



## Haute Ecole Economique et Technique

Avenue du Ciseau, 15

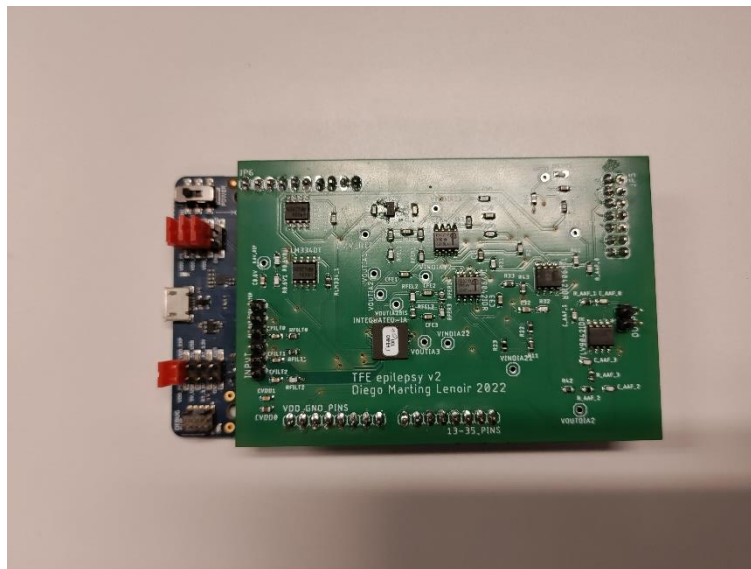
1348 Louvain-La-Neuve

Travail de fin d'études présenté en vue de l'obtention du diplôme de bachelier en  
Informatique et Système : finalité Technologie de l'informatique

---

Amélioration d'un système de détection de crises d'épilepsie : modification du  
régulateur de tension et mise en place du module de transmission et de  
visualisation des données

Diego MARTING LENOIR 3TL1



Rapporteuse : Stéphanie Guérit

[Année Académique 2021-2022](#)

## Remerciement :

Je tiens à remercier toutes les personnes qui m'ont aidé lors de mon travail de fin d'étude.

Tout d'abord, j'adresse mes remerciements à ma rapporteuse Stéphanie Guérit qui m'a aidé à trouver le sujet de ce travail via son réseau à l'UCL et qui m'a guidé durant la préparation de celui-ci.

Je tiens à remercier vivement David Bol et Rémi Dékimpe qui m'ont accepté et fait confiance pour prendre en main le projet. Plus particulièrement à Rémi Dékimpe qui m'a épaulé et a répondu à toutes mes questions par rapport au projet.

Pour finir je tiens à remercier toutes les personnes qui m'ont aidé lors de la rédaction de ce document.

## Table des matières :

Introduction : .....	2
VENG : .....	4
Projet : .....	5
- Méthodologie .....	5
- Fonctionnement.....	5
- Objectif.....	5
- PCB .....	6
- Transfert des données du PCB vers Apollo3 Blue .....	10
- Transfert Apollo3 Blue vers Pc .....	13
- Acquisition et affichages des données sur ordinateur : .....	16
Historique du projet :.....	22
Conclusion :.....	23
Bibliographie :.....	25

## Annexe :

Annexe 1: Schéma PCB .....	26
Annexe 2: Board PCB .....	26
Annexe 3: Code source python .....	27
Annexe 4: Code source Apollo3 .....	28

## Introduction :

L'épilepsie est une famille de maladie neurologique dont le point commun est une prédisposition cérébrale qui engendre des crises spontanées qui touche plus de 50 millions de personnes. Cette maladie se manifeste souvent chez les jeunes ou les plus de 65 ans. Deux personnes sur trois peuvent prendre des médicaments pour se soigner. Afin d'endiguer cette maladie, quelques pistes de solutions se dégagent : tandis que certaines personnes sont aptes à subir une opération chirurgicale, d'autres personnes peuvent être traitées par des stimulations au niveau du cerveau.

Il existe deux solutions :

- The deep brain stimulation (DBS)

Cette option est très invasive, des électrodes sont directement insérées profondément dans le cerveau et présentent un risque non négligeable.

- The vagus nerve stimulation (VNS)

Cette méthode stimule le nerf vagus grâce à des électrodes. Ce nerf est directement connecté au cerveau. Cette méthode est moins invasive que le DBS elle présente donc moins de risque.

Dans ce travail, on utilise la méthode VNS. On peut utiliser celle-ci de deux façons. La première est de tout le temps stimuler le nerf, mais celle-ci est consommatrice d'énergie et n'est pas spécialement viable sur le long terme. La deuxième manière, est de stimuler le nerf seulement quand on détecte la possibilité qu'une crise se déclenche. Dans ce cas, il faut récupérer des signaux via des électrodes pour détecter la crise. Le problème est que le signal est d'environ  $7.1 \mu V_{rms}$  et qu'il y a des interférences. Il faut donc implémenter un système qui récupère les données, les amplifie, enlève le bruit, le tout en étant à basse consommation.

Ce travail est composé de différentes parties :

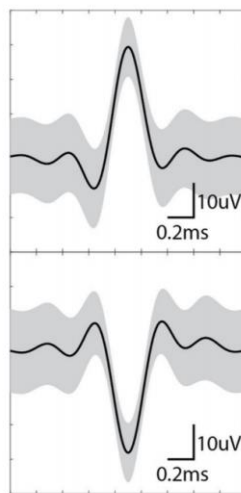
- VENG
- Méthodologie
- Le fonctionnement
- Travail précédent
- Objectif
- PCB
- Transfert des données du PCB vers Apollo3 Blue
- Transfert Apollo3 Blue vers Pc
- Acquisition et affichages de données sur Pc
- Historique du projet
- Conclusion

Ces différentes parties expliquent comment fonctionne chaque système mis en place lors de ce projet. Ce travail se clôture par une brève conclusion qui résumera ce projet et proposera une vision future de celui-ci.

## VENG :

Les informations utilisées pour expliquer le signal VENG proviennent du mémoire de Jaminon-De Roeck, Chen-Terry cf. [1].

Le nerf vagus est porteur d'informations parasympathiques et innerve plusieurs organes. On peut par exemple y détecter des changements respiratoires et cardiaques lors d'une crise. Ce nerf est directement relié au cerveau où les crises sont déclenchées. Pour les détecter, il faut identifier les biomarqueurs associés. D'après des expériences, ceux-ci prennent une forme triphasique comme constaté sur l'image suivante.



*Figure 1: Forme triphasique d'un biomarqueur dans le nerf vagus due à des crises. Cf.[1]*

Dans le VENG, les formes triphasiques sont des rafales de basse et haute amplitude synchrones à la respiration et au rythme cardiaque. Les amplitudes moyenne et de crête dépendent de l'électrode utilisée et de la distance entre les électrodes. En moyenne, le temps d'une pointe est inférieur à 1.5ms. Son amplitude est de  $7.1 \pm 2.3 \mu V$  quand on utilise des électrodes brassards tripolaires avec 2mm d'espacement entre chaque électrode. L'amplitude de crête est différente entre les pics positifs et négatifs.

Positif :  $20.7 \pm 6.6 \mu V$

Néfatif :  $24.1 \pm 7.7 \mu V$

## Projet :

### **- Méthodologie**

Pour la réalisation de ce travail j'ai été amené à travailler avec l'UCL et plus particulièrement avec Rémi Dékimpe. En effet, grâce à Stéphanie Guérit, rapporteuse de ce projet, j'ai pu participer au projet en collaboration avec l'UCL qui a pour but d'améliorer un mémoire réalisé en 2021 par Jaminon-De Roeck Chen-Terry. Ce travail avait pour objectif de créer un printed circuit board (PCB) qui reçoit un signal en entrée puis amplifie le signal pour le ressortir à une pin. Apparemment, il avait déjà établi un code permettant de traiter le signal et le transmettre, mais ce code n'a pas été fourni.

Ce projet a été un réel défi pour moi. En effet, je me suis retrouvé avec des informations qui ne m'étaient pas familier. J'ai donc dû me plonger dans de la documentation afin de comprendre les différents termes utilisés dans le mémoire, avant de vraiment me plonger dans la réalisation du travail.

J'ai aussi dû comprendre comment fonctionnent les différents systèmes utilisés lors du travail, car n'étant pas habitué à utiliser l'environnement de l'Apollo, j'ai passé beaucoup de temps essayer de saisir au mieux son fonctionnement.

### **- Fonctionnement**

Le projet a pour but d'acquérir un signal, l'amplifier, enlever le bruit et le transmettre sur un ordinateur pour visualiser les données. Donc dans un premier temps, on vient récupérer les données grâce à des électrodes pour ensuite les faire passer sur le PCB qui va les amplifier, pour ensuite transmettre à l'Apollo les données tout en enlevant le bruit. Pour finir l'Apollo va envoyer les données vers un ordinateur grâce à un port USB. Un programme python viendra les lire sur un port COM pour ensuite les afficher.

### **- Objectif**

L'objectif est d'apporter des améliorations à l'ancien travail. Pour cela, il fallait réduire la consommation du PCB en modifiant le Low-Dropout regulator (LDO), pour ensuite transférer les signaux du PCB vers l'Apollo3 Blue en les échantillonnant pour enfin transférer les données sur un ordinateur via un câble USB pour les afficher.

- PCB

En me basant sur le travail précédent, celui-ci mettait en place un LDO qui valait 71.3% de la consommation totale. Et donc, il proposait de changer le LDO par le TPS7A0512PDBZR, qui ne faisait que 10.8  $\mu$ W à la place de 504 pour le LDO actuel. Cependant, ce celui-ci n'était malheureusement plus disponible. Il a fallu opter pour un autre LDO qui est le MCP1703AT-1202E/CB. Ce dernier permet toujours de réguler la tension à 1.8V.

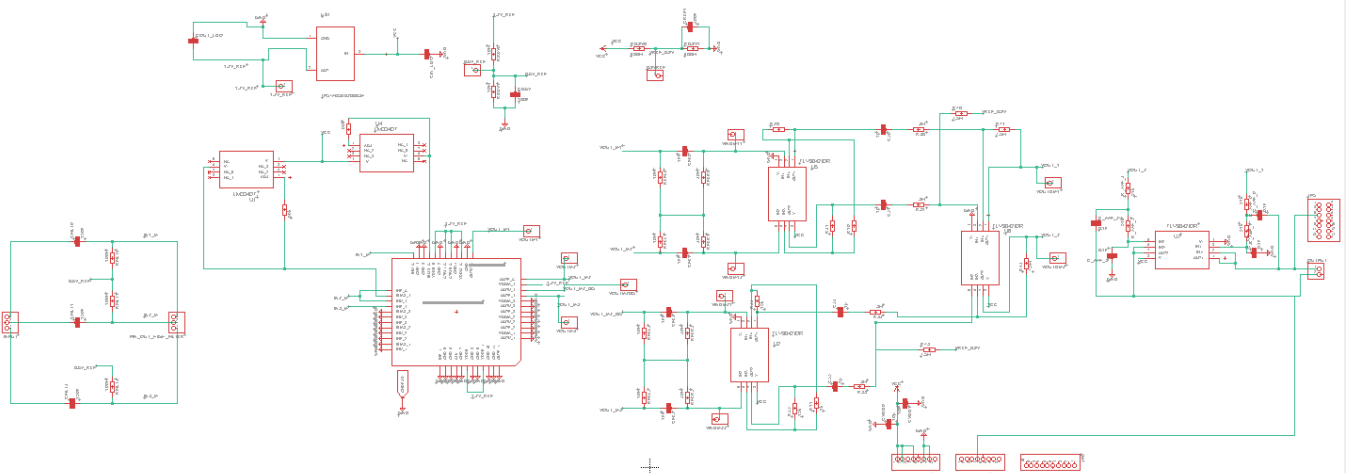
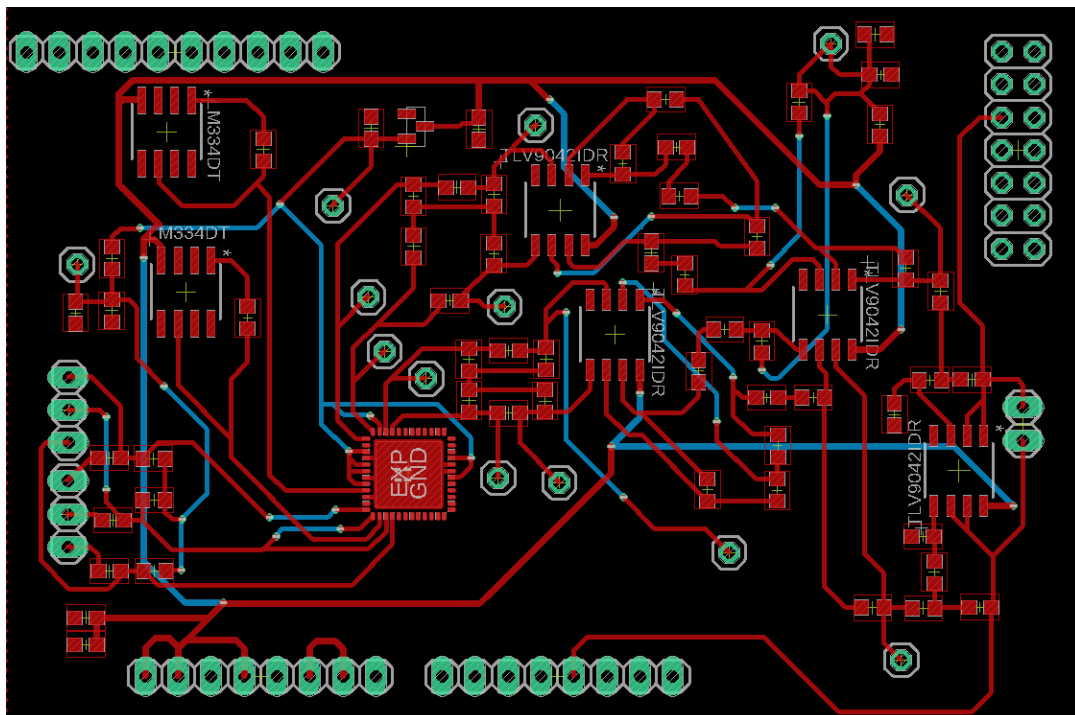


Figure II : Schéma du PCB



*Figure III: Board du PCB*



Afin de changer le LDO, il faut revoir le schéma du PCB comme ci-dessous.

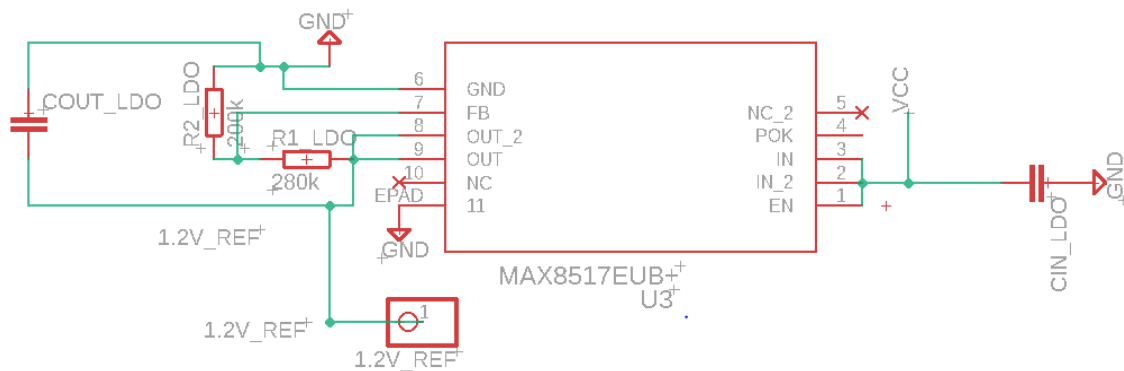


Figure IV: Schéma ancien LDO

Par celui-ci avec des modifications apportées :

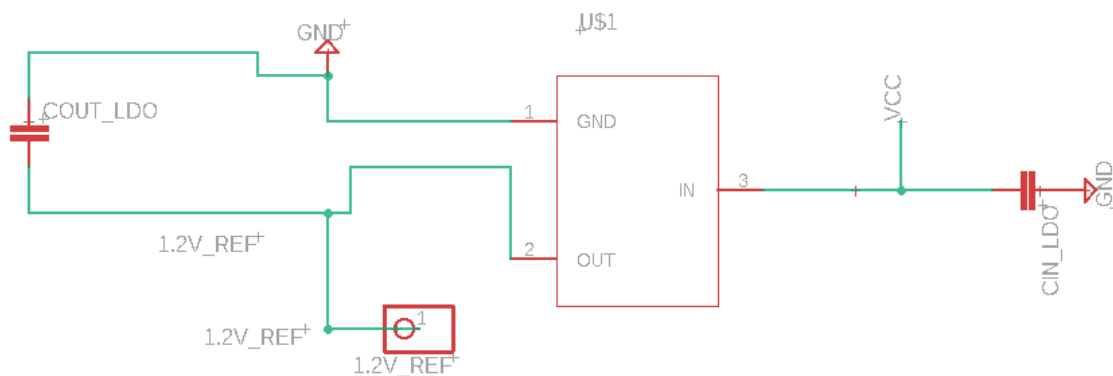


Figure V: Schéma nouveau LDO

Pour ce faire, il a fallu créer une librairie sur Eagle grâce à la datasheet du LDO. Ensuite, il faut enlever les résistances de sortie comme indiqué dans la documentation. Une fois la librairie validée et le schéma changé, la demande d'impression de la carte a été effectuée. Dès que la carte a été reçue, elle a été assemblée avec les différents composants.

Dès que toutes ces étapes ont été finies, avec l'aide de l'équipe de l'UCL on a testé le PCB pour vérifier son bon état de fonctionnement. Le résultat final est que la consommation finale du PCB à lui tout seul par rapport à l'ancien travail a diminué, il est passé d'un total de 700  $\mu\text{W}$  à 288  $\mu\text{W}$ . Le nouveau LDO ne consomme plus que 86  $\mu\text{W}$  à la place de 504  $\mu\text{W}$ . À la page suivante vous retrouvez le détail de la consommation avec un graphique de l'ancien projet comparé au nouveau.

Graphique de la consommation de l'ancien projet :

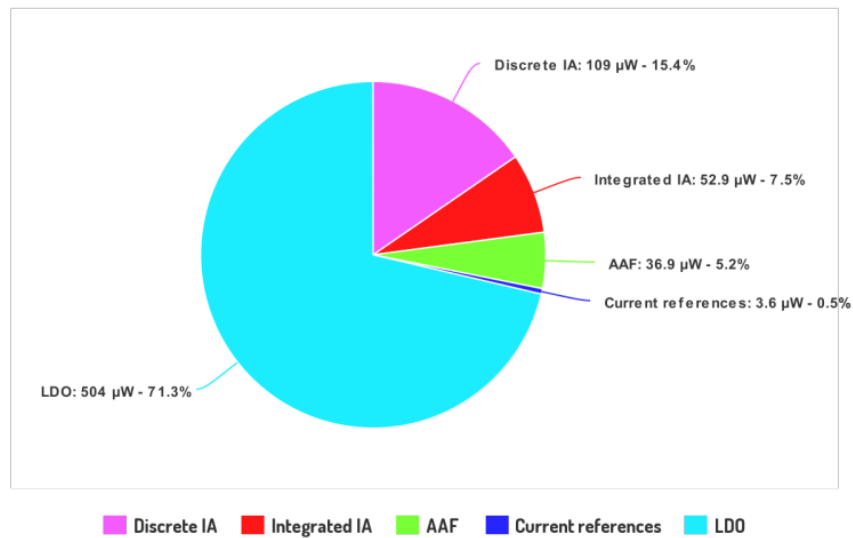


Figure VI: Graphique consommation ancien projet cf. [1]

Graphique de la consommation avec le nouveau LDO :

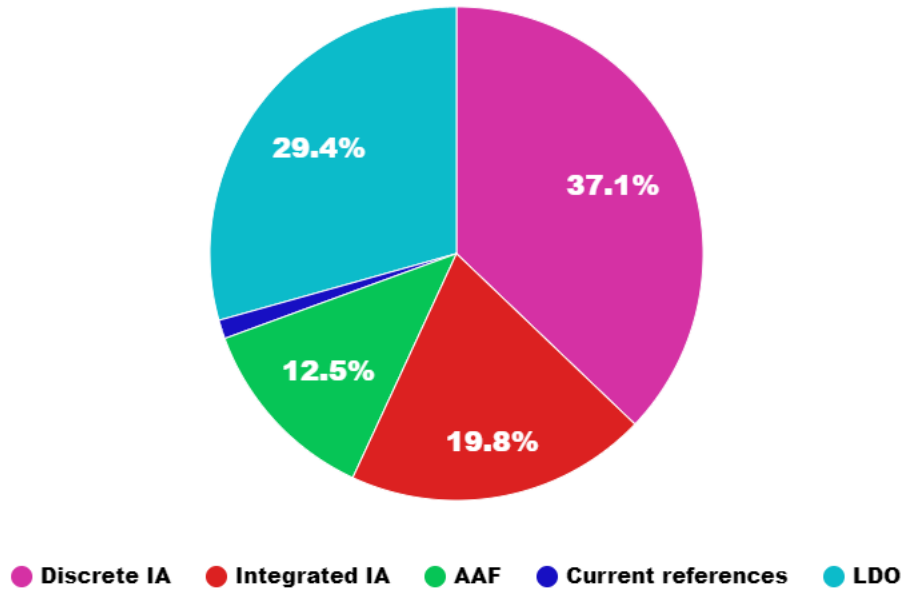


Figure VII: Graphique consommation nouveau projet

On peut voir sur le nouveau graphique que le LDO n'est plus la source principale de la consommation du système. Malheureusement le TPS7A0512PDBZR est censé donner 10  $\mu$ W alors qu'ici avec le nouveau LDO on atteint les 86  $\mu$ W. Il faudrait peut-être à l'avenir, si cela en vaut la peine, changer le LDO. Actuellement on a un gain de 412  $\mu$ W ce qui n'est pas du tout négligeable. Après, ce qui pourrait être aussi intéressant par la suite c'est de refaire une structuration du Board pour essayer d'avoir quelque chose de plus propre et optimisé. Mais cela ne nuit en rien à l'utilisation de la carte.

## - Transfert des données du PCB vers Apollo3 Blue

Tout d'abord avant de transférer un signal il faut expliquer ce qu'est l'Apollo 3 Blue. C'est une carte qu'on va venir emboîter dans le PCB. Cette carte munie d'un microprocesseur, va permettre d'y injecter du code et de traiter les informations reçues analogiquement pour les numériser. Une fois cette opération effectuée on a la possibilité de faire à peu près ce que l'on veut avec.

Donc dans un premier temps notre préoccupation est de recevoir les données du PCB. On va alors utiliser l'ADC (Analog-to-Digital Converter) qui permet de convertir l'analogique en numérique.

Pour commencer, il faut connecter l'Apollo3 et le PCB ensemble. Sur le PCB on fait ressortir le signal à une pin et celle-ci est connectée à la pin 32 de l'Apollo.

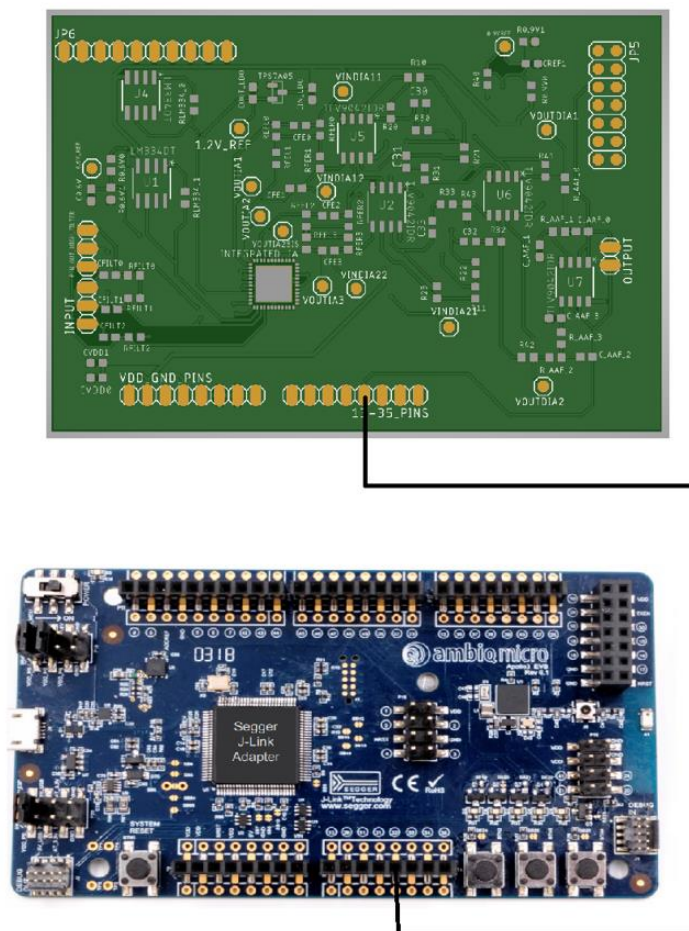


Figure VIII: Association PCB avec Apollo3

Au niveau du code, on va configurer l'Apollo pour dire que la pin32 utilise l'ADC en utilise le canal 0. Pour ce faire, on va débloquent la configuration GPIO et on sélectionne l'ADC au pad 32. Ensuite on l'active en lui donnant sa configuration qui est celle-ci (le code source est repris d'un repository mais modifié pour notre utilisation cf. [10]) :

```
u32Cfg =
| _VAL2FLD(ADC_CFG_CKMODE,ADC_CFG_CKMODE_LLCKMODE) //Low Latency Clock Mode
| _VAL2FLD(ADC_CFG_CLKSEL,ADC_CFG_CLKSEL_HFRC) //HFRC clock
| _VAL2FLD(ADC_CFG_TRIGSEL,ADC_CFG_TRIGSEL_SWT) //Software trigger
| _VAL2FLD(ADC_CFG_REFSEL,ADC_CFG_REFSEL_INT2P0) //internal 2V reference
| _VAL2FLD(ADC_CFG_LPMODE,ADC_CFG_LPMODE_MODE1) //Low Power Mode 1
| _VAL2FLD(ADC_CFG_RPTEN,ADC_CFG_RPTEN_SINGLE_SCAN) //Single scan
| _VAL2FLD(ADC_CFG_ADCEN,1); //Enable ADC

//
// Using slot 0
//
ADC->SL0CFG = 0;
ADC->SL0CFG.b.CHSEL0 = ADC_SL0CFG_CHSEL0_SE4; //use ADCSE4 as input;
ADC->SL0CFG.b.ADSEL0 = ADC_SL0CFG_ADSEL0_AVG_1_MSRMT; //Average over 1 measurements
ADC->SL0CFG.b.PRMODE0 = ADC_SL0CFG_PRMODE0_P14B; //use 14-bit 1.2MS/s
ADC->SL0CFG.b.WCEN0 = 0; //window comparator mode disabled
ADC->SL0CFG.b.SLEN0 = 1; //slot enabled

ADC->WULIM.b.ULIM = 0; //Window comparator upper limit (not used)
ADC->WLLIM.b.LLIM = 0; //Window comparator lower limit (not used)

ADC->INTEN.b.WCINC = 1; //enable window comparator voltage incursion interrupt
ADC->INTEN.b.WCEXC = 1; //enable window comparator voltage excursion interrupt
ADC->INTEN.b.FIFOVR1 = 1; //enable FIFO 100% full interrupt
ADC->INTEN.b.FIFOVR2 = 1; //enable FIFO 75% full interrupt
ADC->INTEN.b.SCNCMP = 1; //enable ADC scan complete interrupt
ADC->INTEN.b.CNVCMP = 1; //enable ADC conversion complete interrupt

ADC->CFG = u32Cfg;

ADC->INTCLR = 0xFFFFFFFF; //clear interrupts
ADC->SWT = 0x37; //trigger ADC
```

*Figure IX: Code de la configuration de l'ADC*

Plusieurs tests ont été effectués avec un signal qui variait de 0V à 1.8V sur différentes fréquences avec un nombre différent de mesure prise :

- 14 mesures :

- 1KHZ

Le signal de sortie était coupé on ne recevait que la tension entre 0.80V et 1V.

- 500HZ

Le signal de sortie était presque correct mais avec parfois un manque d'informations.

- 1 mesure :

- 1KHZ

Le signal de sortie était le signal attendu.

- 500HZ

Le signal de sortie était le signal attendu.

Pour conclure cette partie, le transfert fonctionne bien du PCB vers l'Apollo. Cependant, il faudrait tester quel est le bon compromis entre le nombre de mesures faites et l'envoi d'une donnée. De cette manière, on pourrait être plus précis vis-à-vis du signal de sortie. Il faudrait voir aussi s'il est possible d'optimiser le code en passant par la mémoire DMA.

## - **Transfert Apollo3 Blue vers Pc**

Une fois l'Apollo ayant numérisé le signal grâce à l'ADC, on peut commencer à transférer ces données sur le PC. Pour cela, il existe plusieurs solutions :

### - BLE :

Le BLE (Bluetooth low energy), est une technique de transmission qui complète le bluetooth. Comparé à celui-ci, il permet un débit de 1Mbit/s pour une consommation d'énergie 10 fois moindre. Ce qui lui permet d'être mis en application dans des montres connectées etc.

### - UART :

L'UART dit universal asynchronous receiver / transmitter est un protocole qui est dédié à l'échange de données entre deux appareils. Il suffit de deux câbles entre l'émetteur et le récepteur pour recevoir et émettre dans les deux sens.

Dans notre cas les deux modes de transmission se valent. Toutefois, il est à signaler que le BLE est plus intéressant à long terme pour ce projet. Cependant, l'UART est utilisé à cause d'un manque de temps et de compréhension du module BLE. Cela ne va en rien entraver l'objectif premier de ce projet.

La configuration de l'UART se fait de cette manière (comme pour l'ADC le code vient du même repository cf. [10]) :

```
//  
  
PWRCTRL->DEVPWREN |= (1 << PWRCTRL_DEVPWREN_PWRUART0_Pos);  
  
while(PWRCTRL->DEVPWRSTATUS_b.HCPA == 0) __NOP();  
  
//  
// Enable clock / select clock...  
//  
pstcUart->CR = 0;  
pstcUart->CR_b.CLKEN = 1;           //enable clock  
pstcUart->CR_b.CLKSEL = 1;         //use 24MHz clock  
  
//  
// Disable UART before config...  
//  
pstcUart->CR_b.UARTEN = 0;         //disable UART  
pstcUart->CR_b.RXE = 0;           //disable receiver  
pstcUart->CR_b.TXE = 0;           //disable transmitter  
  
//  
// Starting UART config...  
//  
  
// initialize baudrate before all other settings, otherwise UART will not be initialized  
SystemCoreClockUpdate();  
ConfigureBaudrate(pstcUart,u32Baudrate,24000000UL);  
  
// initialize line coding...  
pstcUart->LCRH = 0;  
pstcUart->LCRH_b.WLEN = 3;         //3 = 8 data bits (2..0 = 7..5 data bits)  
pstcUart->LCRH_b.STP2 = 0;         //1 stop bit  
pstcUart->LCRH_b.PEN = 0;         //no parity  
  
//  
// Enable UART after config...  
//  
pstcUart->CR_b.UARTEN = 1;         //enable UART  
pstcUart->CR_b.TXE = 1;           //enable transmitter  
}
```

Figure X: Code de la configuration de l'UART

Les points importants à savoir sur le code, c'est qu'on va transmettre des données sur 8 bits et il y a seulement la transmission d'active. On n'active pas la réception car cela n'est pas nécessaire. L'Apollo3 n'a pour but que de transmettre les données. À noter que l'Apollo est configuré à 230400 Baud Rate, cette valeur permet de définir la vitesse de transmission sur le câble, ça correspond au nombre de bits transmis par seconde.



Une fois la configuration terminée, on va aller dans la boucle principale du programme qui effectue le transfert de données de l'ADC à l'UART.

```
while(1)
{
    if (ADC->INTSTAT_b.CNVCMP == 1)
    {
        ADC->INTCLR_b.CNVCMP = 1;

        if ((_FLD2VAL(ADC_FIFO_COUNT,ADC->FIFO) > 0) && (_FLD2VAL(ADC_FIFO_SLOTNUM,ADC->FIFO) == 0))
        {
            u32Data = _FLD2VAL(ADC_FIFO_DATA,ADC->FIFO) >> 6;
            sprintf(data, "%d-",u32Data);
            PutStringUart(UART0,data);
            ADC->FIFO = 0; // pop FIFO
        }
    }

    if (ADC->INTSTAT_b.SCNCMP == 1)
    {
        ADC->INTCLR_b.SCNCMP = 1;
        ADC->SWT = 0x37;
    }
}
```

*Figure XI: Code de la boucle d'envoi*

Dans cette boucle, on va vérifier que l'ADC contient quelque chose et que son canal est bien le 0. Une fois la vérification effectuée on vient récupérer les données dans l'ADC et les convertir en une chaîne de caractères pour pouvoir rajouter un symbole qui permettra plus tard de distinguer les différentes données reçues. Une fois la conversion faite, on envoie la chaîne de caractères dans l'UART. À la fin on vérifie bien qu'on enlève la valeur de l'ADC pour pouvoir faire une nouvelle mesure.

Pour conclure, l'UART fonctionne très bien. Pour une future amélioration de ce projet, il pourrait être intéressant de permettre l'envoi des données avec le BLE. Ce qui pourrait être intéressant pour pouvoir utiliser l'ensemble du système sans être attaché à un ordinateur.

## - Acquisition et affichages des données sur ordinateur :

Maintenant que l'Apollo peut transmettre des données via l'UART, il faut créer un script qui permet de recevoir ces données et les afficher. Ce script est fait en python car ce langage de programmation permet, grâce à de simples librairies de mettre en application et d'exécuter un script très facilement.

Le but est d'afficher sur un graphique le signal qu'on reçoit via l'UART au moment où on lance le script. Premièrement, on va importer les différents modules à utiliser pour faire fonctionner le script :

```
1  import serial
2  import matplotlib.pyplot as plt
3  import numpy as np
4  import sys
```

*Figure XII: Modules python à importer*

Le module serial permet de recevoir les données de l'Apollo3.

Matplotlib est le module qui va afficher un graphique de nos données.

Numpy va permettre d'effectuer des opérations mathématiques avancées.

Le dernier module sys va lui, arrêter le code en renvoyant un message d'erreur.

Ensuite on va découper le code en plusieurs fonctions :

### - read()

Cette fonction est celle qui lit les données envoyées sur le port COM, qui les traite et les stocke dans un tableau. Elle va lire les 1000 premières données reçues. Elle va retourner deux variables end et datas.

### - graph(datas)

Cette fonction va afficher les données récupérées par la fonction read. Elle prend en paramètre les données enregistrées.

Dans la fonction read, il faut commencer par dire au programme sur quel port COM écouter et quel est la vitesse de transmission des données. On définit donc dans le code le Baud Rate à 230400. Et le port COM est demandé quand on lance le script.

```
6
7  def read():
8      s = serial.Serial('COM'+sys.argv[1])
9      s.baudrate = 230400
```

*Figure XIII: Code de l'initialisation de la connexion Apollo3 Pc*

Une fois la configuration de la communication établie on va définir différentes variables :

```
10      datas = []
11      data = ''
12      countExit = 0
```

*Figure XIV: Variables*

Datas est un tableau qui permet de stocker toutes les données reçues.

Data est une chaîne de caractère qui va recevoir les données et jusqu'au caractère d'échappement. Pour ensuite ajouter la valeur à datas et ensuite se vider.

CountExit teste combien de fois on passe dans la boucle. Elle permet de tester si on est sur le bon port COM.

Une fois les variables établies, on peut passer à la boucle principale :

```
13  ✓   while True:
14      countExit = countExit + 1
15  ✓   if countExit == 150000:
16      return False,False
17
18      temp = s.read_all().decode("UTF-8")
19  ✓   if temp != '':
20  ✓       for i in temp:
21  ✓           if i == '-':
22               count += 1
23               datas.append(round(int(data)/16384 * 1.8,2))
24               data = ''
25  ✓       else:
26               data += i
27  ✓   if len(datas)==1000:
28       return False,datas
```

Figure XV: Code de la lecture et de l'enregistrement des données

La première étape dans cette boucle est de vérifier grâce à un compteur qu'on n'est pas sur le mauvais port COM. En effet, si le compteur est devenu trop élevé et qu'on n'a toujours pas eu les données affichées, c'est qu'on n'est pas sur le bon port. Dans ce cas-là, on retourne deux valeurs à False, ce qui permettra de dire que le code doit se terminer.

Ensuite, on vient mettre dans la variable temp les données lues sur le port. On reçoit normalement les données en bits donc on va venir décoder celles-ci avec le format UTF-8. Ainsi on n'aura plus à traiter le format bits mais une chaîne de caractères.

Une fois les données reçues, on regarde si la valeur reçue n'est pas ''. Comme ça on évite d'ajouter les données qui ne sont pas nécessaires au script.

Par après, on boucle sur la variable temp testée dans un premier temps pour voir si on retrouve le caractère qui permet de donner la fin d'une valeur '-'. Sinon on rajoute le caractère dans la variable data. Si on trouve le caractère de fin alors on ajoute data dans notre tableau en utilisant la formule suivante :  $\text{valeur} / 2^{\text{nbrbits}} (\text{ici } 14 \text{ via l'ADC}) * \text{tension de référence} (1.8 \text{ V})$ . Cette formule permet de convertir les valeurs de l'ADC qui sont des entiers vers les vraies valeurs de tension.

A la fin de cette boucle il y a un compteur qui permet de vérifier si on a bien atteint les 1000 données souhaiter. Si on les a bien reçus on retourne False et datas ce qui respectivement va mettre un terme à la boucle de test qui se situe plus loin et va renvoyer les données.

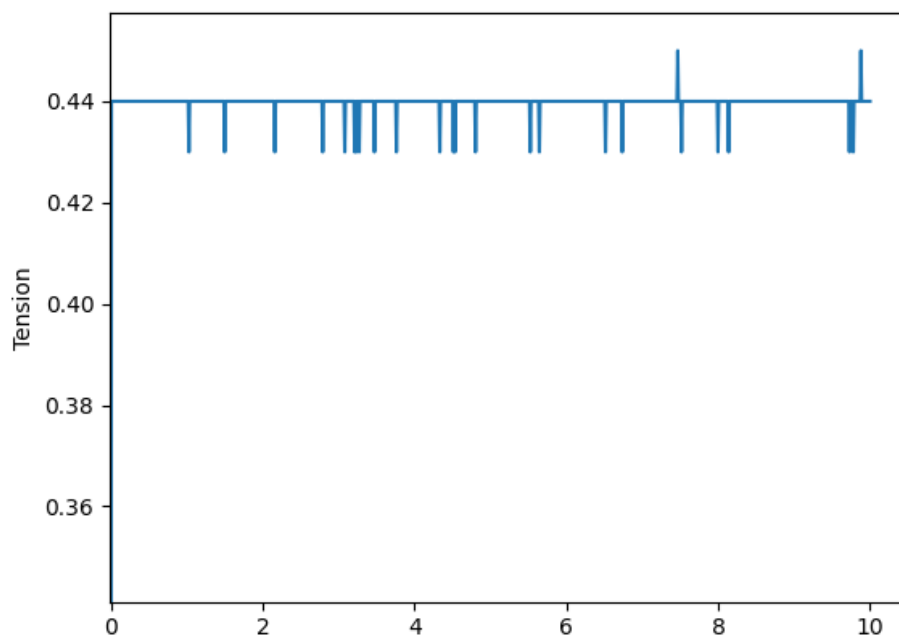
Dans la fonction graph, on va vouloir afficher un graph avec les 1000 données reçues.

```
29 def graph(datas):  
30     x=np.linspace(0,10,1000)  
31     plt.plot(x,datas)  
32     plt.ylabel('Tension')  
33     plt.xlabel("")  
34     plt.show()
```

*Figure XVI: Code du graphique*

Donc on va dans un premier temps définir l'axe des abscisses en donnant un intervalle 0 à 10. En ayant que 1000 valeurs souhaitées dans celui-ci. Ensuite on va envoyer les données sur le graphique.

On voit sur le graphique ci-dessous une tension de 0.44 V.



*Figure XVII: Graphique de la tension reçu sur le script python*

Pour finir, le programme est exécuté dans une boucle qui permet d'éviter des erreurs.

```
37  √ if __name__ == "__main__":
38      stop = True
39      count = 0
40  √  while stop:
41  √      try:
42          stop,datas = read()
43  √      except:
44          count = count+1
45  √          if count == 20:
46              print("Port COM incorrect")
47              sys.exit()
48  √  if datas == False:
49              print("Port COM incorrect")
50              sys.exit()
51      graph(datas)
52
```

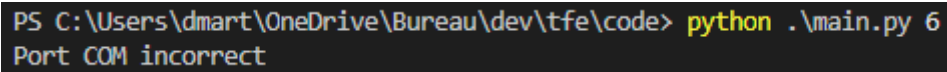
*Figure XVIII: Code de la gestion du script*

Dans un premier temps on initialise stop et count qui permettent de tester le nombre d'erreurs du programme et d'arrêter celui-ci quand on dépasse la limite.

Donc dans la boucle tant que stop est égale à True alors on va exécuter le programme. On va utiliser try et except qui vont nous permettent de continuer à lancer read jusqu'à 20 fois, car parfois lors de la transmission, il y a parfois des erreurs de réception qui stop le programme direct. Donc pour contourner cette erreur, on va boucler 20 fois jusqu'au moment où on a plus l'erreur et que le script démarre. Sinon, on arrête le programme car on ne sera pas sur le bon port COM, car le bouclage de 20 permet à chaque fois de lancer la lecture sur le bon port.

Ensuite, on va récupérer stop et datas. Stop va arrêter la boucle et datas ce sont les données reçues par la lecture qui vont être envoyées dans la fonction graph. Si datas vaut False alors il y a eu une erreur de port COM on quitte donc le programme.

Pour exécuter le programme il faut lancer la console et donner en paramètre le numéro du port utilisé.



```
PS C:\Users\dmart\OneDrive\Bureau\dev\tfe\code> python .\main.py 6
Port COM incorrect
```

*Figure XIX: Commande à exécuter pour lancer le script*

Pour conclure, on pourrait améliorer le code pour essayer de recevoir les données en temps réel et ne pas à chaque fois exécuter le script pour recevoir les 1000 premières données. Et pourquoi ne pas essayer, si cela est pertinent, de stocker les données sous un fichier texte.

## Historique du projet :

20/01 réunion projet + ressources

21/04 assemblage PCB sans LDO

27/04 LDO

06/05 LDO

01/06 test du projet

Lors de ce travail, j'ai effectué plusieurs réunions avec Rémi Dékimpe pour avoir les ressources du projet assemblé la carte avec les différents composants et à la fin tester si le tout fonctionne bien.

Initialement, le planning s'agencait de cette manière :

- Mars : board du PCB finie
- Avril : Transfert du signal du PCB vers l'Apollo
- Mai : Transfert et affichage des données sur ordinateur

Cependant, les délais de ce planning n'ont pas pu être respectés. Le board a été fini début avril. Et le reste du code a été fini fin mai. Ce retard est dû à plusieurs facteurs. Le premier est le temps de travail réduit à cause du stage qui est effectué au même moment. Ensuite, le fait de ne pas avoir pu réaliser les tests de chez moi car le matériel nécessaire aux tests se trouvait à l'UCL. Par conséquent, à cause de ce retard le module BLE n'a pas été implémenté et l'UART a été préféré pour être sûr d'avoir fini dans les temps.



## Conclusion :

L'épilepsie est une famille de maladie neurologique qui touche plus de 50 millions de personnes. Cette maladie se manifeste souvent chez les jeunes ou les plus de 65 ans.

En utilisant la méthode VNS, on stimule le nerf seulement quand on détecte la possibilité qu'une crise se déclenche. On récupère donc des signaux via des électrodes pour détecter la crise. Le signal étant d'environ  $7.1 \mu\text{Vrms}$  ayant des interférences. Il fallait donc implémenter un système qui récupère les données, les amplifie, enlève le bruit le tout en étant à basse consommation.

Le signal qui nous intéresse est le VENG, ce signal est émis par le nerf vagus et permet de détecter quand une crise d'épilepsie se déclenche grâce à des électrodes.

Ce projet est effectué en collaboration avec l'UCL en se basant sur le travail de Jaminon-De Roeck, Chen-Terry. Il a pour but d'améliorer le travail précédent en changeant le LDO, et faisant le transfert du signal vers l'Apollo qui va les envoyer sur un ordinateur pour les afficher.

Ensuite, malgré les différentes difficultés liées rencontrées lors du projet, je suis parvenu à le mener à bien et à le finir dans les temps. Ces difficultés étaient principalement liées à des problèmes de compréhension de l'environnement de travail. Au fur et à mesure de l'avancement, les zones sombres ont laissé place à des éclaircissements qui m'ont permis de poursuivre.

Pour commencer, l'objectif d'améliorer l'ancien mémoire en changeant le LDO, faisant le transfert de données du PCB vers l'Apollo vers le PC pour finalement les afficher est bien fonctionnel.

Pour la partie du LDO sur le PCB, il pourrait être intéressant de réduire la consommation avec un autre LDO. En effet, celui qui est actuellement installé baisse quand même de 4x la consommation mais il est sûrement encore possible de la baisser avec le LDO qui était prévu à la base pour ce projet.

Les données converties par l'ADC arrivent bien sur le script python dans un format qu'on va ensuite convertir via une formule pour avoir les bonnes valeurs de tension. À ce niveau-ci, il n'y a plus grand-chose à faire, peut-être vérifier s'il est possible d'optimiser le code en passant par la mémoire DMA et baisser la consommation d'énergie sur l'Apollo. Il faudrait aussi tester quel est le bon compromis entre le nombre de mesures faites et l'envoi d'une donnée.

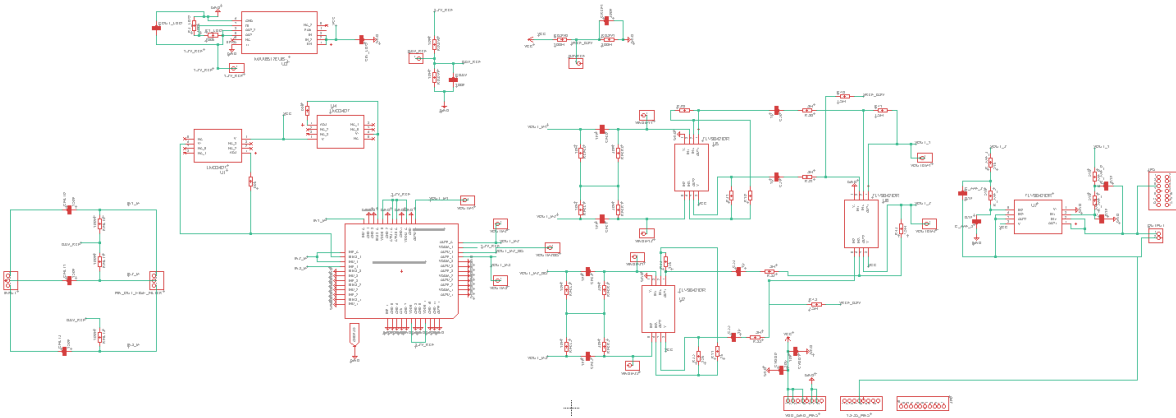
Le script python reçoit et affiche bien les données. Mais il n'affiche que les 1000 premières données. Il est sûrement possible d'afficher les données en temps réel sans à chaque fois relancer le script. Cela pourrait permettre d'avoir une meilleure idée du signal reçu.

Finalement, nous pouvons dire que l'objectif de ce projet est atteint. Il pourrait être intéressant que les futures personnes qui reprennent ce projet mettent en place les différentes améliorations citées précédemment.

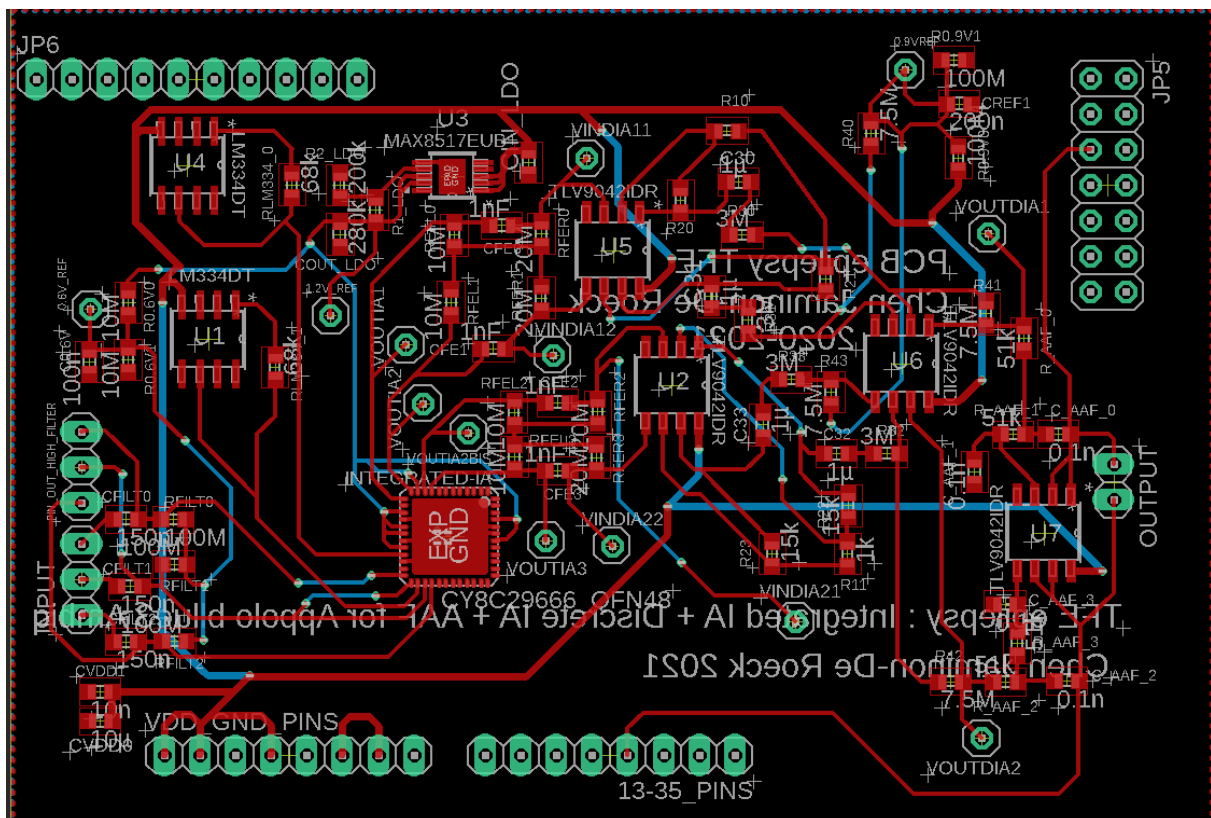
## Bibliographie :

- [1] Jaminon-De Roeck, Chen-Terry. Implementation and optimization of an ultra-low power vagus nerve sensing system for epileptic seizures detection. Ecole polytechnique de Louvain, Université catholique de Louvain, 2021. Prom. : Bol, David. <http://hdl.handle.net/2078.1/thesis:30551>
- [2] Lars Stumpp, Hugo Smets, Simone Vespa, Joaquin Cury, Pascal Doguet, Jean Delbeke, Emmanuel Hermans, Christian Sevcencu, Thomas N. Nielsen, Antoine Nonclercq, and Riem El Tahry. Recording of spontaneous vagus nerve activity during pentylenetetrazol-induced seizures in rats. Journal of Neuroscience Methods, 343:108832, 2020.
- [3] Ambiq, Apollo 3 Blue, <https://ambiq.com/apollo3-blue/>
- [4] Ambiq, Apollo 3 Blue datasheet, <https://ambiq.com/wp-content/uploads/2020/10/Apollo3-Blue-MCU-Datasheet.pdf>
- [5] Ambiq, Apollo 2 Blue, <https://ambiq.com/apollo2-blue/>
- [6] Ambiq, Apollo Blue, <https://ambiq.com/apollo/>
- [7] Ambiq, Apollo 4 Blue, Ambiq, Apollo 3 Blue, accessed January 26 2021, <https://ambiq.com/apollo4-blue/>
- [8] Mouser, "MAX8518EUB+T", <https://www.mouser.be/datasheet/2/256/MAX8516-1514578.pdf>
- [9] Mouser, "MCP1709AT-1202 E/CB", <https://www.mouser.be/ProductDetail/Microchip-Technology/MCP1703AT-1202E-CB?qs=PZkJWYMe7ld%252BTcdUZg0YJQ%3D%3D>
- [10] [https://github.com/schreiner/Apollo3Blue\\_SoftwareExamples](https://github.com/schreiner/Apollo3Blue_SoftwareExamples)
- [11] Texas Instrument, "TPS7A05 ", [https://www.ti.com/lit/ds/symlink/tps7a05.pdf?HQS=dis-mous-null-mouser-mode-dsf-pf-null-ww&ts=1623233199595&ref\\_url=https%253A%252F%252Fwww.mouser.co.il%252F](https://www.ti.com/lit/ds/symlink/tps7a05.pdf?HQS=dis-mous-null-mouser-mode-dsf-pf-null-ww&ts=1623233199595&ref_url=https%253A%252F%252Fwww.mouser.co.il%252F)
- [12] <https://matplotlib.org/stable/api/index>
- [13] <https://pyserial.readthedocs.io/en/latest/shortintro.html>

Annexe 1: Schéma PCB



Annexe 2: Board PCB



```

1  import serial
2  import matplotlib.pyplot as plt
3  import numpy as np
4  import sys
5
6
7  def read():
8      """
9      This function read the datas from the port COM.
10     The return of the function is:
11     - False: that can tell to the main function to stop the loop.
12     - Datas: This is the data that we read. But if we don't read a data this variable return false.
13     """
14     s = serial.Serial('COM'+sys.argv[1]) #Initialise the connection
15     s.baudrate = 230400
16     datas = []
17     data = ''
18     countExit = 0
19     while True:
20         countExit = countExit + 1
21         if countExit == 150000: #This is tell to the programm that we have any data because we still get out t
22             return False,False
23
24         temp = s.read_all().decode("UTF-8") # temps take the data in UTF-8 format
25         if temp != '': # avoid the empty string
26             for i in temp:
27                 if i == '-': #tell us the end of one sample of data
28                     count += 1
29                     datas.append(round(int(data)/16384 * 1.8,2)) # Add the data to a table with all the datas.
30                     data = ''
31                 else:
32                     data += i
33                 if len(datas)==1000: #stop the function when we have 1000 samples of data
34                     return False,datas
35
36 def graph(datas): #Make a graph with the data
37     x=np.linspace(0,10,1000)
38     plt.plot(x,datas)
39     plt.ylabel('Tension')
40     plt.xlabel("")
41     plt.show()
42
43
44 if __name__ == "__main__":
45     stop = True
46     count = 0
47     while stop:
48         try:
49             stop,datas = read()
50         except:
51             count = count+1
52             if count == 20:
53                 print("Port COM incorrect")
54                 sys.exit()
55     if datas == False:
56         print("Port COM incorrect")
57         sys.exit()
58     graph(datas)

```

```

1  | *****
2  * Copyright (C) 2016, Fujitsu Electronics Europe GmbH or a      *
3  * subsidiary of Fujitsu Electronics Europe GmbH.              *
4  *                                                              *
5  * This software, including source code, documentation and related materials *
6  * ("Software") is developed by Fujitsu Electronics Europe GmbH ("Fujitsu") *
7  * unless identified differently hereinafter for Open Source Software.      *
8  * All rights reserved.                                          *
9  *                                                              *
10 * This software is provided free of charge and not for sale, providing test *
11 * and sandbox applications. Fujitsu reserves the right to make changes to   *
12 * the Software without notice. Before use please check with Fujitsu        *
13 * for the most recent software.                                           *
14 *                                                              *
15 * If no specific Open Source License Agreement applies (see hereinafter),   *
16 * Fujitsu hereby grants you a personal, non-exclusive,                  *
17 * non-transferable license to copy, modify and compile the                *
18 * Software source code solely for use in connection with products          *
19 * supplied by Fujitsu. Any reproduction, modification, translation,        *
20 * compilation, or representation of this Software requires written         *
21 * permission of Fujitsu.                                                 *
22 *                                                              *
23 * NO WARRANTY: This software is provided as-is with no warranty of any kind *
24 * (German Unter Ausschluss jeglicher Gewährleistung), express or implied,  *
25 * including but not limited to non-infringement of third party rights,      *
26 * merchantability and fitness for use.                                     *
27 *                                                              *
28 * In the event the software deliverable includes the use of               *
29 * open source components, the provisions of the respective governing        *
30 * open source license agreement shall apply with respect to such software  *
31 * deliverable. Open source components are identified in the read-me files  *
32 * of each section / subsection.                                           *
33 *                                                              *
34 * German law applies with the exclusion of the rules of conflict of law.   *
35 *                                                              *
36 *                               ***                                       *
37 * September 2016                                                         *
38 * FUJITSU ELECTRONICS Europe GmbH                                         *
39 *                                                              *
40 *****/
41 /*****/
42 /** \file main.c
43 **
44 ** \brief UART + ADC for Apollo3
45 **
46 ** Main Module
47 **
48 **
49 *****/

```

```

50
51 /*****
52 /* Include files
53 /*****
54 #include "mcu.h"
55 #include "base_types.h"
56 #include "stdio.h"
57
58 /*****
59 /* Prototype functions
60 /*****
61
62 void ConfigureBaudrate(UART0_Type* pstcUart, uint32_t u32Baudrate, uint32_t u32Uart
63 void PutCharUart(UART0_Type* pstcUart, uint8_t u8Char);
64 void PutStringUart(UART0_Type* pstcUart, char_t *pu8Buffer);
65 uint8_t GetCharUart(UART0_Type* pstcUart);
66 void UartInit(UART0_Type* pstcUart, uint32_t u32Baudrate);
67 static volatile uint32_t u32Counter; //ms counter
68 void delay(uint32_t delayMs);
69
70
71 /*****
72 /* Procedures and functions
73 /*****
74 void SysTick_Handler(void)
75 {
76     u32Counter++;
77 }
78
79 void delay(uint32_t delayMs)
80 {
81     uint32_t u32End = u32Counter;
82     u32End += delayMs;
83     while(u32End != u32Counter) __NOP();
84 }
85 /**
86
87
88 ****
89 ** \brief Init UART...
90 **
91 ** \param pstcUart      UART pointer
92 **
93 ** \param u32Baudrate    Baudrate
94 **
95 ****
96 void UartInit(UART0_Type* pstcUart, uint32_t u32Baudrate)

```

```

97 {
98     //
99     // Enable UART clocking
100    //
101
102    PWRCTRL->DEVPWREN |= (1 << PWRCTRL_DEVPWREN_PWRUART0_Pos);
103
104    while(PWRCTRL->DEVPWRSTATUS_b.HCPA == 0) __NOP();
105
106    //
107    // Enable clock / select clock...
108    //
109    pstcUart->CR = 0;
110    pstcUart->CR_b.CLKEN = 1;           //enable clock
111    pstcUart->CR_b.CLKSEL = 1;         //use 24MHz clock
112
113    //
114    // Disable UART before config...
115    //
116    pstcUart->CR_b.UARTEN = 0;         //disable UART
117    pstcUart->CR_b.RXE = 0;           //disable receiver
118    pstcUart->CR_b.TXE = 0;           //disable transmitter
119
120
121    //
122    // Starting UART config...
123    //
124
125    // initialize baudrate before all other settings, otherwise UART will not be
126    SystemCoreClockUpdate();
127    ConfigureBaudrate(pstcUart,u32Baudrate,24000000UL);
128
129    // initialize line coding...
130    pstcUart->LCRH = 0;
131    pstcUart->LCRH_b.WLEN = 3;         //3 = 8 data bits (2..0 = 7..5 data
132    pstcUart->LCRH_b.STP2 = 0;         //1 stop bit
133    pstcUart->LCRH_b.PEN = 0;         //no parity
134
135    //
136    // Enable UART after config...
137    //
138    pstcUart->CR_b.UARTEN = 1;         //enable UART
139    pstcUart->CR_b.TXE = 1;           //enable transmitter
140 }
141
142 /**
143  ****

```



```

144  ** \brief Set baudrate
145  **
146  ** \param pstcUart      UART pointer
147  **
148  ** \param u32Baudrate   Baudrate
149  **
150  ** \param u32UartClkFreq UART clock
151  **
152  *****/
153 void ConfigureBaudrate(UART0_Type* pstcUart, uint32_t u32Baudrate, uint32_t u32UartClkFreq)
154 {
155     uint64_t u64FractionDivisorLong;
156     uint64_t u64IntermediateLong;
157     uint32_t u32IntegerDivisor;
158     uint32_t u32FractionDivisor;
159     uint32_t u32BaudClk;
160
161     //
162     // Calculate register values.
163     //
164     u32BaudClk = 16 * u32Baudrate;
165     u32IntegerDivisor = (uint32_t)(u32UartClkFreq / u32BaudClk);
166     u64IntermediateLong = (u32UartClkFreq * 64) / u32BaudClk;
167     u64FractionDivisorLong = u64IntermediateLong - (u32IntegerDivisor * 64);
168     u32FractionDivisor = (uint32_t)u64FractionDivisorLong;
169
170     //
171     // Integer divisor MUST be greater than or equal to 1.
172     //
173     if(u32IntegerDivisor == 0)
174     {
175         //
176         // Spin in a while because the selected baudrate is not possible.
177         //
178         while(1);
179     }
180     //
181     // Write the UART regs.
182     //
183     pstcUart->IBRD = u32IntegerDivisor;
184     pstcUart->IBRD = u32IntegerDivisor;
185     pstcUart->FBRD = u32FractionDivisor;
186 }
187

```

```

187
188  /**
189  *****
190  ** \brief sends a single character (no timeout !)
191  **
192  ** \param pstcUart  UART pointer
193  **
194  ** \param u8Char    Data to send
195  **
196  *****/
197  void PutCharUart(UART0_Type* pstcUart, uint8_t u8Char)
198  {
199      while(pstcUart->FR_b.TXFF) __NOP();
200      pstcUart->DR = u8Char;
201  }
202
203  /**
204  *****
205  ** \brief sends a complete string (0-terminated)
206  **
207  ** \param pstcUart  UART pointer
208  **
209  ** \param Pointer to (constant) file of bytes in mem
210  **
211  *****/
212  void PutStringUart(UART0_Type* pstcUart, char_t *pu8Buffer)
213  {
214      while (*pu8Buffer != '\0')
215      {
216          PutCharUart(pstcUart,*pu8Buffer++);      // send every char of string
217      }
218  }
219

```

```

219
220  /**
221  ****
222  ** \brief Main function
223  **
224  ** \return int return value, if needed
225  ****
226  int main(void)
227  {
228      uint32_t u32Cfg;
229      uint32_t u32Data;
230      char data[10];
231      SystemCoreClockUpdate(); //update clock variable SystemCoreClock (defined by CMSIS)
232      SysTick_Config(SystemCoreClock / 1000); //setup 1ms SysTick (defined by CMSIS)
233      //application initialization area
234
235      GPIO->PADKEY = 0x73; //unlock GPIO configuration
236      GPIO->PADREGI_b.PAD32FNCSEL = GPIO_PADREGI_PAD32FNCSEL_ADCSE4; //select ADC4 at PAD32
237      GPIO->PADKEY = 0;
238
239      //
240      // Enable power for ADC and wait power is enabled
241      //
242      PWRCTRL->DEVPWREN_b.PWRADC = 1;
243      while(PWRCTRL->ADCSTATUS_b.ADCPWD == 0) __NOP();
244
245      ADC->CFG = 0;
246
247
248      u32Cfg =
249      | _VAL2FLD(ADC_CFG_CKMODE,ADC_CFG_CKMODE_LLCKMODE) //Low Latency Clock Mode
250      | _VAL2FLD(ADC_CFG_CLKSEL,ADC_CFG_CLKSEL_HFRC) //HFRC clock
251      | _VAL2FLD(ADC_CFG_TRIGSEL,ADC_CFG_TRIGSEL_SWT) //Software trigger
252      | _VAL2FLD(ADC_CFG_REFSEL,ADC_CFG_REFSEL_INT2P0) //internal 2V reference
253      | _VAL2FLD(ADC_CFG_LPMODE,ADC_CFG_LPMODE_MODE1) //Low Power Mode 1
254      | _VAL2FLD(ADC_CFG_RPTEN,ADC_CFG_RPTEN_SINGLE_SCAN) //Single scan
255      | _VAL2FLD(ADC_CFG_ADCEN,1); //Enable ADC
256
257      //
258      // Using slot 0
259      //
260      ADC->SL0CFG = 0;
261      ADC->SL0CFG_b.CHSEL0 = ADC_SL0CFG_CHSEL0_SE4; //use ADCSE8 as input;
262      ADC->SL0CFG_b.ADSEL0 = ADC_SL0CFG_ADSEL0_AVG_1_MSRMT; //Average over 16 measurements
263      ADC->SL0CFG_b.PRMODE0 = ADC_SL0CFG_PRMODE0_P14B; //use 14-bit 1.2MS/s
264      ADC->SL0CFG_b.WCEN0 = 0; //window comparator mode disabled
265      ADC->SL0CFG_b.SLEN0 = 1; //slot enabled
266

```

```

265
266 ADC->WULIM_b.ULIM = 0; //Window comparator upper limit (not used)
267 ADC->WLLIM_b.LLIM = 0; //Window comparator lower limit (not used)
268
269 ADC->INTEN_b.WCINC = 1; //enable window comparator voltage incursion interrupt
270 ADC->INTEN_b.WCEXC = 1; //enable window comparator voltage excursion interrupt
271 ADC->INTEN_b.FIFO0VR1 = 1; //enable FIFO 100% full interrupt
272 ADC->INTEN_b.FIFO0VR2 = 1; //enable FIFO 75% full interrupt
273 ADC->INTEN_b.SCNCMP = 1; //enable ADC scan complete interrupt
274 ADC->INTEN_b.CNVCMP = 1; //enable ADC conversion complete interrupt
275
276 ADC->CFG = u32Cfg;
277
278 ADC->INTCLR = 0xFFFFFFFF; //clear interrupts
279 ADC->SWT = 0x37; //trigger ADC
280
281
282
283 UartInit(UART0,230400); //init UART with 115200 baud
284
285 GPIO->PADKEY = 0x00000073; //unlock pin selection
286
287 GPIO->PADREGF_b.PAD22FNCSEL = 0; //set UARTTX on pin 22
288 GPIO->PADREGF_b.PAD22INPEN = 1; //enable input on pin 22
289 GPIO->CFG_b.GPIO220UTCFG = 1; //output is push-pull
290
291
292 GPIO->PADKEY = 0; //lock pin selection
293
294
295 while(1)
296 {
297     if (ADC->INTSTAT_b.CNVCMP == 1)
298     {
299         ADC->INTCLR_b.CNVCMP = 1;
300
301         if ((_FLD2VAL(ADC_FIFO_COUNT,ADC->FIFO) > 0) && (_FLD2VAL(ADC_FIFO_SLOTNUM,ADC->FIFO) == 0))
302         {
303             u32Data = _FLD2VAL(ADC_FIFO_DATA,ADC->FIFO) >> 6;
304             sprintf(data, "%d-",u32Data);
305             PutStringUart(UART0,data);
306             ADC->FIFO = 0; // pop FIFO
307         }
308     }
309
310     if (ADC->INTSTAT_b.SCNCMP == 1)
311     {
312         ADC->INTCLR_b.SCNCMP = 1;
313         ADC->SWT = 0x37;
314     }
315 }
316
317 return 0;
318
319
320
321 /******
322 /* EOF (not truncated) */
323 /******
324

```