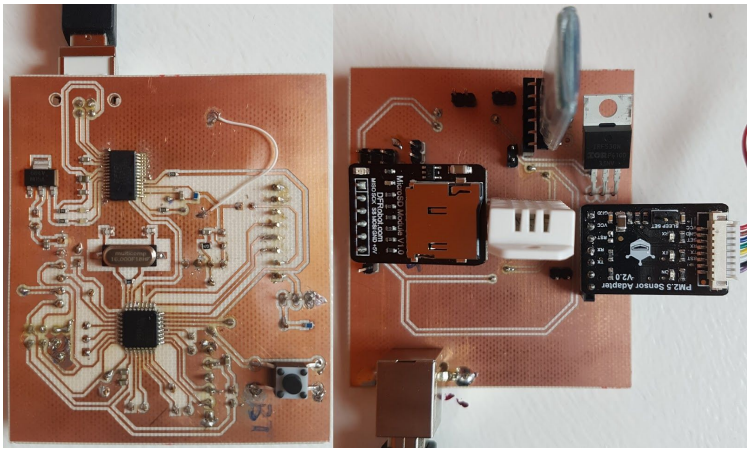
Projet n°63
Etude de la consommation d'un capteur de pollution



Encadrants : M. BOE, M. Redon, M. VANTROYS

IMA4
Année 2018-2019

LORTHIOS Victor
OBLED Juliette

Remerciements

Nous tenons particulièrement à remercier nos encadrants, M. Boé, M. Redon et M. Vantroys pour leur aide et leur disponibilité. Nous pensons notamment au début de notre projet avec l'importante partie électronique où leurs conseils nous ont été très précieux, ainsi que pendant les vacances où nous avons pu accéder aux salles et matériels de l'école.

Nous remercions également M. Flamen, qui a été très présent pour nous conseiller avec beaucoup de bienveillance.

Enfin, nous remercions nos camarades qui n'ont pas hésité à nous apporter de l'aide quand nous en avions besoin.

Sommaire

Introduction

De nos jours, la pollution est un enjeu sanitaire majeur. En effet, les polluants atmosphériques sont à l'origine de nombreux décès en France et dans le monde (48000 victimes en France selon l'agence de santé publique). Pour réduire et éviter les endroits où la pollution est la plus dense, il est très intéressant de pouvoir analyser la présence de ces particules fines. Ces dernières années, de nombreux moyens ont été mis en oeuvre pour surveiller le taux de pollution. On retrouve par exemple de nombreux objets connectés composés de capteurs capables de mesurer la qualité de l'air. Cela permet notamment de sensibiliser la population à ce problème.

Dans le cadre de notre 4ème année d'étude d'ingénieur dans la spécialité IMA, nous avons donc décidé de travailler sur l'étude de la consommation d'un capteur de pollution simple. L'objectif est donc d'établir un modèle, pouvant déterminer à l'avance l'autonomie de notre circuit, l'idéal étant d'avoir un système efficace consommant le moins d'énergie possible.

Tout d'abord, nous allons voir les détails et contraintes de notre projet, les choix techniques associés ainsi que notre positionnement face à la concurrence. Dans un second temps, nous expliquerons les capteurs et codes associés utilisés. Ensuite, nous verrons la réalisation de la carte électronique, d'abord avec un module arduino puis la version améliorée finale. Enfin, nous verrons les consommations associées à nos différents éléments selon les configurations, et le modèle utilisé pour estimer l'autonomie d'une batterie liée à notre système.

1. Cahier des charges

1.1. Objectifs et choix techniques

Comme évoqué plus tôt, ce projet consiste à modéliser la consommation d'un capteur de pollution. Notre système sera donc composé d'un microcontrôleur atmega, un capteur de particules, un capteur de température/humidité, une carte SD ainsi qu'un système Bluetooth pour transmettre les données.

Dans un premier temps, nous allons travailler avec une carte Arduino afin de se familiariser avec les différents éléments et établir les programmes. Cependant, aucune consommation énergétique ne sera relevée sur cette configuration puisque l'Arduino possède trop de fonctions parasites qui peuvent altérer nos valeurs. C'est pour cela que nous allons ensuite réaliser un système embarqué uniquement composé des éléments nécessaires (cités plus haut). Le premier objectif est donc d'obtenir des valeurs cohérentes et ainsi qu'un système fonctionnel. Pour réaliser la carte électronique, nous utiliserons le logiciel Altium pour lequel l'école peut nous fournir la license. Pour le code, nous utiliserons le logiciel Arduino pour sa facilité d'utilisation.

Dans un second temps, le système doit être le moins énergivore possible. Théoriquement, c'est le capteur de particules PM2.5 qui consomme le plus de courant. C'est donc sur cet élément que nous allons disposer différentes configurations : nous allons tester le mode continu du capteur, le mode veille, et également le fait de l'éteindre électriquement. Cette dernière méthode est faisable avec l'utilisation d'un interrupteur sur l'alimentation du capteur. Comme vu en cours d'électronique, les transistors mosfet ont la capacité - selon leur type bien sûr - de pouvoir agir comme un interrupteur. De ce fait, en commandant le mosfet on pourra éteindre électriquement le PM2.5 sans impact sur le reste du système. Tout cela doit évidemment être pris en compte dans la première partie du projet.

Enfin, l'objectif final est de réaliser différentes études énergétiques en vue de définir un modèle pouvant anticiper l'autonomie de la batterie liée à notre système. Pour cela, nous étudierons d'abord les données reçues par le capteur PM2.5 selon les configurations évoquées. Ensuite, nous évaluerons laquelle des configurations est la plus appropriée selon la période de prise entre deux mesures. Pour finir, nous verrons comment estimer la durée de vie d'une batterie en prenant en compte cette fois tous les éléments de notre système. Afin de pouvoir effectuer ces études énergétiques, de nombreux headers seront disposés sur la carte afin de relever facilement le courant et la tension traversant les composants.

1.2. Etat de l'art

Avant de commencer le projet, nous avons réalisé une étude du marché pour savoir comment se baser sur le fonctionnement de ce capteur. Nous avons notamment trouvé deux entreprises qui se positionnent dans le même usage que notre système.

Tout d'abord l'entreprise française Plume Labs, fondée en 2014 dans le but d'améliorer notre quotidien face à la pollution. Elle commercialise Flow, un objet de 12,5 cm de haut et 4cm de large permettant d'analyser la pollution de l'air en temps réel. En appuyant sur un bouton, on peut relever la concentration des principaux polluants dans l'air.

Le résultat s'affiche directement sur l'objet, par l'allumage de leds : une couleur sombre correspond à un taux de pollution élevé et une couleur claire un air sain. Les résultats peuvent également être transmis via bluetooth sur le téléphone afin d'être affichés sur l'application Air Report. Cette application est composée d'une carte permettant de géolocaliser le téléphone et ainsi représenter le taux de pollution de chaque endroit de la ville. Toutes les personnes utilisant Flow peuvent donc contribuer à améliorer cette carte. Ce capteur permet donc d'évaluer ses parcours dans la ville, et inciter les gens à passer par des endroits plus sains. Il est également utilisable dans la maison.
Étant si petit, résistant (fait d'acier) et ne pesant que 70g, Flow est transportable partout. Le capteur se recharge facilement sur ordinateur ou secteur grâce à un câble USB-C vers USB, et est supposé avoir une autonomie de 24h. Ce capteur est donc très avancé mais coûte tout de même 179 euros.
Nous verrons par la suite de ce rapport si nous arrivons à une autonomie aussi importante, nous pouvons cependant nous avancer sur le prix puisque la totalité des capteurs coûte environ 93 euros (mais sans support ni batterie).

Nous avons ensuite Enedis qui propose Pollutrack, un système du mesure de pollution de l'air installé sur des véhicules électriques. Il permet de mesurer la pollution près du trafic dans des grandes villes telles que Paris ou Lille. Plus de 300 capteurs mobiles Pollutrack sont utilisés en complément de stations qui mesurent en temps réel la quantité de particules nocives dans l'air. Le dispositif permet de récupérer un très grand nombre de mesures. Grâce à ces mesures, il est possible de comparer le niveau de pollution d'une rue à une autre, d'un quartier à un autre et d'ainsi repérer les zones de concentration de la pollution.

Nous avons alors réfléchi à comment pourrait être utilisé notre capteur. Le site https://www.airparif.asso.fr/ mesure et cartographie précisément la pollution de Paris et de son agglomération en temps réel. L'ensemble de leurs données permettent de produire une carte avec au mieux une résolution de 10 mètres par mesure. En prenant l'exemple d'un piéton marchant à la vitesse de 4 km/h il faudrait une mesure toutes les 9 secondes pour avoir une résolution de 10 mètres. Si on prend le cas d'un cycliste roulant à la vitesse de 20km/h il faudrait une mesure toutes les 2 secondes pour atteindre une résolution de 10m. Enfin, le capteur peut également être utilisé de manière fixe dans une usine par exemple, nos modélisations permettraient à l'entreprise de connaître combien de temps le capteur serait utilisable et donc estimer le moment de la journée le plus propice pour effectuer des mesures. Dans ce cas, une période plus grande de plusieurs minutes jusqu'aux heures pourrait être envisagée.

Nous avons pu alors commencer la réalisation de notre capteur.

2. Etudes et programmes des différents éléments

2.1. Capteur de particules PM2.5

Le premier objectif fut de trouver une solution pour pouvoir brancher et récupérer les données reçues par le capteur sur une arduino UNO. Ainsi après quelques recherches sur internet, nous avons trouvé un document expliquant le branchement à réaliser ainsi que la forme des données reçues par le capteur. Le capteur communique via un port série.

Communication protocol

Serial port baudrate: 9600; Parity: None; Stop Bits: 1; packet length is fixed at 32 bytes.

Start Character 1	0x42(fixed bit)
Start Character 2	0x4d(fixed bit)
Frame Length 16-byte	Frame Length = 2*9+2 (data+check bit)
Data 1, 16-byte	concentration of PM1.0, ug/m3
Data 2, 16-byte	concentration of PM2.5, ug/m3
Data 3, 16-byte	concentration of PM10.0, ug/m3
Data 4, 16-byte	Internal test data
Data 5, 16-byte	Internal test data
Data 6, 16-byte	Internal test data
Data 7, 16-byte	the number of particulate of diameter above 0.3um in 0.1 liters of air
Data 8, 16-byte	the number of particulate of diameter above 0.5um in 0.1 liters of air
Data 9, 16-byte	the number of particulate of diameter above 1.0um in 0.1 liters of air
Data 10, 16-byte	the number of particulate of diameter above 2.5um in 0.1 liters of air
Data 11, 16-byte	the number of particulate of diameter above 5.0um in 0.1 liters of air
Data 12, 16-byte	the number of particulate of diameter above 10.0um in 0.1 liters of air
Data 13, 16-byte	Internal test data
Check Bit for Data Sum, 16-byte	Check Bit = Start Character 1 + Start Character 2 + ...all data

Format des données envoyés par le capteur

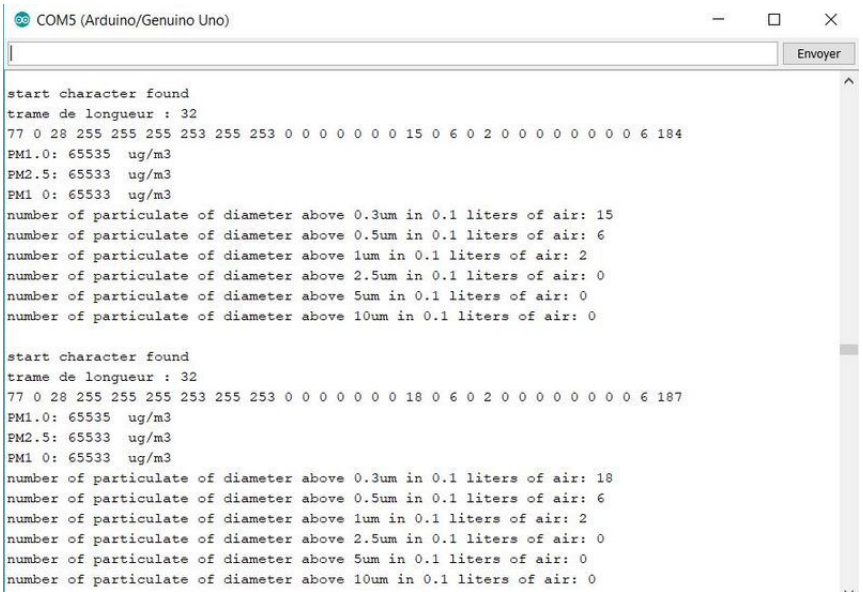
Dans un premier temps, nous avons réalisé le branchement du capteur sur une carte arduino. Pour cela, on branche simplement le vcc et le gnd du capteur aux 5v et gnd de la carte arduino, on branche également les Rx Tx du capteur aux Tx Rx de la carte arduino (pour éviter un conflit avec le moniteur série, on utilise la fonction "SoftwareSerial" qui nous permet de de créer une liaison série sur des pins choisis sur la carte). On connecte enfin la pin sleep à une pin de l'arduino définie en sortie pour pouvoir contrôler la mise en veille du capteur. Une fois le branchement effectué, nous avons simplement essayé d'afficher la trame telle quelle. Pour cela on vérifie le caractère de début de trame (avec la fonction find) et on place le reste des données dans un buffer de taille correspondant au reste des données. On est alors capable d'observer les données.

```
32
66 77 0 28 255 255 255 253 255 253 0 0 0 0 0 0 0 13 0 6 0 4 0 0 0 0 0 0 0 0 6 184
```

Affichage de la trame brute avec les données à récupérer encadrées

On est également capable de mettre en veille le capteur en pilotant la pin sleep. Nous avons aussi effectué des test en déposant de la poussière de craie à proximité du capteur. Ainsi on observe bien que les valeurs des différentes particules changent. Cependant, on observe un problème. En effet, pour les valeurs de concentration des différentes particules (celles encadrés en rouge), quand le capteur ne détecte aucune particule il affiche la valeur max plutôt que d'afficher 0. C'est un problème lié au capteur puisqu'en fin de projet, nous avons eu l'occasion de travailler avec un second capteur identique qui lui fonctionnait sans problème.

Nous avons ensuite étoffé le programme pour qu'il récupère les valeurs utiles et qu'il les affiche plus proprement. On obtient le résultat suivant.



L'objectif de notre projet étant l'étude énergétique de ce capteur, une autre option que le mode sleep pour économiser de l'énergie est l'extinction électrique du capteur. Pour cela, nous avons utilisé un transistor MOSFET qui nous permet d'éteindre le capteur sur commande.
Pour ce faire, on connecte sur la breadboard le MOSFET de la façon ci-contre. On connecte la grille à la commande qui sera en réalité une pin de l'arduino. On branche le mosfet entre la pin gnd du capteur et la masse. la résistance entre la grille et la source est une résistance de pull down. Nous avons choisi une résistance de 100kΩ.
On est ainsi capable avec un simple "digital write" de contrôler l'état du capteur.

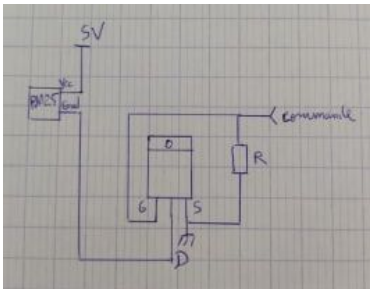


schéma du mosfet utilisé pour contrôler l'état du capteur

Pour les mesures énergétique futures, nous avons réalisé un programme qui met le capteur en mode sleep pendant un certain temps, le rallume, attend un certain temps, éteint le capteur électriquement et finalement le rallume. Nous avons utilisé la fonction "millis" qui n'est pas bloquante contrairement à la fonction "delay" qui aurait suspendu toutes les autres tâches. Nous expliquerons l'intérêt de ce programme dans la suite du rapport.
Tous les codes utilisés sont trouvables en annexe.

2.2. Capteur de température DHT22

Le DHT22 est un capteur qui permet de mesurer une température comprise entre -40° et 80° avec une précision de 0.5° et le taux d'humidité avec une précision de 2%. Il est bien plus précis que son confrère le DHT11. Contrairement au PM2.5, le DHT22 n'a besoin que d'une broche d'entrée sortie pour communiquer avec l'Arduino. Il utilise un protocole de communication spécifique. La démarche est la suivante : l'Arduino (maître) réveille le capteur (esclave) pendant un certain temps. Pendant ce temps, le capteur se réveille et prépare une mesure. Une fois ce temps écoulé, l'arduino va se mettre à écouter pendant que le capteur émet un signal pour montrer qu'il est prêt. Le capteur va finalement transmettre à l'arduino un paquet de 5 octets contenant toutes les informations nécessaires.

Table 5: AM2302 Communication format specifier

Name	Single-bus format definition
Start signal	Microprocessor data bus (SDA) to bring down a period of time (at least 800μ s) [1] notify the sensor to prepare the data.
Response signal	Sensor data bus (SDA) is pulled down to 80μ s, followed by high-80μ s response to host the start signal.
Data format	Host the start signal is received, the sensor one-time string from the data bus (SDA) 40 data, the high first-out.
Humidity	Humidity resolution of 16Bit, the previous high; humidity sensor string value is 10 times the actual humidity values.
Temp.	Temperature resolution of 16Bit, the previous high; temperature sensor string value is 10 times the actual temperature value; The temperature is the highest bit (Bit15) is equal to 1 indicates a negative temperature, the temperature is the highest bit (Bit15) is equal to 0 indicates a positive temperature; Temperature in addition to the most significant bit (Bit14 ~ bit 0) temperature values.
Parity bit	Parity bit = humidity high + humidity low + temperature high + temperature low

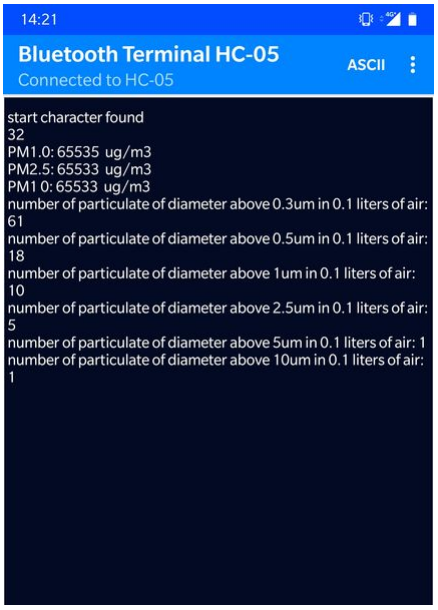
Format d'une trame de communication

Le branchement est très simple : on alimente le capteur depuis la carte arduino et on branche la broche DATA sur une broche d'entrée sortie de l'arduino. On s'assurera également de connecter la broche DATA à une résistance pull-up reliée à VCC.

Pour le code, il y avait énormément d'informations et de sources disponibles sur internet que nous avons décidé d'en récupérer une qui a très bien fonctionnée.

2.3. Module Bluetooth HC05

Le HC05 est un module Bluetooth qui communique en série avec une arduino et envoie et/ou reçoit des données. Dans notre cas, nous l'avons juste utilisé en émetteur afin d'envoyer en bluetooth les données reçues par le PM2.5 et le DHT22. Il est composé de quatre broches (vcc,gnd,rx,tx). Pour le faire fonctionner, on l'alimente via la carte arduino (broches vcc et gnd) et on relie les rx,tx vers les tx,rx de la carte arduino (pin 0 et 1). Ainsi, à la place d'observer les valeurs via le moniteur série, on les envoie via le module. En installant une application jouant le rôle de terminal bluetooth ("Bluetooth terminal hc-05") sur un smartphone, on observe les données envoyées par le capteur.



Affichage des données transmises sur le smartphone

On est désormais capable d'envoyer les données du capteur via bluetooth mais il serait également intéressant de pouvoir stocker les valeur récupérées par le capteur si personne n'est en mesure de les récupérer via le bluetooth.

2.4. Module carte SD

Le module carte SD est conçu pour accueillir une carte SD. Il permet de lire et d'écrire des données sur la carte grâce à une communication via un bus SPI. Le bus SPI est un bus de communication série synchrone (synchronisé avec une horloge). Il fonctionne en maître esclave. Ici le maître sera la carte arduino et l'esclave sera le module. On trouve sur les cartes arduino des ports dédiés à la communication SPI. Pour connecter le module, on l'alimente via la carte arduino et on branche les broches MISO et MOSI (qui transmettent les données) et la broche SCK (l'horloge) aux pin dédiées à cet effet sur l'arduino. On doit également connecter le CS (chip select) pour pouvoir activer la communication avec la carte SD.

Une fois cela fait, il ne reste plus qu'à réaliser un programme permettant d'ouvrir la carte et d'y écrire des données. Pour se faire, nous avons utilisé la bibliothèque SD prévue à cet effet. Ainsi nous avons conçu des fonctions qui permettent chaque démarrage de créer deux fichiers textes (les écraser s'ils existent déjà) : l'un est destiné à stocker les valeur du PM2.5 et l'autre à stocker celles du DHT22. Pour pouvoir analyser les données par la suite, il faut récupérer le temps correspondant à chaque mesure. Pour ce faire, on utilise la fonction "millis" qui est une fonction permettant de récupérer le temps écoulé depuis que le

programme s'est lancé. ainsi, à chaque fois qu'on relève une mesure, on appelle "millis" pour récupérer le temps écoulé. On s'assure également de séparer chaque donnée par un espace afin de traiter plus facilement les données sur une feuille de calcul.

Temps	Humidité	Température
0,01	46,4	20,8
1,62	46,4	20,8
2,57	46,4	20,8
3,55	46,4	20,8
4,53	46,4	20,8

DHT22 - Bloc-notes

Fichier Edition Format Affichage

```
0.01 46.40 20.80
1.62 46.40 20.80
2.57 46.40 20.80
3.55 46.40 20.80
4.53 46.40 20.80
```

Exemple d'écriture sur la carte sd pour le DHT22

Les différents éléments à intégrer à notre carte ont donc tous été testé. Nous avons réalisé un programme regroupant les codes utilisés pour chaque élément afin de simuler le système complet.

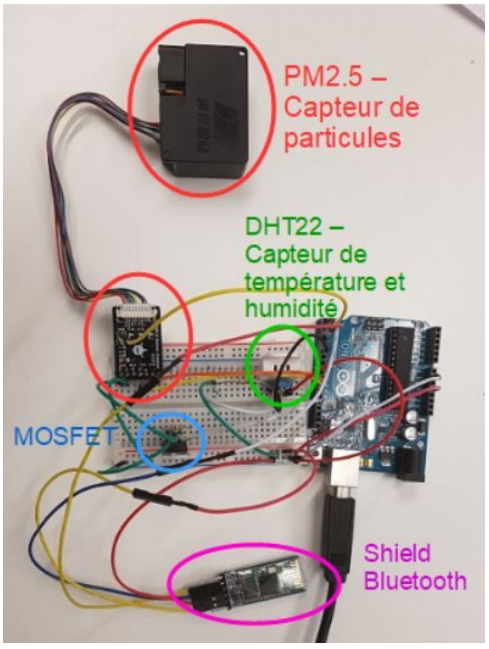


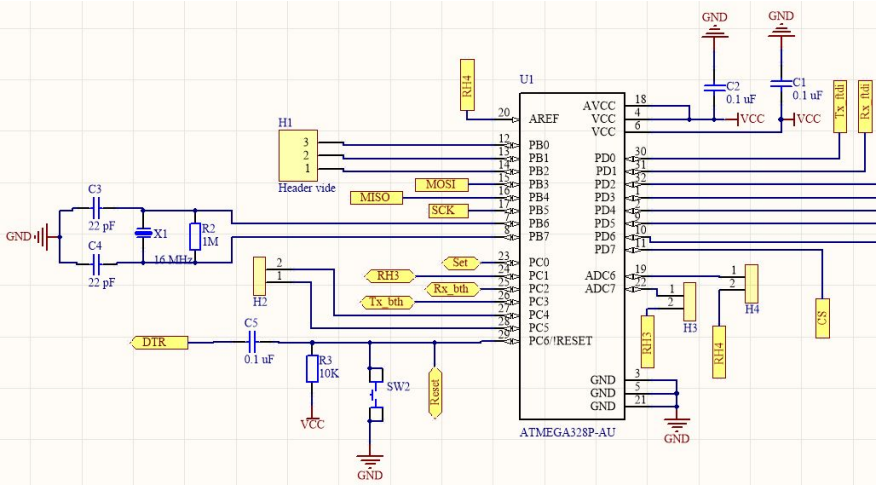
photo des différents éléments présentés connectés sur l'arduino

3. Réalisation des cartes électroniques

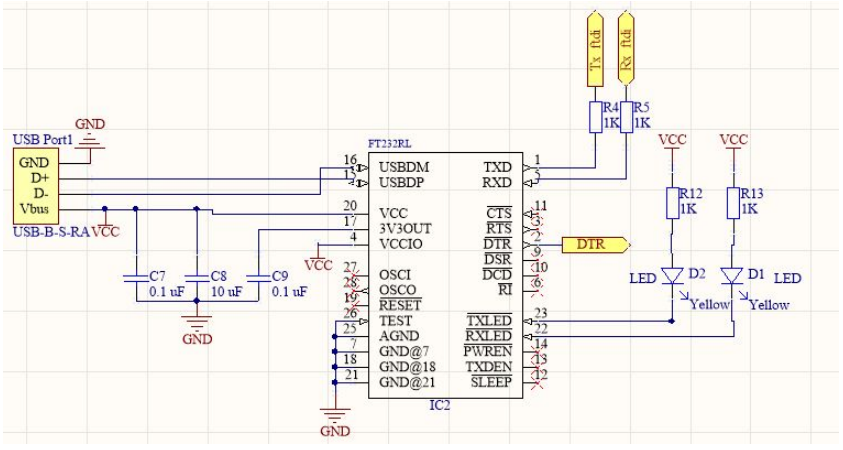
En parallèle, nous avons donc commencé à réaliser la carte électronique sur Altium. Comme évoqué plus tôt, l'Arduino est très pratique pour tester facilement si nos connections et programmes fonctionnent, cependant pour notre projet où nous devons étudier la consommation de chaque élément, il est indispensable de réaliser une carte simplifiée sans fonctions parasites.

3.1. Schématique de la carte principale

Commençons par l'élément central de notre système : le microcontrôleur. Au vu des fonctions assez basiques que nous utilisons, nous nous sommes inspirés des schématiques Arduino UNO et nous avons décidé d'utiliser un Atmega328p-AU pour gérer tous nos éléments. Combiné à ce microcontrôleur nous avons un FTDI et un connecteur USB, qui permettent de téléverser les programmes de l'ordinateur vers l'Atmega.



connections atmega328p-AU partie1

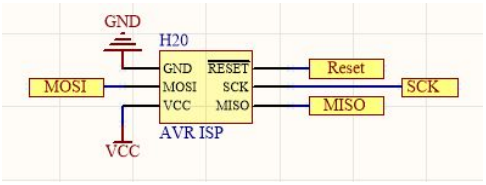


connections FTDI

On peut donc voir sur la partie atmega que nous avons repris les schématiques de l'Arduino au niveau du quartz, du reset et de l'alimentation (que nous protégeons avec des capacités de découplage). Les headers H1, H2, H3, H4 sont des pins non utilisées que nous faisons quand même "sortir" sur la carte au cas où nous aurions un problème. Il y a quatre header plutôt qu'un seul afin que le PCB que nous verrons plus tard soit plus facile et comprenne moins de vias.

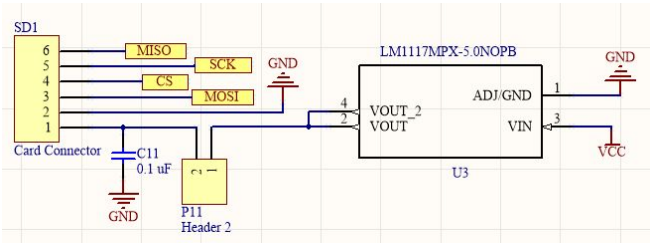
Le FTDI est très simplement connecté au port USB selon sa datasheet. Nous avons ajouté des led pour s'assurer que les programmes se téléversent bien (et qu'il n'y pas de problème au moins sur cette partie de la carte).
L'atmega et le FTDI sont reliés au niveau du reset (broche DTR), et surtout par les connections RX et TX pour le transfert de données.

Comme dit précédemment, nous avons réalisé tous nos programmes sur le logiciel Arduino IDE. Afin de les téléverser correctement sur l'Atmega, nous avons besoin d'un module ISP.



Les broches MISO et MOSI permettent la transmission des données, et SCK correspond à l'horloge. Elles sont connectées respectivement aux pins PB3, PB4, PB5 de l'atmega comme énoncé dans les datasheets.

De plus, puisque ces broches sont nécessaires aux transferts de données, elles sont donc évidemment également utilisées pour l'utilisation de la carte SD qui a besoin de récupérer les données de l'Atmega. Le connecteur s'alimentant en 3.3v, nous avons ajouté un régulateur 5v/3.3v en entrée. Il y a également le CS (chip select) connecté à une des broches de l'Atmega, ainsi qu'une capacité de découplage.



On peut remarquer qu'il y a des headers en série avec l'alimentation de chaque gros élément de notre système. En effet, le but est de pouvoir lire le courant et la tension passant dans chacun afin de pouvoir par la suite effectuer nos relevés et ainsi réaliser nos études énergétiques.
Le module bluetooth, le PM2.5 et le DHT22 sont donc liés de manière identique à notre montage Arduino, avec la seule différence qu'ils ont tous un header pour la lecture de puissance et qu'il y a des capacités de découplage aux masses. Nous avons également ajouté un header particulier pour le PM2.5 pour câbler notre pin sleep. Nous avons utilisé un header *2 afin d'en plus pouvoir accéder à la masse par un câble. Cela nous a été utile pendant les tests.

On peut retrouver le schématique complet en annexe.

3.2. Routage du PCB et soudure des composants de la carte principale

Nous avons alors pu router la carte. Cette partie fut très longue. En effet, il y a beaucoup de composants et nous n'avions au début pas vraiment de bonne technique pour correctement faire le routage. Chaque petit changement dans notre schématique impliquait de reprendre à presque zéro notre PCB.

En suivant les conseils de nos encadrants, nous avons donc placé nos composants traversants sur le "Bottom" de notre carte, et les composants CMS sur le "Top". Cela permet de souder toutes les connections d'un seul côté de la carte, et d'avoir toutes les sorties de header qui ne posent pas de problème avec les petits composants.
Nous avons fait attention à laisser de la place pour les éléments comme le PM2.5 qui sont en longueur et non en hauteur. Cependant, nous n'avons pas laissé assez de place pour certains éléments et nous avons donc dû utiliser des petits câbles pour les connecter et éviter de trop les tordre et les abimer. Cela n'a pas posé de problème mais est assez dommage puisque l'on perd l'intérêt d'utiliser une carte.
Nous avons également pris soin de disposer les capacités de découplage près des éléments auxquels elles réfèrent.

Une fois tous les éléments placés, nous avons donc routé la carte. Nous avons donc deux plans de masse (un de chaque côté) pour une belle uniformisation de cette dernière. Nous avons évité au mieux les angles aigus et les zones isolées. Enfin, nous avons placé des vias pour résoudre les dernières erreurs. Plusieurs vérifications avec les encadrants ont été faites avant d'imprimer la carte.

Pour imprimer la carte justement, nous avions dans un premier temps été conseillés de la faire réaliser par des industriels. Cependant, cette dernière a pris du retard et il a alors été nécessaire de l'imprimer à l'école pour ne pas perdre trop de temps.

Nous avons donc pu souder la carte. Nous avons d'abord souder tous les composants CMS à l'aide du four à refusion de M. FLAMEN puis les vias. Nos encadrants nous ont ensuite aidé à tester si les éléments les plus importants (atmega, ftdi, isp, connecteur USB) étaient correctement connectés et arrivaient à communiquer entre eux. Pour cela, nous avons téléversé un bootloader pour pouvoir comme dit plus tôt pour pouvoir ensuite téléverser les programmes depuis le FTDI. Puis nous avons essayé de faire clignoter les LED de sécurité d'abord du FTDI puis d'une Arduino connectée à l'Atmega pour savoir si le microcontrôleur ne présentait aucun problème. Fort heureusement pour nous et parce que notre schématique et PCB étaient corrects, nous n'avons pas eu de problème de connection. Nous avons donc pu souder le reste des composants, ou plutôt tous les headers ainsi que le mosfet.
Après avoir fait des tests de connection ainsi que quelques retouches, nous avons essayé nos différents programmes, avec succès.

Les deux faces du PCB ainsi que les photos de la carte une fois imprimée et soudée sont en annexes.

3.3. Montage High-side

Nous avons également essayé de réaliser un deuxième circuit permettant de mesurer la valeur du courant passant dans nos capteurs. En connectant les headers de notre circuit principal au header de gauche, nous allons pouvoir lire la valeur du courant sur le header de droite. Nous nous sommes inspirés de la datasheet LTC6800 pour réaliser ce circuit.
Il a fallu cependant adapter la valeur de la résistance en entrée du LTC6800. En effet, sa valeur dépend du composant pour lequel on va mesurer le courant.

Après quelques recherches, nous avons pu trouver les valeurs de courant suivantes :
~120mA pour le PM2.5, entre 1 et 15 mA pour le DHT22 et entre 15 et 60 mA pour le HC-05. On a donc une plage de variation du courant égale à [1;120] mA. Pour correctement relever le courant, nous devons donc avoir en sortie une tension variant de 0 à 5V pour le courant allant de 0 à 120mA.

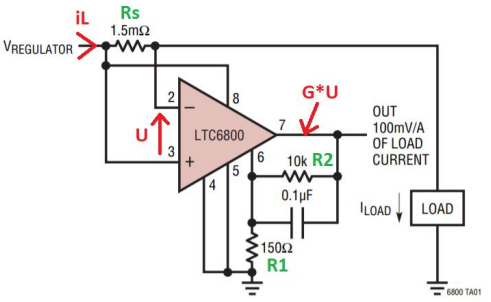


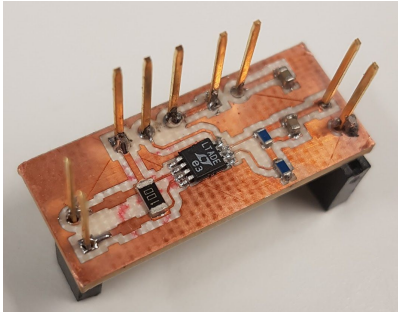
Schéma des grandeurs sur le montage d'utilisation du LTC6800

Nous nous basons sur le schéma ci-contre trouvé dans les datasheet du LTC6800 pour les calculs.

U=Rs*iL
Vout=G*U=G*Rs*iL => G=Vout(Rs*iL)
Or G=1+R2/R1
En prenant Vout/iL = 40mV/mA on a bien 4.8V pour 120mA. Soit G*Rs = 40mV/mA. On en déduit G et Rs.
Rs est limité par la puissance qu'il peut supporté. Si on prend Rs = 10 Ohm (G=4), on a donc une puissance de $P=Rs*iL^2$, $Pmax=10*0.12^2=0.144W$. Avec une résistance supportant 0.25W (selon ses propriété), on peut donc prendre cette résistance de 10 Ohm.
G est défini par R2 et R1. On doit donc prendre R2/R1=3 donc par exemple R2 = 30 kOhm et R1 = 10kOhm. Comme 30kOhm n'est pas une valeur normalisée, on prend R2=27kOhm, notre gain sera plus petit.

Nous avons alors réalisé le schématique ainsi que le PCB de cette carte, puis nous l'avons imprimé et soudé.

Cependant, au moment de l'utiliser, nous avons remarqué qu'il y avait des problèmes au niveau de nos headers et que les mesures étaient finalement plus faciles à réaliser au multimètre et à l'oscilloscope (en plus d'être plus précises). Nous ne nous sommes donc pas servis de cette carte.

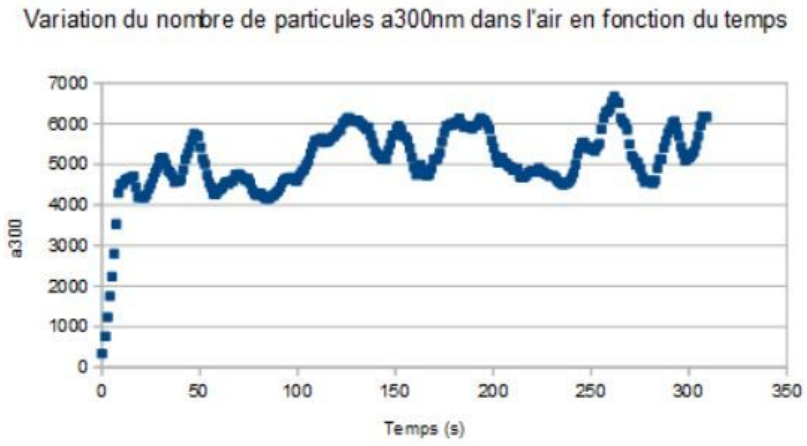


4. Études énergétiques

4.1. Étude des données envoyées par le PM2.5

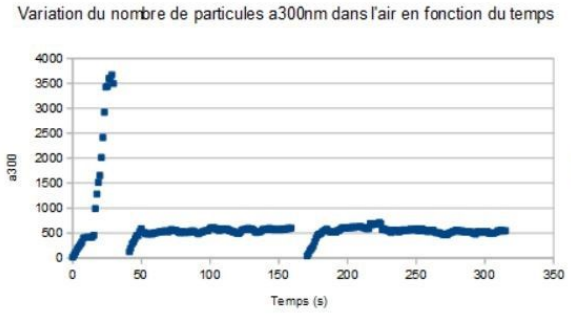
Comme évoqué précédemment, pour faciliter l'étude des données, nous allons prendre les valeurs stockées dans les différents fichiers textes présents sur la carte sd. Les données ont été enregistrées de façon à être facilement intégrées à une feuille de calcul (fichier excel). Ainsi il nous suffit de copier coller les données des fichiers textes sur la feuille de calcul et d'utiliser une option permettant de changer de colonne pour chaque espace.

On a donc réalisé plusieurs tests afin d'observer l'évolution de la quantité et de la concentration de particules au cours du temps. Pour les concentrations de particules, on est toujours livré au même problème (elles sont à la valeur max plutôt que d'être à zéro). On va donc concentrer notre étude sur les autres données. Pour les résultats observés en environnement non poussiéreux, ils sont très instables et on ne peut pas en tirer grand chose (voir annexe). Par contre quand on est en environnement poussiéreux les résultats sont plus intéressants. Pour obtenir un environnement poussiéreux, nous plaçons simplement de la poussière de craie à proximité du capteur. Pour la suite des analyses, on ne montrera que la variation du nombre de particules a300nm dans l'air en fonction du temps car les résultats obtenus sont bien plus exploitables que pour les autres particules (voir annexe).

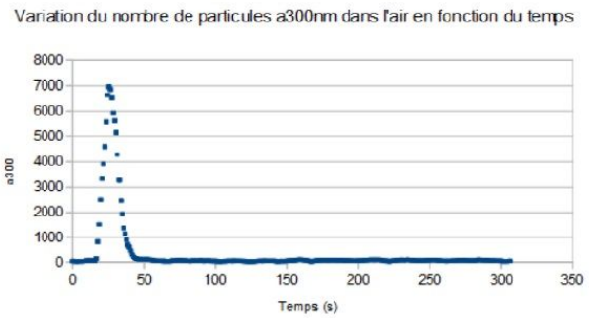


Comme expliqué précédemment, pour économiser de l'énergie entre différentes plages de mesure, on a le choix entre couper électriquement le capteur ou alors le mettre en mode veille. Pour l'étude énergétique il est important de connaître le temps de réponse (quand le capteur se remet en route) nécessaire pour obtenir une valeur cohérente. En effet ce temps de réponse risque de différer entre les deux mode ce qui influera sur l'énergie consommée "pour rien". est donc intéressant d'avoir un second capteur jouant le rôle de référence afin de déterminer plus précisément l'instant où les mesures re-deviennent correctes.

Dans un premier temps, nous avons eu le PMS7003. Le PMS7003 est également un capteur pm2.5 qui a exactement les même caractéristiques que le SEN0177. Cependant celui ci devait certainement être de moins bonne qualité. En effet, après plusieurs tests, le capteur n'affiche pas de valeurs cohérentes et cela ne correspond pas du tout avec les valeurs du SEN0177. Même avec un dépôt de poussière, les données transmises ne changeaient que très peu (voire pas du tout). Heureusement, un second capteur SEN0177 ayant été trouvé, nous avons pu faire des mesures avec un capteur identique à notre capteur initial. Les valeurs sont bien plus cohérentes même si on observe une différence d'échelle entre les deux.



capteur PM2.5 depuis notre carte (subit les tests)

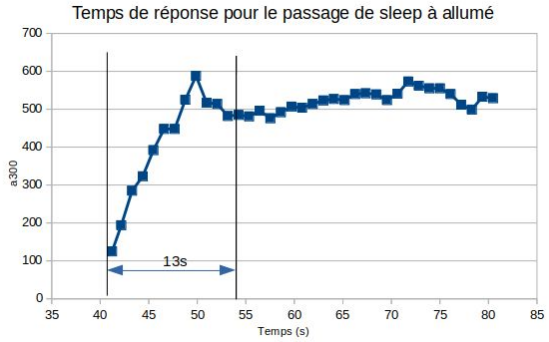


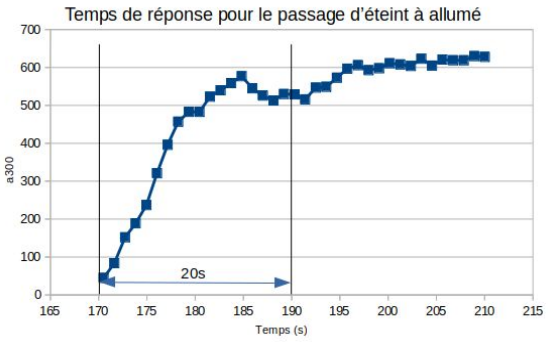
capteur PM2.5 depuis un arduino (référence)

On observe des coupures sur le graphe de gauche car on éteint électriquement le capteur et on le met en mode veille pour observer son temps de réponse.
En observant l'allure des deux courbes, nous avons d'abord pensé que cette référence allait permettre de réaliser les test espérés. La différence d'échelle entre les deux est d'environ 500 dans un environnement non poussiéreux. Quand on applique de la poussière (moment où on observe les pics sur les graphes), l'allure semble identique mais nous n'avons pas trouvé de lien convaincant entre les deux (l'écart n'est pas constant). La précision de ce type de capteur semble être limitée.

On s'intéresse maintenant aux temps de réponse pour les deux modes (sleep/allumé et éteint/allumé). La référence étant un échec, nous avons simplement estimé le moment où les valeurs se stabilisent. Avec des interprétations, on est tout de même capable de déterminer approximativement le temps de montée pour les deux modes (on récupère sur la courbe précédente les moments qui nous intéressent).

On observe ci dessous le temps de réponse pour les deux modes (on observe la réponse du capteur pendant 40 secondes).





On remarque que le temps de réponse du capteur est sensiblement plus rapide quand on passe du mode sleep au mode normal (7s plus rapide). Ainsi on privilégiera certainement cette méthode si on souhaite effectuer des relevés à une fréquence élevée. On notera tout de même qu'il est absurde de mettre en veille/d'éteindre le capteur si on effectue des relevés toutes les 13 secondes ou moins.

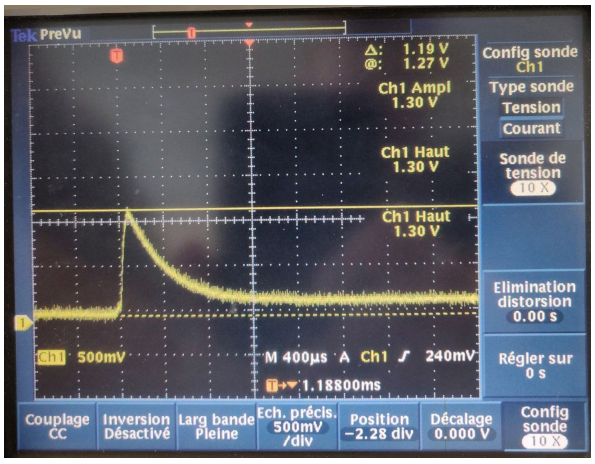
4.2. Mesures des puissances consommées

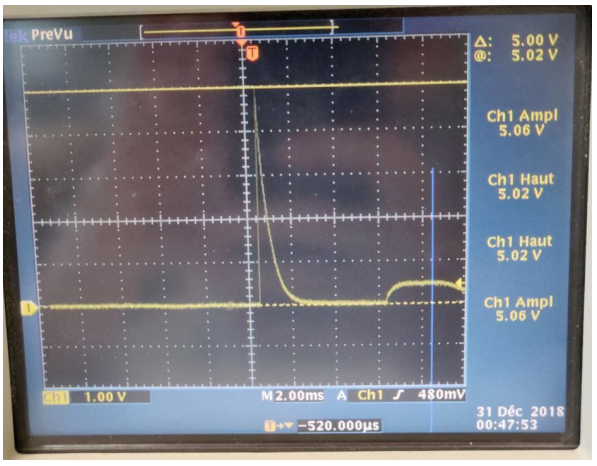
Nous avions initialement pour objectif d'utiliser le montage high-side que nous avons réalisé afin mesurer la puissance consommée par nos différents éléments. Cependant, nous n'avons pas réussi à le faire fonctionner. Nous avons donc utilisé un multimètre avec lequel on a mesuré la tension et le courant consommé pour chaque capteur. Nous avons simplement relié les entrées du multimètre aux différents header prévu pour effectuer les relevés. Pour le courant, on connecte simplement le multimètre en série. Pour la tension, on connecte le COM à la masse du circuit (header prévu à cet effet, évoqué précédemment), et le V au potentiel de l'élément en question. Il ne nous reste plus qu'à calculer la puissance avec P=U*I.

On obtient ainsi les résultats suivants:
- Pour le capteur DHT22 on a une consommation d'environ 1,2mA pour 5V soit P=6mW
- Pour le shield Bluetooth on a une consommation de 43/44mA (contre 50mA sur la datasheet) pour 5V soit P=215mW
- Pour le module de carte SD on a une consommation de 19mA pour 3,36V soit P=63mW
- Pour le capteur de particules, on différencie la consommation classique en marche qui vaut entre 30 et 50mA pour 5V soit P = 200mW et la consommation en mode veille qui vaut environ 9mA pour 5V soit P= 45mW. Pour le mode veille on observe un courant largement supérieur à celui indiqué par la datasheet (200uA). On pense que cela est dû à l'adaptateur du capteur PM2.5 qui fausse le résultat notamment à cause des led qu'il utilise. Pour avoir une consommation précise, il faudrait souder des fils à l'entrée de l'adaptateur pour mesurer la puissance réelle dissipée sans l'adaptateur. Nous effectuerons par la suite les calculs pour la valeur théorique mais aussi pour la valeur relevée.

Pour une étude énergétique plus poussée, nous avons observé les pics de courants du PM2.5 lorsqu'on l'allume électriquement (avec le MOSFET) et lorsque l'on sort du mode SLEEP. Etant impossible de mesurer ces pics avec un multimètre, nous avons donc opté pour l'oscilloscope. Nous n'avions cependant que des sondes de tension. Pour palier à ce problème, nous avons observé la chute de tension au borne d'une résistance de très petite valeur (10ohm) pour pouvoir ainsi en déduire le courant consommé. On règle ensuite l'oscilloscope de façon à observer le pic grâce au mode "single seq".

On a les relevés suivants, avec à gauche le pic de courant lors de la sortie du mode sleep vue à l'oscilloscope, et à droite celui après une mise en tension électrique (mosfet) :





Pour le mode sleep on a donc un fonctionnement en veille consommant 45mW, un pic de courant au redémarrage allant jusqu'à 130mA, contre 9mA en mode sleep continu et 40mA en mode relevé de valeurs. La courbe peut être approchée par une exponentielle puisqu'on peut retrouver des ressemblances avec les courbes de décharge de condensateurs. Cependant, nous avons fait le choix de l'approcher par une fonction affine afin d'en déduire la consommation énergétique.
Pour le capteur éteint par le mosfet on a 0W de consommés et un pic de courant au redémarrage allant jusqu'à 500mA. De même, nous avons fait le choix de l'approcher par une fonction affine.

4.3. Étude énergétique

Nous avons dans un premier temps calculé l'énergie consommée dans le capteur, dans les trois cas (normal, utilisation du mode sleep, utilisation du mosfet).

Nous avons considéré une période T entre deux relevés de valeurs. Ces valeurs sont considérées comme cohérentes et c'est pour cela que nous ré utilisons les données déduites précédemment : le temps entre la fin du mode sleep (respectivement le moment où l'on rallume électriquement) et le relevé doit correspondre à 13s (respectivement 20s).
C'est pour cela que notre calcul d'énergie se décompose en trois parties. Comme il faut laisser trs=13sec (trm=20sec) avant d'effectuer le relevé, on calcule l'énergie véhiculée pendant le mode sleep (le capteur éteint par le mosfet) entre 0s et T - trs (T - trm).

Puis pendant le pic de courant approximé par l'aire d'un triangle rectangle, avec pic_sleep le point de courant le plus haut, tpic_sleep le temps avant de retrouver un courant équivalent au courant normal (pic_mosfet et tpic_mosfet). Enfin, le reste de la période est un fonctionnement normal (jusqu'à temps que les valeurs soient cohérentes) c'est-à-dire la puissance normale pendant trs-tpic_sleep (trm-tpic_mosfet).

On obtient donc les calculs suivants :

E_{Normal} = tension * courant_normal * T

E_{Sleep} = tension * courant_sleep * (T-trs)
+ tension * pic_sleep * tpic_sleep * ½
+ tension * courant_normal * (trs - tpic_sleep)

E_{Mosfet} = tension * 0 * (T-trm)
+ tension * pic_mosfet * tpic_mosfet * ½
+ tension * courant_normal * (trs - tpic_mosfet)

On remarque donc que l'énergie consommée lors de l'utilisation du mosfet ne dépend pas de la période, tant que celle-ci est supérieure à trm=20s. Ici on a :

tension = 5V
courant_normal = 0,04A
trm = 20s ; pic_mosfet = 0,5A ; tpic_mosfet = 1,6 ms ; soit E_{Mosfet} = 4,00148 J
trs = 13s ; pic_sleep = 0,13A ; tpic_sleep = 0,8ms

On cherche alors à partir de quelle valeur de T l'utilisation du mode sleep est recommandée. On prend un premier cas avec courant_sleep = 0,009A et un autre avec courant_sleep = 0,0002A (datasheet).

- Cas 1 :
  $E_{s,e} = 5 * 0,009 * T - 0,585 + 2,6.10^{-4} + 2,59984 = 5 * 0,009 * T + 2,0151$
  $T < \frac{4,00148 - 2,0151}{5 * 0,009}$ soit T < 44,14s pour la valeur expérimentale de courant_sleep
- Cas 2 :
  $E_{s,t} = 5 * 0,0002 * T - 0,013 + 2,6.10^{-4} + 2,59984 = 5 * 0,0002 * T + 2,5871$
  $T < \frac{4,00148 - 2,5871}{5 * 0,0002}$ soit T < 1414,38s pour la valeur théorique de courant_sleep

Nous faisons ensuite de même en comparant l'utilisation du mode sleep et aucune mise en veille (mode normal). On a alors le mode sleep privilégié pour T > 13,0006s pour le courant_sleep expérimental et T > 13,0005s pour la valeur théorique.

La variable courant_sleep est donc peu importante pour déterminer quelle utilisation entre le mode sleep et le mode normal est à privilégier, dans tous les cas il faut considérer qu'en dessous de 13,1 secondes il est préférable de laisser le capteur toujours allumé. On peut noter que c'est complètement en adéquation (et sûrement lié) avec notre valeur de trs (13 secondes avant d'obtenir des relevés cohérents).
En revanche, lors de la comparaison entre l'utilisation du mode veille et le mosfet, la valeur courant_sleep a un impact très important : avec notre valeur expérimentale, le mosfet est largement préféré puisque le mode sleep ne peut être utilisé pour T entre 13s et 44s soit une très faible fenêtre de mesures. Avec la valeur théorique il faut une période d'une vingtaine de minutes avant de pouvoir utiliser judicieusement le mosfet.

Pour une période entre deux mesures données nous connaissons maintenant quelle est la meilleure configuration :

T <= 13	13 < T < 44	44 <= T
normal	utilisation du mode sleep	utilisation du mosfet

configuration selon T, valeur expérimentale de courant_sleep

T <= 13	13 < T < 1414	1414 <= T
normal	utilisation du mode sleep	utilisation du mosfet

configuration selon T, valeur théorique de courant_sleep

4.4. Estimation de l'autonomie d'une batterie

Nous avons alors essayé de trouver un modèle permettant de prédire l'autonomie d'une batterie selon le mode utilisé. Pour cela, les calculs sont légèrement différents. En effet, une batterie se caractérise par ses Ampères-heure et non ses Joules. Nous avons donc calculé la valeur moyenne du courant traversant le PM2.5 (calculs en annexes). Ces calculs ont des valeurs qui correspondent bien à notre précédent point (correspondance des courants moyens aux valeurs extrêmes de T). Une fois cette valeur trouvée, on l'ajoute au reste des valeurs de courant des autres composants. On divise alors la valeur en A-h de la batterie par cette valeur de courant moyen total, ce qui nous donne une valeur en heures de l'autonomie de la batterie.

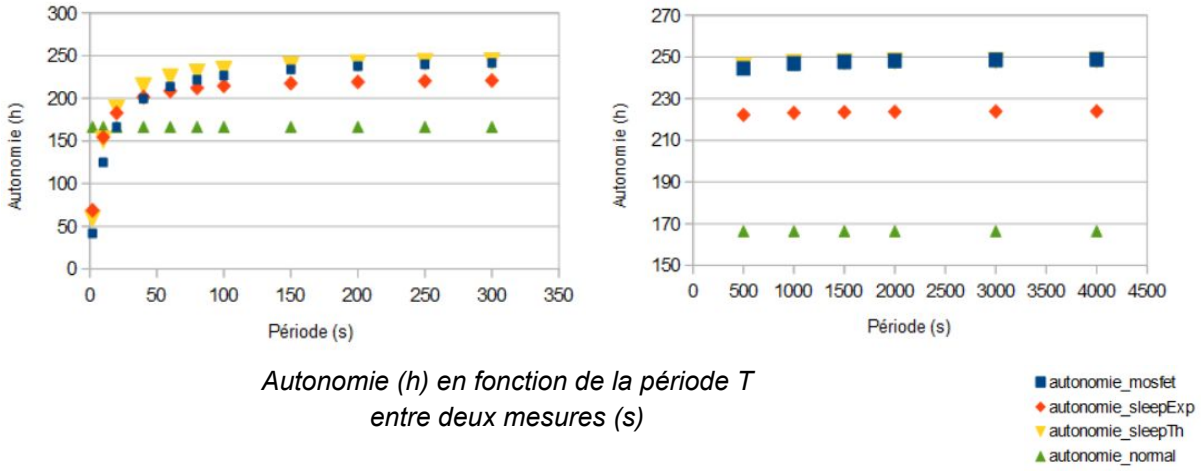
Voici ci-dessous les différentes valeurs moyennes du courant fonction de T :

ipm2_5	Fonctionnement normal	courant_sleep
	Utilisation du mode sleep	(1/T)*[courant_sleep *(T - trs) + courant_normal * (trs – tpic_sleep) + pic_sleep * tpic_sleep - (tpic_sleep²/2) * (tpic_sleep + 2 * (T – trs)]
	Utilisation du mosfet	(1/T)*[(courant_normal)*(trm – tpic_mosfet) + pic_mosfet * tpic_mosfet - (tpic_mosfet²/2) * (tpic_mosfet + 2* (T – trm)]
isd_card		0,019A
idht22		0,0012A
ibluetooth		0,0435A
iatmega		0,0165A

La somme de tous ces courants forme permet donc de diviser le nombre d'Ah donné par la batterie étudiée et ainsi obtenir le nombre d'heures d'autonomie.

Nous avons également mis tous ces calculs dans un programme (en annexe) afin de pouvoir tester différents types de batterie.

Pour une batterie de 20Ah facilement trouvable dans le commerce, nous obtenons les résultats suivants :



Autonomie (h) en fonction de la période T entre deux mesures (s)

On peut encore une fois voir que l'utilisation continue du capteur PM2.5 n'est conseillée que jusqu'à une dizaine de secondes. L'utilisation du mode sleep reste privilégiée jusqu'à une quarantaine de secondes en utilisant la valeur expérimentale de son courant en mode veille contre 1500 pour la valeur théorique où l'utilisation du mosfet est alors une meilleure solution. Cependant, on remarque que si l'on considère la valeur théorique de courant_sleep, l'autonomie entre cette configuration et l'arrêt électrique est très proche.

Au niveau de la durée de la batterie, on plafonne à 250 heures. En revanche, cela n'est possible que pour une fréquence de prise de mesures très faibles. Comme expliqué au tout début du rapport, si l'on veut que notre système soit transportable et utilisable pour cartographier une ville par exemple, alors il faut absolument relever des valeurs en continu et l'autonomie de la batterie se limite à environ 166 heures. Cela reste convaincant comparé au capteur Flow mentionné dans l'état d'art. Bien sûr, au vu du poids de ce dernier la batterie avait sûrement une capacité plus faible, mais une capacité de 20Ah correspond à des batteries classiques pour téléphone portable qui sont très légères et transportables très facilement. Seule la taille pourrait éventuellement plus poser problème.

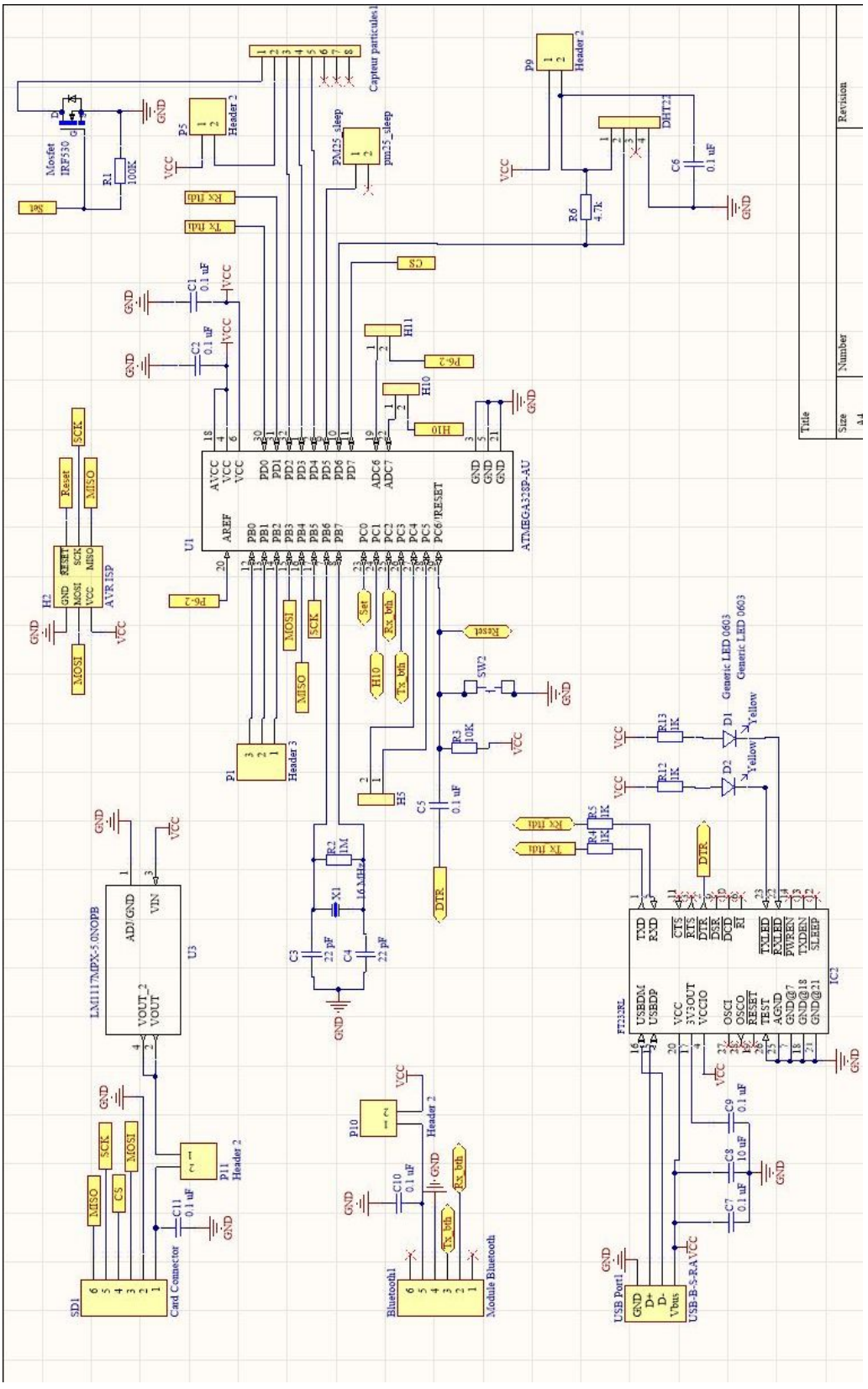
Si notre capteur est utilisé pour effectuer des relevés toutes les heures ou encore quotidiens, alors l'arrêt électrique du capteur de particules est à privilégier puisque jusqu'à 100h d'autonomie peuvent être gagnées sur une batterie de capacité 20Ah. Dans ce cas, le module de communication devrait être à revoir, puisqu'un module bluetooth y perdrait son sens. En enlevant ce module on peut gagner plus de 300 heures d'autonomie. Dans la même optique, le module SD pourrait être programmé différemment afin de ne pas perdre les fichiers quand il est éteint, et ainsi toute la carte pourrait être mise en veille avec l'Atmega.

Conclusion

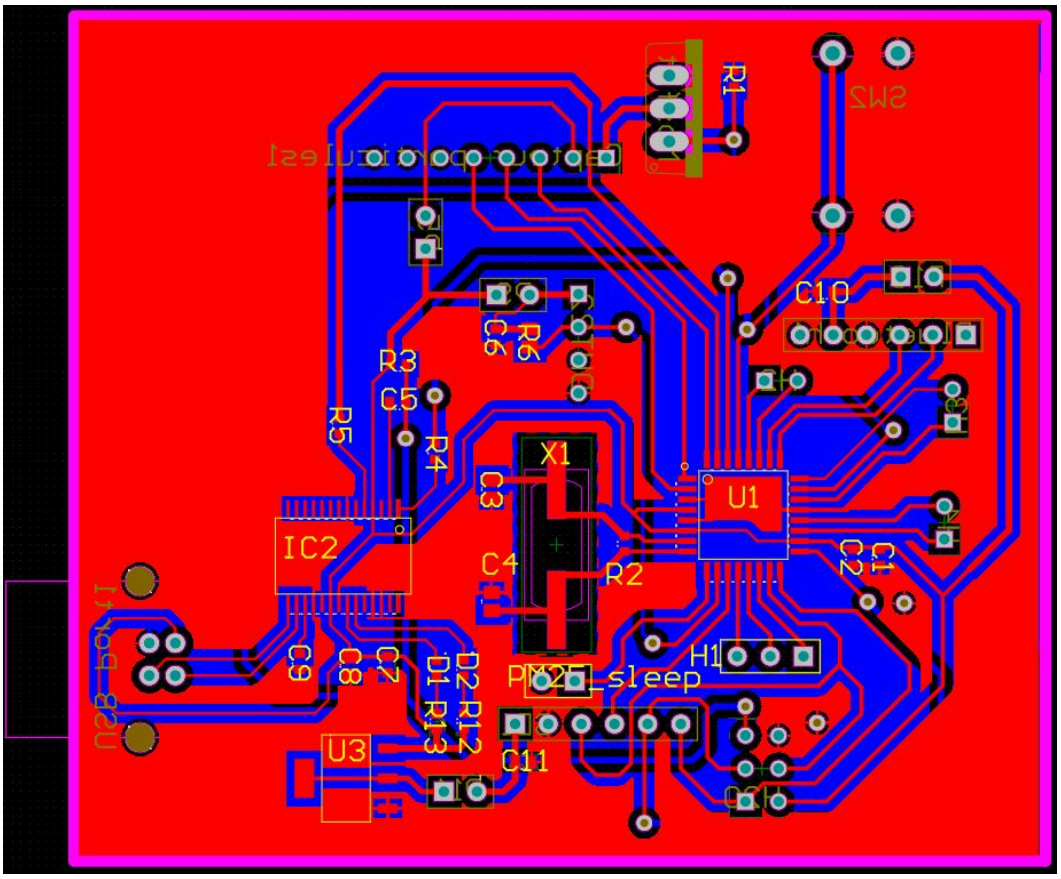
Lors de notre projet, nous avons mis en place un capteur de pollution simplifié. A l'aide de composants simples, nous avons d'abord formé un premier circuit avec une Arduino. Nous avons ensuite réalisé une carte simplifiée afin de ne pas avoir de pertes inutiles dans notre consommation d'énergie. Ces deux parties pouvant paraître assez futiles, nous y avons passé beaucoup de temps car ce sont des étapes fatidiques qui, si elles sont survolées, pouvaient avoir beaucoup de répercussion sur le reste du projet. Notre carte fonctionnant alors très bien, nous avons pu réaliser tous nos tests de consommation et en déduire un modèle pouvant estimer la durée de vie d'une batterie.

Nous avons remarqué que notre capteur est difficilement optimisable pour une fréquence de mesures élevée, correspondant à un piéton, un cycliste ou une voiture. En effet, pour de telles fréquences, les capteurs ne peuvent pas se permettre de se mettre en veille. Si nous pensons alors à une utilisation fixe (comme par exemple une borne de mesure dans une ville) la fréquence des relevés sera plus faible, on pourrait alors concevoir un système très économe en énergie où le microcontrôleur se mettrait en veille entre chaque mesure.

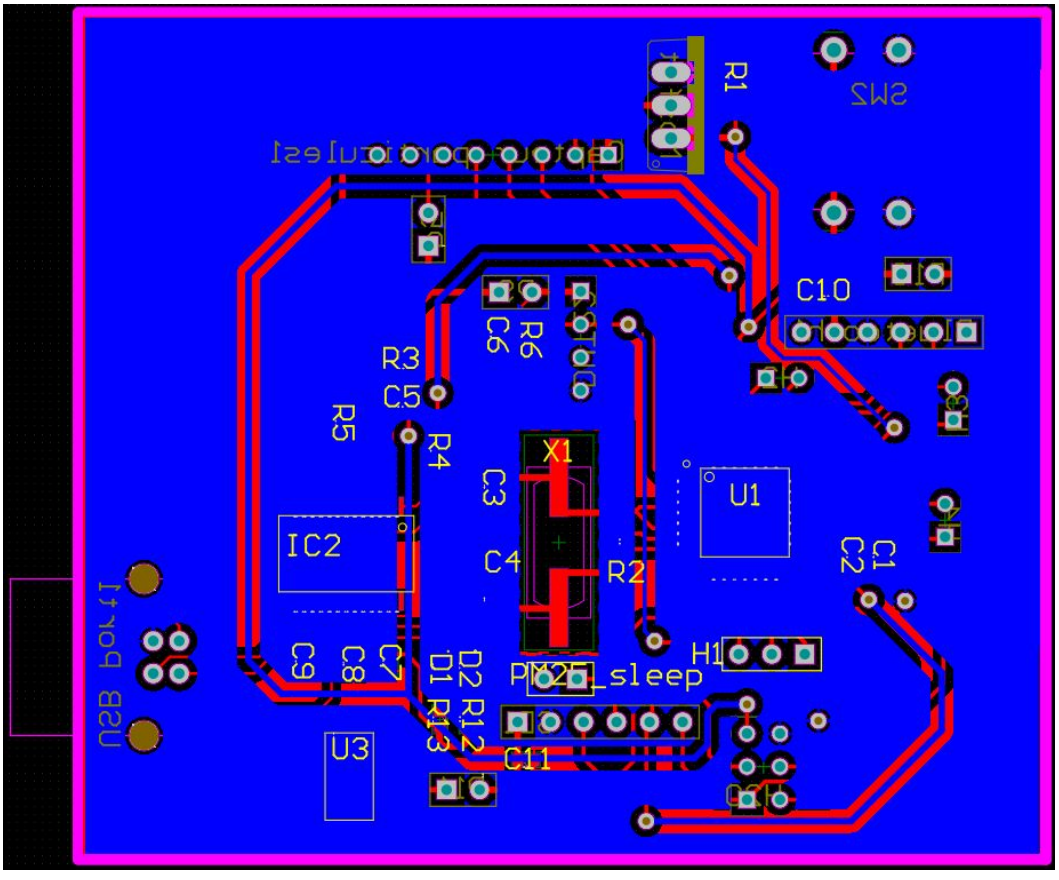
Annexes



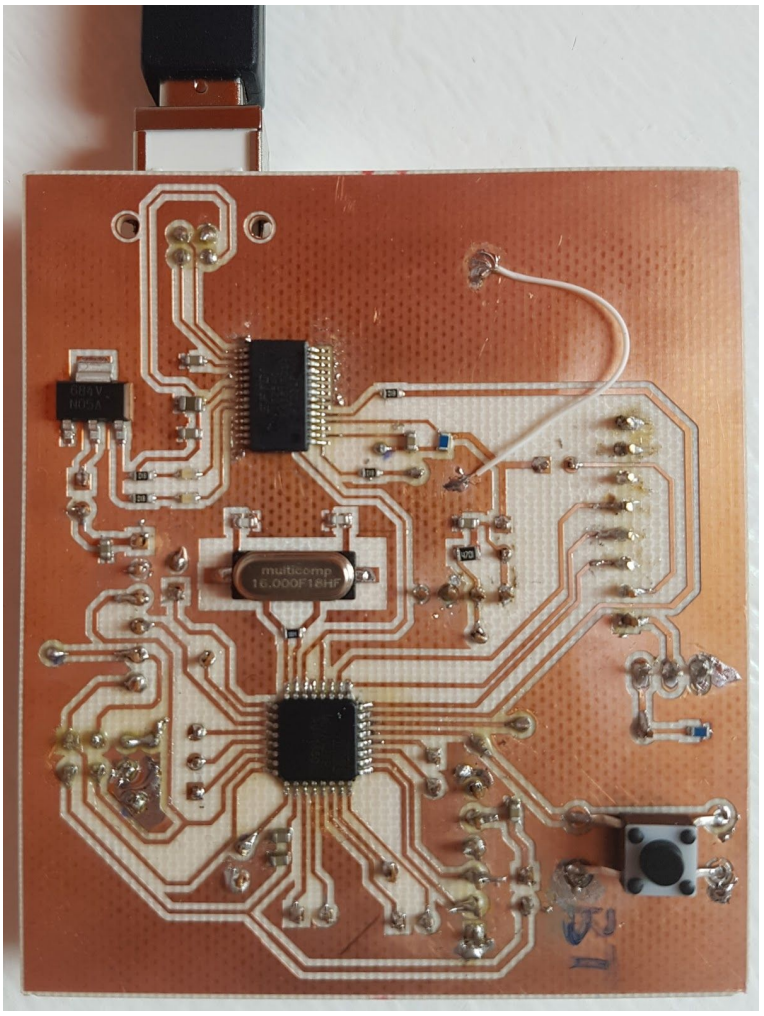
schématique du circuit principal



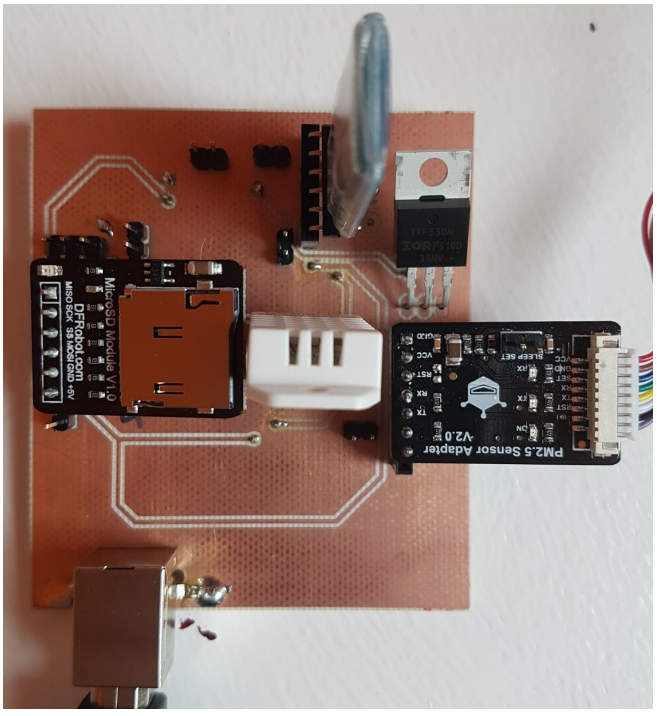
PCB du circuit principal - top



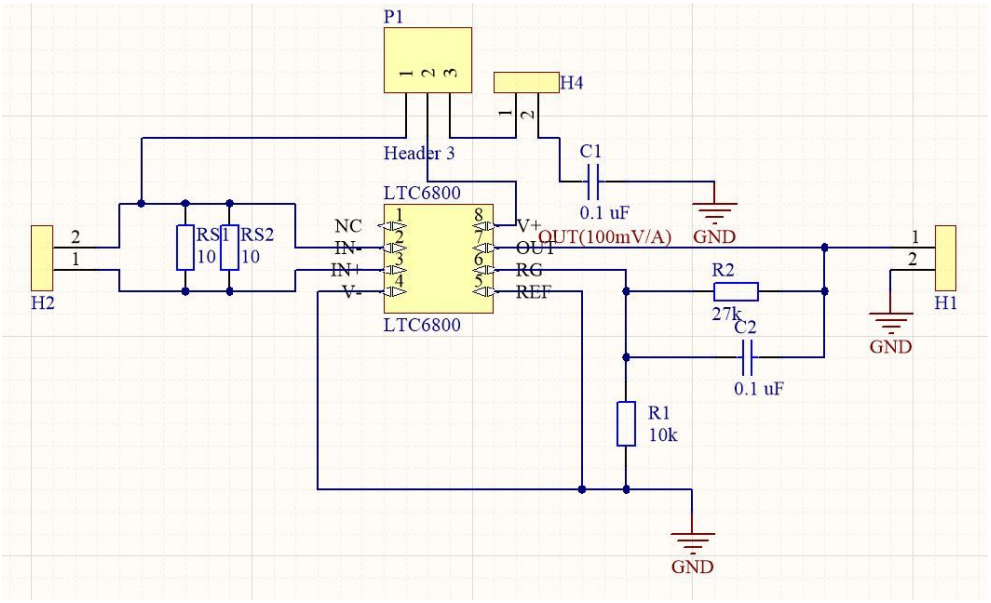
PCB du circuit principal - bottom



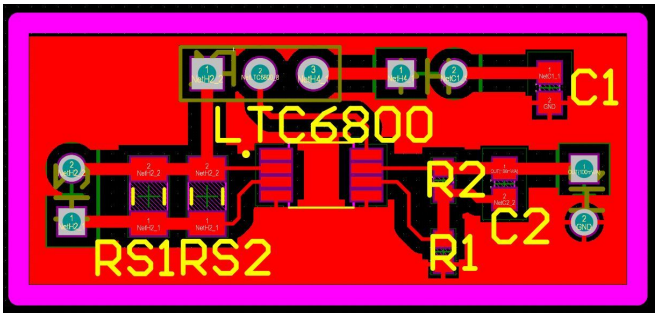
carte imprimée du circuit principal - top



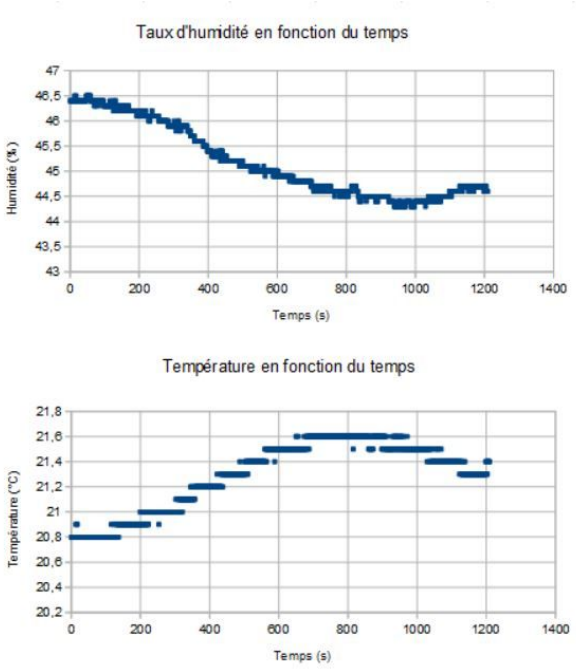
carte imprimée du circuit principal - bottom



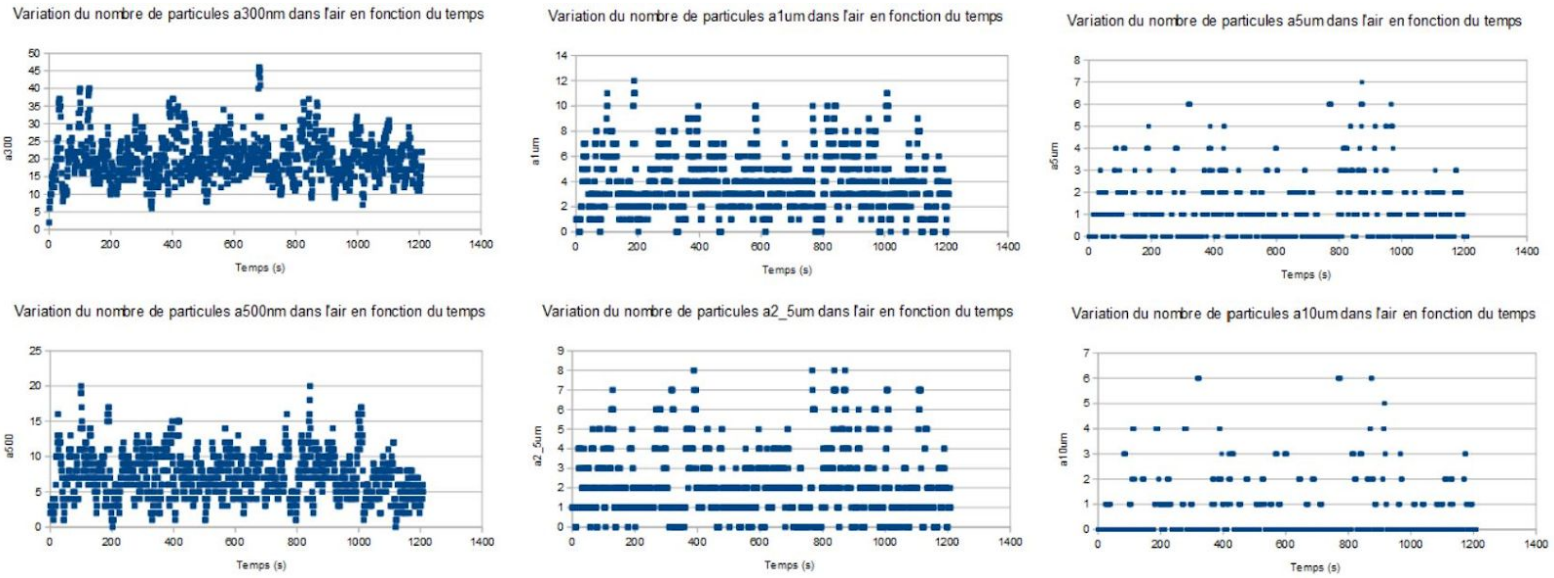
schématique du montage pour mesurer le courant



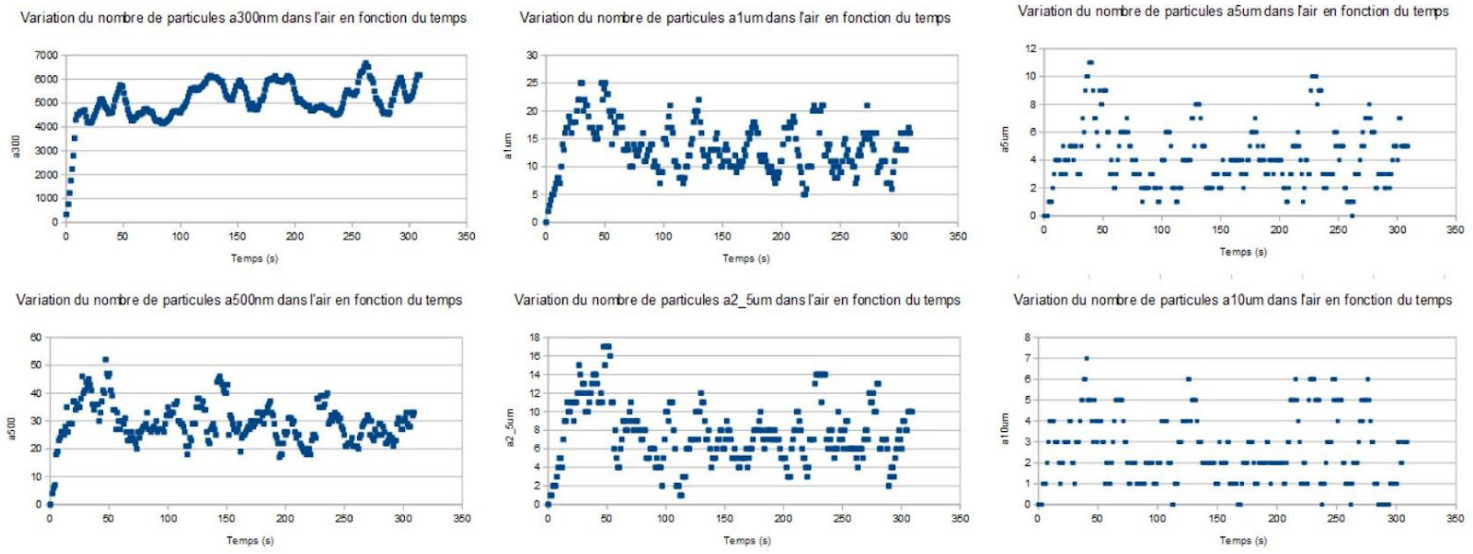
PCB du montage pour mesurer le courant



valeurs reçues sur le DHT22 en fonction du temps



Graphes des données reçues par le PM2.5 en environnement non poussiéreux



Graphes des données reçues par le PM2.5 en environnement poussiéreux

Code global du capteur

```
#include <Arduino.h>
#include <SoftwareSerial.h>
#include <SPI.h>
#include <SD.h>


#define LENG 31   //0x42 + 31 bytes equal to
32 bytes
SoftwareSerial PMSerial(4, 3);// RX, TX

const byte set = 5;//Broche pour le sleep du
capteur PM2.5
const byte mosfet = 14;//Sortie controllant le
MOSFET
const byte BROCHE_CAPTEUR = 6;//Broche
"DATA" du capteur DHT22

/*Différentes variables du PM2.5*/
unsigned char buf[LENG];
static char start_byte_1 = 0x42;
static char start_byte_2 = 0x4d;
unsigned int pm_01 = 0;
unsigned int pm_2_5 = 0;
unsigned int pm_10 = 0;
unsigned int a_300_nm = 0;
unsigned int a_500_nm = 0;
unsigned int a_1_um = 0;
unsigned int a_2_5_um = 0;
unsigned int a_5_um = 0;
unsigned int a_10_um = 0;

unsigned long currentTime;
int is_pause = 0;

/* Code d'erreur de la fonction readDHT22() */
const byte DHT_SUCCESS = 0;        // Pas
d'erreur
const byte DHT_TIMEOUT_ERROR = 1;  //
Temps d'attente dépassé
const byte DHT_CHECKSUM_ERROR = 2; //
Données reçues erronées
const int TMP_MESURE=500; // en ms. pour
DHT22 il ne faut pas plus de deux mesures par
seconde
float temperature, humidity;

File myFile;

void setup()
{
  Serial.begin(9600);
  init_sd();
  init_pm25();
  /* Place la broche du capteur DHT22 en
entrée avec pull-up */
  pinMode(BROCHE_CAPTEUR,
INPUT_PULLUP);



}

void loop() {

    currentTime = millis();
    /*
    if((currentTime > 20000 && is_pause==0) ||
(currentTime > 50000 && is_pause==1)) {
      digitalWrite(set,!digitalRead(set));
      is_pause++;
    }
    else if ((currentTime > 80000 &&
is_pause==2) || (currentTime > 110000 &&
is_pause==3)) {
      digitalWrite(mosfet,!digitalRead(mosfet));
      is_pause++;
    }*/
    donnees_pm25();
    donnees_dht22();
    pm_sd();
    dht_sd();
    //read_sd("pm25.txt");
    //read_sd("DHT22.txt");

    delay(TMP_MESURE);
}


void init_sd(){
  const byte CS_PIN = 7;
  pinMode (CS_PIN, OUTPUT);
  Serial.print("Initializing SD card...");
   while (!SD.begin(CS_PIN)) {
    Serial.println("initialization failed!");
    //while (1);
    delay(1000);
  }
  if(SD.exists("pm25.txt")){
    SD.remove("pm25.txt");
  }
  if(SD.exists("DHT22.txt")){
    SD.remove("DHT22.txt");
  }
  Serial.println("initialization done.");

}

void read_sd(String s){
  myFile = SD.open(s);
  if (myFile) {
    Serial.println("lecture du fichier");

    // read from the file until there's nothing else
in it:
    while (myFile.available()) {
      Serial.write(myFile.read());
    }
    // close the file:
    myFile.close();
  } else {
    // if the file didn't open, print an error:
    Serial.println("error opening pm25.txt");
  }
}

void pm_sd(){
  myFile = SD.open("pm25.txt", FILE_WRITE);
    if (myFile) {

myFile.print((currentTime/1000.0));myFile.print(
" ");
    myFile.print(pm_01);myFile.print(" ");
    myFile.print(pm_2_5);myFile.print(" ");
    myFile.print(pm_10);myFile.print(" ");
    myFile.print(a_300_nm);myFile.print(" ");
    myFile.print(a_500_nm);myFile.print(" ");
    myFile.print(a_1_um);myFile.print(" ");
    myFile.print(a_2_5_um);myFile.print(" ");
    myFile.print(a_5_um);myFile.print(" ");
    myFile.println(a_10_um);
    myFile.close();
    Serial.println("done pm2.5.");
    }
    else{
      Serial.println("error opening pm25.txt");
    }
}

void dht_sd(){
  myFile = SD.open("DHT22.txt",
FILE_WRITE);
  if (myFile) {

myFile.print((currentTime/1000.0));myFile.print(
" ");
    myFile.print(humidity);myFile.print(" ");
    myFile.println(temperature);
    myFile.close();
    Serial.println("done dht22.");  }
    else{
      Serial.println("error opening DHT22.txt");
    }
}
/*FONCTIONS DU CAPTEUR DE PARTICULE
PM2.5*/

//Fonction d'initialisation utilisée dans le setup
void init_pm25 () {
  PMSerial.begin(9600);
  PMSerial.setTimeout(1500);
//Serial.setTimeout() sets the maximum
milliseconds to wait
  pinMode(set, OUTPUT);
  pinMode(mosfet, OUTPUT);
  digitalWrite(mosfet, 1);
  digitalWrite(set, 1);
}

//affichage des données reçues
void donnees_pm25() {
  if (PMSerial.find(start_byte_1)) {
      //Serial.println("start character found");
      //Serial.print("trame de longueur : ");
      PMSerial.readBytes(buf, LENG);
      //Serial.print((PMSerial.readBytes(buf,
LENG) + 1), DEC);
      Serial.println();
      if (buf[0] == start_byte_2) {
        if (test_checksum()) {
          //debug();
          pm_01 = get_pm_01(buf);
          pm_2_5 = get_pm_2_5(buf);
          pm_10 = get_pm_10(buf);

          a_300_nm = above_300_nm(buf);
          a_500_nm = above_500_nm(buf);
          a_1_um = above_1_um(buf);
          a_2_5_um = above_2_5_um(buf);
          a_5_um = above_5_um(buf);
          a_10_um = above_10_um(buf);
        }     }   }
}

//fonction test du bit checksum
int test_checksum() {
  int check_sum = 0;
  for (int i = 0; i < (LENG - 2); i++) {
    check_sum += buf[i];
  }
  check_sum += 0x42;
  if (check_sum == ((buf[LENG - 2] << 8) +
buf[LENG - 1])) {
    return 1;
  }
  else return 0;
}

void debug() {
  for (int i = 0; i < LENG; i++) {
    Serial.print(buf[i], DEC);
    Serial.print(" ");
  }
  Serial.println();
}

//fonctions de traitement et récupération de
chaque donnée
unsigned int get_pm_01(unsigned char *buf)
{
  unsigned int pm_01_Val;
  pm_01_Val = ((buf[3] << 8) + buf[4]); //count
PM1.0 value of the air detector module
  return pm_01_Val;
}

unsigned int get_pm_2_5(unsigned char *buf)
{
  unsigned int pm_2_5_Val;
  pm_2_5_Val = ((buf[5] << 8) + buf[6]); //count
PM2.5 value of the air detector module
  return pm_2_5_Val;
}
unsigned int get_pm_10(unsigned char *buf)
```

```
{
  unsigned int pm_10_Val;
  pm_10_Val = ((buf[7] << 8) + buf[8]); //count
PM10 value of the air detector module
  return pm_10_Val;
}
unsigned int above_300_nm(unsigned char
*buf)
{
  unsigned int above_300_nm_val;
  above_300_nm_val = ((buf[15] << 8) +
buf[16]);
  return above_300_nm_val;
}
unsigned int above_500_nm(unsigned char
*buf)
{
  unsigned int above_500_nm_val;
  above_500_nm_val = ((buf[17] << 8) +
buf[18]);
  return above_500_nm_val;
}

unsigned int above_1_um(unsigned char *buf)
{
  unsigned int above_1_um_val;
  above_1_um_val = ((buf[19] << 8) + buf[20]);
  return above_1_um_val;
}

unsigned int above_2_5_um(unsigned char
*buf)
{
  unsigned int above_2_5_um_val;
  above_2_5_um_val = ((buf[21] << 8) +
buf[22]);
  return above_2_5_um_val;
}

unsigned int above_5_um(unsigned char *buf)
{
  unsigned int above_5_um_val;
  above_5_um_val = ((buf[23] << 8) + buf[24]);
  return above_5_um_val;
}

unsigned int above_10_um(unsigned char *buf)
{
  unsigned int above_10_um_val;
  above_10_um_val = ((buf[25] << 8) + buf[26]);
  return above_10_um_val;
}


/*FONCTION DU CAPTEUR DE
TEMPERATURE DHT22 */

//affichage des données reçues
void donnees_dht22(){
  switch (readDHT22(BROCHE_CAPTEUR,
&temperature, &humidity)) {
  case DHT_SUCCESS:
    //affichage des données
    break;
  case DHT_TIMEOUT_ERROR:
    Serial.println(F("Pas de reponse !"));
    break;
  case DHT_CHECKSUM_ERROR:
    Serial.println(F("Pb de communication !"));
    break;
  }
}

//récupération des données en valeurs lisibles
apres read_brut
byte readDHT22(byte pin, float* temperature,
float* humidity) {
  /* Lit le capteur */
  byte data[5];
  byte ret = read_brut(pin, data, 1, 1000);
  /* Détecte et retourne les erreurs de
communication */
  if (ret != DHT_SUCCESS)
    return ret;
  /* Calcul la vraie valeur de la température et
de l'humidité */
  float fh = data[0];
  fh *= 256;
  fh += data[1];
  fh *= 0.1;
  *humidity = fh;
  float ft = data[2] & 0x7f;
  ft *= 256;
  ft += data[3];
  ft *= 0.1;
  if (data[2] & 0x80) {
    ft *= -1;
  }
  *temperature = ft;
  /* Ok */
  return DHT_SUCCESS;
}

//Première récupération des données (en
valeurs brutes)
byte read_brut(byte pin, byte* data, unsigned
long start_time, unsigned long timeout) {
  data[0] = data[1] = data[2] = data[3] = data[4]
= 0;
  // start_time est en millisecondes
  // timeout est en microsecondes
  /* Conversion du numéro de broche Arduino
en ports / masque binaire "bas niveau" */
  uint8_t bit = digitalPinToBitMask(pin);
  uint8_t port = digitalPinToPort(pin);
  volatile uint8_t *ddr = portModeRegister(port);
// Registre MODE (INPUT / OUTPUT)
  volatile uint8_t *out =
portOutputRegister(port); // Registre OUT
(écriture)
  volatile uint8_t *in = portInputRegister(port);
// Registre IN (lecture)

  /* Conversion du temps de timeout en nombre
de cycles processeur */
  unsigned long max_cycles =
microsecondsToClockCycles(timeout);

  /* Evite les problèmes de pull-up */
  *out |= bit;  // PULLUP
  *ddr &= ~bit; // INPUT
  delay(100);   // Laisse le temps à la résistance
de pullup de mettre la ligne de données à HIGH

  /* Réveil du capteur */
  *ddr |= bit;  // OUTPUT
  *out &= ~bit; // LOW
  delay(start_time); // Temps d'attente à LOW
causant le réveil du capteur

  /* Portion de code critique - pas d'interruptions
possibles */
  noInterrupts();

  /* Passage en écoute */
  *out |= bit;  // PULLUP
  delayMicroseconds(40);
  *ddr &= ~bit; // INPUT

  /* Attente de la réponse du capteur */
  timeout = 0;
  while(!(*in & bit)) { /* Attente d'un état LOW */
    if (++timeout == max_cycles) {
      interrupts();
      return DHT_TIMEOUT_ERROR;
    }
  }

  timeout = 0;
  while(*in & bit) { /* Attente d'un état HIGH */
    if (++timeout == max_cycles) {
      interrupts();
      return DHT_TIMEOUT_ERROR;
    }
  }
  /* Lecture des données du capteur (40 bits) */
  for (byte i = 0; i < 40; ++i) {
    /* Attente d'un état LOW */
    unsigned long cycles_low = 0;
    while(!(*in & bit)) {
      if (++cycles_low == max_cycles) {
        interrupts();
        return DHT_TIMEOUT_ERROR;
      }
    }
    /* Attente d'un état HIGH */
    unsigned long cycles_high = 0;
    while(*in & bit) {
      if (++cycles_high == max_cycles) {
        interrupts();
        return DHT_TIMEOUT_ERROR;
      }
    }
    /* Si le temps haut est supérieur au temps
bas c'est un "1", sinon c'est un "0" */
    data[i / 8] <<= 1;
    if (cycles_high > cycles_low) {
      data[i / 8] |= 1;
    }
  }

  /* Fin de la portion de code critique */
  interrupts();
  /* Vérifie la checksum */
  byte checksum = (data[0] + data[1] + data[2] +
data[3]) & 0xff;
  if (data[4] != checksum)
    return DHT_CHECKSUM_ERROR; /* Erreur
de checksum */
  else
    return DHT_SUCCESS; /* Pas d'erreur */
```

On utilise $\langle i \rangle = \frac{1}{T}\int_T i(t)\,dt$

- fonctionnement normal

$$\langle i_N \rangle = \frac{1}{T}\int_0^T \text{courant\_normal} \cdot dt = \text{courant normal}$$

- utilisation du mode sleep

$$\langle i_S \rangle = \frac{1}{T}\left(\int_0^{T-trs} \text{courant\_sleep}.\,dt + \int_{T-trs}^{T-trs+tpic\_sleep} (-tpic\_sleep.t + pic\_sleep)dt + \int_{T-trs+tpic\_sleep}^{T} \text{courant\_normal} \cdot dt \right)$$

$$= \frac{1}{T}\left(\begin{array}{l} \text{courant\_sleep}(T-trs) + \text{courant\_normal}\,(T-T+trs-tpic\_sleep) \\ + pic\_sleep(T-trs+tpic\_sleep-T+trs) - \left[tpic\_sleep.\frac{t^2}{2} \right]_{T-trs}^{T-trs+tpicsleep} \end{array} \right)$$

$$\left[t^2\right]_{T-trs}^{T-trs+tpicsleep} = T^2 + trs^2 + tpic\_sleep^2 + 2\times(tpic\_sleep*T - trs*T + tpic\_sleep*trs) - T^2 - 2*T*trs + trs^2$$

$$= tpic\_sleep^2 + 2*tpic\_sleep\,(T+trs)$$

$$\Rightarrow \langle i_S \rangle = \frac{1}{T}\left(\begin{array}{l} \text{courant\_sleep}(T-trs) + \text{courant\_normal}\,(trs - tpic\_sleep) \\ + pic\_sleep.\,tpic\_sleep - \frac{tpic\_sleep^2}{2}\left(tpic\_sleep + 2(T+trs)\right) \end{array} \right)$$

- utilisation du mosfet

$$\langle i_M \rangle = \frac{1}{T}\left(\int_0^{T-trm} 0\,dt + \int_{T-trm}^{T-trm+tpic\_mosfet} (-tpic\_mosfet.t + pic\_mosfet)dt + \int_{T-trm+tpic\_mosfet}^{T} \text{courant\_normal}.\,dt \right)$$

$$= \frac{1}{T}\left(\begin{array}{l} 0 + \text{courant\_normal}\,(trm - tpic\_mosfet) + pic\_mosfet.\,tpic\_mosfet \\ - \frac{tpic\_mosfet^2}{2}\left(tpic\_mosfet + 2\times(T+trm)\right) \end{array} \right)$$

calcul du courant moyen traversant le PM2.5 selon les configurations

Code pour le calcul de l'autonomie d'une batterie

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

#define B 1 // quand b vaut 1 on est en expÃ©rimental sinon on est thÃ©orique
#define courant_sleep_exp (float)0.009
#define courant_sleep_th (float)0.0002
#define courant_normal (float)0.04
#define trs (float)13
#define trm (float)20
#define tpic_sleep (float) 0.0008
#define tpic_mosfet (float)0.0016
#define pic_sleep (float) 0.13
#define pic_mosfet (float) 0.5
#define isd_card (float) 0.019
#define idht22 (float) 0.0012
#define ibluetooth (float) 0.0435
#define i328p (float) 0.0165


int main(){
   int t,b;
   float autonomie,ipm_2_5;
   float heures,minutes,secondes;
   printf("Bonjour !\nentrez le temps entre chaque mesure (en secondes) ");
   scanf("%d",&t);
   printf("Entrez la capacitÃ© de la batterie en Ah ");
   scanf("%d",&b);
   if(B==1){
      if(t<=13){
         ipm_2_5 = courant_normal;
      }
      else if(t<=41 && t>13){
         ipm_2_5 = (1.0/t)*(courant_sleep_exp*(t-trs)+courant_normal*(trs-tpic_sleep)+pic_sleep*tpic_sleep-((tpic_sleep*tpic_sleep)/2)*(tpic_sleep+2*(t-trs))); // moyenne du courant       }
      else{
         ipm_2_5 = (1.0/t)*((courant_normal)*(trm-tpic_mosfet)+pic_mosfet*tpic_mosfet-((tpic_mosfet*tpic_mosfet)/2)*(tpic_mosfet+2*(t-trm))) ;
      }
   }
   else{
      if(t<=13){
         ipm_2_5 = courant_normal;
      }
      else if(t<=1414 && t>13){
         ipm_2_5 = (1.0/t)*(courant_sleep_th*(t-trs)+courant_normal*(trs-tpic_sleep)+pic_sleep*tpic_sleep-((tpic_sleep*tpic_sleep)/2)*(tpic_sleep+2*(t-trs))); // moyenne du courant
      }
      else{
         ipm_2_5 = (1.0/t)*((courant_normal)*(trm-tpic_mosfet)+pic_mosfet*tpic_mosfet-((tpic_mosfet*tpic_mosfet)/2)*(tpic_mosfet+2*(t-trm))) ;
      }
   }


autonomie=b/(ipm_2_5+isd_card+idht22+ibluetooth+i328p);
   heures = autonomie;
   minutes = (autonomie-(int)autonomie)*60;
   secondes = (minutes-(int)minutes)*60;
   printf("ipm_2_5 = %f l'autonomie est de %d heures, %d minutes, et %d secondes\n",ipm_2_5,(int)heures,(int)minutes,(int)secondes);
   return 0;
}
```