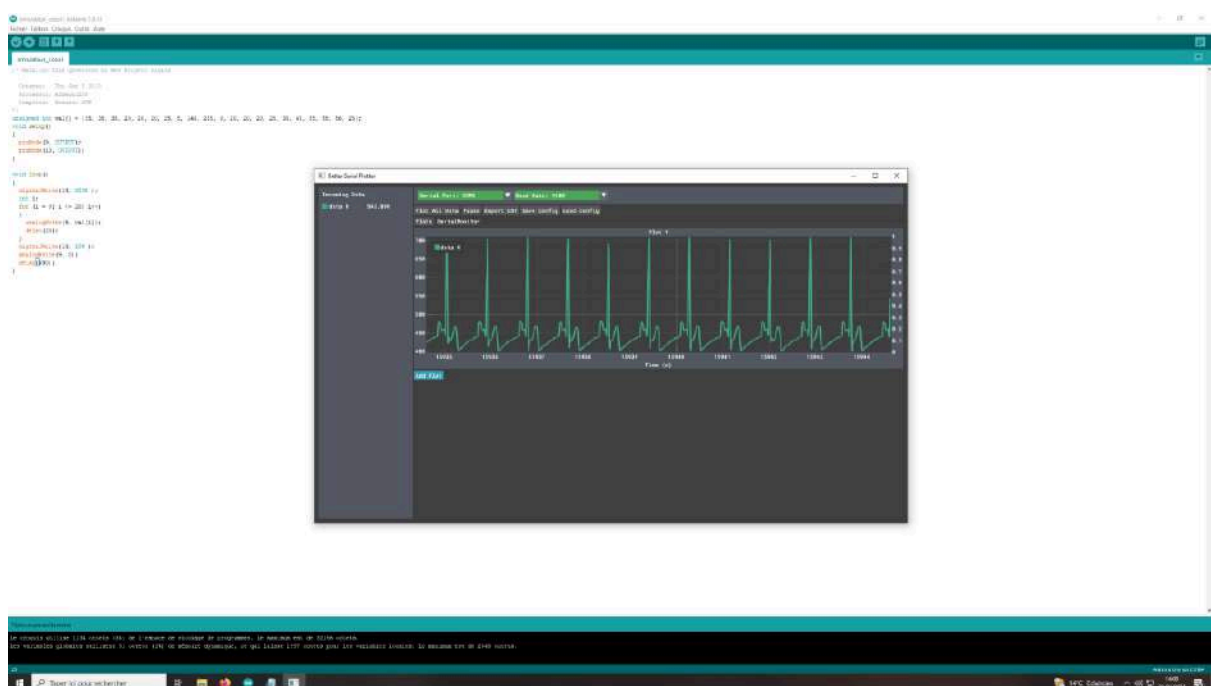
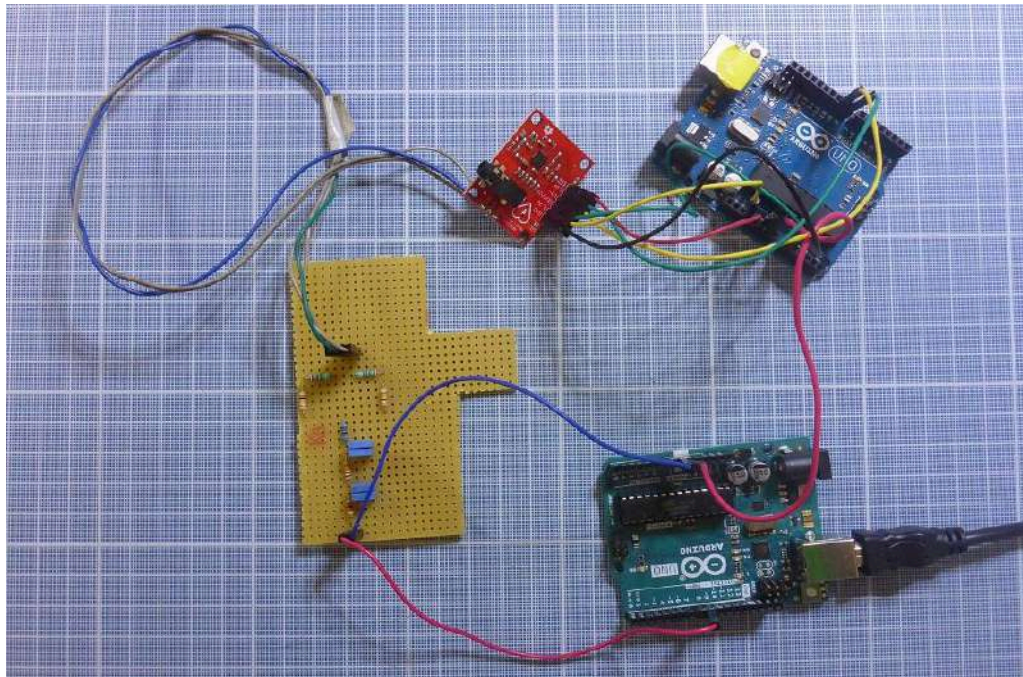
Capteur de pouls Sparkfun SEN 12650 corps:

Schéma de câblage depuis Fritzing



Après avoir suivi un tutoriel pour prendre en main le capteur de pouls lien ci dessous, j'ai rencontré des difficultés concernant la visualisation du graphique sur Arduino avec le **Serial Plotter**. J'ai donc sollicité l'aide du professeur du **Ceri**, qui avait déjà effectué des tests avec ce module. Il était venu dans la salle, mais cela ne fonctionnait toujours pas. En conséquence, il m'a apporté ses propres cartes et a réalisé une simulation de l'électrocardiogramme, ce qui nous a permis de faire des tests.

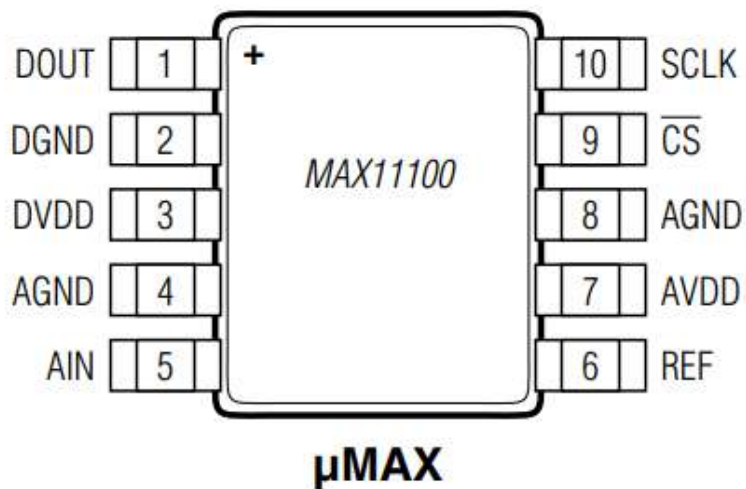
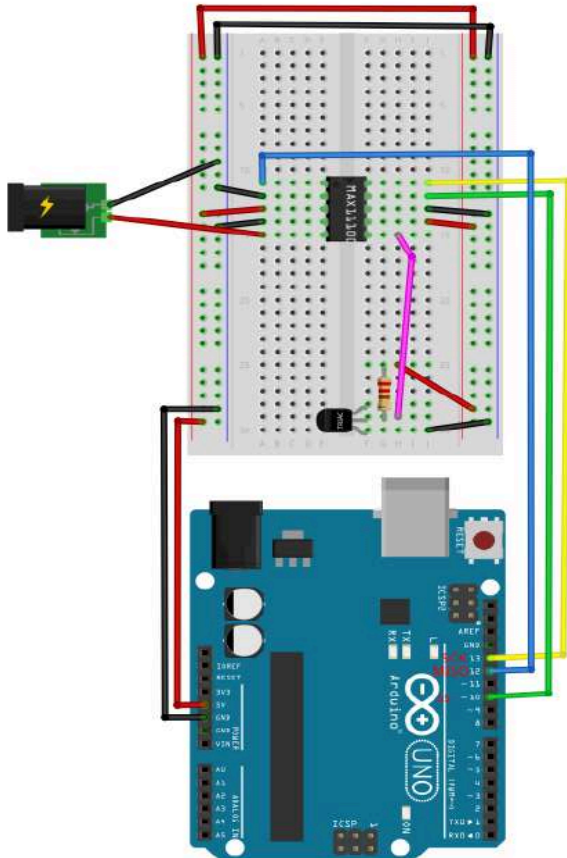
<https://learn.sparkfun.com/tutorials/ad8232-heart-rate-monitor-hookup-guide>



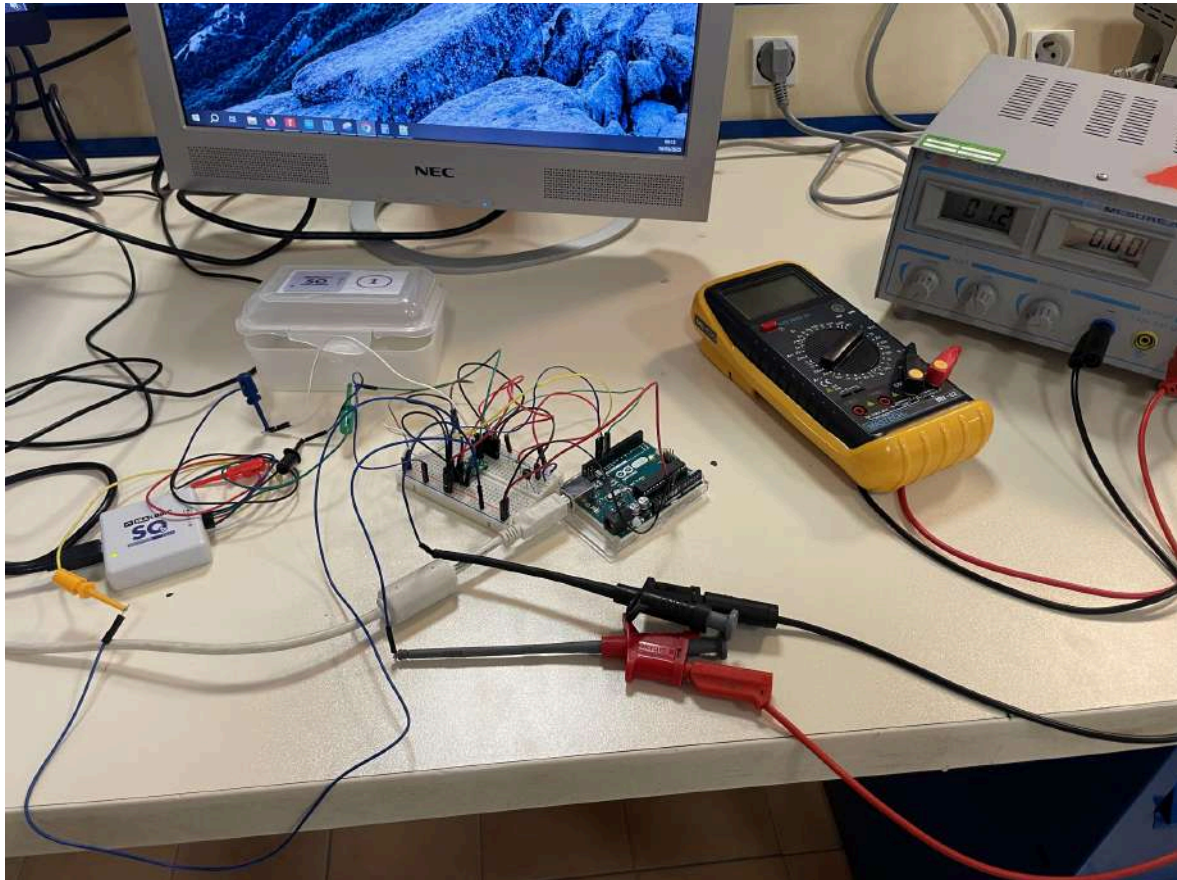
Convertisseur Analogique-Numérique MAX11100 (ADC) 16 bits:

Schéma de câblage du CAN:

J'ai connecté une alimentation à l'entrée analogique du **CAN** ainsi qu'une **tension de référence de 2,5V**. J'ai utilisé uniquement les fils **MISO**, **SCLK** et **SS**, mais **pas MOSI**, car nous devons seulement lire les valeurs et non les écrire.



Après avoir réalisé mon câblage rapide sur Fritzing et l'avoir fait vérifier par le professeur, je suis passé au câblage réel. J'ai ensuite téléversé le programme du MAX1100 sur l'Arduino et modifier le code afin qu'il affiche également la tension d'entrée.



<pre> 12 #include <Arduino.h> 13 #include <ADC_MAX1100.h> 14 15 ADC_MAX1100 ADC_EC(10); 16 uint16_t raw_reading = 0; 17 18 19 void setup() 20 { 21 Serial.begin(9600); 22 } 23 24 void loop() 25 { 26 27 raw_reading = ADC_EC.readAdc(); 28 Serial.print(raw_reading); 29 Serial.print(" -> "); 30 Serial.print(raw_reading*2.50/65536); 31 Serial.println("V"); 32 delay(1000); 33 } </pre>	Output Serial Monitor x Message (Enter to send message to 'Arduino Uno' on 'COM3') <pre> 30311 -> 1.16V 30310 -> 1.16V 30208 -> 1.15V 30311 -> 1.16V 30310 -> 1.16V 30311 -> 1.16V 30304 -> 1.16V 30311 -> 1.16V 30311 -> 1.16V 30311 -> 1.16V 30311 -> 1.16V 30307 -> 1.16V 30311 -> 1.16V 30311 -> 1.16V 30311 -> 1.16V 30310 -> 1.16V 30304 -> 1.16V 30311 -> 1.16V 30311 -> 1.16V </pre>
---	---

Pour une tension d'alimentation de **1,2 V**, la tension mesurée en sortie est d'environ **1,16 V**. On constate ainsi une légère différence entre la valeur attendue et la valeur réelle, ce qui indique que la précision de l'alimentation n'est pas totalement optimale



```

30209 -> 1.15V
30211 -> 1.15V
30211 -> 1.15V
30208 -> 1.15V
30304 -> 1.16V
30304 -> 1.16V
30211 -> 1.15V
30304 -> 1.16V
30259 -> 1.15V
30304 -> 1.16V
30305 -> 1.16V
30311 -> 1.16V
30259 -> 1.15V
30211 -> 1.15V
30259 -> 1.15V
30319 -> 1.16V
30256 -> 1.15V
30305 -> 1.16V

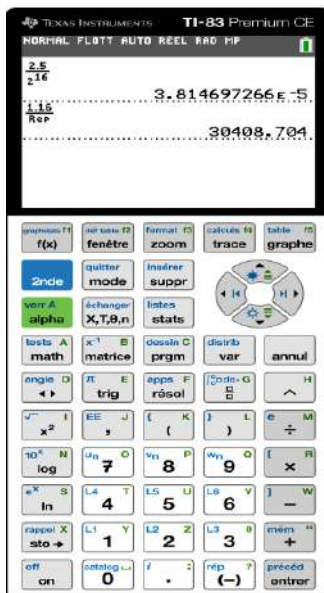
```

J'ai donc réalisé le calcul afin de vérifier la cohérence des résultats avec la théorie, en déterminant le quantum selon la formule :

$$q = \frac{U_{max}}{2^n} \quad \text{valeur numérique} = \frac{\text{tension entrée}}{q}$$

Avec **U_{max}**: tension d'entrée, et **n** : nombre de bits

Les résultats obtenus s'avèrent globalement cohérents avec la théorie, comme le montre le calcul ci-dessous. Il convient toutefois de noter que la tension indiquée par l'alimentation n'est pas parfaitement précise, ce qui peut introduire une légère marge d'erreur dans les mesures.



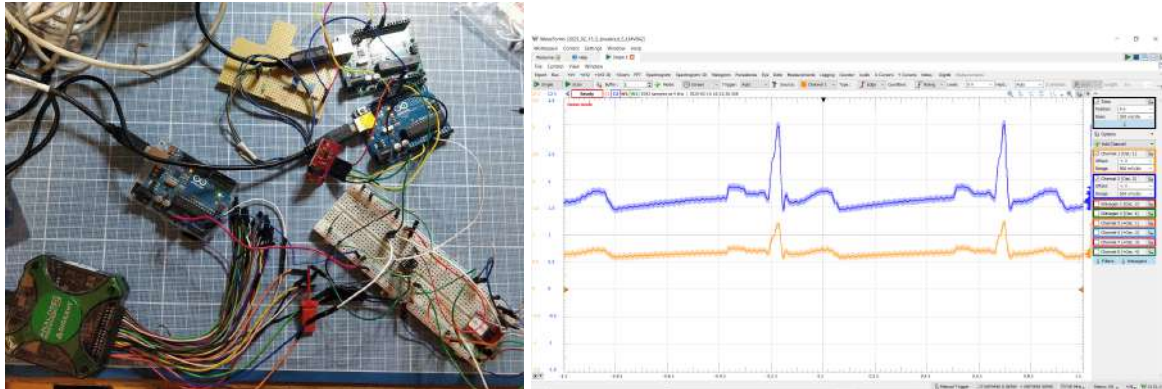
```

30209 -> 1.15V
30211 -> 1.15V
30211 -> 1.15V
30208 -> 1.15V
30304 -> 1.16V
30304 -> 1.16V
30211 -> 1.15V
30304 -> 1.16V
30259 -> 1.15V
30304 -> 1.16V
30305 -> 1.16V
30311 -> 1.16V
30259 -> 1.15V
30211 -> 1.15V
30259 -> 1.15V
30319 -> 1.16V
30256 -> 1.15V
30305 -> 1.16V

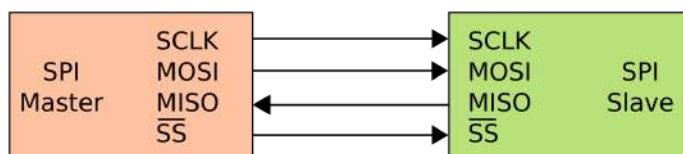
```

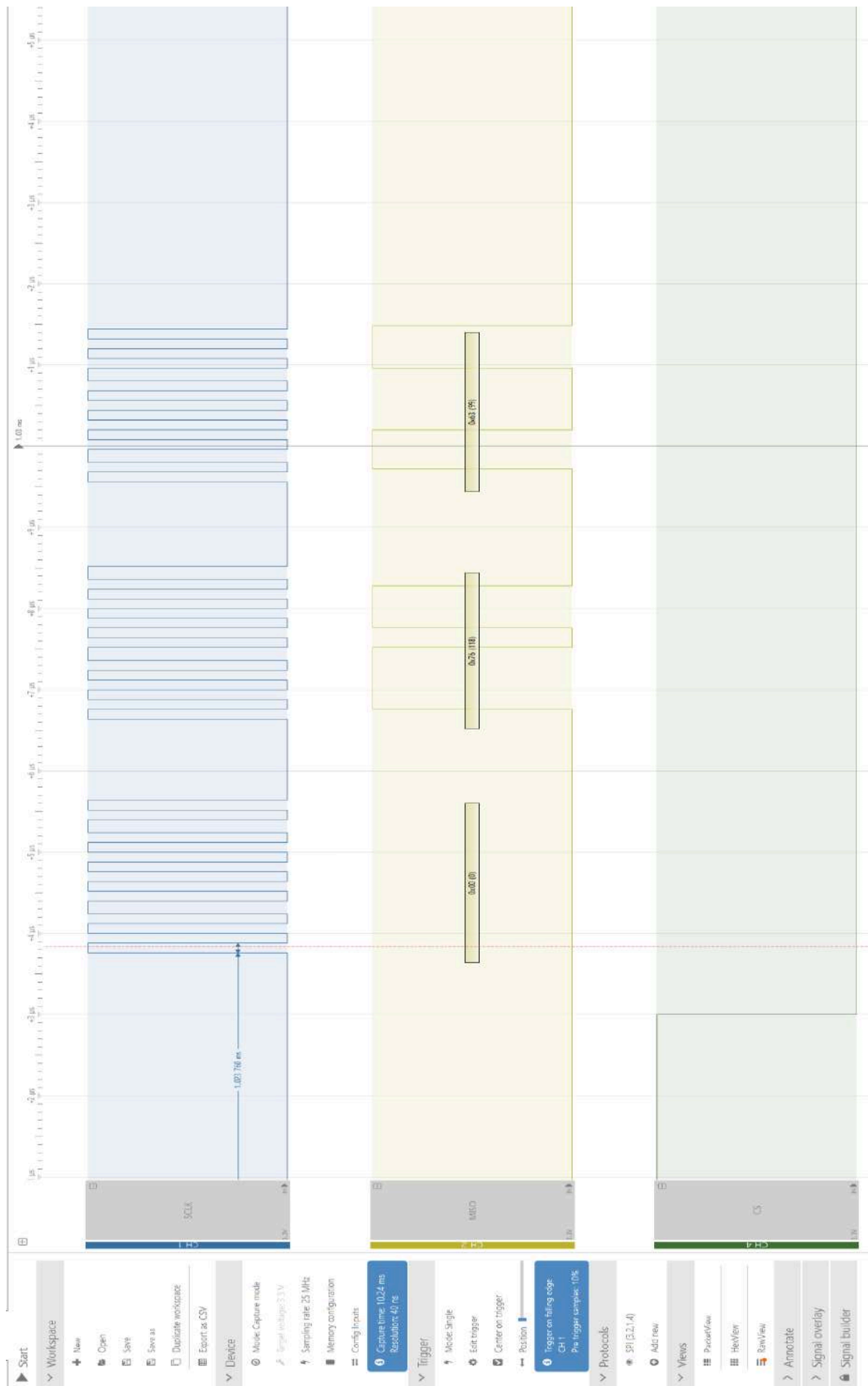
Essaie amplificateur isolé

J'ai câblé l'**amplificateur isolé** et j'ai injecté le signal provenant du simulateur Arduino à son entrée. À l'aide de l'oscilloscope, j'ai visualisé l'**entrée en orange** et la **sortie en bleu**. On observe que les deux signaux ont la même forme, mais que le signal de sortie est amplifié par rapport à celui d'entrée.



Ensuite, j'ai utilisé un analyseur logique (Ika Logic) pour visualiser la trame envoyée sur le bus SPI et vérifier si tout était cohérent. Avant cela, je vais d'abord rappeler le principe de fonctionnement du bus SPI. Il s'agit d'un protocole de communication série synchrone où le maître génère un signal d'horloge (**SCLK**) pour synchroniser l'échange de données avec un ou plusieurs esclaves. La sélection de l'esclave s'effectue via le signal **CS** (Slave Select) : lorsqu'il est à l'état bas (LOW), l'esclave correspondant est activé et peut communiquer avec le maître. Les données sont ensuite échangées bit par bit sur les lignes **MOSI** (Master Output, Slave Input) et **MISO** (Master Input, Slave Output). L'envoi et la réception des bits se font à chaque front d'horloge, selon un mode défini (front montant ou descendant), permettant un transfert rapide et fiable. La ligne **MOSI** n'est pas utilisée dans mon projet, car la **Raspberry pi** ne transmet pas de données vers le périphérique.





On peut observer que, dans la trame, le signal **SS** passe à un **niveau bas**, puis envoie deux quartets de 0, soit un octet (1 byte), suivi de l'octet de poids fort (MSB). Ensuite, un décalage de **8 bits** est effectué, permettant d'envoyer l'octet de poids faible (**LSB**), puis à nouveau l'octet de poids fort. La valeur hexadécimale **0x7663** correspond ainsi à **30 307** en décimal.

J'ai analysé le fichier **.cpp** du programme, que j'ai mis en évidence en jaune fluo, ainsi qu'une partie de la documentation du CAN. Tout semble cohérent avec ce que je constate dans la trame. On remarque que le processus commence par l'envoi de deux quartets à 0, suivis de la donnée.

```
digitalWrite(_CS, LOW); //activate Chipselect
// there is no MOSI for this chip so lets just send bytes of 0s
byte leading8zeros = SPI.transfer(0x00);
byte hi = SPI.transfer(0x00);
byte lo = SPI.transfer(0x00);
digitalWrite(_CS, HIGH); //deactivate Chipselect
```

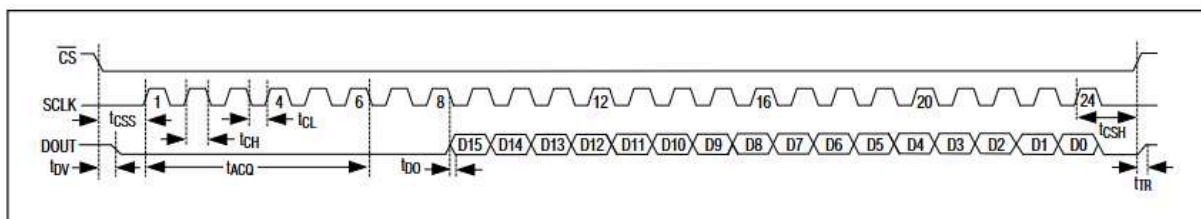
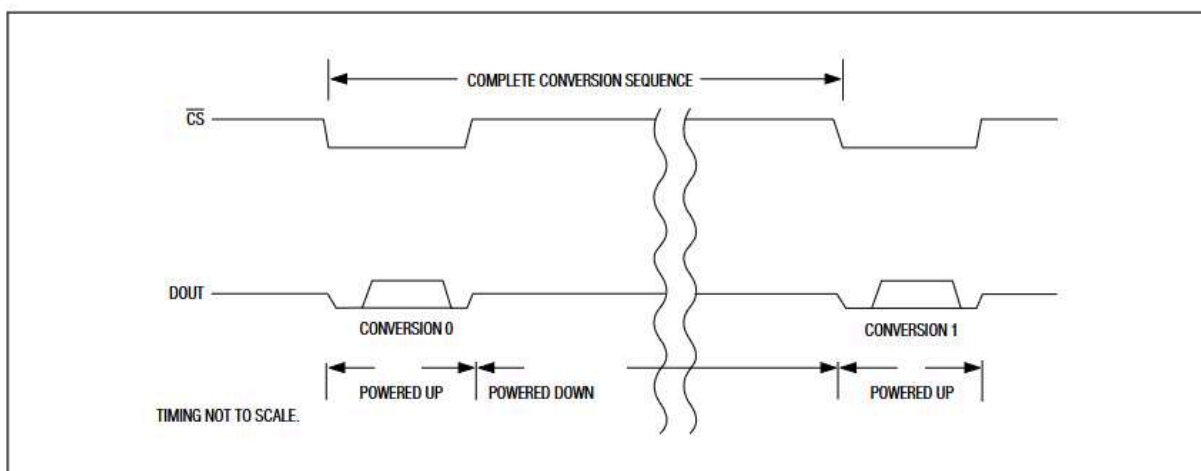
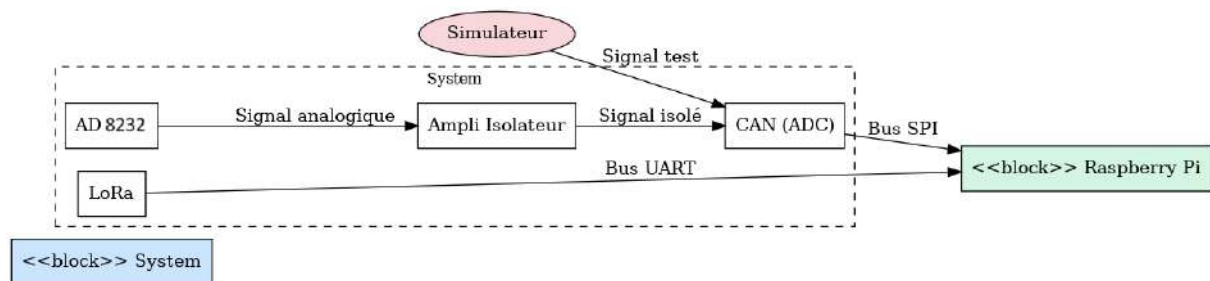


Figure 6. External Timing Diagram



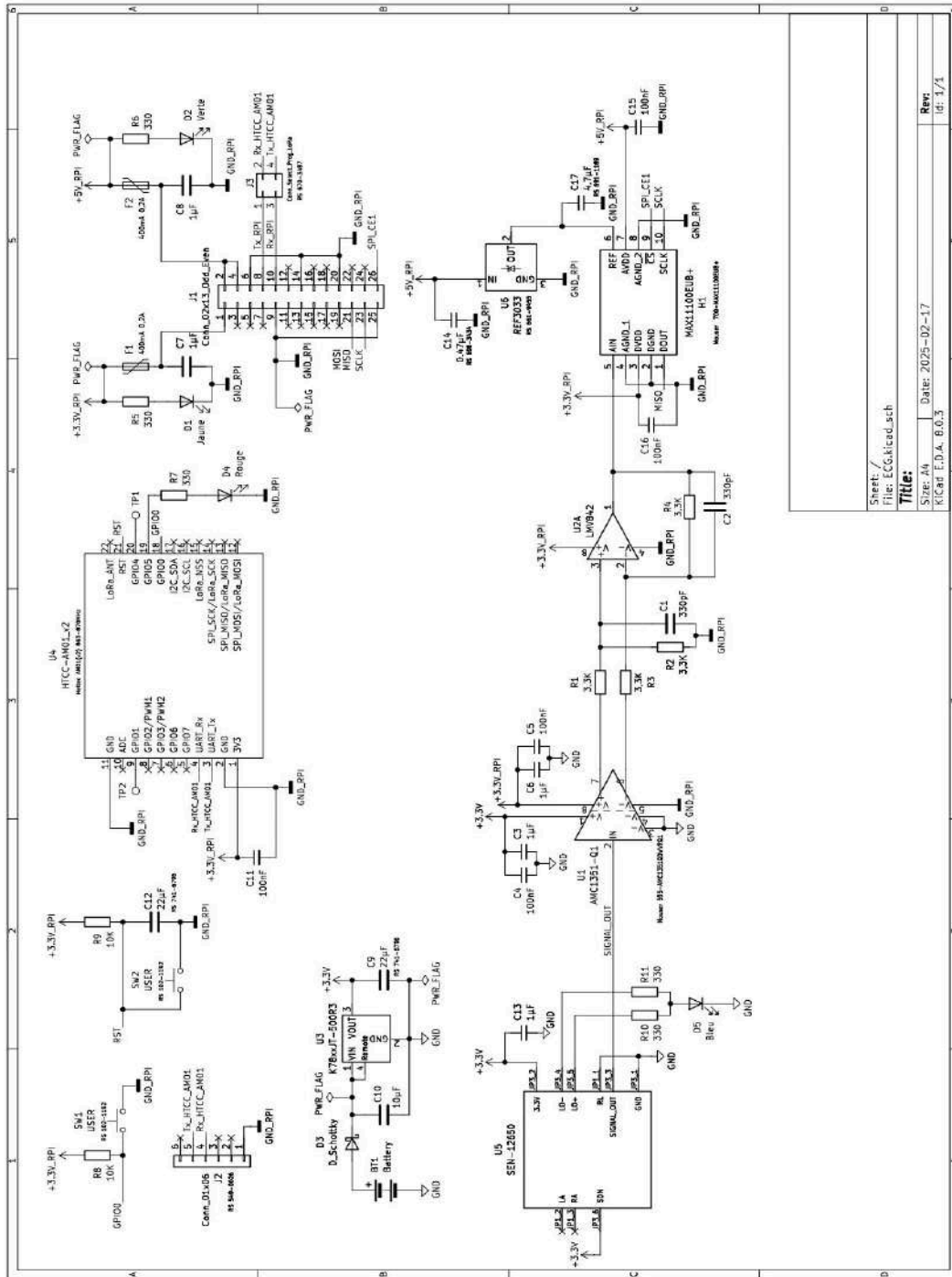
Après avoir vérifié que tous mes modules fonctionnaient correctement sur une **Arduino Uno**, j'ai décidé de réaliser un diagramme SysML pour illustrer la manière dont ils sont connectés à ma carte. Ce diagramme m'a permis de mieux visualiser l'agencement des différents modules et de m'assurer que chaque connexion était bien prise en compte. En utilisant SysML, j'ai pu modéliser précisément les composants et tracer les liaisons entre eux afin d'avoir une représentation claire et compréhensible de mon montage. Cela m'a également aidé à repérer d'éventuelles erreurs ou améliorations à apporter avant de passer à l'étape suivante de mon projet.

Diagramme sysml



J'ai entrepris la réalisation de mon schéma structurel sur **KiCad**. Pour cela, je me suis appuyé sur un schéma d'exemple fourni par mon professeur, qui m'a servi de base. Au fil de l'avancement, j'ai apporté plusieurs modifications afin d'adapter le schéma à mes besoins spécifiques et aux contraintes de mon projet. J'ai pris soin d'organiser les composants de manière claire et logique pour faciliter la conception du circuit et anticiper d'éventuelles corrections. Cette étape m'a permis de mieux comprendre l'architecture de mon montage et d'optimiser son fonctionnement avant de passer à l'étape suivante.

Schéma du professeur



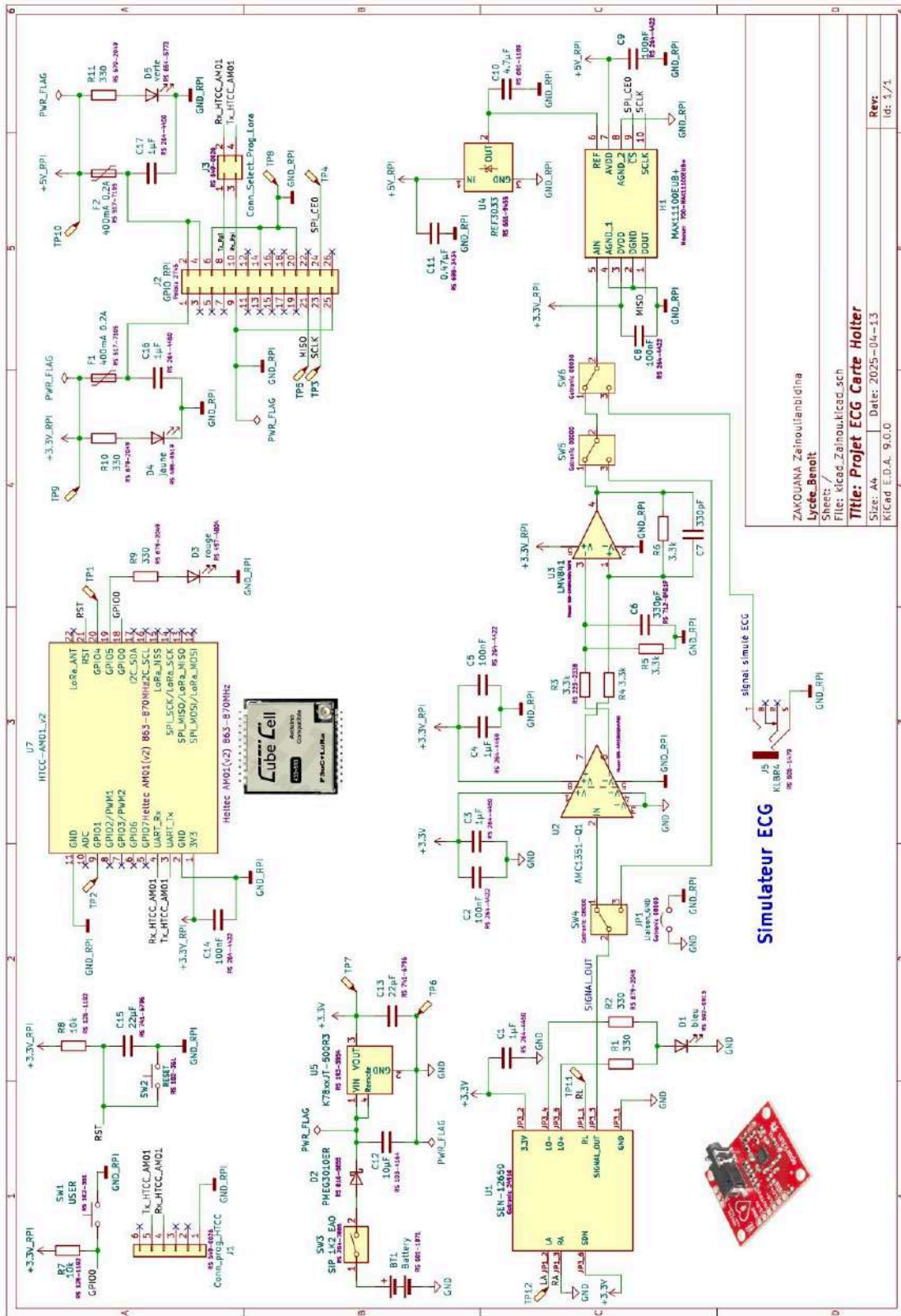
Sheet: /
File: ECG.ecad.sch

Title:

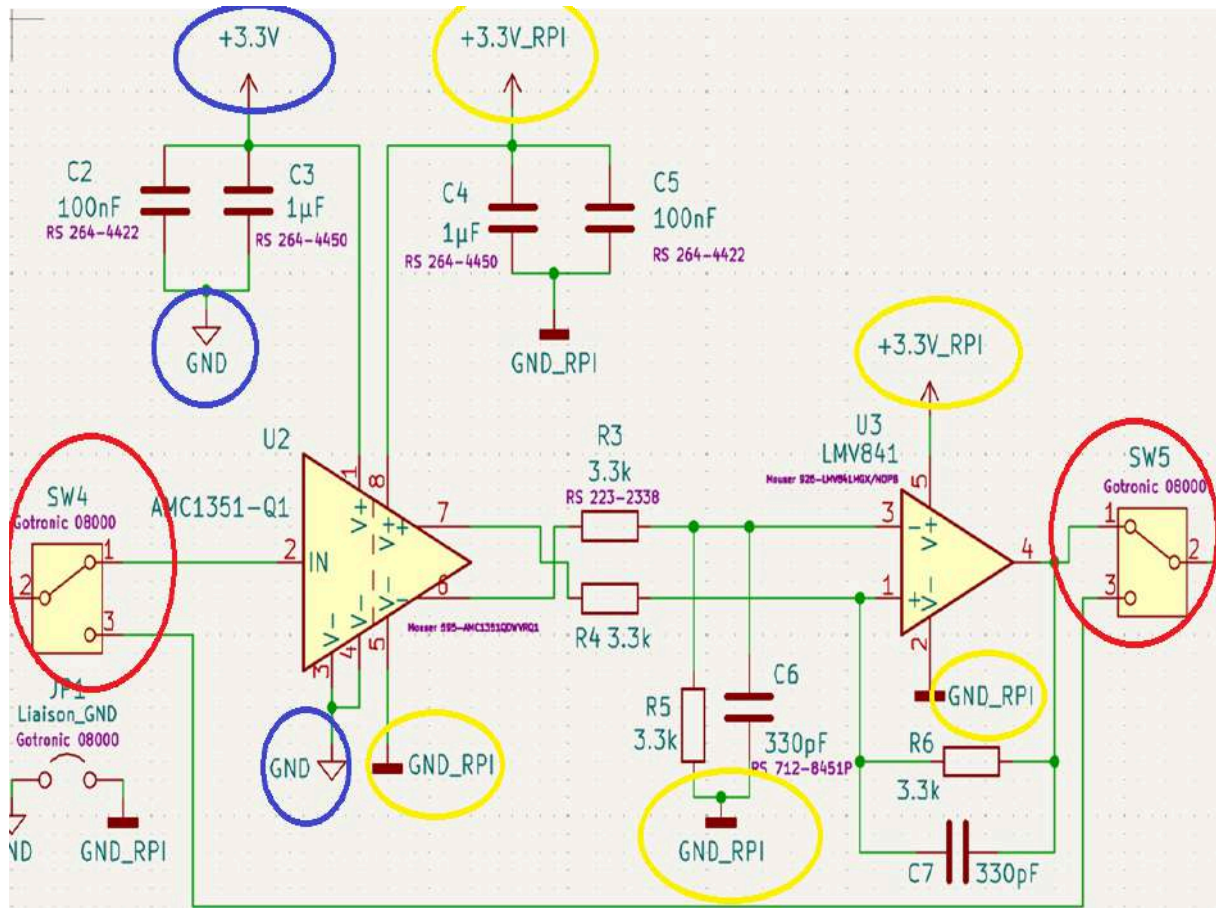
Size: A4 Date: 2025-02-17

Rev: R1 Rev: R1

Mon schéma



On peut clairement voir, entouré en rouge, les modifications que j'ai apportées au schéma. J'ai ajouté deux cavaliers entouré en **rouge** permettant de diriger le signal soit vers l'amplificateur isolateur, utile pour les mesures sur de vraies personnes, soit de le faire passer directement sans amplification. De plus, j'ai ajouté un autre cavalier pour pouvoir injecter directement le signal du simulateur de mon collègue ER, sans passer par l'amplificateur isolateur. On peut également remarquer que j'ai différencié les deux alimentations entouré en **jaune** et **bleu** en raison de l'isolation galvanique : **3.3V** provenant de la pile et **3.3V_RPI**, ainsi que leurs **GND** respectifs.



L'**AMC1351-Q1** est un **amplificateur isolateur** utilisé notamment dans les **appareils médicaux** en raison de sa précision et de sa capacité à assurer une isolation galvanique entre les circuits. Il utilise une modulation **On-Off Keying (OOK)** pour transmettre un signal numérique à travers une barrière d'isolation. Un signal haute fréquence de **480 MHz** est émis pour représenter un bit "1", tandis qu'aucun signal n'est envoyé pour un bit "0". Ce signal est ensuite récupéré, démodulé et converti en analogique par un **convertisseur numérique-analogique (DAC)** interne, bien que la documentation ne fournisse pas de détails sur son fonctionnement. En sortie, un **LMV 842** est utilisé pour filtrer et amplifier le signal afin d'éliminer les bruits indésirables et améliorer la qualité du signal récupéré. Il est crucial de respecter les valeurs des condensateurs pour garantir le bon fonctionnement de l'amplificateur isolateur. Cela est illustré dans l'extrait de la documentation ci-dessous.

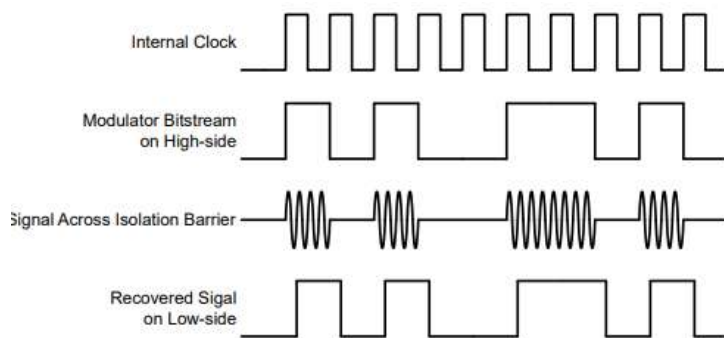
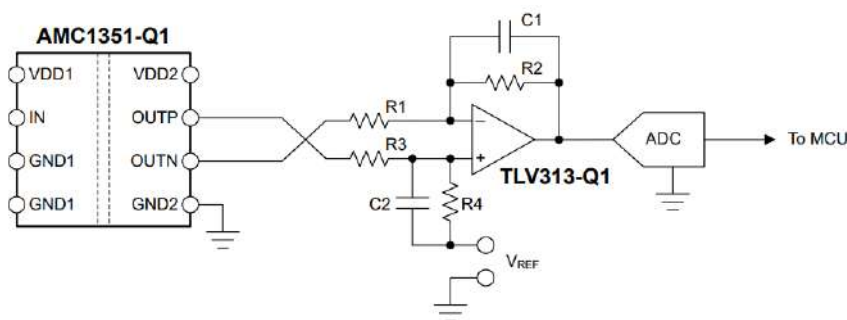
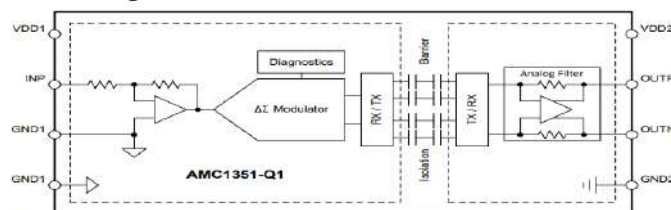


Figure 7-1. OOK-Based Modulation Scheme

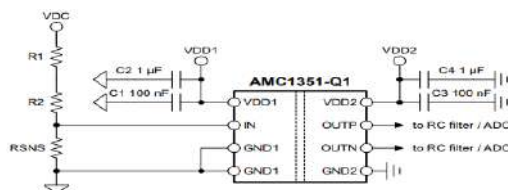
La documentation de l'amplificateur recommandait le modèle **TLV313-Q1**, mais le professeur disposait d'un **LMV842**, que nous avons testé avec succès. Étant donné son bon fonctionnement, nous avons décidé de l'utiliser.



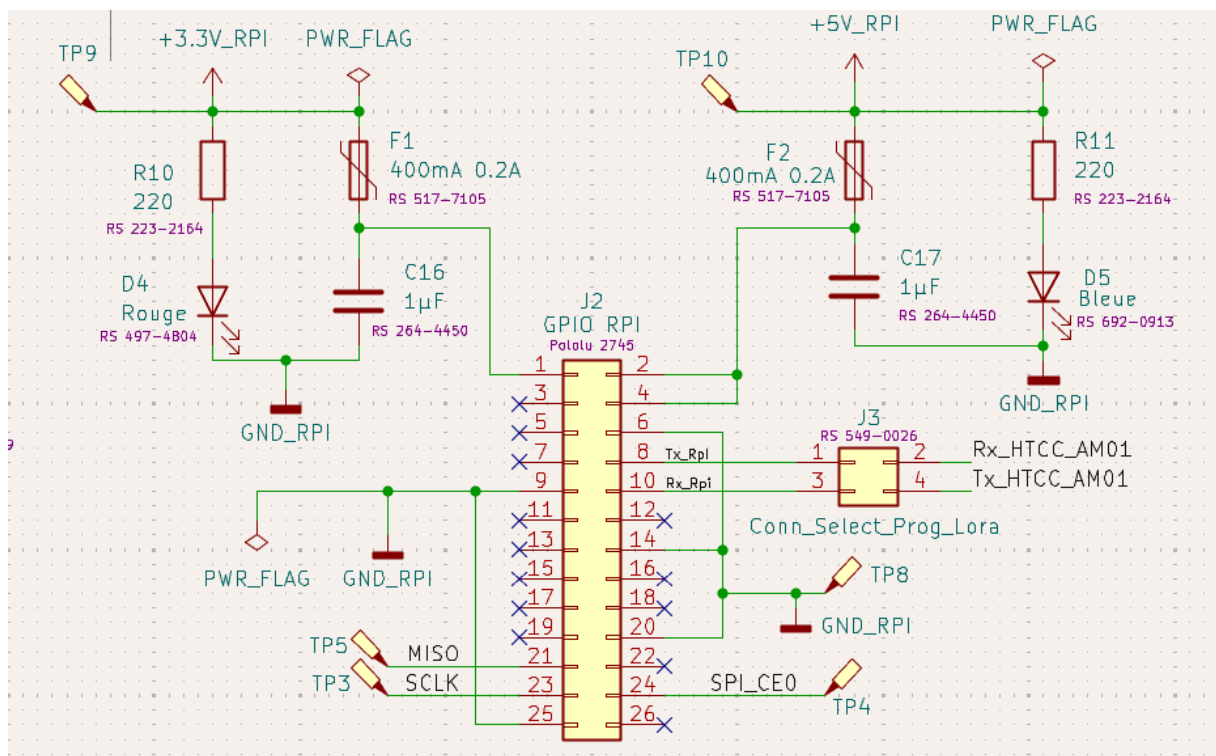
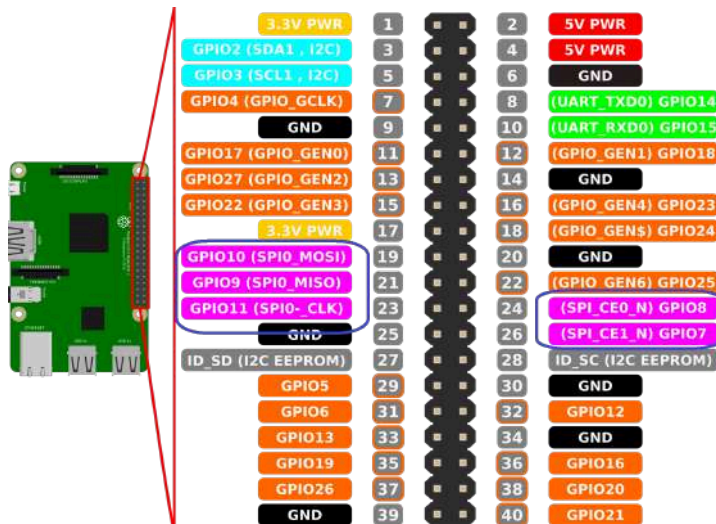
7.2 Functional Block Diagram



7.3 Feature Description



Concernant le convertisseur analogique-numérique sur le schéma structurel, j'ai modifié la tension de référence, qui était de **2,5 V** lors de mes essais, et qui est maintenant de **3,3 V**. Le brochage change également, car nous utiliserons finalement une carte **Raspberry Pi 4**. Ainsi, le **CS** de l'**ADC** est connecté au **GPIO24**, le **SCLK** au **GPIO23** et le **MISO** au **GPIO21** **MOSI** ne sera pas utilisé dans mon cas on envoie pas de données, comme illustré sur les images ci-dessous.



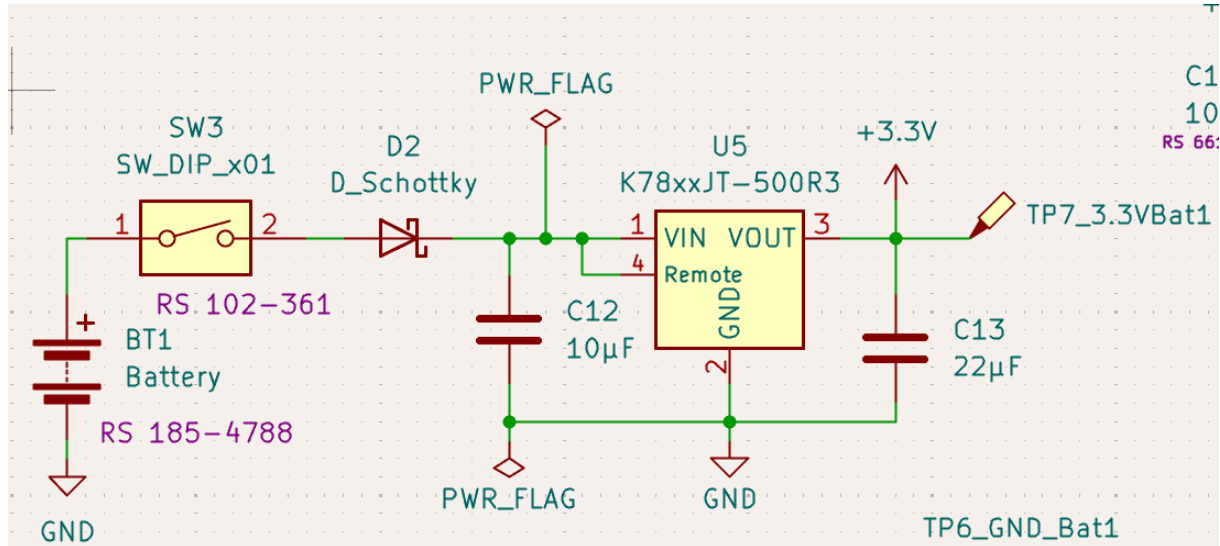
On peut également remarquer que j'ai ajouté des LED témoins sur les deux alimentations de la Raspberry Pi : une **LED Rouge** pour la tension de **3,3 V** avec une résistance de **200 Ω** et une **LED Bleu** pour le **5 V** avec une résistance de **220Ω**. J'ai aussi intégré des fusibles réarmable de **400 mA** ainsi que des condensateurs de découplage aux extrémités des alimentations. De plus, tous les GND sont reliés ensemble, et j'ai ajouté un connecteur à 2 broches sur les broches 8 et 10 pour la programmation du module **LoRa**.

calcul des résistance pour les leds: $R = \frac{U_{alim} - U_{led}}{\text{courant led}} =$

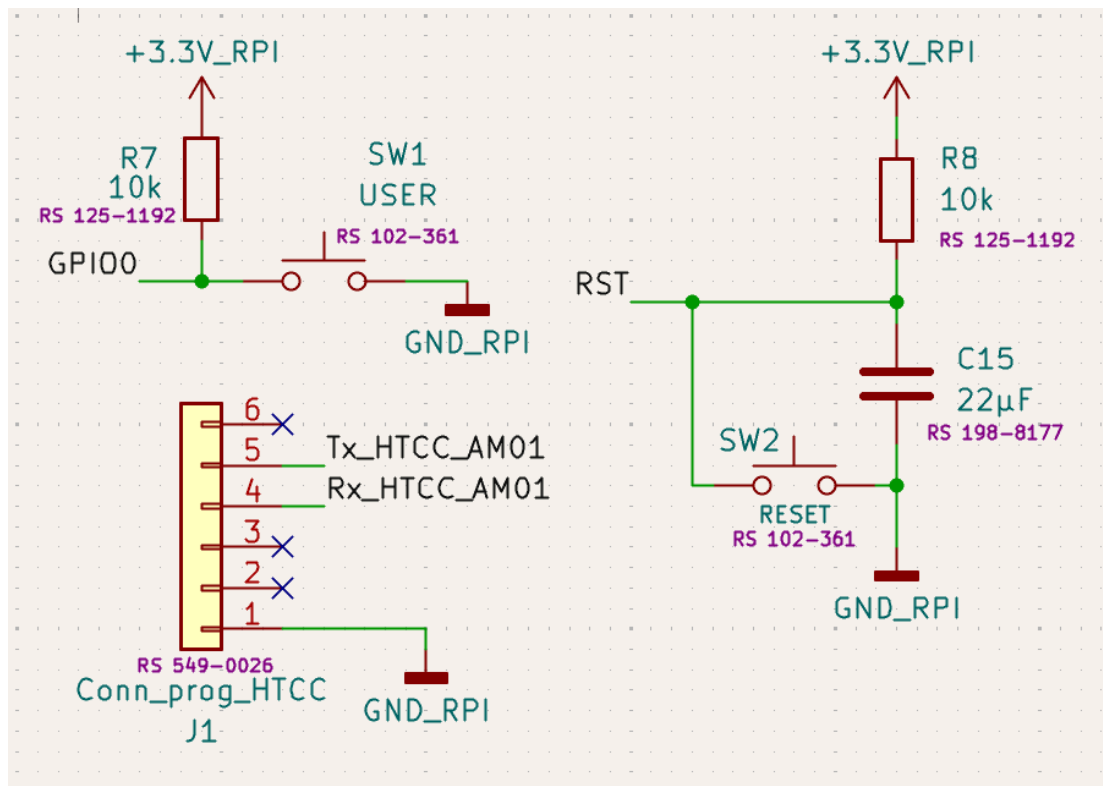
Led **Rouge** $R = \frac{3.3 - 2}{5.9 \times 10^{-3}} = 220\Omega$ valeur normalisé **220Ω**

Led **Bleu** $R = \frac{5 - 3.3}{7.7 \times 10^{-3}} = 220\Omega$ valeur normalisé **220Ω**

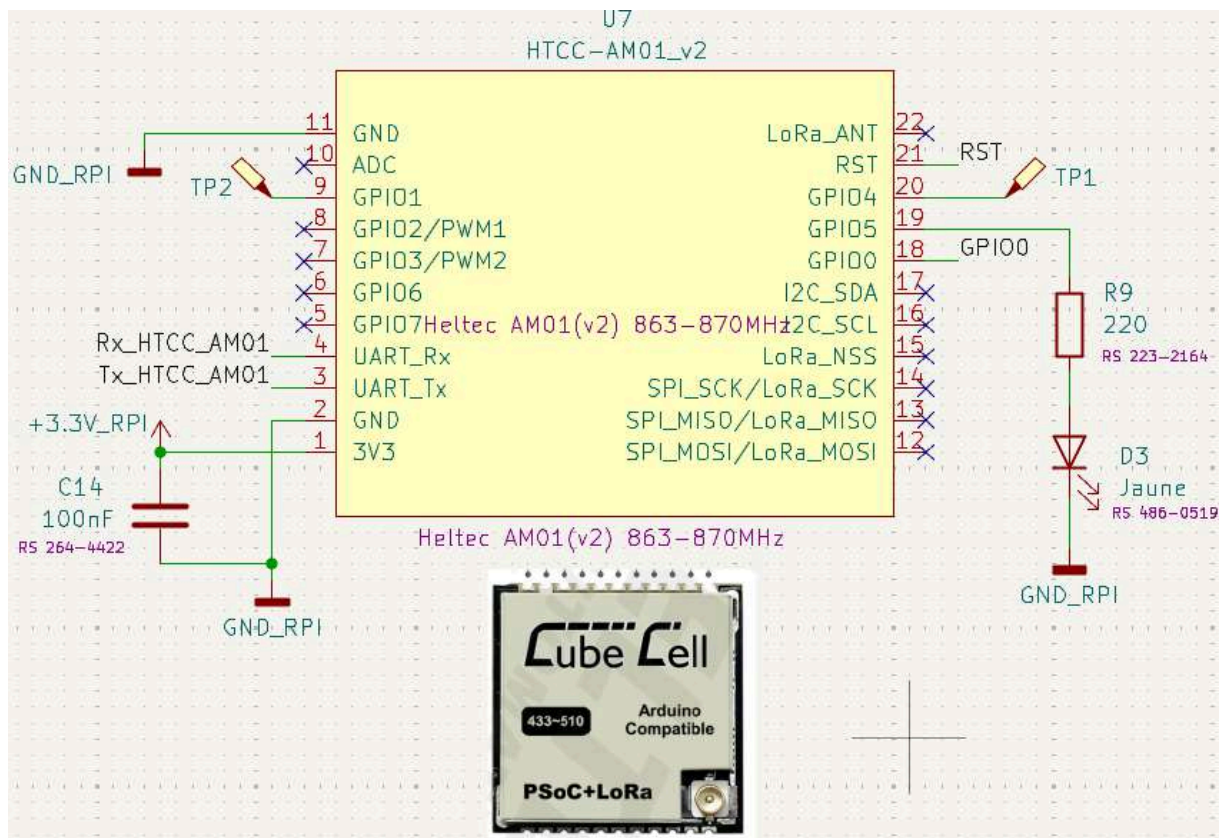
Pour l'alimentation par pile, j'ai opté pour un convertisseur **9V DC - 3.3V DC**. Ce modèle n'est plus disponible sur le marché, mais mon professeur en avait en stock, donc nous l'avons utilisé. J'ai également ajouté un interrupteur ainsi qu'une **diode Schottky** afin de prévenir toute incidence en cas d'inversion de polarité (+ et -).



J'ai aussi intégré un **bouton poussoir** avec une résistance de pull-up pour configurer le module LoRa, ainsi qu'un **bouton de réinitialisation**.



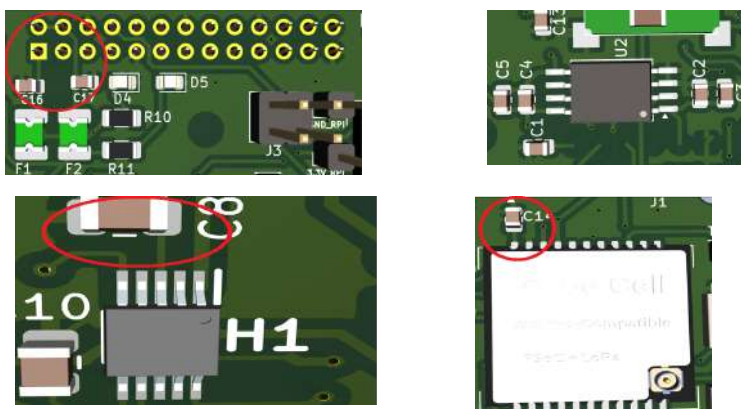
Pour le **module LoRa**, j'ai également ajouté une **LED jaune** en tant que témoin, et il communique en **UART (Tx-Rx)** avec la **Raspberry Pi**, avec un condensateur de découplage pour stabiliser l'alimentation.



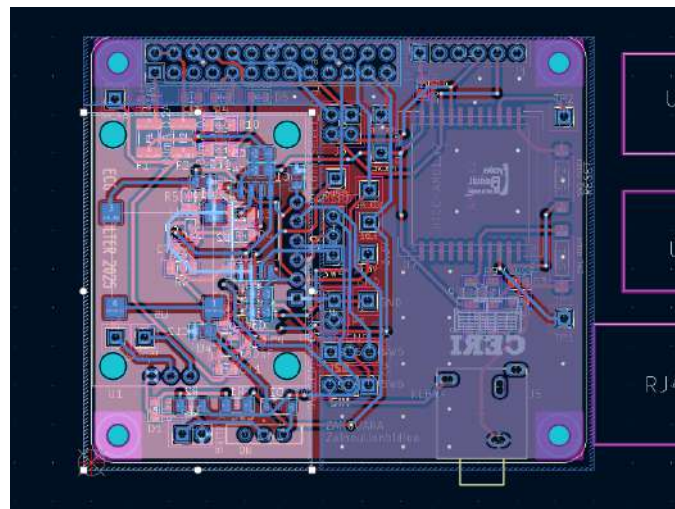
Il me reste à finaliser le routage de la carte, générer les fichiers Gerber ainsi que la liste des composants, puis passer commande pour la fabrication du circuit imprimé et l'achat des composants. Une fois ces étapes réalisées, je pourrai procéder au câblage et aux premiers tests afin de valider le bon fonctionnement du montage.

Routage

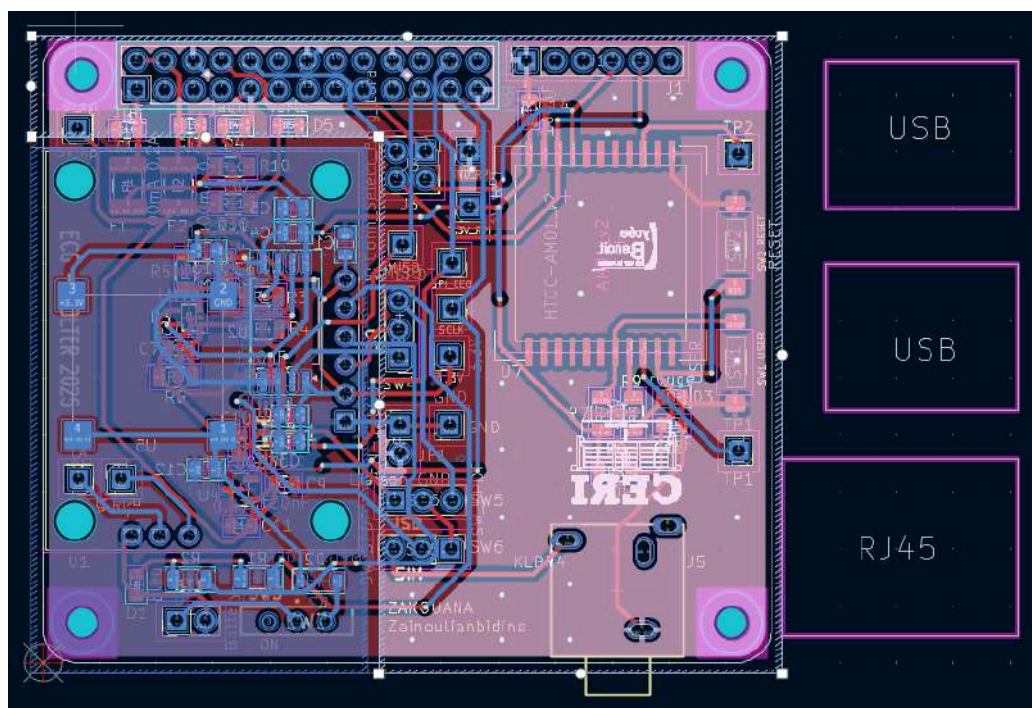
Concernant le routage, j'ai utilisé deux plans de masse distincts : **GND** et **GND_RPI**, afin de limiter les interférences entre alimentations et d'**isoler galvaniquement le circuit lié au Raspberry Pi**. Cette séparation permet également de **protéger la personne sur laquelle les mesures sont prises**. Les condensateurs de découplage ont été placés au plus près des broches d'alimentation.



Plan de masse 3.3V à gauche;

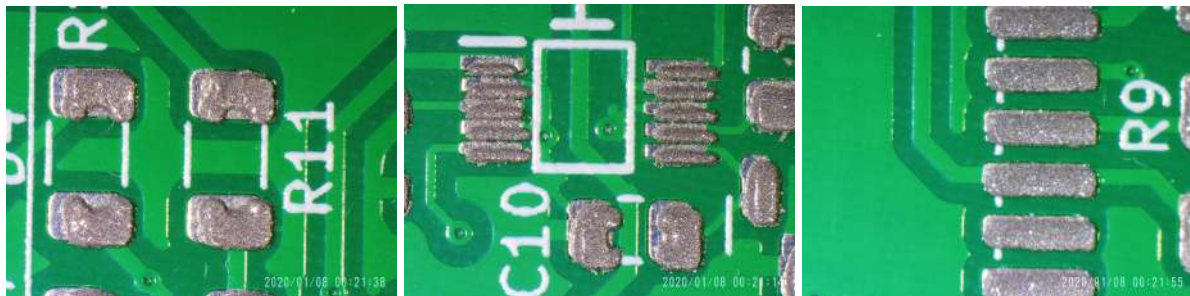


Plan de masse 3.3V_RPI à droite et dessous.

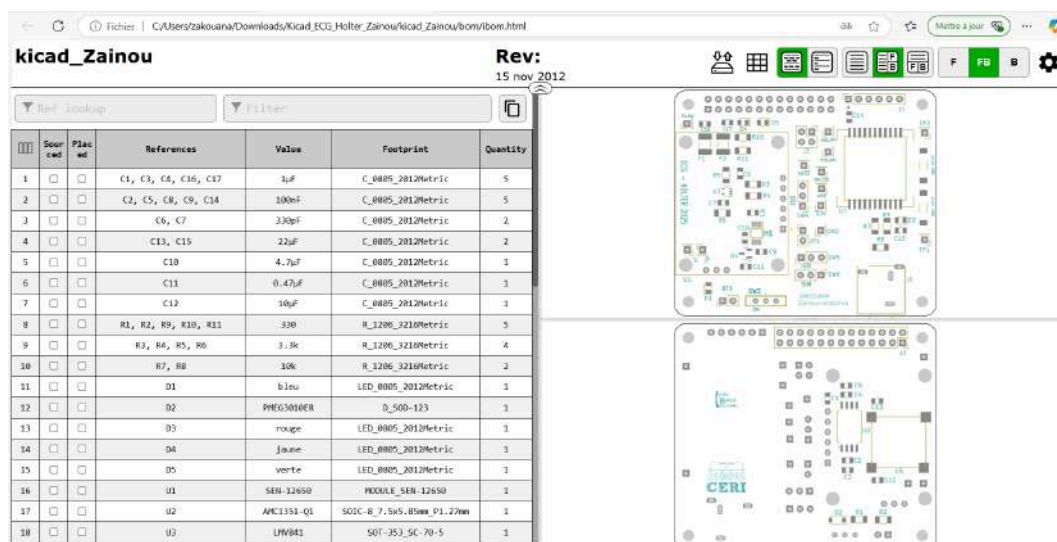


Fabrication de la carte

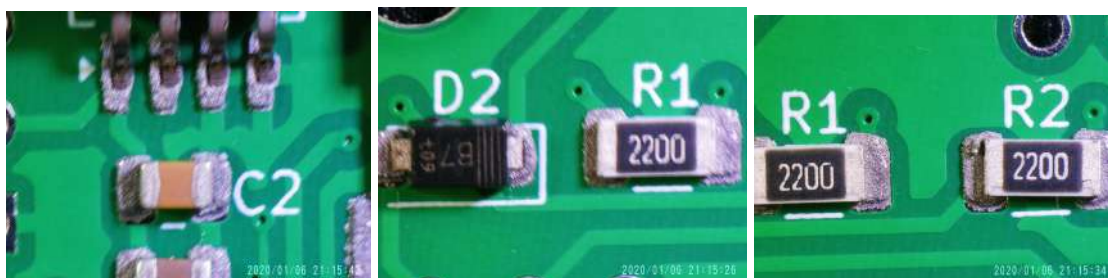
La première étape consiste à nettoyer soigneusement la carte avec de l'alcool **isopropylique** afin d'éliminer toute trace de poussière ou de graisse. Ensuite, à l'aide du **stencil**, j'applique la pâte à braser sur l'ensemble des empreintes des composants. Une fois cette opération réalisée, je vérifie minutieusement la répartition de la pâte à l'aide d'une loupe, comme illustré sur les photos ci-dessous.



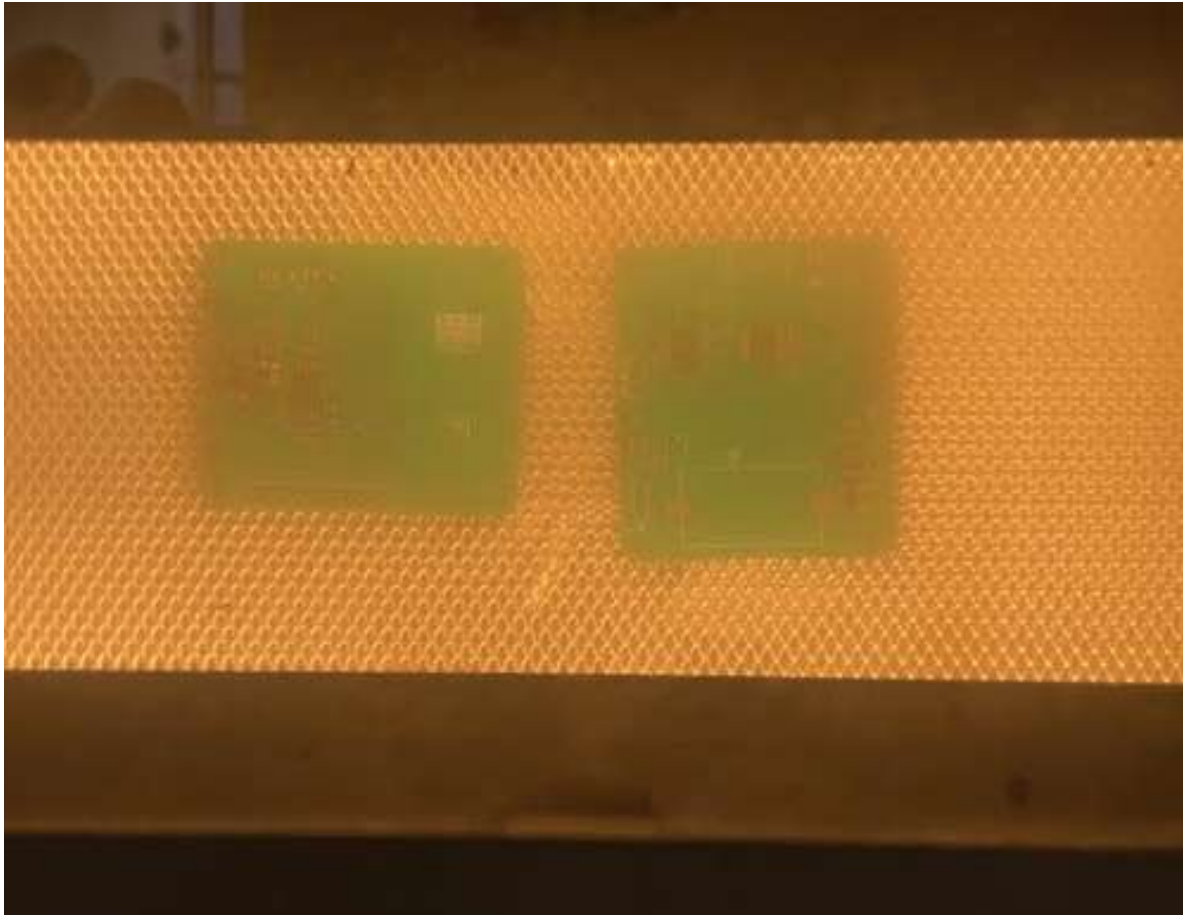
La deuxième étape consiste à placer les composants sur la carte. Pour m'aider dans cette tâche, j'ai utilisé un plugin "Interactive html boom" téléchargeable pour KiCad qui facilite le positionnement des composants sur les couches **front** (dessus) et **bottom** (dessous). Ce plugin m'a permis de placer rapidement et avec précision tous les composants, comme le montrent les images ci-dessous.



La troisième étape consiste à vérifier le bon placement des composants à l'aide d'une loupe, afin de s'assurer qu'ils sont correctement positionnés et bien en contact avec la pâte à braser sur la carte.



Après avoir placé et vérifié tous les composants, j'ai inséré soigneusement la carte dans le four, réglé à 250 °C, pendant 2.50 seconde minutes.



Enfin, pour la dernière étape, j'insère minutieusement la carte dans le four à refusion, préalablement réglé à **230 °C**. La carte y reste pendant **2 minutes et 50 secondes**, ce qui permet à la pâte à braser de fondre et de souder correctement tous les composants. Le résultat est visible sur les photos ci-dessous.



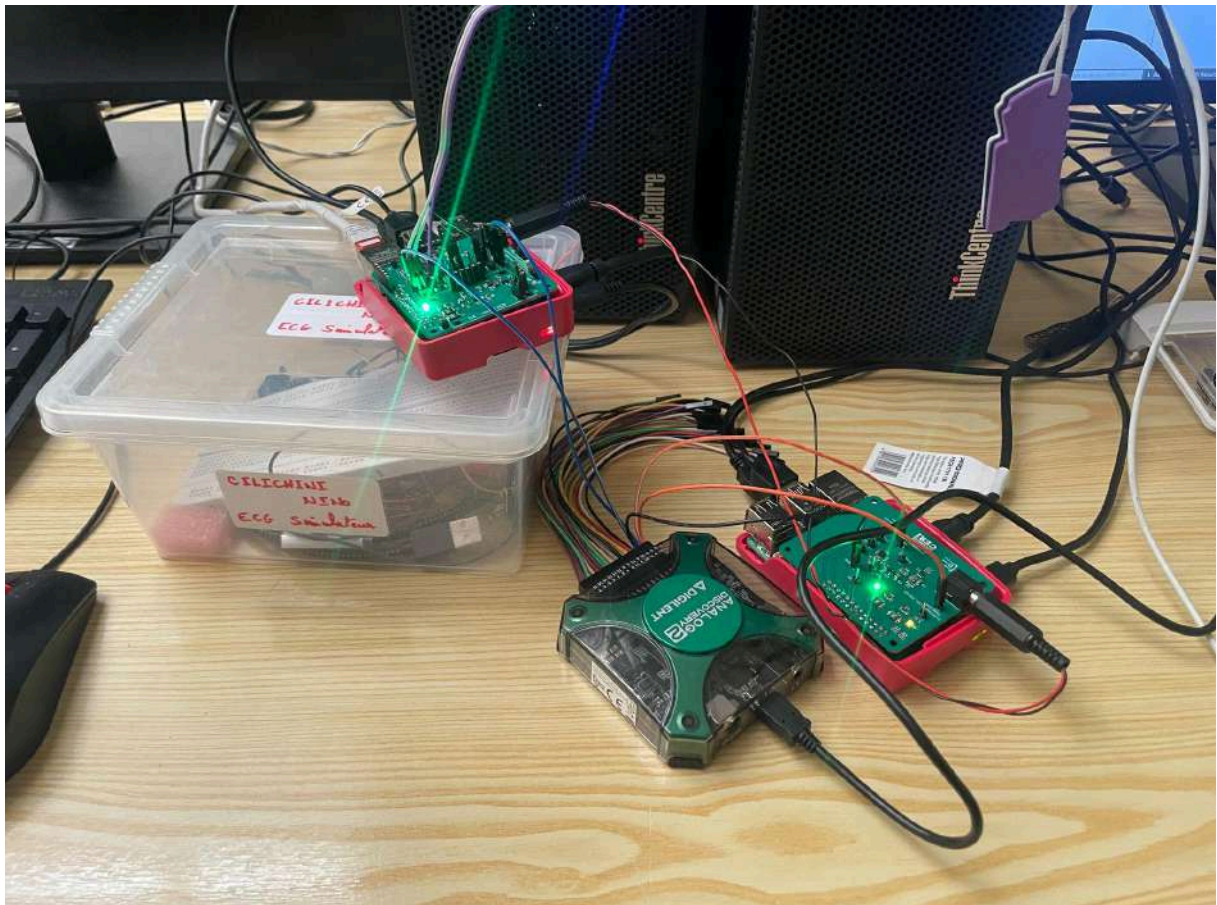
Cette méthode de fabrication permet d'obtenir une carte propre, fonctionnelle et bien soudée. Le respect de chaque étape, de la préparation à la refusion, garantit une bonne fiabilité du montage et réduit considérablement les risques de défauts liés à la soudure des composants **CMS**.

Après avoir vérifié le bon fonctionnement de ma carte à l'aide de la fiche utilisateur que j'ai rédigée, j'ai contrôlé les deux circuits d'alimentation séparés. J'ai ensuite injecté un signal provenant du **simulateur Arduino** mis à disposition par le professeur du CERI. Par la suite, j'ai fabriqué un câble jack permettant de récupérer le signal émis par la **carte du simulateur** de mon collègue ER. Afin de visualiser le signal transmis, j'ai connecté un oscilloscope entre la sortie de son simulateur et l'entrée de ma carte. Les programmes des deux IR concernés (la partie Holter et le simulateur) ont ainsi pu être testés. Vous pouvez voir le résultat ci-dessous.

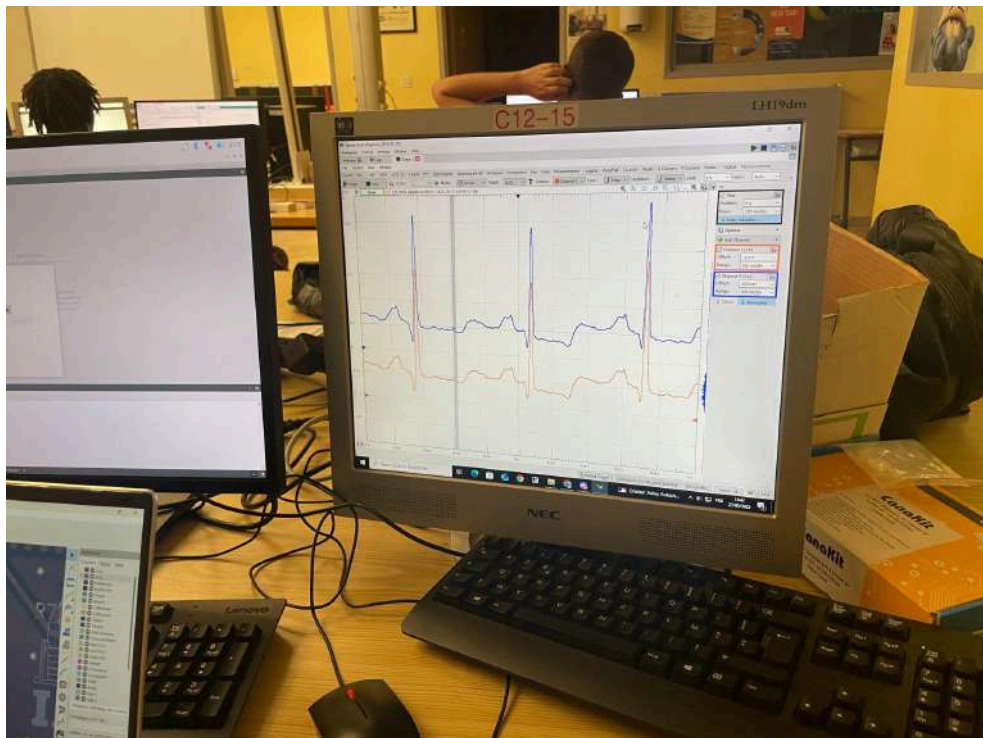
Jack male male



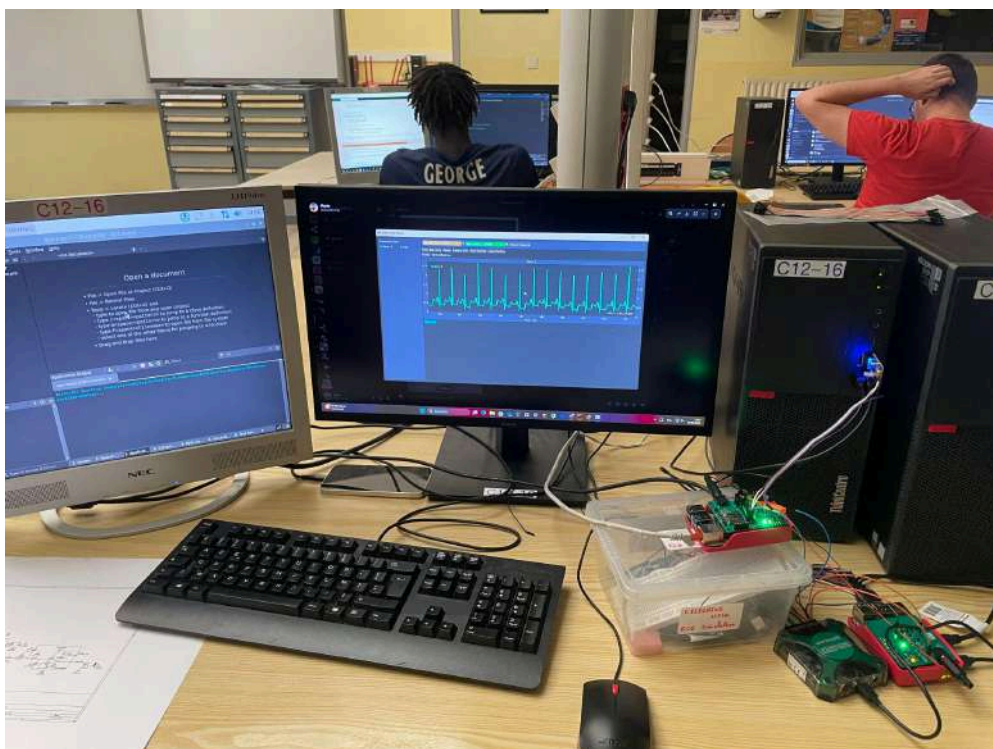
Branchement des deux carte avec cable jack:



Oscilloscope entrée et sortie de la carte simulateur



Reception signal sur la ma carte holter



systeme operationnel



Résumé du projet:

Dans le cadre de ce projet, j'ai conçu une carte électronique pour un Holter ECG connecté, ce qui m'a permis de mettre en pratique et d'approfondir de nombreuses compétences techniques. J'ai appris à comprendre le fonctionnement de composants spécifiques comme le capteur de pouls, l'amplificateur isolé ou encore le convertisseur analogique-numérique, ainsi qu'à les intégrer dans un système cohérent. Ce projet m'a également permis de découvrir concrètement comment router une carte électronique avec KiCad, générer les fichiers Gerber, passer commande pour la fabrication du PCB, et effectuer moi-même la soudure des composants CMS en respectant les étapes de fabrication (pâte à braser, placement, refusion). J'ai aussi renforcé ma capacité à appliquer la théorie vue en cours à un cas réel, notamment en analysant les signaux via oscilloscope ou en calculant les résistances et tensions nécessaires. Ce projet m'a beaucoup appris sur la rigueur, l'autonomie et l'interdisciplinarité, en lien étroit avec les contraintes de la sécurité médicale.

Partie Personnelle
Nino CILICHINI
(ER 1)

ECG : SIMULATEUR (ER)

Nino CILICHINI

Lycée Alphonse Benoît

Mon projet consiste à schématiser, router et cabler une carte électronique qui fera office de simulateur, ce dernier comportera plusieurs composants afin d'exploiter par la suite de manière visuelle la représentation d'un rythme cardiaque correspondant à des anomalies provenant de mesures réalisées sur de vrais patients dans le domaine médical (cardiologie).

Cependant je m'occupe essentiellement de la partie électronique car la partie informatique est gérée par un autre étudiant avec qui, malgré les différentes connaissances, je suis souvent en relation afin de confectionner et perfectionner ce simulateur.

DIAGRAMME DE GANTT :

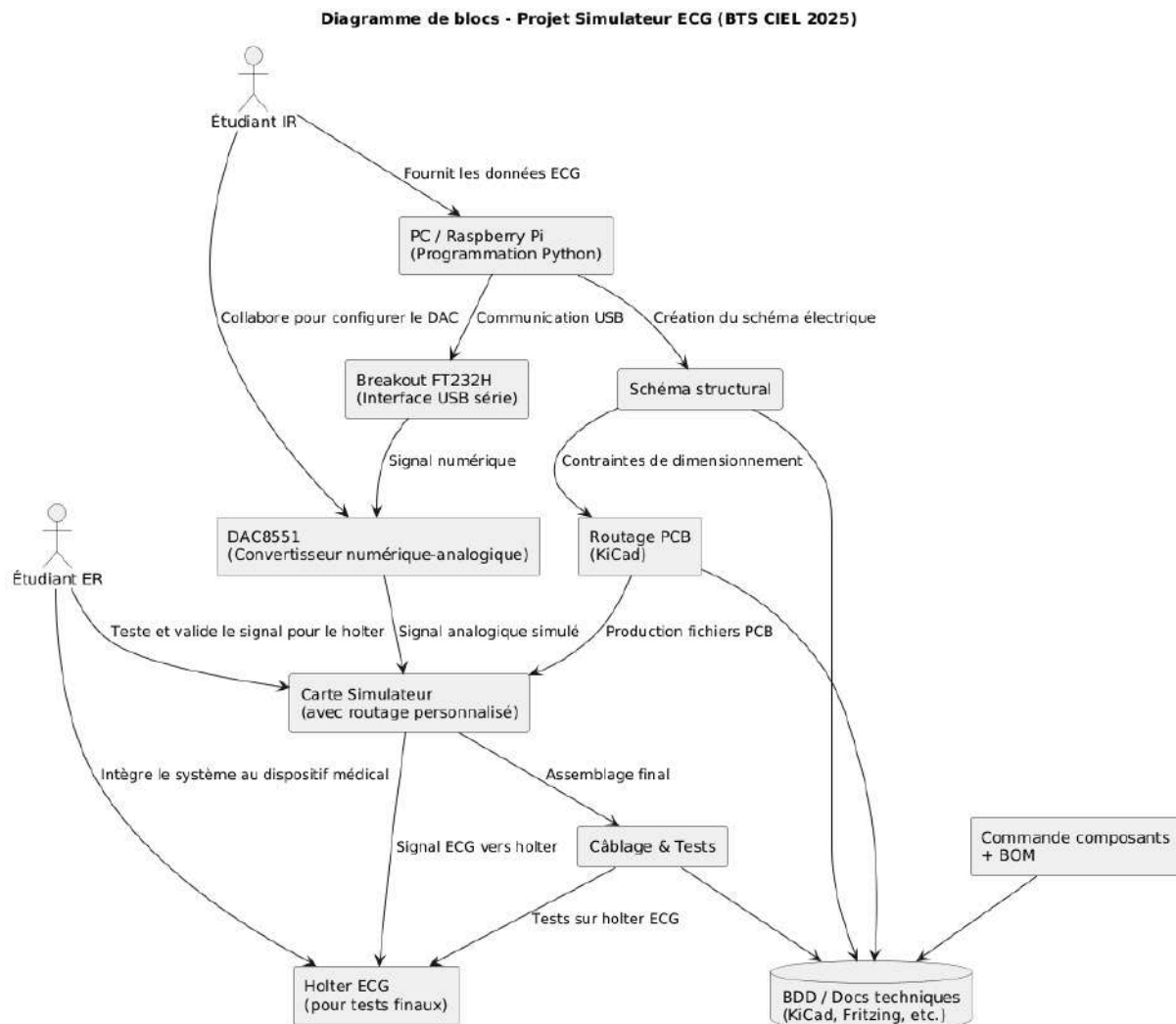
Prévisionnel :

▲ Evaluations du projet	Lun 03/02/25	Mar 27/05/25		
Revue 1	Lun 03/02/25	Mer 05/02/25		
Revue 2	Lun 31/03/25	Mer 02/04/25		
Dépôt dossier techniq	Mar 27/05/25	Mar 27/05/25		
Soutenance finale E62 (Revue 3)	Mar 27/05/25	Lun 09/06/25		
Epreuve E4	Mer 14/05/25	Mer 14/05/25		
Epreuve E1	Mar 13/05/25	Mar 13/05/25		
Vacances d'hiver	Sam 08/02/25	Dim 23/02/25	12	
Vacances printemps	Sam 05/04/25	Lun 21/04/25	14	
▲ Projet	Lun 06/01/25	Ven 13/06/25		
Prise de connaissance du dossier du projet.	Lun 06/01/25	Mar 07/01/25		
Commencer les premiers tests avec les composants mis a disposition.	Lun 13/01/25	Mar 04/02/25	11	
Améliorer les essais et essayer de produire un schéma correct.	Lun 24/02/25	Mar 25/02/25	8	
Effectuer le routage tout en respectant les différents conseils.	Lun 03/03/25	Mar 01/04/25		
Cabler la carte et effectuer les premiers essais.	Mar 22/04/25	Mar 06/05/25	9	
Tester la carte avec les coéquipiers pour s'assurer de son fonctionnement.	Lun 12/05/25	Lun 26/05/25	15	
Finir le dossier personnel.	Mar 27/05/25	Mar 10/06/25	16	
Fichier de fabrication PCB finalisés	Mar 22/04/25	Mar 22/04/25		
Rédaction du dossier	Lun 06/01/25	Mar 24/06/25		
1er BTS Blanc	Jeu 06/02/25	Jeu 06/02/25		
2nd BTS Blanc	Jeu 03/04/25	Jeu 03/04/25		
Soutenance E61	Lun 17/03/25	Mar 18/03/25		

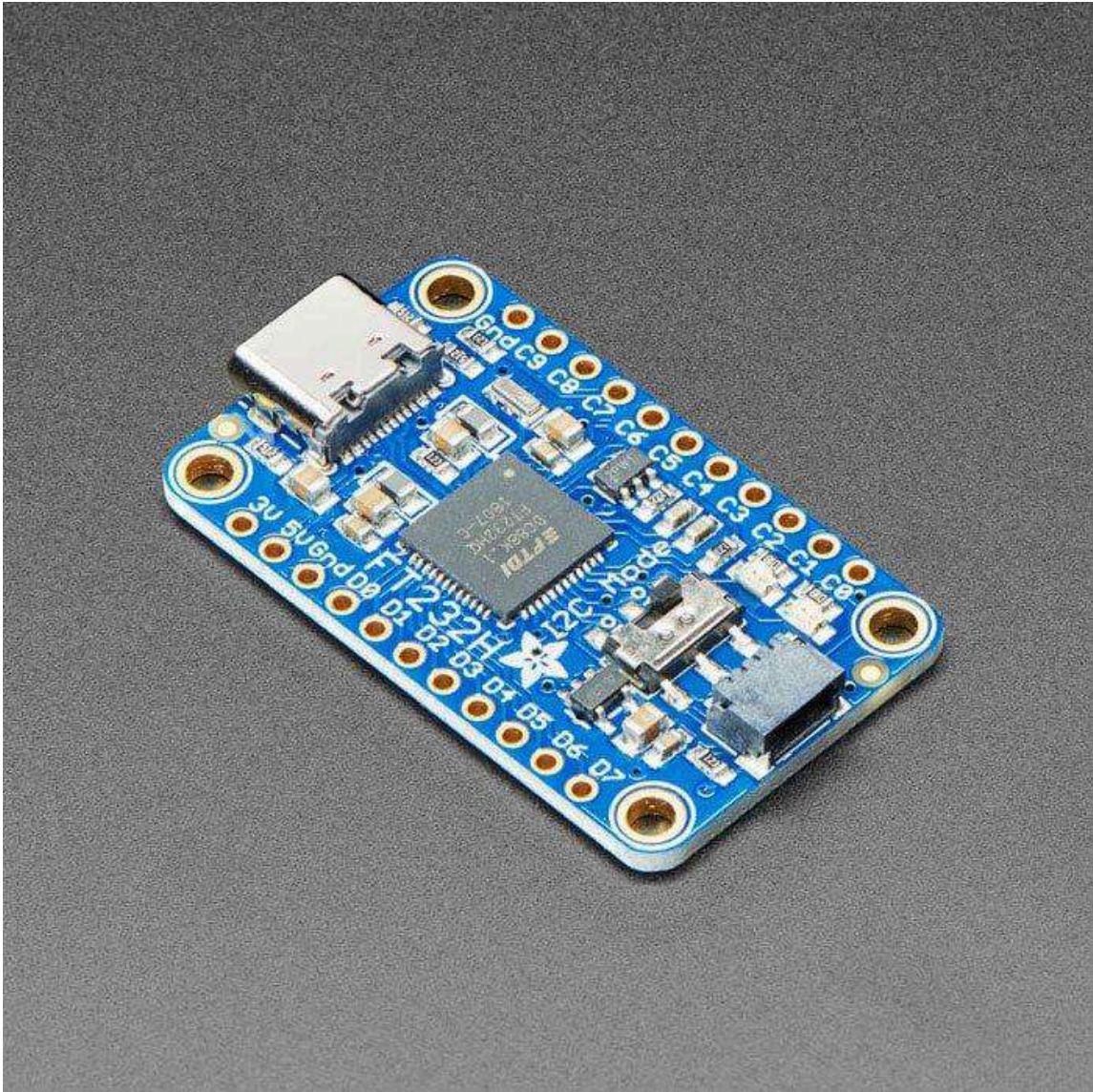
Réal :

Epreuve E1	Mar 13/05/25	Mar 13/05/25		
Vacances d'hiver	Sam 08/02/25	Dim 23/02/25	12	
Vacances printemps	Sam 05/04/25	Lun 21/04/25	14	
▲ Projet	Lun 06/01/25	Ven 13/06/25		
Specifications générales, prise de connaissance du dossier du projet ainsi que du travail des coéquipiers.	Lun 06/01/25	Mar 07/01/25		
Analyser, corriger, cabler et tester les différents composants comprenant le Breakout, le DAC ainsi que l'amplificateur opérationnel.	Lun 13/01/25	Mar 04/02/25	11	
Etudier plusieurs types de filtre passe bas afin de choisir le plus optimal puis le tester pour voir son fonctionnement. Produire le schéma structurel de l'ensemble.	Lun 24/02/25	Mar 25/02/25	8	
Effectuer le routage d'une première carte puis l'améliorer en respectant les règles de conformité ainsi que produire les fichiers afin que la fabrication du PCB soit sous-traitée.	Lun 03/03/25	Mar 01/04/25		
Cabler la carte et effectuer les premiers essais.	Mar 22/04/25	Mar 06/05/25	9	
Combiner la carte produite avec le travail des autres élèves afin de voir son bon fonctionnement avec le reste du projet.	Lun 12/05/25	Lun 26/05/25	15	
Documenter la mise en service de la carte finalisée ainsi que finir le dossier personnel.	Mar 27/05/25	Mar 10/06/25	16	
Fichier de fabrication PCB finalisés	Mar 22/04/25	Mar 22/04/25		
Rédaction du dossier	Lun 06/01/25	Mar 24/06/25		
1er BTS Blanc	Jeu 06/02/25	Jeu 06/02/25		

DIAGRAMME DE BLOCS/EXIGENCES :



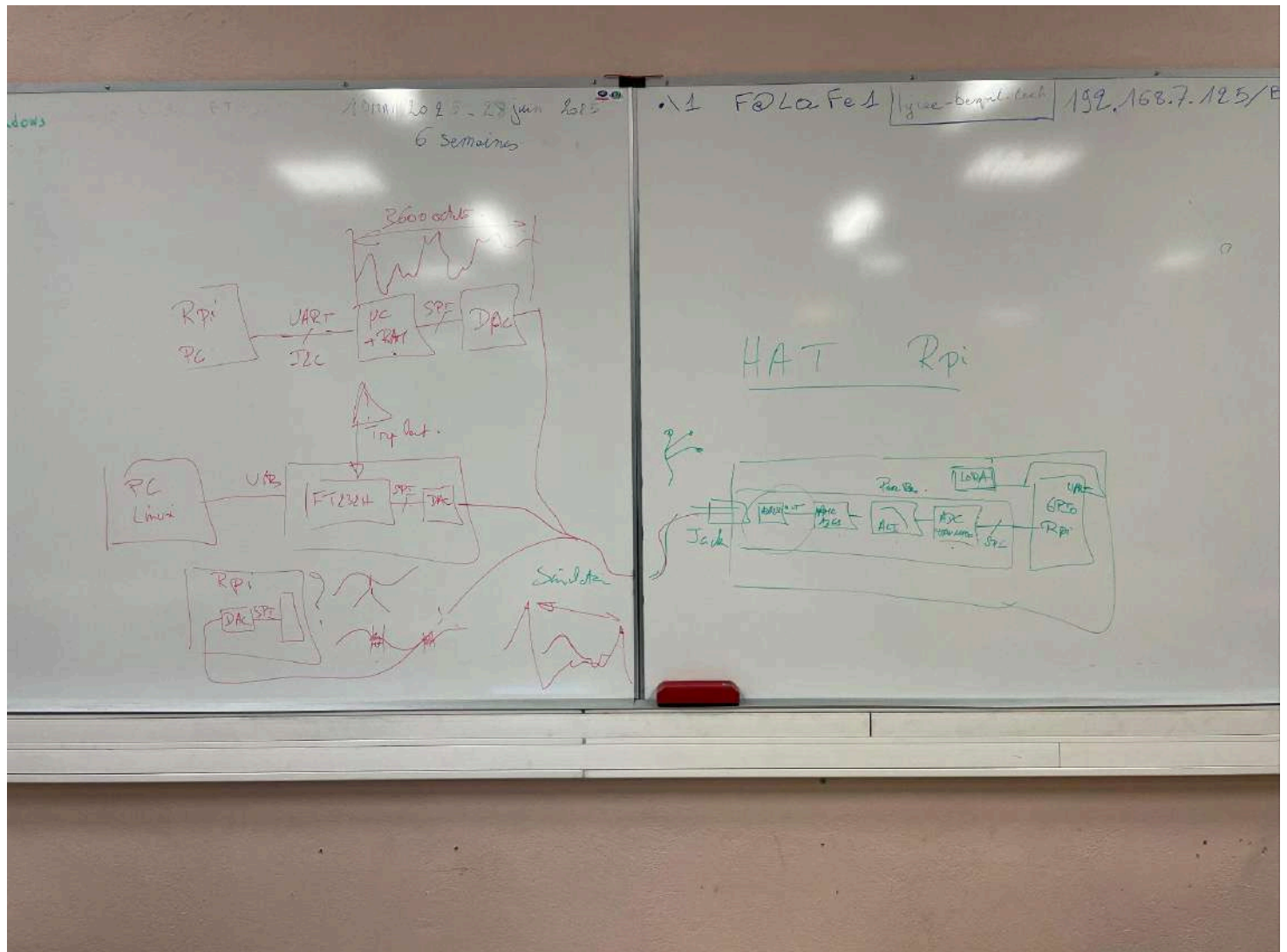
La première idée de ce projet aurait été de partir sur ce Breakout : le FT232H, une carte d'expérimentation qui correspondait parfaitement à la simplicité et objectif du simulateur,



Cependant nous nous sommes rendu compte, que le microcontrôleur de ce dernier était trop lent, et qu'il était donc impossible de gérer les 360Hz soit les 3600 octets en 10 secondes.

Nous avons donc cherché d'autres solutions plus optimales, comme une solution incluant un Raspberry Pi beaucoup plus rapide.

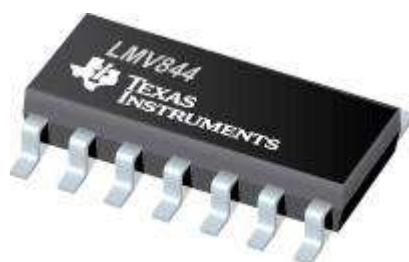
LES DIFFÉRENTES OPTIONS (HYPOTHÈSE) :



Comme dit précédemment nous avons donc choisi la 3eme solution du tableau avec le Raspberry Pi car cela permettait de répondre à nos besoins sans trop de complications, néanmoins il est bon de préciser que la dernière option s'offrant à nous aurait été encore plus précise mais beaucoup plus complexe.

Nous utiliserons donc une Raspberry, en association avec un convertisseur numérique analogique, un amplificateur opérationnel ainsi qu'un filtre passe bas afin de lisser le bruit et les parasites.

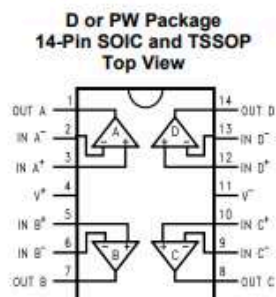
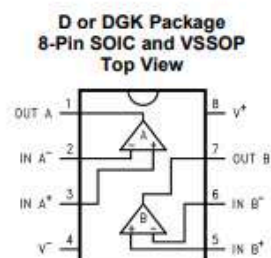
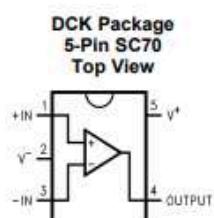
Voici l'amplificateur qui gère les données analogiques en sortie du DAC.



www.ti.com

LMV841, LMV842, LMV844
LMV841-Q1, LMV842-Q1, LMV844-Q1
SNOSAT1H – OCTOBER 2006 – REVISED JULY 2016

5 Pin Configuration and Functions



Pin Functions

PIN		DESCRIPTION
NAME	I/O.	
+IN	I	Noninverting Input
-IN	I	Inverting Input
OUT	O	Output
V+	P	Positive Supply
V-	P	Negative Supply

Et voici le Convertisseur numérique analogique (DAC).

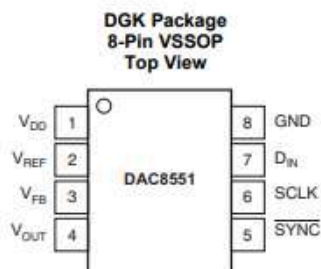


DAC8551

www.ti.com

SLAS429E – APRIL 2005 – REVISED JUNE 2017

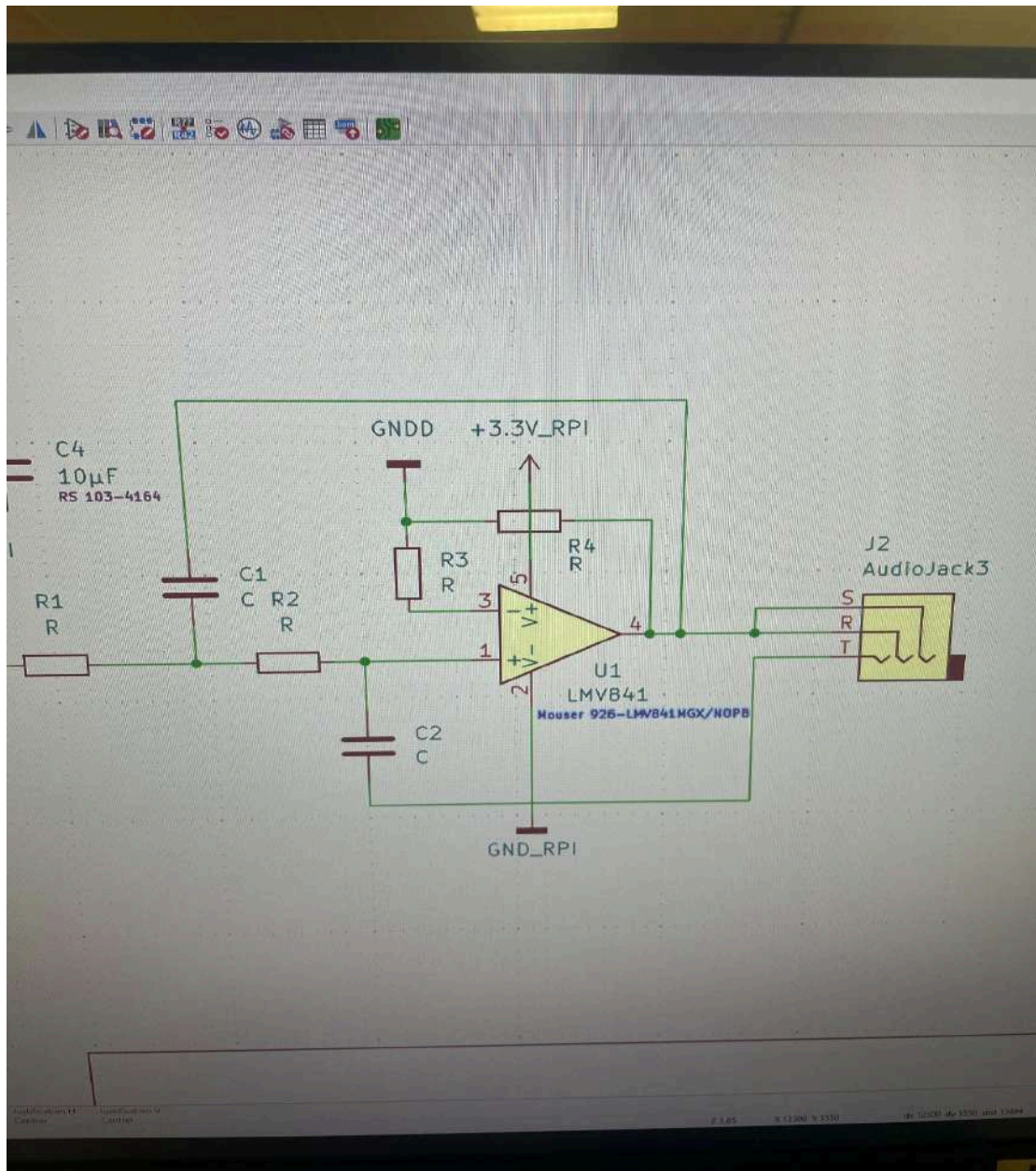
5 Pin Configuration and Functions



Pin Functions

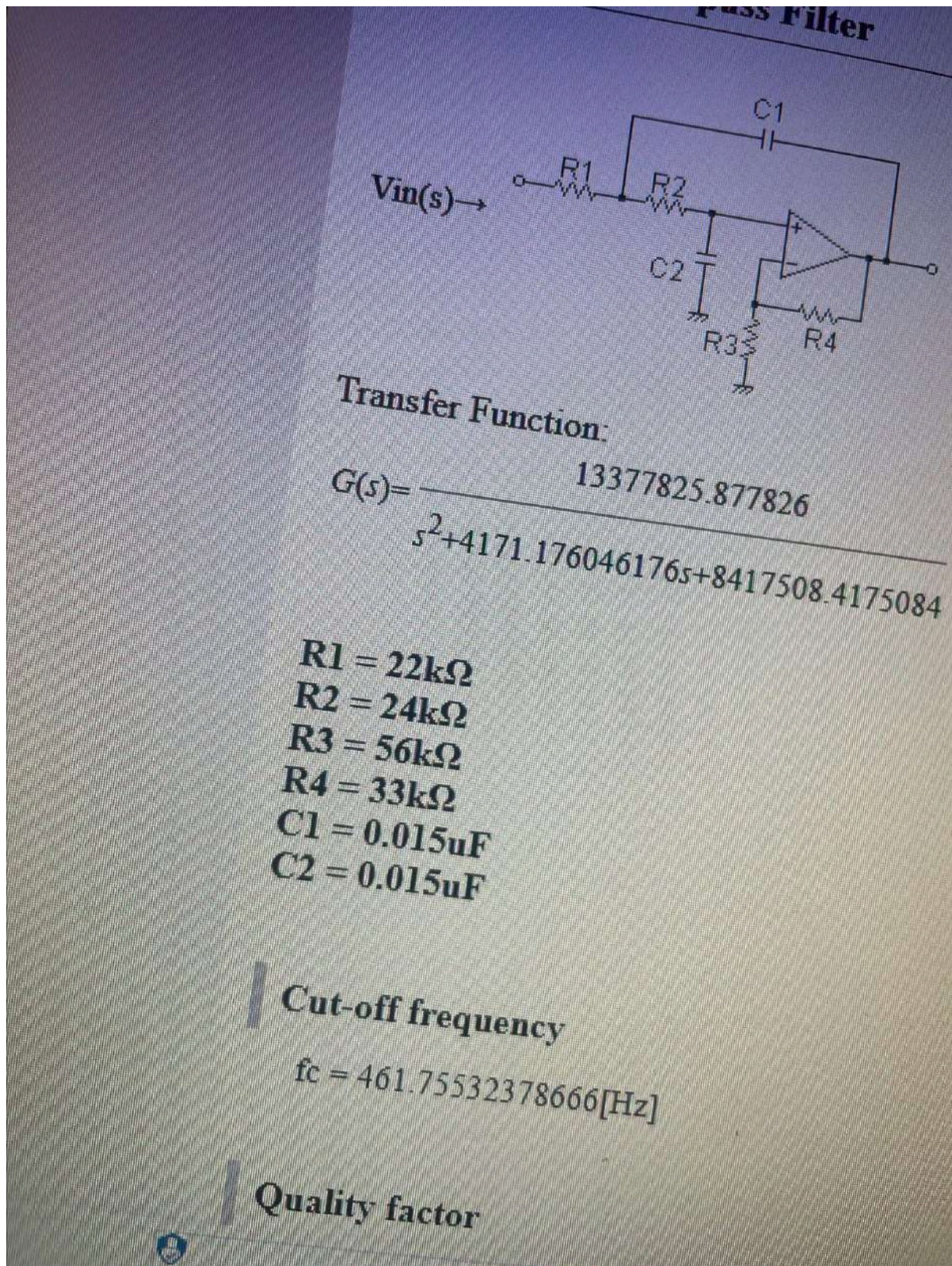
PIN		TYPE	DESCRIPTION
NAME	NO.		
D _{IN}	7	I	Serial data input. Data is clocked into the 24-bit input shift register on each falling edge of the serial clock input. Schmitt-Trigger logic input.
GND	8	GND	Ground reference point for all circuitry on the part
SCLK	6	I	Serial clock input. Data can be transferred at rates up to 30-MHz Schmitt-Trigger logic input.
SYNC	5	I	Level-triggered control input (active LOW). This is the frame synchronization signal for the input data. When SYNC goes LOW, it enables the input shift register and data is transferred in on the falling edges of the following clocks. The DAC is updated following the 24th clock (unless SYNC is taken HIGH before this edge, in which case the rising edge of SYNC acts as an interrupt and the write sequence is ignored by the DAC8551). Schmitt-Trigger logic input.
V _{DD}	1	PWR	Power supply input, 2.7 V to 5.5 V
V _{FB}	3	I	Feedback connection for the output amplifier. For voltage output operation, tie to V _{OUT} externally.
V _{OUT}	4	O	Analog output voltage from DAC. The output amplifier has rail-to-rail operation.
V _{REF}	2	I	Reference voltage input

Voici les premiers essais du filtre passe bas :



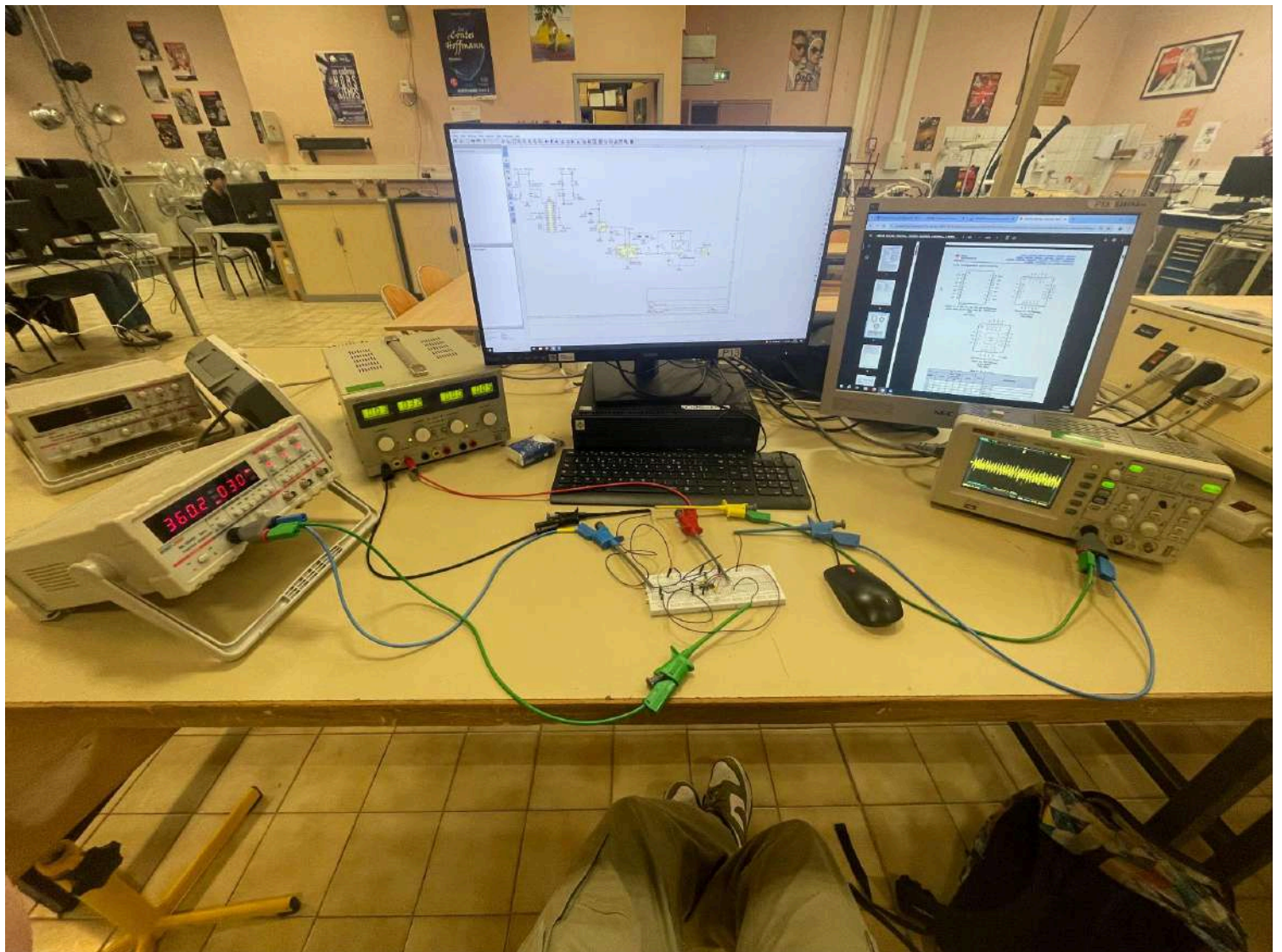
Il est tout de même bon de préciser que nous avons utilisé une structure pour le filtre dites de Sallen Key, elle possède le moins d'inconvénient comparé à son frère la structure dites de Rauch, car ce dernier aurait dû s'alimenter avec une alimentation symétrique (positif et négatif) et le problème majeur de cette structure est qu'elle abaisse obligatoirement l'amplitude du signal ce qui n'est pas optimal dans notre cas, le filtre passe bas de Sallen Key est donc beaucoup plus optimal dans notre système.

J'ai donc utiliser un calculateur en ligne afin de récupérer des valeurs expérimentales des composants qui compose le filtre :

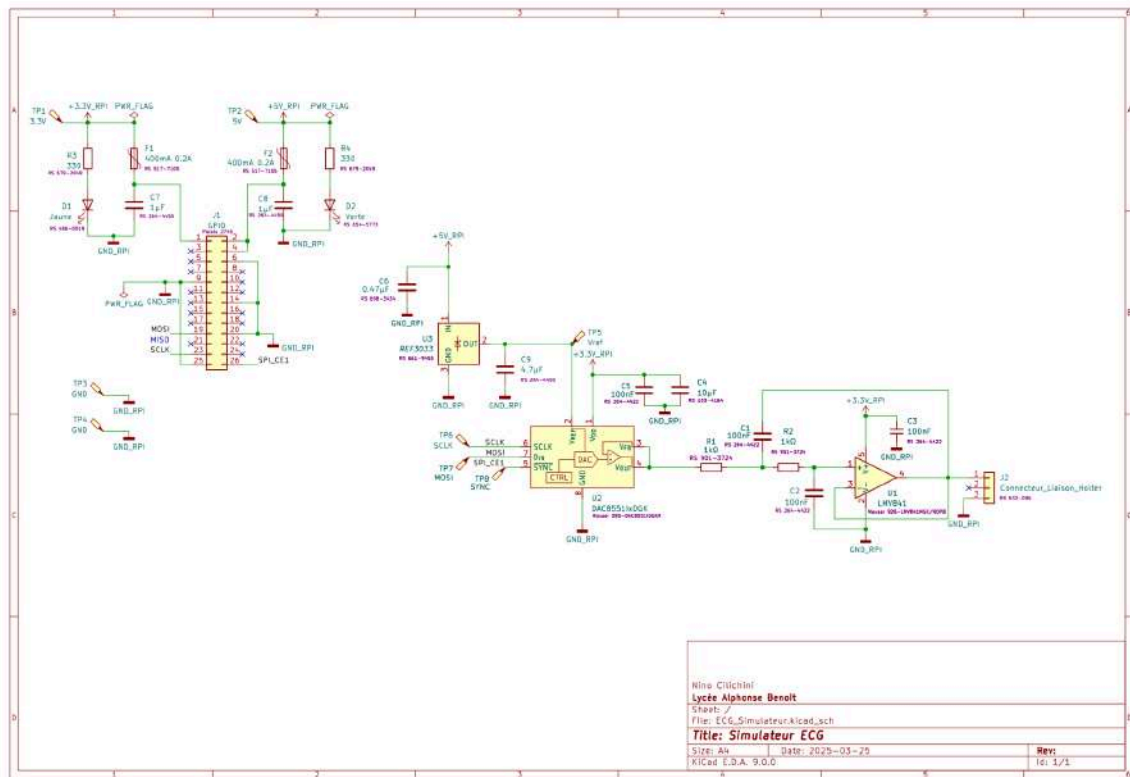


<http://sim.okawa-denshi.jp/en/OPseikiLowkeisan.htm>

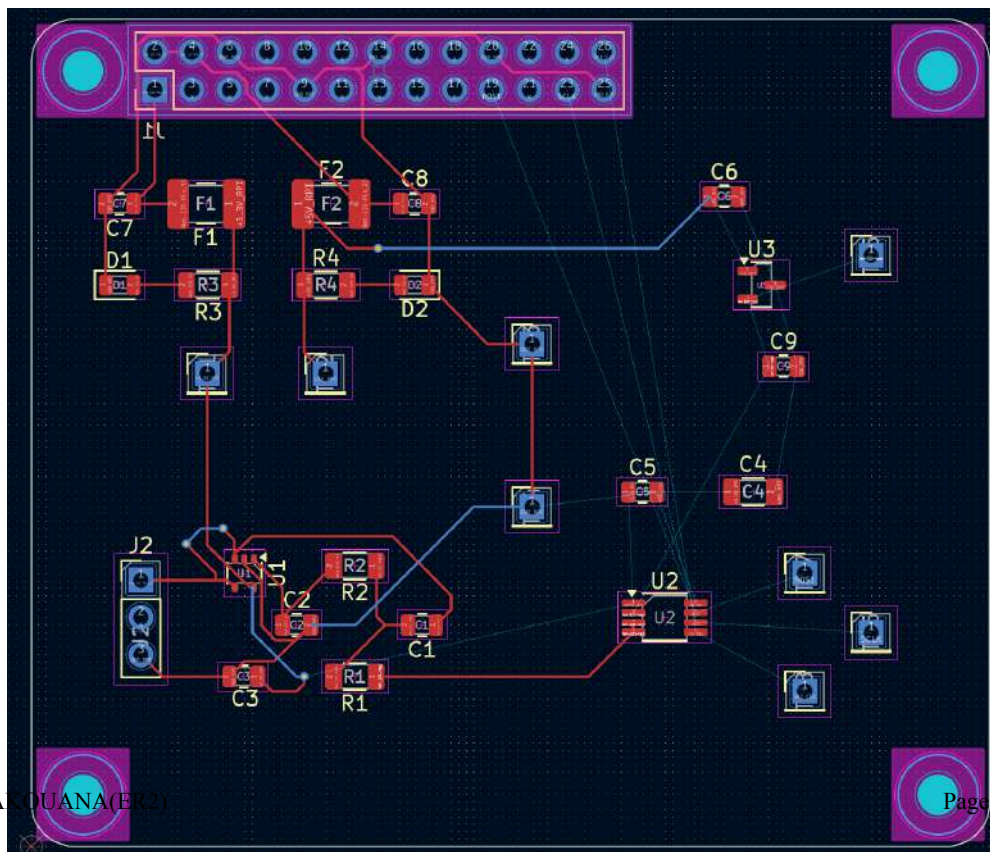
Premier test du filtre à l'aide d'un GBF qui envoie un signal analogique de 360hz simulant l'envoi d'octets, une alimentation continue pour le circuit et un oscilloscope afin de repérer les anomalies :



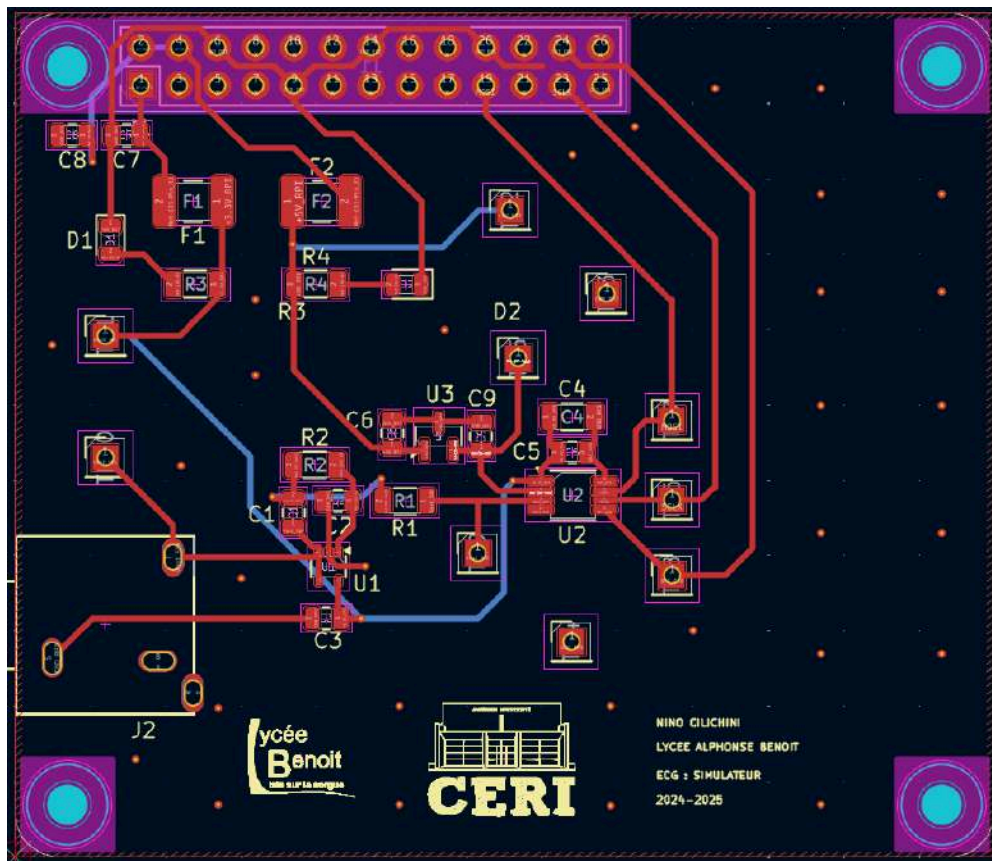
Pour finir et pour l'instant voici le schéma du simulateur avant de commencer le routage de ce projet :



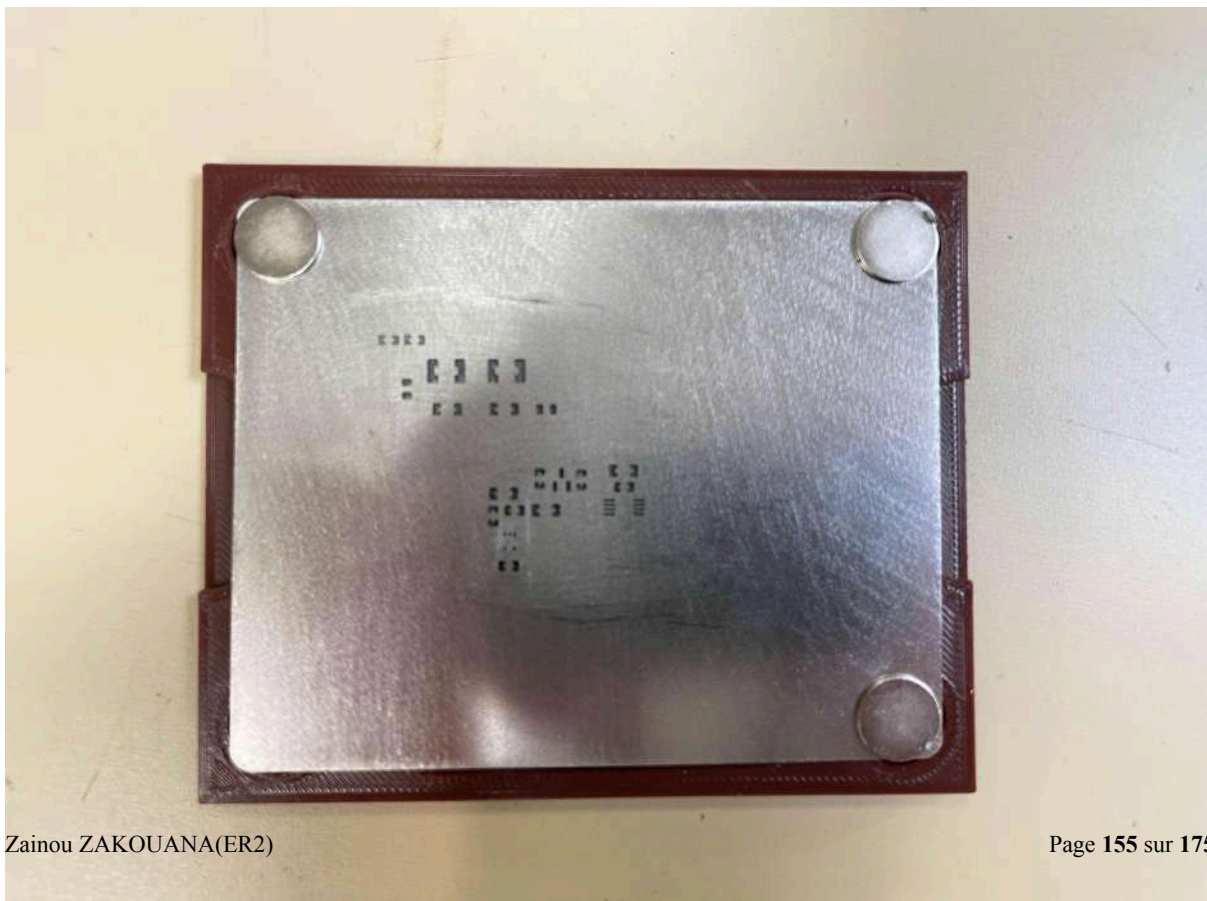
Début du routage :



Fin du routage :



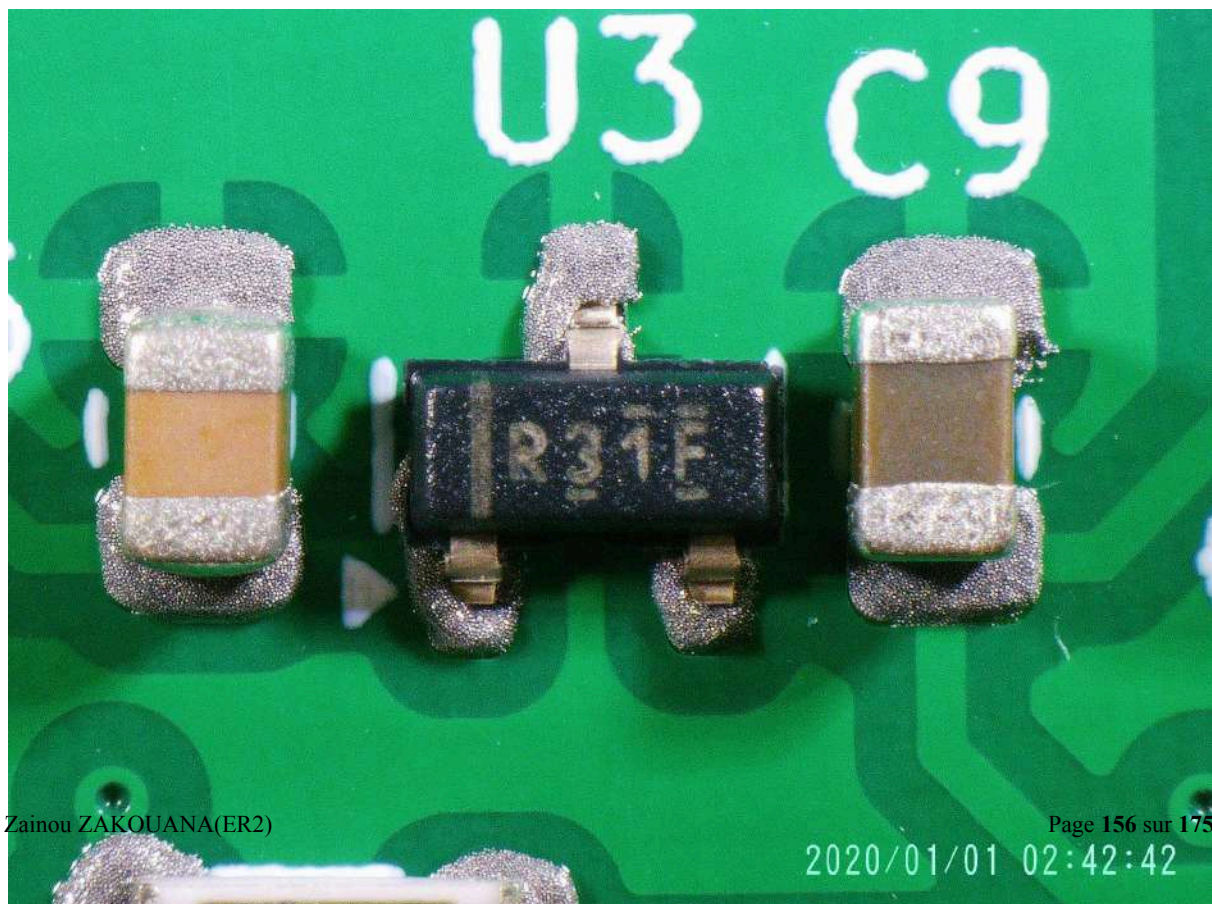
Début du câblage de la carte avec mise en place de la pâte à braser à l'aide du stencil ou pochoir :





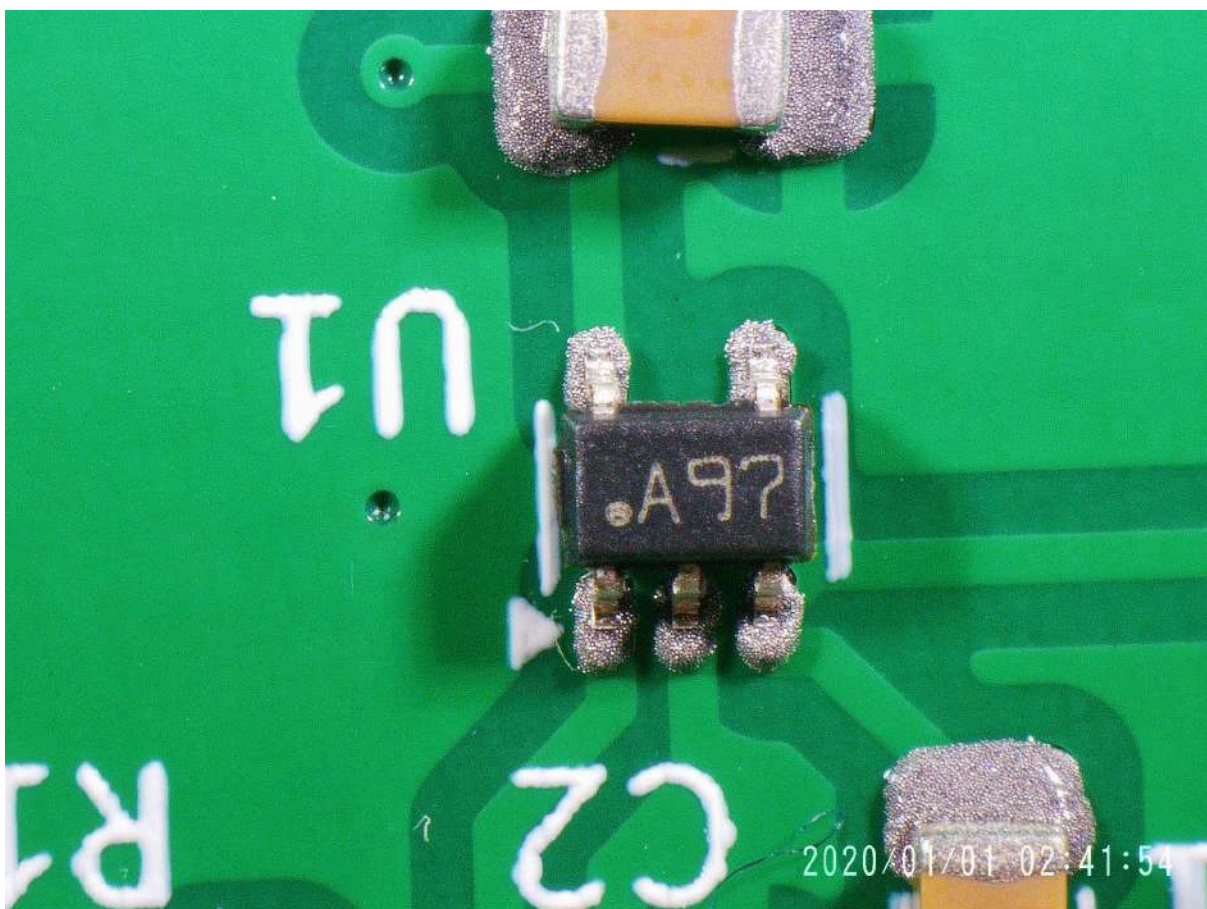
Mise au point sur les 3 composants primaires constituant la carte :

Référence de tension 3,3V :



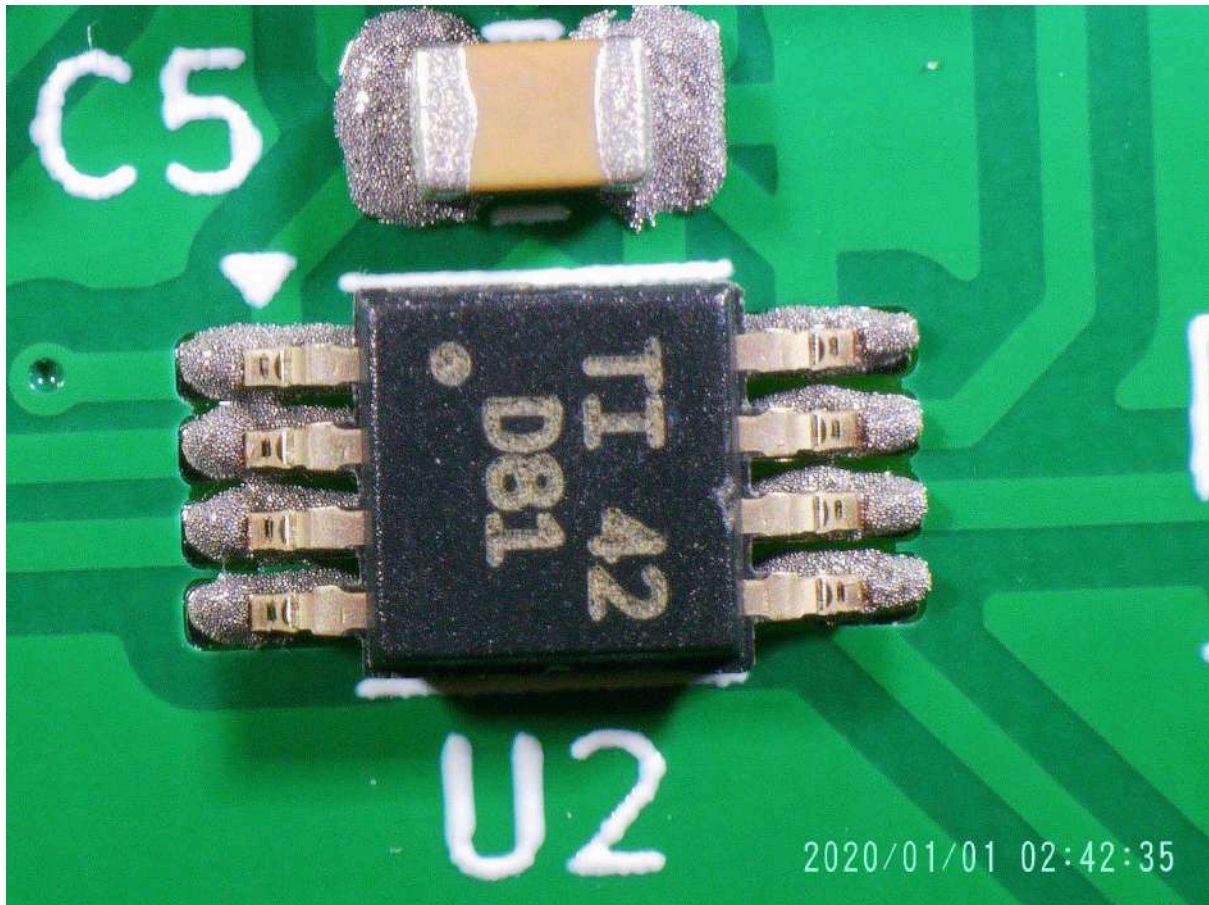
Ce composant joue un rôle fondamental dans la précision et la stabilité du signal analogique généré par le DAC car une variation de la tension d'alimentation du DAC entraîne une variation proportionnelle du signal analogique, cela pourrait donc modifier le quantum de ce dernier (le pas, plus petit écart de tension), il lui faut donc le système le plus précis et stable.

Amplificateur opérationnel :



Ce composant joue un rôle majeur dans le système de filtrage, sa bonne bande passante et sa linéarité permettent de l'utiliser dans des filtres passe-bas, il est aussi adapté pour amplifier les signaux très faibles issue du DAC.

Convertisseur numérique analogique (DAC) :

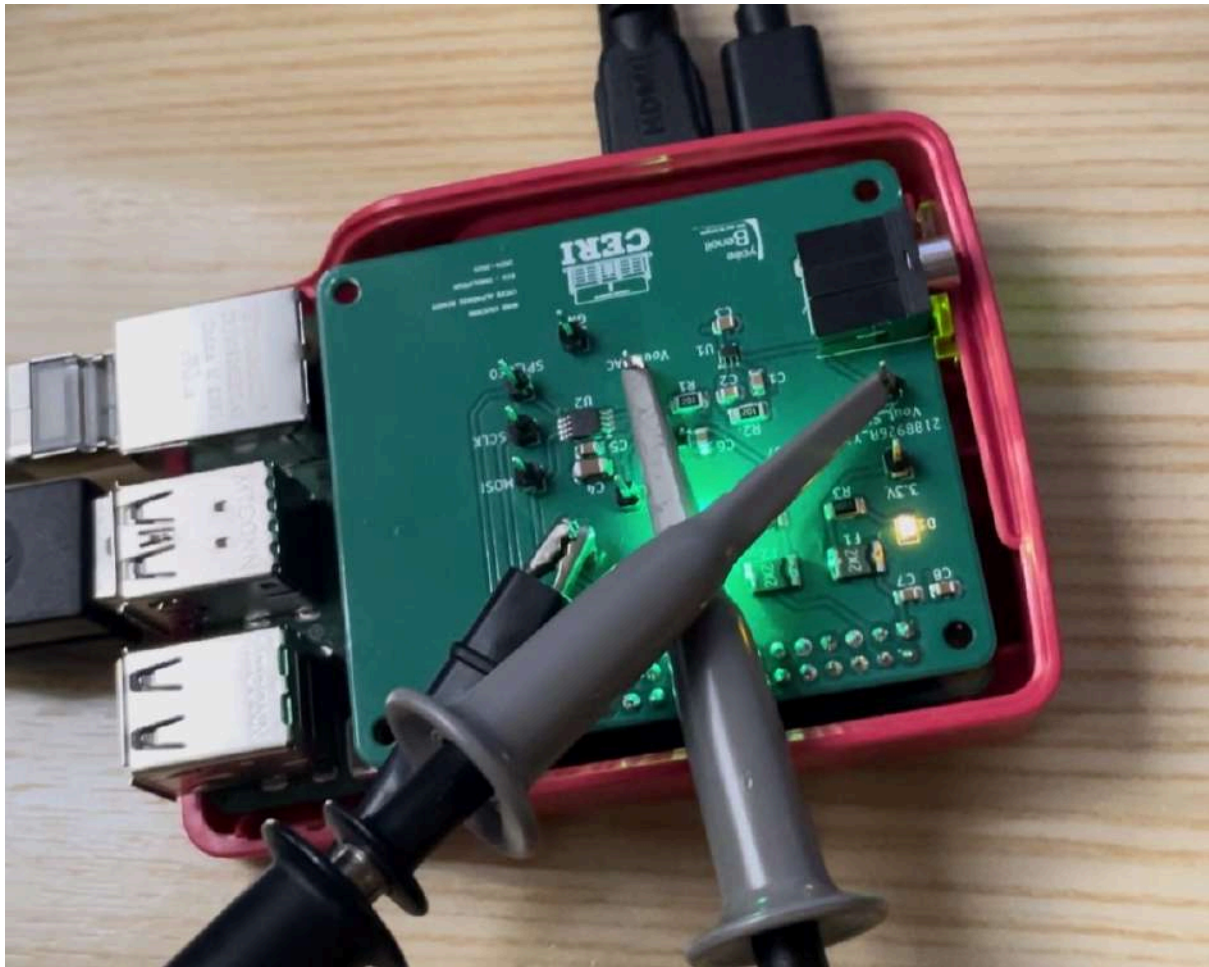


Ce convertisseur est un monocanal, il possède une résolution de 16 bits soit de 65536 valeurs, une interface SPI avec laquelle on communique avec le Raspberry Pi, ainsi qu'une référence de tension externe pour une meilleure précision et stabilité (cité précédemment).

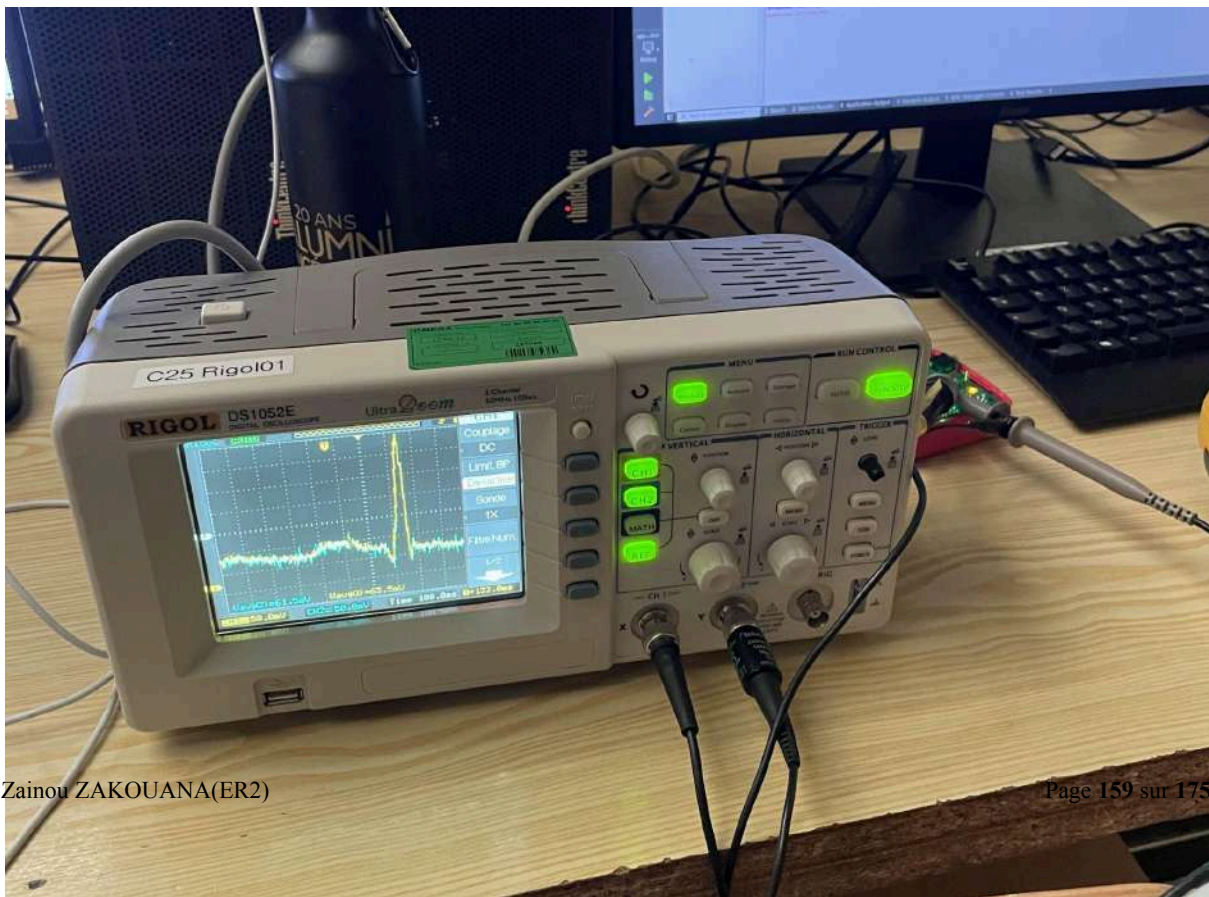


Analyse logique de ce dernier à l'aide d'Analog Discovery, on peut y voir l'horloge ainsi que les données envoyées par le bus SPI.

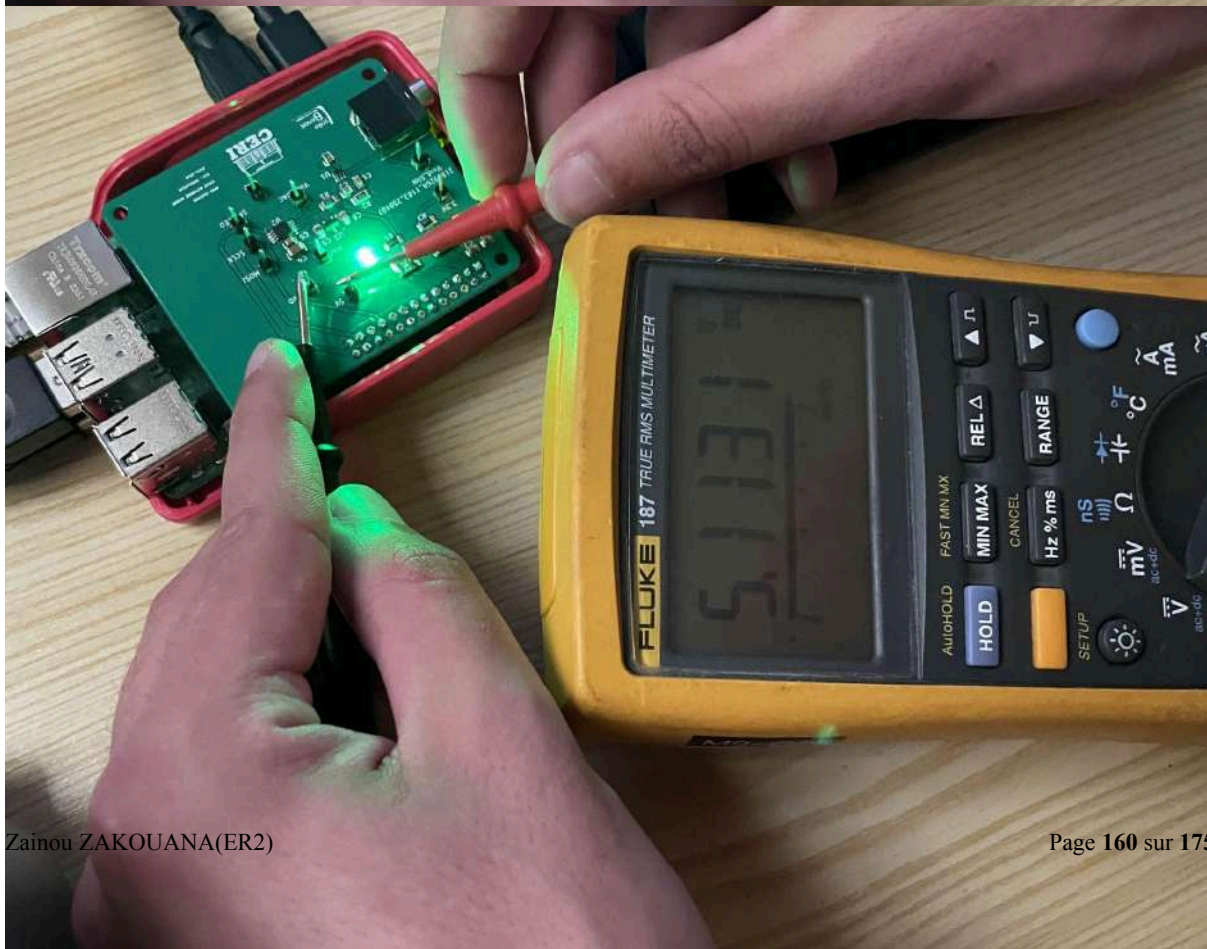
Premier test après fabrication de la carte avec l'étudiant IR :



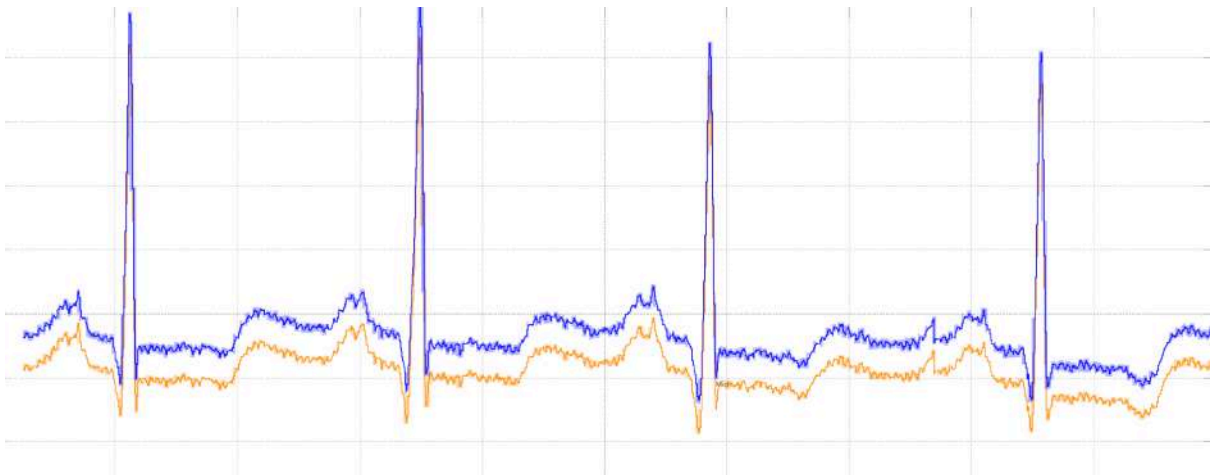
Visualisation du signal à l'aide d'un oscilloscope :



Vérification des tensions principales 3,3V et 5V à l'aide des points de test :



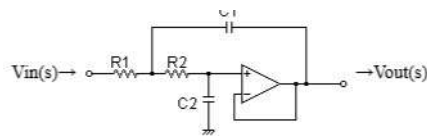
Après vérification et analyse du signal, nous nous sommes rendu compte que les valeurs utilisées pour les composants qui constituaient le filtre passe bas étaient surement mauvaise,



En bleu, le signal en sortie du DAC et en orange le signal normalement filtré, comme on peut voir, il n'y a aucun changement entre les deux signaux, cela veut dire que le filtre passe bas ne joue pas son rôle et qu'il faut donc le réétudié afin de comprendre cette erreur.

Cela étant dit et par raison de simplicité pour ce dernier, nous avons utilisé pour composer ce filtre passe bas, 100nF pour les deux condensateurs et 1k Ohm pour les deux résistances, sachant donc leur valeur, on peut donc s'aider du calculateur en ligne utilisé lors de la phase expérimental pour observer sa fréquence de coupure théorique.

Sauf qu'après calcul la fréquence de coupure actuelle est probablement bien trop grande pour le signal résultant de son inutilité (Fréquence de coupure de 1591 Hz) :



Transfer Function:

$$G(s) = \frac{100000000}{s^2 + 20000s + 100000000}$$

Cut-off frequency

$$f_c = 1591.549430919[\text{Hz}]$$

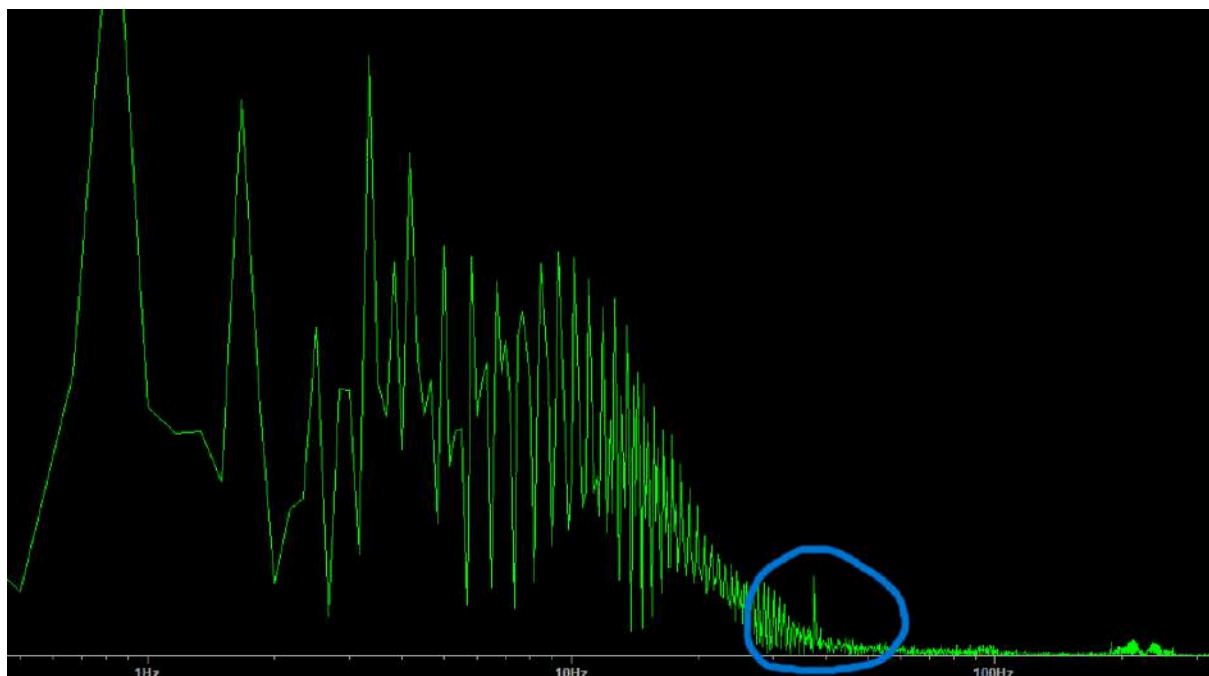
Quality factor

$$Q = 0.5$$

Damping ratio

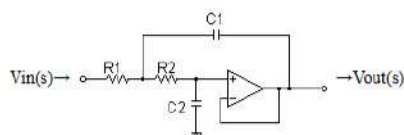
$$\zeta = 1$$

Cependant, pour comprendre cette anomalie, on a eu l'occasion d'utiliser le logiciel de simulation LTspice et de son option de transformation de fourier rapide (FFT) afin d'analyser en profondeur le signal entrant dans le filtre pour ensuite trouver sa fréquence de coupure optimale.



Comme on peut le voir entouré en bleu, voici la plage optimale de fréquence (autour de 40 Hz) qui serait en effet utile pour lisser le signal et ne laisser passer que l'allure de ce dernier et non son bruit/parasite.

Cependant comme dit précédemment le filtre actuel possède une fréquence de coupure bien trop élevée, il faut donc augmenter les valeurs des composants pour y augmenter son action de filtrage, on utilisera donc le calculateur en ligne afin de trouver la valeur exacte des composants à modifier.



Transfer Function:

$$G(s) = \frac{65746.219592373}{s^2 + 512.82051282051s + 65746.219592373}$$

Cut-off frequency

$$f_c = 40.808959767153 [\text{Hz}]$$

Quality factor

$$Q = 0.5$$

Damping ratio

$$\zeta = 1$$

Electric Vehicle Systems

Download Catalog. Configure Your Vehicle Now.
cascadiamotion.com

R1=39k Ω C1=100n F
R2=39k Ω C2=100n F
p:pico, n:nano, u:micro, k:kilo, M:mega

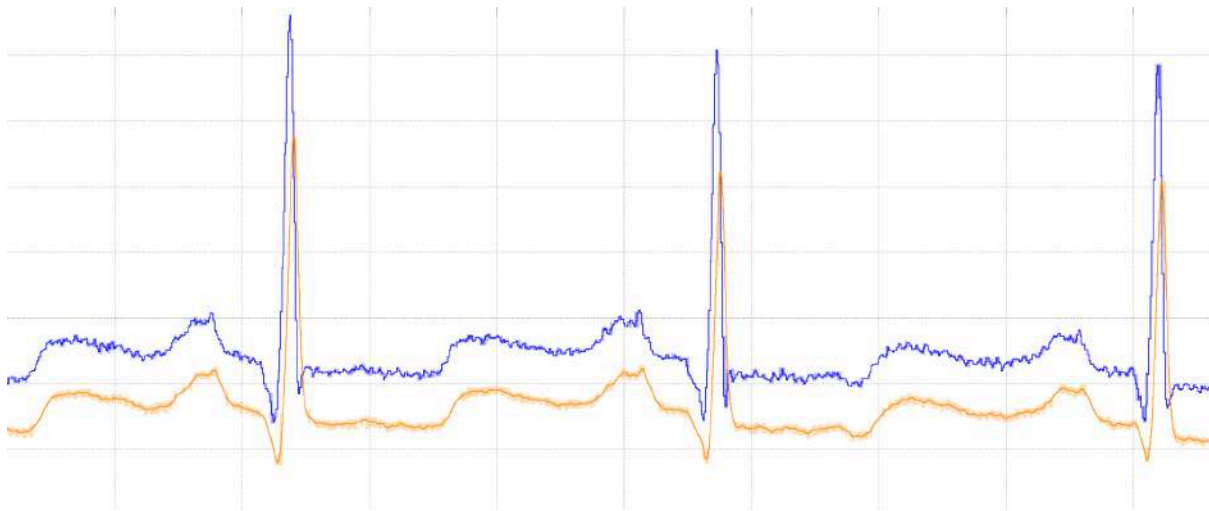
Frequency analysis

- ☒ Bode diagram
 - ☒ Phase ☐ Group delay
- ☒ Nyquist diagram
- ☒ Pole, zero
- ☒ Phase margin
- ☒ Oscillation analysis

Analysis on frequency range:
f1=0 ~ f2=10000 [Hz] (optional)

Nous avons donc changé les deux résistances par des résistances beaucoup plus grandes afin de baisser grandement la fréquence de coupure et atteindre les 40 Hz.

Actuellement les résistances ont été changées par les résistances de 39k Ohm, cependant il nous reste à ré analyser et visualiser le signal à l'aide de l'oscilloscope pour enfin voir le réel impact du filtre passe bas sur le signal simulé.



Pour finir notre hypothèse était donc juste car comme on peut observer dans le graphique le signal Orange correspondant à la sortie du filtre est contrairement au dernier montage réellement filtré, on y voit donc seulement son allure sans trop abaisser son amplitude, ce qui permettra aux étudiants suivant d'analyser précisément les différences entre les multiples anomalies en cardiologie.

MISE EN SERVICE :

- Connecter la carte au Raspberry Pi à l'aide de son hat.
- Brancher l'entièreté des connectiques incluant l'alimentation du Raspberry PI, les périphériques (clavier/souris) ainsi que le port vidéo (HDMI dans notre cas).
- Allumer la Raspberry puis en haut cliquer sur le logo Raspberry allez dans l'onglet programmation puis Qt Creator ensuite ouvrir projet ecg puis ECG, QT, app-simulateur et enfin app-simulateur.pro une fois sur l'interface cliquer sur la flèche verte en bas à gauche puis attendre que le programme se compile et ce lance puis ouvrir fichier choisir son anomalie cardiaque et lancer la simulation.

DÉPANNAGE/MAINTENANCE :

Au cas où la carte ne fonctionne pas, voici quelques conseils.

- Vérifier les deux alimentations du circuit, Le 5V et le 3,3V à l'aide d'un multimètre en le mettant en position tension continu puis à l'aide des pointes de touche coller la pointe noir sur un des GND (masse) de la carte et coller la pointe rouge sur une des tensions et pareil pour l'autre.
- Vérifier l'horloge du convertisseur à l'aide d'un analyseur logique ou tout simplement un oscilloscope, en suivant la méthode de votre outil (oscilloscope, ikalogic ou analog discovery).
- Vérifier la continuité à l'aide du multimètre en mettant les pointes sur les pastilles entre les différents composants et sinon vérifier les soudures car le brasage avec le four à refusion manque parfois d'une mauvaise quantité d'étain.

PRIX DE LA CARTE COMPLÈTE :

Nomenclature

[illegible]

Annexes

FICHE DE MAINTENANCE – ECG HOLTER

Nom du projet : ECG Carte Holter

Concepteur : Zainoulianbidina ZAKOUANA

Date de création : Mai 2025

Référence du schéma : kicad_Zainou.kicad_sch

Version carte : V1.0

Type de maintenance : Préventive / Corrective

Composants critiques et vérifications

Bloc / Composant	Fonction	Symptômes de panne	Méthodes de test	Actions de maintenance
SEN-12650	Capteur ECG	Signal absent ou bruité	Mesure tension OUT via oscillo	Vérifier connexions / remplacer capteur
LMV841	Amplification du signal ECG	Signal trop faible	Vérifier gain via oscillo	Contrôle alim / remplacement
MAX11100	Convertisseur analogique-numérique	Pas de signal numérique	Vérifier SCLK / MISO et CS via analyseur logic	Vérifier soudures, SPI, remplaçant
AMC1351-Q1	Isolation galvanique	Coms instables ou bruitées	Oscillo entre SW4 pin 1 et SW5 pin 1	Vérifier tension d'entrée/condensateur / remplacement
HTCC-AM01	Module LoRa (transmission)	Pas de données envoyées	UART LoRa test	Test communication / mise à jour firmware
REF3033 / K78xx	Régulation d'alimentation	Tension instable ou absente	Mesure tension de sortie doit être à 3.3	Vérifier alim / remplacement
LEDs D1-D5	Indicateurs d'état	Aucune indication visuelle	Test avec multimètre	Remplacement / vérifier résistances
Fusibles F1/F2	Protection	Carte totalement hors tension	Continuité avec multimètre	Remplacement immédiat
Boutons SW1-SW6	Contrôle utilisateur / reset / config	Aucune réponse	Vérification mécanique + ohmmètre	Nettoyage ou remplacement

Outils nécessaires pour la maintenance

- Multimètre numérique
- Oscilloscope
- Fer à souder + étain
- Station d'air chaud
- Alimentation stabilisée (3.3V / 5V)
- Logiciel de programmation LoRa / RPi
- Analyseur logique (UART / SPI)

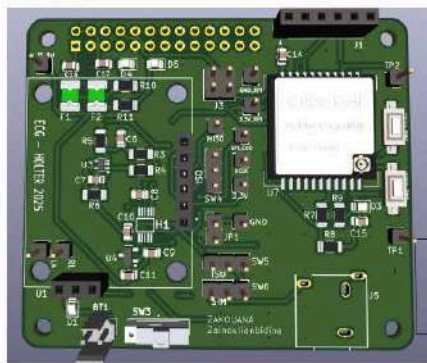
Fréquence recommandée

Type de vérification	Fréquence
Inspection visuelle	Tous les mois
Test électrique complet	Tous les 3 mois
Reprogrammation firmware	Selon évolution
Vérif capteurs	Avant chaque usage

Fiche utilisation carte holter:

Vérification alimentation:

- 3.3V du Module 8232
- 3.3 du TP7
- 3.3V_RPI
- 5V_RPI
- GND et GND_RPI
- Sdn 3.3V



Désignation	Schéma	Positionnement cavalier
Utilisation du simulateur externe (arduino): SW6 pin 3 relier au 2.		
Utilisation Patient avec isolation galvanique: SW4 pin 2 relier au 1, SW5 pin 1 relier au 2 et SW6 pin 1 relier au 2.		
Utilisation simulateur avec le module 8232 sans isolation galvanique: SW4 pin 2 relier au 3, SW5 pin 2 relier au 3 et SW6 pin 1 relier au 2.		

Page 2 – Mode d'Utilisation Détaillé de la Carte Holter

1. Objectif de la carte

Cette carte Holter est conçue pour la capture de signaux ECG à des fins de simulation ou de mesure réelle, avec la possibilité d'utiliser différents modes de fonctionnement (patient réel, simulateur, avec ou sans isolation galvanique). Elle intègre un module LoRa pour la communication distante et divers points de test pour vérification électrique.

2. Alimentation

La carte nécessite les tensions suivantes :

- +3.3V : pour les composants sensibles et les modules (8232, REF3033, etc.)
- +5V : pour l'alimentation générale de certains régulateurs
- GND et GND_RPI : masses de référence (*peuvent être reliées ensemble uniquement si aucun patient réel n'est connecté*)
- SDN : ligne de contrôle à alimenter en 3.3V pour activer le module 8232

Vérification recommandée : Utiliser les points de test, pour valider les tensions.

3. Modes de fonctionnement

Mode	Position des cavaliers / interrupteurs	Description
Simulateur ECG externe (Arduino)	SW6 : pin 3 → 2	Permet de simuler un signal ECG via un générateur externe
Patient réel avec isolation galvanique	SW4 : pin 2 → 1 SW5 : pin 1 → 2 SW6 : pin 1 → 2	Utilise l'AMC1351 pour isoler électriquement le patient
Simulateur sans isolation	SW4 : pin 2 → 3 SW5 : pin 2 → 3 SW6 : pin 1 → 2	Mode de test rapide sans isolement galvanique

4. Connexions clés

- Entrées ECG (RL, LA et RA relier au GND) : connecteurs JP1_x vers le module ECG (SEN-12650)
- Sortie du module 8232 : via SIGNAL_OUT (JP3_3)
- Liaison avec Raspberry Pi : connecteur J2 (GPIO RPi)
- Communication LoRa : via module U7 (HTCC-AM01_v2)
- Programmation LoRa : J3 (Connecteur sélection programme LoRa)

5. Points de test recommandés

Point de test	Signal attendu
TP1 à TP9	Tensions 3.3V / 5V / GND / signal logique
TP12	Signal ECG traité (à oscilloscope)

6. Remarques d'utilisation

- Ne jamais utiliser le simulateur sans vérifier la déconnexion de la partie galvanique pour éviter les interférences.
- En cas d'utilisation sur patient réel, toujours activer l'isolation galvanique.
- Avant chaque session, contrôler l'intégrité des connexions cavaliers et des tensions sur les TP.

Fiche de suivi: Zainou

date:	Description	Défis rencontrés
13/01/2025	2h de réunion d'équipe, 2h de planification sur project	
14/01/2025	4h pour mettre en oeuvre le module AD8232 et téléchargement de toutes les librairies pour les différents circuits	J'ai rencontré des difficultés au niveau du terminal, problème sur l'exploitation du l'électrocardiogramme
20/01/2025	2h pour faire les câblages rapides du module AD8232 et MAX11100 sur Fritzing en me référent des datasheets et téléchargement des	J'ai donc essayé de mettre en service le module avec un professeur du CERI, mais encore une fois, rien ne fonctionnait. J'ai donc décidé de passer au convertisseur analogique-numérique.
21/01/2025	4h de mise en oeuvre du convertisseur analogique numérique MAX11100 avec un LM4040 comme référence de tension	J'ai mis en œuvre le convertisseur et effectué mes premiers tests en utilisant une carte Arduino comme maître pour le bus spi et une alimentation externe comme source de signal analogique. J'ai fait varier ce signal de 0 à 2,5 V, correspondant à la tension de référence du convertisseur, appliquée à son entrée.
27/01/2025	absent	
28/01/2025	J'ai testé et re-testé le module AD8232 avec le professeur de Céri, en utilisant une deuxième carte Arduino qui lui permettra de simuler l'ECG. Grâce au logiciel Serial Plotter, qui est plus performant que celui proposé sur Arduino, nous avons pu observer la forme de la courbe. J'ai aussi câbler l'AMC1351-Q1 d'abord sur fritzing	RAS
24/02/2025	absent	
25/02/2025	saisie de schéma Kicad	RAS
03/03/2025	saisie schéma + montage	RAS
10/03/2025	Récupération de la tram du CAN sur analyseur logique et câblage de l'ampli isolateur	RAS
11/03/2025	Test du convertisseur analogique numérique avec les ir	RAS

17/03/2025	Test simulateur avec le collègue IR	Problème quand on essayer de faire passer le simulateur dans le la carte du holter, on avait un signal dégradé
18/03/2025	Amélioration du schéma avec des modif a faire et j'ai commencé la de rédaction du projet	RAS
25/03/2025	Rédaction dossier + préparation démo	RAS
31/03/2025	passage revue	RAS
01/04/2025	routage	RAS
22/04/2025	Amélioration du dossier	RAS
28/04/2025	réception des carte + fabrication	c12 10micro
29/04/2025	fabrication	RAS
05/06/2025	test de la carte	RAS
06/05/2025	test de la carte	RAS
12/05/2025	préparation dossier + démo	RAS
13/05/2025	préparation dossier + démo	RAS

Liste Matériel Holter(ECG)							
Référence	Désignation	Valeur	Code commande	Fournisseur	Qté	Prix TTC Unitaire (€)	Prix TTC Total (€)
connecteur batterie	Batterie	9V	661-1871	RS Components	1	0,05	0,05
C1, C3, C4, C16	Condensateur CMS	1µF	264-4460	RS Components	4	0,05	0,2
C2, C5, C8, C9, C11	Condensateur CMS	100nF	264-4422	RS Components	5	0,125	0,625
C6, C7	Condensateur CMS	330pF	712-8455	RS Components	2	0,136	0,272
C10	Condensateur CMS	4,7µF	691-1189	RS Components	1	0,122	0,122
C12	Condensateur CMS	0,47µF	698-3434	RS Components	1	0,137	0,137
C13	Condensateur CMS	10µF	103-4168	RS Components	1	0,282	0,282
C13, C15	Condensateur CMS	22µF	741-6796	RS Components	2	0,146	0,292
D1	LED bleue	-	692-0913	RS Components	1	0,254	0,254
D2	Diode PMEG3010ER	-	685-8165	RS Components	1	0,234	0,234
D3	LED rouge	-	497-4904	RS Components	1	0,452	0,452
D4	LED jaune	-	486-0519	RS Components	1	0,157	0,157
D5	LED verte	-	497-4904	RS Components	1	0,452	0,452
F1, F2	Fusible CMS	400mA 0,2A	517-7105	RS Components	2	0,11	0,22
H1	Convertisseur MAX11100EUB+	-	700-MAX11100EUB+	Mouser	1	4,22	4,22
J1	Connecteur prog. HTCC	-	549-0026	RS Components	1	0,6	0,6
J2	Connecteur GPIO RPI	-	2745	Pololu	1	2,6	2,6
J5	Connecteur Select Prog. Lora	-	549-0026	RS Components	1	0,6	0,6
JP1	Liaison GND	-	8000	Gotronic	1	0,6	0,6
R1, R2, R9, R10, R11	Résistance CMS	330 Ohms	679-2049	RS Components	5	0,031	0,155
R3, R4, R5, R8	Résistance CMS	3,3k Ohms	223-2338	RS Components	4	0,031	0,124
R7, R8	Résistance CMS	10k Ohms	125-1192	RS Components	2	0,028	0,056
SW1	Interrupteur USER	-	102-361	RS Components	1	0,557	0,557
SW3	Interrupteur SIP 1K2 EAO	-	204-7855	RS Components	2	1,893	3,786
SW4, SW5, SW6	Interrupteur SPDT	-	8000	Gotronic	3	0,6	1,8
TP1 à TP11	Points de test	-	8000	Gotronic	11	0,6	6,6
U1	Capteur SEN-12650	-	34916	Gotronic	1	2,44	2,44
U2	Amplificateur AMC1351-Q1	-	595-AMC1351QDWVRQ1	Mouser	1	4,98	4,98
U3	Amplificateur LMV841	-	926-LMV841MGX/NOPB	Mouser	1	1,91	1,91
U4	Référence de tension REF3033	-	661-9455	RS Components	1	3,66	3,66
U5	Régulateur de tension	-	193-3954	RS Components	1	3,42	3,42
U7	Module LoRa HTCC-AM01_v2	863-870MHz		Heltec	1	8,88	8,88
PCB	PCB + Stencil	-	-	JLC-PCB	6	6,89	6,86
	NOM	DATE				TOTAL HT GLOBAL	55,77€
Crée	ZAKOUANA	28/04/2025					