MICRO-CONTROLERS AND OPEN-SOURCE HARDWARE

Conception d'un capteur de gaz connecté

Auteurs : Julien CHOUVET et Jean Baptiste LAFFOSSE

28 NOVEMBRE 2018

INSA DE TOULOUSE
5ème année – PTP Innovative Smart System

Sommaire

Table des matières

I - Introduction

Dans le cadre du cours s'intitulant « microcontrôleurs et open-source hardware », nous avons conçu un appareil détectant la présence anormale de gaz rejeté lors de combustion.

Nous avons appréhendé la conception de notre capteur de gaz intelligent et connecté en respectant les concepts relatifs au cours et ceux de l'Open Hardware. En effet notre projet comprend l'utilisation d'un microcontrôleur, l'Arduino UNO, de KiCAD pour le développement d'un shield, et de The Thing Networks (TTN) pour envoyer les données du capteur à travers le réseau LoRa et représenter les valeurs sur un dashboard.

Nous allons décrire dans ce rapport toutes les étapes qui nous ont amenées à terminer le capteur connecté. Dans un premier temps, nous allons décrire le cahier des charges que nous avons suivi. Ensuite, nous allons aborder la conception du circuit en précisant le dimensionnement du détecteur de dépassement et en testant le circuit sur LTSpice. Nous avons aussi ajouté un buzzer afin de signifier la présence anormale de gaz de combustion puis aussi un bouton RESET sur le shield. Puis, nous allons expliquer comment les données sont acheminés de l'Arduino vers le dashboard, c'est-à-dire de l'envoie des données puis de la réception. Nous avons ajouté une fonctionnalité de calibration à distance du capteur. Finalement, nous avons développé un shield que nous voulons plugger sur l'Arduino Uno à l'aide de KiCAD.

Ce rapport est à la fois une documentation de notre capteur de gaz connecté, mais il a aussi pour but d'expliquer comment réaliser par soi-même le capteur, comme la philosophie de l'Open source le veut. Nous avons ainsi mis à disposition tous nos fichiers sur Github :

https://github.com/JulienChvt/Microcontroller_Open_Source_Hardware

II - Cahier des charges

Afin de concevoir un capteur intelligent, ou « smart device », le capteur connecté doit respecter les exigences suivantes :

- Réaliser la mesure de la concentration des gaz de combustion, et calibrer.
- Utiliser le réseau LoRa et récupérer les valeurs du capteur sur TTN.
- Détecter la présence anormale de gaz.
- Prévenir l'utilisateur d'un danger sur TTN et par l'intermédiaire d'un buzzer.
- Calibrer le capteur à la demande de l'utilisateur.
- Installer un bouton RESET sur le shield.
- Utiliser du matériel et logiciel open-source.

III - Hardware

Cette première partie aborde le développement du Hardware utilisé pour le capteur connecté afin de respecter le cahier des charges. Nous allons donc expliquer le choix de notre montage afin de détecter la présence d'un danger. Puis nous allons parler de la mise en place du buzzer et du bouton RESET.

III - A. Conception du Trigger de Schimtt

La meilleure solution pour détecter un dépassement est l'utilisation des interruptions. Elles permettent en effet de ne pas altérer le déroulement normal du programme et de gérer des cas exceptionnels.

La valeur du capteur est mesurée par un convertisseur analogique numérique (CAN). Or sur l'Arduino UNO, il n'est pas possible d'attacher une interruption sur une entrée analogique, c'est-à-dire nous ne pouvons pas détecter quand la concentration de gaz de combustion dépasse la valeur d'un seuil. Il faut alors passer par les entrées digitales.

Nous devons donc pour cela utiliser un montage qui va nous indiquer quand la concentration est au-dessus ou pas du seuil. En cas de danger, le montage mettra à l'état haut la pin #3 sur laquelle nous pouvons attacher une interruption et traiter par la suite l'avertissement de l'utilisateur (partie software).

III - A.1. Choix du circuit

Pour détecter le dépassement du seuil, plusieurs choix de montage s'offrent à nous. Nous décidons pour cela d'utiliser un Trigger de Schmitt. En effet, nous pouvons facilement choisir la tension de sortie de notre trigger et ainsi permettre de détecter les changements de niveaux logiques par l'Arduino grâce à l'entrée digitale. De plus, à partir d'un simple potentiomètre, nous pouvons régler la tension de seuil à laquelle nous voulons déclencher notre alerte.

Finalement, l'avantage principal du Trigger de Schmitt est qu'il permet, de par son hystérésis, d'éviter les rebonds potentiels causés par une source bruitée. Or, c'est le cas de notre capteur de gaz qui nous donne une valeur approximative et changeante en fonction du temps. Ainsi, avec notre Trigger de Schmitt, nous pouvons fixer une valeur seuil, et le Trigger va changer de valeur qu'une seule fois.

Voici le schéma de notre montage :

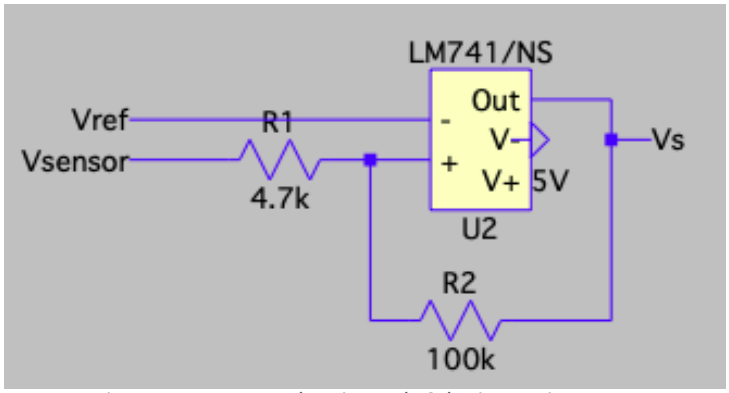


Figure 1 – Montage du Trigger de Schmitt non inverseur

III - A.2. Calcul des résistances

a) Les tensions de seuil

Nous pouvons maintenant choisir les valeurs de nos résistances afin de déclencher l'alerte à un seuil donné. Voici les tensions de seuils du Trigger :

$$V_h = V_e\left(1 + \frac{R_1}{R_2}\right) + V_{sat}(\frac{R_1}{R_2})$$
$$V_b = V_e\left(1 + \frac{R_1}{R_2}\right) - V_{sat}(\frac{R_1}{R_2})$$

Avec V_{sat} la tension d'alimentation de l'amplificateur opérationnel (AOP), puis V_h et V_b respectivement les tensions pour passer de l'état bas à l'état haut et inversement. Nous voulons pouvoir détecter des valeurs proches de $1.3\ V$, ce qui correspond à une concentration suspecte en gaz de combustion.

Nous pouvons donc prendre comme tension de comparaison $V_e = 1V$ (que nous pouvons ajuster avec un potentiomètre), $R_1 = 4.7\ k\Omega$ et $R_2 = 100\ k\Omega$. Nous obtenons donc les valeurs suivantes pour les tensions de seuil V_h et V_b :

$$V_h = 1.282\ V \text{ et } V_b = 0.812\ V$$

Ce qui nous laisse une marge de bruit de $V_h - V_b = 0.47\ V$, ce qui est largement suffisant pour prendre en compte le bruit et laisser l'alerte allumer pendant un certain moment le temps que la valeur du capteur redescende.

b) Niveaux logiques

Nous devons prendre en compte les niveaux logiques de l'entrée digitale de l'Arduino Uno. Nous savons que l'Arduino peut détecter des niveaux logiques bas en dessous de $0.3 * V_{cc} = 1.5\ V$, et pour les niveaux logiques hauts, il faut que la tension soit supérieure à $0.6 * V_{cc} = 3.0\ V$.

Or les AOP disponibles en salle de TP ne sont pas rail-to-rail, c'est-à-dire que la tension de sortie ne pourra jamais atteindre les tensions d'alimentation $V+$ et $V-$. Nous devons donc s'assurer pour l'AOP choisit que la tension de sortie sera soit $1\ V$ soit $4V$.

Nous pouvons donc dire que le Trigger de Schmitt non-inverseur nous permettra de déclencher l'interruption de l'Arduino Uno quand la concentration de gaz combustible devient dangereuse. C'est ainsi que nous avons choisi l'AOP LM741.

III - A.3. Simulation sur LTSpice

Nous pouvons maintenant simuler notre montage sur LTSpice afin de nous assurer du bon fonctionnement du Trigger de Schmitt.

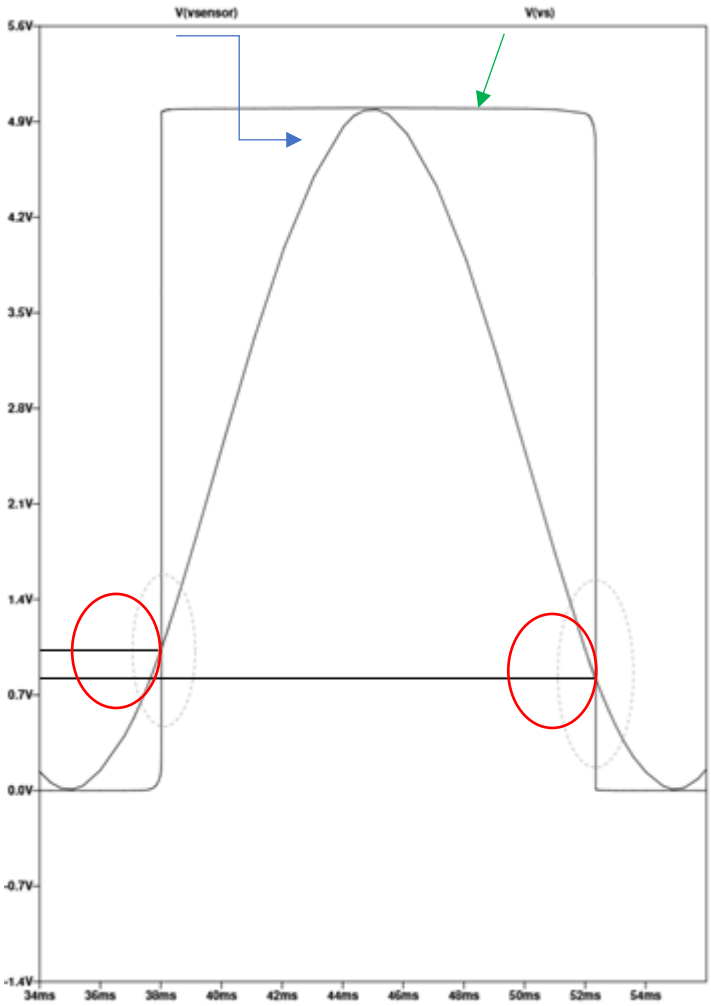


Figure 2 – Simulation LTSpic - Représentation de la tension de sortie (signal carré) en fonction de la valeur donnée par le capteur de gaz (sinus).

Nous pouvons relever les deux valeurs de tensions de seuil auxquelles le Trigger de Schmitt se déclenchent.

$$V_h = 1.1\ V \text{ et } V_b = 0.8\ V$$

Ensuite les deux états bas et hauts en sortie du Trigger de Schmitt ne sont pas représentatifs de la réalité. Nous avons testé notre montage sur breadboard, et nous obtenons bien des valeurs compatibles pour l'Arduino.

III - B. Mise en place du buzzer et du bouton RESET

Afin de respecter le cahier des charges, nous devons implémenter un bouton RESET sur le shield pour garantir une réinitialisation à la demande de l'utilisateur, ainsi qu'un buzzer pour indiquer un dépassement anormal de la concentration de gaz de combustion.

Dans le circuit d'un montage et dans le programme, nous devons prendre en compte ces deux éléments en fonction du matériel disponible en salle de TP.

III - B.1. Le buzzer et l’indicateur lumineux

Le buzzer est constitué d’un piézoélectrique relié à une membrane qui une fois excitée avec une certaine fréquence peut faire du bruit. Nous devons donc connecter notre buzzer à une PWM afin de pouvoir faire varier la fréquence, et donc le son du buzzer. De plus, nous pouvons ajouter une LED pour ajouter un indicateur lumineux. Voici le schéma de notre circuit :

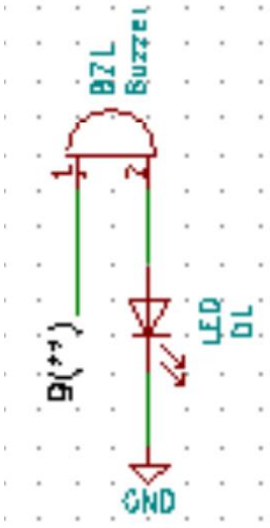


Figure 3 – Schéma BUZZER et LED

Nous avons relié le buzzer à la pin #9 où nous pouvons utiliser une PWM, et la LED au GND afin qu’elle s’allume quand la PWM est à l’état haut.

III - B.2. Le bouton RESET

L'Arduino peut être réinitialisé quand le pin RESET est mise à l'état bas. Nous devons connecter par l'intermédiaire d'un bouton la pin RESET au GND. Voici le schéma de notre circuit :

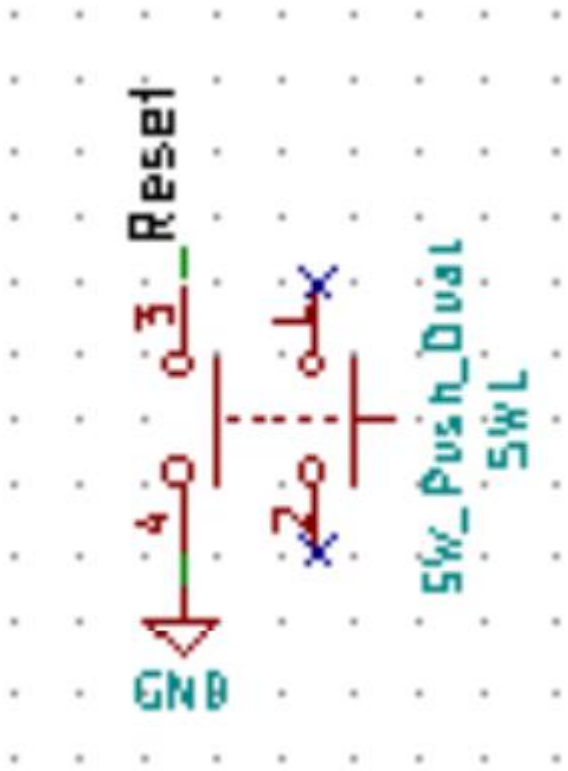


Figure 4 – Schéma bouton RESET

Nous avons utilisé ce bouton car disponible en salle de TP, et peu encombrant sur le shield de l'Arduino Uno.

Maintenant que l'ensemble de notre circuit est dimensionné et testé, nous pouvons détailler la partie software qui permet de gérer l'envoie des données, la calibration du capteur de gaz et l'avertissement d'un dépassement.

IV - Software

Cette partie explique le software que nous avons développé pour acquérir et traiter les données du capteur de gaz, ainsi que la mise en place des alertes pour prévenir d'un danger.

Nous développerons ensuite l'envoie de message sur le réseau LoRa et leur traitement sur la plateforme The Things Network.

Enfin, nous verrons comment nous avons pu créer une interface utilisateur avec NodeRED.

IV - A. Mesure et traitement des données du capteur de gaz

Le capteur de gaz utilisé ici est le *Grove MQ-9*. La connexion à la carte Arduino se fait de la manière suivante :

Arduino	Gas Sensor
5V	VCC
GND	GND
NC	NC
Analog A0	SIG

Figure 5 - Connection Arduino - MQ9

Nous avons ensuite créé une fonction *GasSensorAverageVolt* qui renvoie une valeur moyenne sur 100 mesures faites par le capteur de gaz. Ce choix d'un échantillon de 100 mesures permet d'avoir une mesure plus précise. De plus, comme il ne faut que 100 µs pour lire une valeur sur un pin analogique de l'Arduino Uno, cette valeur d'échantillonnage n'introduit pas de latence significative dans l'exécution du programme. Enfin, cette valeur moyenne est convertie en volt puis retournée sous forme d'un *float*.

```
//This function calculates an overage over 100 values sent by the gaz sensor,
//converts it in voltage and return this value (float)
float GasSensorAverageVolt(){
  float tmp_sensor_value = 0, sensor_volt, sensor_value;

  for (int i=0 ; i<100 ; i++){
    tmp_sensor_value += analogRead(A0);
  }

  sensor_value = tmp_sensor_value/100.0;
  sensor_volt = sensor_value/1024*5.0;

  return sensor_volt;
}
```

Figure 6 – Fonction GasSensorAverageVolt

IV - B. Gestion des interruptions – Création des alertes

Le but ici est de mettre en place une alerte lorsque la valeur renvoyée par le capteur de gaz dépasse une certaine valeur seuil signifiant un danger.

Comme nous l'avons vu dans la première partie, le trigger de Schmitt mis en place nous permet d'avoir une mise à l'état haut du pin numérique numéro 3 de l'Arduino, lorsque la valeur seuil est dépassée.

Ainsi, nous avons le fonctionnement suivant :

- Passage à l'état haut du pin #3 si $V_{Gas\text{-}Sensor} > V_{Seuil}$
- Passage à l'état bas du pin #3 lorsqu'on revient à $V_{Gas\text{-}Sensor} < V_{Seuil}$

Nous avons donc choisi de créer une interruption se déclenchant lorsqu'on a un changement (CHANGE) d'état de la pin #3.

```
#define interruptPin 3

//Monitor interruptPin and execute GasAlert function when the interruptPin goes from low to high or from high to low
attachInterrupt(digitalPinToInterrupt(interruptPin), GasAlert, CHANGE);
```

Figure 7 - Définition de l'interruption sur le pin #3

Lorsque cette interruption est levée, la fonction *GasAlert* est déclenchée.

```
//Function call by attachInterrupt to set the global var alert to true if interruptPin is set to HIGH
//or set alert to false and stop the alarm if interruptPin is set to LOW
//WARNING : This function must have no parameter and return nothing
//          delay() doesn't work on interrup function
void GasAlert(){
  if(digitalRead(interruptPin)){
    debugSerial.println("----- GAS ALERT -----");
    alert = true;
  }
  else{
    alert = false;
    GasAlertFinish();
  }
}
```

Figure 8 - Fonction GasAlert

Cette dernière vérifie alors dans quel état se trouve la pin #3 :

- Si *interruptPin* est à l'état haut alors on met la variable boolean *alert* a *true*
- Sinon (interruptPin à l'état bas), on met la variable boolean *alert* a *false* et on appelle la fonction *GasAlertFinish* qui arrête le buzzer.

La variable *alert* est une variable globale. Dans la fonction principale *loop* du code Arduino, on vérifie l'état de cette variable. Si cette dernière est égale à *true* c'est donc qu'une alerte est en cours. On active alors le buzzer et on appelle la fonction *SendAlertMessage* qui envoie un message d'alerte sur le réseau LoRa.

Il est a noté que ces opérations ne peuvent être faites dans la fonction *GasAlert* car l'interruption et l'instruction d'envoie des messages sur le réseau LoRa (*ttn.send* présent dans *SendAlertMessage*) utilisent tous les deux l'unique *serial* de l'Arduino Uno. De plus, il est préférable de faire en sorte que la fonction appelée par l'interruption s'exécute le plus rapidement possible afin de ne pas rater la possible levée d'autres interruptions.

Si *alert = false*, alors aucune alerte n'est en cours. Dans ce cas, nous avons choisi d'envoyer toutes les 5 secondes la valeur du capteur de gaz sur le réseau Lora et de l'afficher sur le moniteur série de l'IDE Arduino.

```
void loop() {

    //Check if there is an alert
    //If yes, start the alarm & send an alert message (byte[0]=1)
    //We need to do this in the loop cause it's impossible to send message on the interrupt function
    //because both the interruption and the ttn.send(t) use the serial
    if (alert){
      SendAlertMessage();
      //Start Alarm
      MsTimer2::start();
    }
    else{
      //Send every 5 seconds the value of the gas sensor
      byte data[3];
      data[0] = 00;
      data[1] = 00;
      data[2] = 00;

      Payload(0, data);

      debugSerial.print("Sensor volt : ");
      debugSerial.print(data[1]+data[2]/100.0);
      debugSerial.println(" V");

      //Send data to TTN
      ttn.sendBytes(data, sizeof(data));

    }

    delay(5000);
}
```

Figure 9 - Fonction principale loop

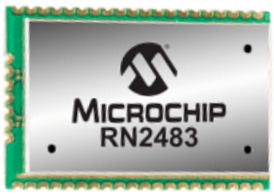
Nous verrons les fonctions *SendAlertMessage, Payload, ttn.sendBytes et ttn.poll* en détail dans les parties suivantes.

IV - C. Envoie de messages sur le réseau LoRaWAN & Traitement des données

Comme nous avons pu le voir dans la partie précédente, nous avons utilisé le réseau LoRaWAN pour envoyer la valeur du capteur de gaz et les alertes. Pour se faire nous avons utilisé *The Things Network*, réseau LoRaWAN open source. Le but étant par la suite de créer une interface utilisateur pour afficher ces informations.

IV - C.1. Transceiver Microchip RN2483

Afin de communiquer sur le réseau LoRa, nous utilisons le module *RN2483* de *Microchip*. La connexion avec la carte Arduino est la suivante :



Arduino	RN2483
10	TX
11	RX
12	RST
3V3	3V3
GND	GND

Enfin, nous ouvrons un serial au baud rate de 57600 (cf fig 10).

IV - C.2. The Things Network

The Things Network (TTN) est un réseau LoRa communautaire mondial. Ce dernier nous permet d'utiliser gratuitement le réseau déployé.

Pour se faire, il faut créer un compte puis créer une application et enfin enregistrer un device. Enfin, il faut importer la librairie *TheThingsNetwork.h* dans notre code Arduino, puis créer un objet *TheThingsNetwork* et enfin procéder à l'activation pour se connecter au réseau TTN. Il est a noter que nous avons choisi ici la méthode d'activation *Activation By Personalization (ABP)*.

```
#include <TheThingsNetwork.h>

SoftwareSerial loraSerial(RX, TX);
TheThingsNetwork ttn(loraSerial, debugSerial, freqPlan);
void setup() {
  loraSerial.begin(57600);

   //TTN Activation Method : ABP
  ttn.personalize(devAddr, nwkSKey, appSKey);

}
```

Figure 10 - Initialisation du serial LoRa et activation de la connexion avec TTN

IV - C.3. Envoie des données sur TTN

Maintenant que le réseau est configuré, nous pouvons envoyer des données. Ces dernières sont envoyées sous forme de bytes. Ainsi, nous créons une fonction pour former les messages sur Arduino, que nous décoderons ensuite sur TTN.

a) Création et envoi des messages sur l'Arduino

Nous avons décidé d'envoyer des messages dont la payload était de 3 bytes :

- **1er byte :** 00 = Pas d'alerte ; 01 = Alerte
- **2ème byte :** Partie entière de la valeur renvoyée par *GasSensorAverageVolt* à laquelle on soustrait *calibrationGazSensor*.
- **3ème byte :** Partie décimale de la valeur renvoyée par GasSensorAverageVolt à laquelle on soustrait calibrationGazSensor.

Nous verrons dans la *partie 2.4* à quoi correspond la variable *calibrationGazSensor*.

```
//Define the payload to send to TTN
//
//Arguments :
//    - alert (int) => 0 : no gas alert ; 1 : gas alert
//    - payload[] (byte) => data to send
//          Byte 0 = alert
//          Byte 1 = Integer part of the float value return by the GasSensorAverageVolt function
//          Byte 2 = Decimal part of the float value return by the GasSensorAverageVolt function
void Payload(int alert, byte payload[]){

  float tmp_sensor_volt;
  int intPart, decPart;

  //----------- Byte 0 ------------
  payload[0] = alert;

  //--------- Bytes 1 & 2 ---------
  //Get the float value of the gas sensor
  //Split the integer part and the decimal part
  //Put respectively the integer part and the decimal part into payload[1] and payload[2]
  tmp_sensor_volt = GasSensorAverageVolt()-calibrationGazSensor;

  if (tmp_sensor_volt < 0){
    tmp_sensor_volt = 0;
  }

  intPart = (int)tmp_sensor_volt;
  decPart = (tmp_sensor_volt-intPart)*100;

  payload[1] = intPart;
  payload[2] = decPart;
}
```

Figure 11 - Fonction de création du message à envoyer sur TTN

Cette fonction prend en argument un *int* « alert », qui vaut 0 s'il n'y a pas d'alerte ou 1 si une alerte est en cours, et un tableau de *byte* « payload » qu'il va remplir suivant la règle citée ci-dessus. Notre message est donc finalement un tableau de 3 bytes.

Une fois le message créé, nous l'envoyons sur le réseau à l'aide de la fonction (cf fig.9) :

ttn.sendBytes(data, sizeof(data))

b) Réception et décodage des messages sur TTN

Sur le site internet de TTN, nous pouvons voir les messages arriver dans l'onglet « data ».

Ces derniers arrivent sous forme brute, c'est-à- dire que nous voyons les 3 bytes que nous avons envoyés. Il est possible de traiter les données reçues afin de les rendre plus lisibles. Pour cela, nous implémentons sous TTN les fonctions « decoder » et « validator », de l'onglet « Payload Format ».

```
//---------------------------------------------------
//-------------- DECODER FUNCTION for TTN ------------
//---------------------------------------------------
//
//Decodes the payload (3 bytes) sent by the Arduino

function Decoder(bytes, port) {
  // Decode an uplink message from a buffer
  // (array) of bytes to an object of fields.
  var decoded = {};

  if (port === 1){
    //First byte of the payload message send by the arduino is the gas alert message : 00 = no alert ; 01 = alert

    //Check if bytes[0] is a valid value (0 or 1)
    //If not, the validator function will invalidate the payload :
    //Invalid messages will be dropped and not published to services that have subscribed to this application's messages.
    if (port === 1 && (bytes[0] === 0 || bytes[0] === 1)) {
      decoded.GasAlert = Boolean(bytes[0]);
    }

    //Second byte of the payload message send by the arduino is the integer part of the float value return by the gas sensor
    //Third byte of the payload message send by the arduino is the part of the float value return by the gas sensor
    var intPart = bytes[1];
    var decPart = bytes[2]/100.0;

    decoded.GasSensorVoltage = intPart+decPart;
  }

  return decoded;
}
```

Figure 12 - Fonction Decoder sur TTN

```
//---------------------------------------------------
//-------------- VALIDATOR FUNCTION for TTN -----------
//---------------------------------------------------
//
//Check if the first byte of the payload is a boolean
//Return false if the decoded message is invalid and should be dropped (in this case the GasAlert field won't be display)

function Validator(decoded, port) {

  if (port === 1 && typeof decoded.GasAlert !== 'boolean') {
    return false;
  }

  return true;
}
```

Figure 13 - Fonction Validator sur TTN

Une fois ces deux fonctions créées, nous obtenons les données décodées :

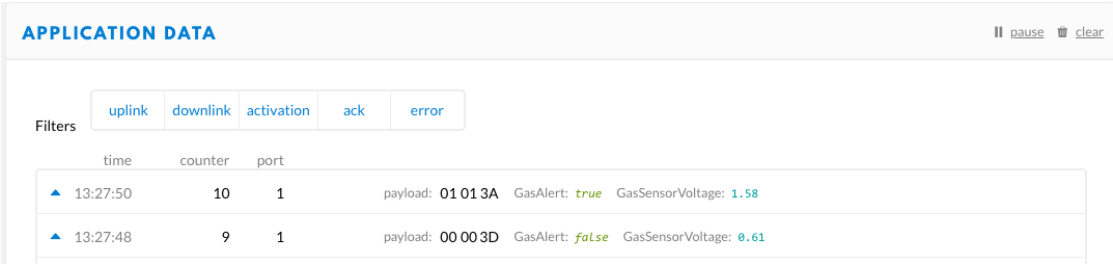
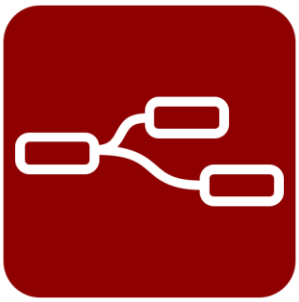


Figure 14 - Uplink messages décodés et validés sous TTN

IV - C.4. Node-RED



Afin de créer une interface utilisateur, nous utilisons Node-RED, un outil de développement graphique permettant de connecter ensemble les hardwares, APIs et services en ligne. Node-RED peut être directement associé à TTN, permettant ainsi de récupérer les messages échangés sur le réseau.

Nous avons mis en place une jauge permettant de visualiser la valeur du capteur de gaz, ainsi qu'un message indiquant si une alerte de gaz est en cours.

Le schéma Node-RED est le suivant :

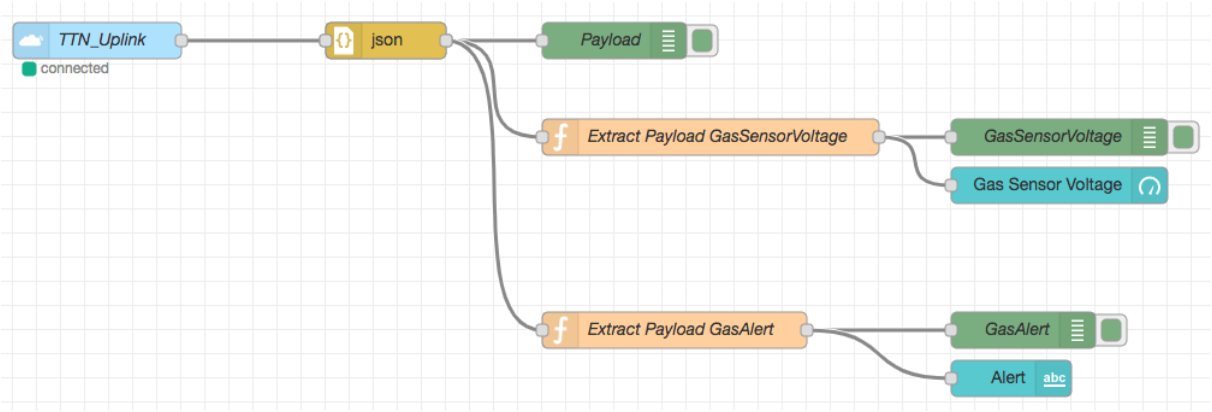


Figure 15 – NodeRED flow

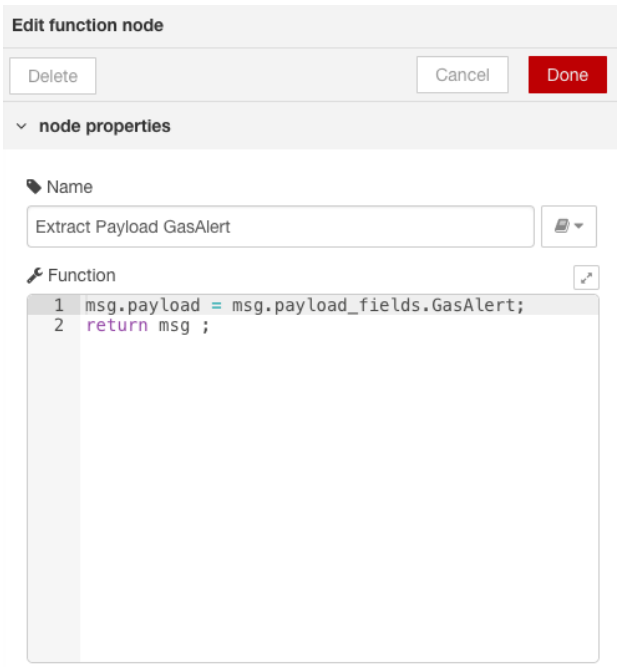


Figure 17 - Fonction "Extract Payload GasAlert"

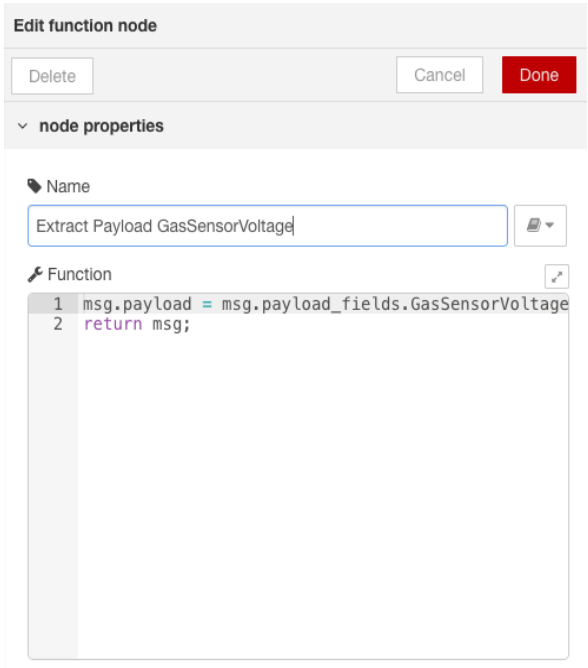


Figure 16 - Fonction "Extract Payload GasSensorVoltage"

Le Dashboard obtenu est le suivant :

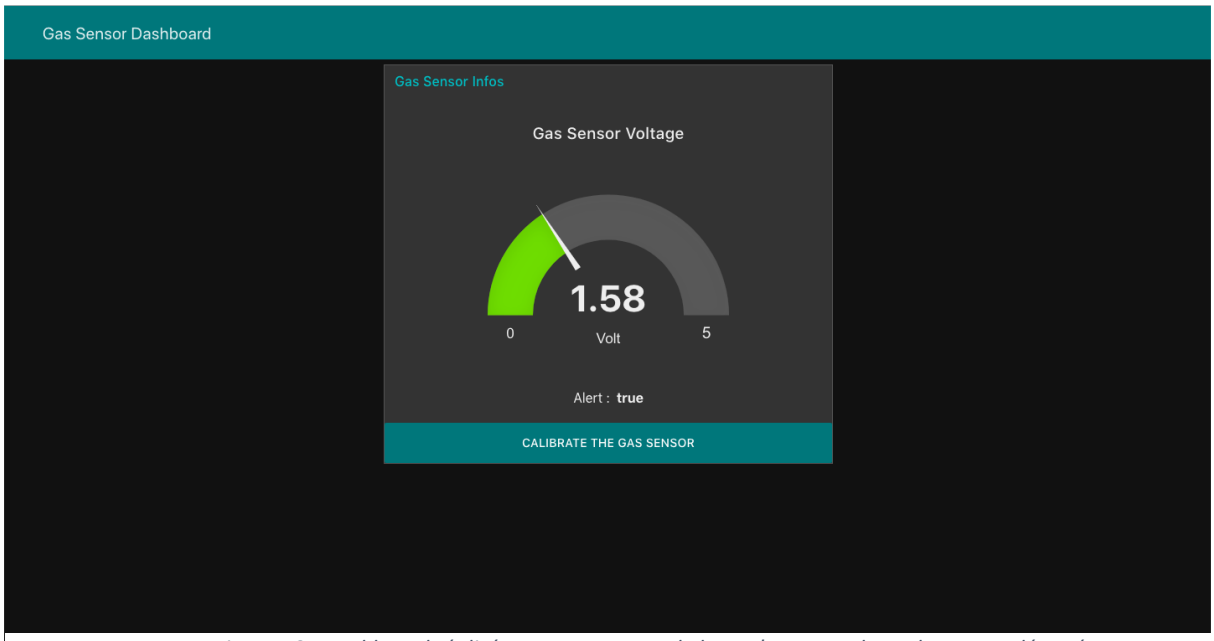


Figure 18 - Dashboard réalisé avec TTN - Exemple lorsqu'une une alerte de gaz est détectée

IV - D. Downlink messages - Calibration du capteur

Nous avons ici voulu utiliser les downlink messages. A l'inverse des uplink messages qui sont envoyés du capteur vers les antennes du réseau LoRa et reçu sur TTN, les donwlink messages sont eux envoyés depuis TTN vers le capteur (plus exactement vers la puce LoRa présente dans le capteur).

IV - D.1. Principe général

Nous avons rajouté à notre interface *NodeRED* un bouton permettant à l'utilisateur de calibrer le capteur de gaz. Lorsque ce dernier appuie sur ce bouton, un downlink message est envoyé vers le capteur. Le schéma *NodeRED* est le suivant :

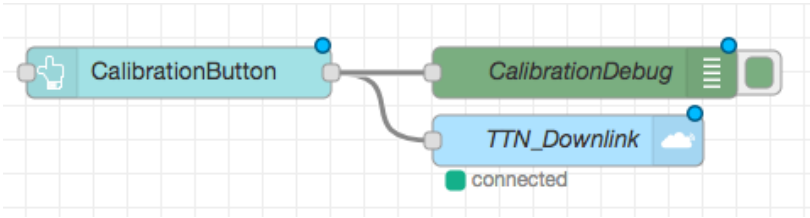


Figure 19 - Schéma NodeRED pour le bouton de calibration

Le bouton « *CalibrationButton* » a été programmé pour envoyer une *payload = « 05 »* lorsqu'il est activé :

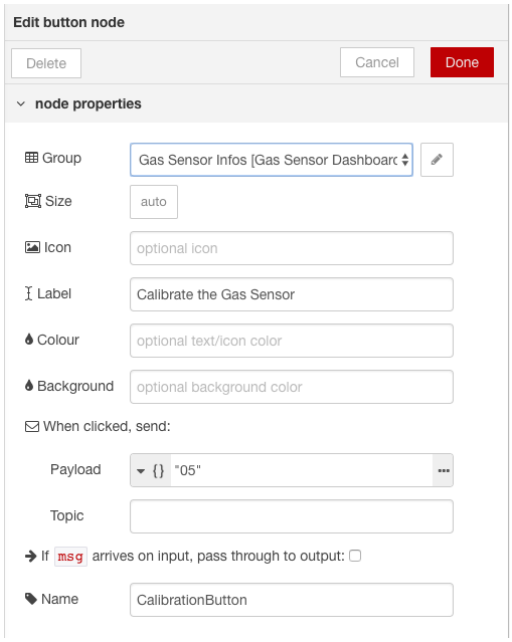


Figure 20 - Settings du bouton CalibrationButon

Ainsi, coté capteur, nous savons que lorsque l'on reçoit un downlink message avec une *payload = « 05 »*, cela signifie que l'utilisateur souhaite calibrer le capteur.

IV - D.2. Réception des downlink messages

Au niveau de la couche MAC des transceivers LoRa, il existe 3 classes différentes A, B et C dont les caractéristiques principales sont les suivantes :

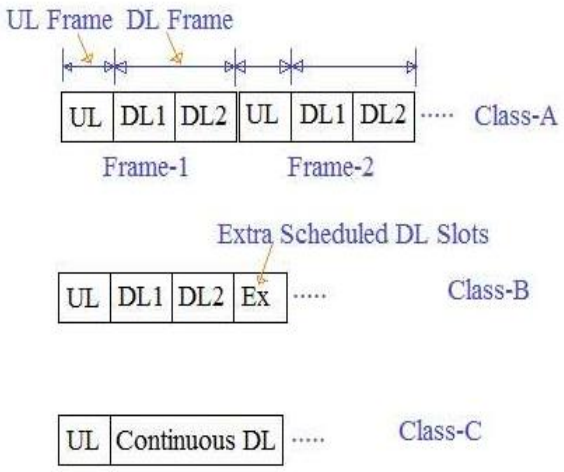


Figure 21 - Illustration des différentes classes au niveau MAC des devices LoRaWAN

- Classe A : Après chaque uplink message (*UL*) envoyé, deux « fenêtres » sont ouvertes permettant la réception de downlink message (*DL*). Les downlink messages reçus en dehors de ces 2 fenêtres ne seront pas pris en compte.
- Classe B : Reprend le fonctionnement de la classe A mais rajoute une fenêtre de réception (*Ex*) des downlink message supplémentaire dont l'ouverture périodique est synchronisée avec la gateway.
- Classe C : La fenêtre de réception est ouverte en permanence permettant ainsi la réception des downlink messages à n'importe quel moment (sauf quand un uplink message est envoyé).

Ces classes ont été pensées dans un soucis d'économie d'énergie, enjeu primordial dans le domaine des objets connectés. Ainsi, les devices de classe A consommeront moins que ceux de classe B qui consomment moins que ceux de classe C.

Le transceiver RN2483 utilisé dans ce projet est un classe A. Il nous est donc possible de recevoir un downlink message qu'après avoir envoyé notre uplink message de mise à jour de la valeur du capteur de gas (cf fig 9).

Ainsi, dans l'état actuel, c'est-à-dire avec l'envoie d'un uplink message toutes les 5 secondes, il est possible que nous manquions une demande de calibration de l'utilisateur. Pour remédier à cela, une évolution possible de la fonction principale *loop* est la suivante :

```
void loop() {

    while (i<3){
      ttn.poll();
      i++;
      delay(2000);
    }
      //Check if there is an alert
      //If yes, start the alarm & send an alert message (byte[0]=1)
      //We need to do this in the loop cause it's impossible to send message on the interrupt function
      //because both the interruption and the ttn.send(t) use the serial
      if (alert){
        SendAlertMessage();
        //Start Alarm
        MsTimer2::start();
      }
      else{
        //Send every 5 seconds the value of the gas sensor
        byte data[3];
        data[0] = 00;
        data[1] = 00;
        data[2] = 00;

        Payload(0, data);

        debugSerial.print("Sensor volt : ");
        debugSerial.print(data[1]+data[2]/100.0);
        debugSerial.println(" V");

        //Send data to TTN
        ttn.sendBytes(data, sizeof(data));

      }

      i=0;
}
```

Figure 22 - Mise à jour de la fonction loop pour éviter de rater un downlink message

Ici, nous appelons la fonction *ttn.poll* toutes les 2 secondes. Cette dernière envoie un simple byte permettant ainsi l'ouverture de 2 fenêtres de réception. Puis, toutes les 6 secondes nous effectuons le même algorithme que précédemment. Il serait possible de faire un *poll* toutes les secondes pour augmenter la précision mais nous avons remarqué qu'à cette fréquence d'envoi, tous les messages ne parviennent pas jusqu'à TTN.

Sous TTN, on peut ainsi voir les uplink messages de la fonction *ttn.poll* (payload : 00) ainsi que les downlink messages correspondant à une demande de calibration de la part de l'utilisateur (payload : 05) :

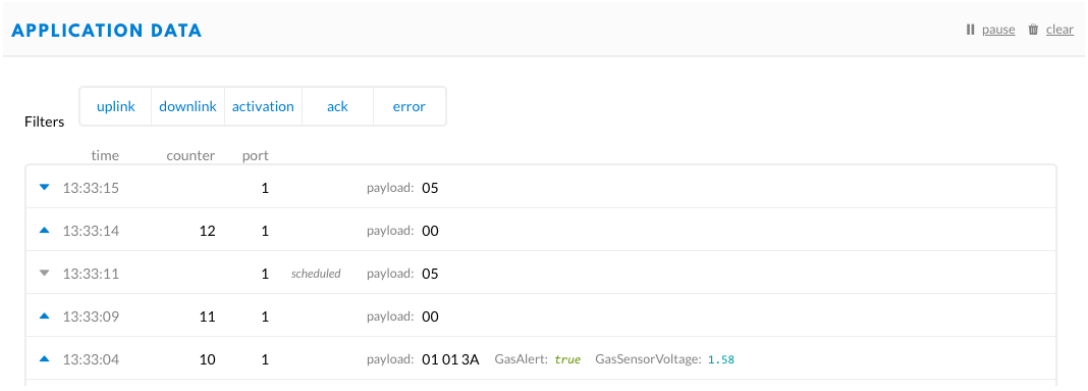


Figure 23 - Visualisation des uplink poll messages et downlink messages sur TTN

Enfin, pour que la réception des messages soit prise en compte, il est important d'appeler, dans la fonction *setup*, la fonction de callback :

ttn.onMessage(calibration)

Cette dernière appelle la fonction *calibration* lorsqu'un message est reçu.

IV - D.3. Fonction de calibration du capteur de gaz

La fonction *calibration* possède 3 arguments :

- **payload* : données reçues dans le downlink message
- *size* : taille, en bytes, de la payload
- *port* : port sur lequel on a reçu le downlink message

```
//Function used to display, on the serial monitor, the incoming TTN downlink messages
void calibration(const uint8_t *payload, size_t size, port_t port)
{
  debugSerial.println("-- DOWNLINK MESSAGE --");
  debugSerial.print("Received " + String(size) + " bytes on port " + String(port) + ":");

  for (int i = 0; i < size; i++)
  {
    debugSerial.print(" " + String(payload[i]));
  }

  debugSerial.println();

  //If the calibration button is press on the Node Red Dashboard (=> TTN downlink message = 05)
  //Then define the calibration offset of the gas sensor
  if (payload[0]==5 && payload[1]==0 && size==1)
  {
    calibrationGazSensor = GasSensorAverageVolt();
    debugSerial.print("Calibration OK ! Offset = ");
    debugSerial.println(calibrationGazSensor);
  }
}
```

Figure 24 - Fonction calibration appelée lorsqu'un downlink message est reçu

Dans un premier temps, cette fonction affiche sur le *serial monitor* les informations (size, port) et le contenu du message reçu. Puis dans un second temps, vérifie si la payload reçue est bien égale à « 05 », c'est-à-dire que ça correspond bien à un appuie sur le bouton *CalibrationButton* par l'utilisateur. Si tel est le cas, alors la variable globale *calibrationGazSensor* prend la valeur retournée par la fonction *GasSensorAverageVolt* décrite précédemment.

La partie Software et Hardware ont été testé ensemble, en effet elles dépendent l'une de l'autre. Nous devons nous assurer du bon fonctionnement du projet afin de respecter les exigences du cahier des charges. Nous pouvons maintenant concevoir le shield sur KiCAD qui nous permettra d'intégrer tous les composants sur l'Arduino.

V - Partie KiCAD

KiCAD est un logiciel open-source qui permet la création de schémas électroniques et de Printed Circuit Board (PCB). KiCAD a l'avantage de pouvoir regrouper différentes interfaces dans le but de réaliser l'ensemble des étapes de la réalisation d'un circuit imprimé. Nous l'avons donc utilisé pour réaliser notre shield qui doit regrouper le Trigger de Schmitt, le buzzer, le bouton RESET et le transceiver LoRa.

Dans un premier temps, nous allons expliciter les étapes que nous avons réalisé pour modéliser notre puce RN2483 que nous avons intégré à notre shield. Puis nous allons réaliser le schéma électronique, la liste des empreintes et finir par le routage de tous les composants.

V - A. Conception du module de la puce RN2483

Nous disposons d'une puce LoRa intégrée à un shield réalisé par Azzy's Electronics, dont voici un aperçu :

Figure 25 – Shield de la puce RN2483

Nous devons donc créer un modèle du shield afin de l'intégrer dans notre PCB. Dans un premier temps, nous allons dessiner le symbole qui va représenter la puce LoRa avec les connexions, puis nous allons dessiner l'empreinte de la puce LoRa.

V - A.1. Création du symbole de la puce LoRa

Nous avons créé le symbole de la puce LoRa afin de faire apparaitre toutes les pins du Shield. Nous avons annoté toutes les connexions, précisé les pins qui ne nécessitaient pas de connexion et le type des connexions (entrée, sortie, power). Vous pouvez retrouver le symbole de notre puce LoRa en **Annexe**.

V - A.2. Création du l'empreinte du shield de la puce LoRa

De même pour la puce LoRa, nous avons créé une empreinte afin de reprendre les dimensions exactes du shield développé Azzy's Electronics. Nous avons veillé à respecter la taille des pins, la numérotation des broches et à créer un dossier « .pretty » contenant le symbole avec l'empreinte. Vous pouvez retrouver l'empreinte de la puce LoRa en **Annexe.**

Nous pouvons maintenant utiliser le symbole et l'empreinte de la puce dans notre projet KiCAD.

V - B. Création du circuit sur Eeschema

Nous pouvons réaliser le circuit de notre montage grâce à *Eeschema*. Pour cela nous partons d'un board pour un Arduino Uno, nous pouvons compléter le projet avec notre circuit. Vous pouvez trouver le circuit en **Annexe**.

Nous avons intégré les différents éléments relatifs au capteur de gaz connecté :

- Le Trigger de Schmitt.
- Le buzzer avec l'indicateur lumineux.
- Le bouton RESET.
- Le connecteur pour le capteur de gaz.
- Le shield de la puce LoRa.

Ensuite, après numéroté et vérifié l'ensemble des connexions, nous devons lister les empreintes pour chaque composant en fonction des éléments disponibles en salle de TP, vous pouvez trouver la liste de toutes les empreintes en **Annexe**.

Finalement, à partir de la liste des empreintes, nous pouvons générer la netlist et commencer à faire le routage du PCB.

V - C. Création du circuit imprimé sur Pcbnew

Pour créer le circuit imprimé, nous utilisons Pcbnew. Pour réaliser le routage, nous devons suivre des règles simples qui sont les suivantes :

- Paramétrer la largeur des pistes des alimentations.
- Ne pas faire d'angles à 90° avec les pistes.
- Faire un plan de masse sur l'ensemble du PCB.
- Placer un texte afin de repérer la face recto du verso.
- Vérifier le routing avec l'outil Design Rule Check (DRC).

VI - Conclusion

Pour conclure, le cours sur les microcontrôleurs et l'open-source hardware nous a donné les clés pour designer et concevoir un capteur connecté. Nous avons commencé par ajouter petit à petit l'ensemble des fonctionnalités attendues en testant tour à tour le hardware et le software pour terminer le projet.

Nous avons dimensionné le Trigger de Schmitt afin détecter un dépassement anormal de la concentration en gaz de combustion. Nous avons implémenté un buzzer et un bouton RESET. Puis nous nous sommes occupés de la communication entre l'Arduino Uno et la Gateway LoRa de l'INSA, que cela soit pour la transmission des données, la demande de calibration de l'utilisateur, ou même pour avertir l'utilisateur d'un possible danger. Nous avons créé un dashboard qui permet l'affichage des valeurs, ainsi que des messages d'alerte, et la demande de calibration de la part de l'utilisateur. Finalement, nous avons terminé par la création d'un shield Arduino sur KiCAD.

Une étude approfondie de la consommation d'énergie peut être faite sur le capteur afin de définir comme un objet connecté. L'amélioration du programme en ajoutant des « sleep », en changeant les modes de consommation de l'Arduino UNO entre deux envoies, ou même en réduisant la fréquence d'émission des données sont des pistes à suivre pour rendre autonome le système. Cependant, le capteur de gaz dont nous disposons n'est pas adapté : il consomme beaucoup trop du fait de sa résistance de chauffe et son temps de réponse est relativement élevé.

Le shield n'a pas encore été développé par l'INSA, et donc testé. Cependant les tests préalables suggèrent que le capteur connecté rempli les exigences du cahier des charges.

VII - Annexe

Datasheets

MQ-9 Gas Sensor Datasheet: for C0/Combustible Gas
http://www.haoyuelectronics.com/Attachment/MQ-9/MQ9.pdf

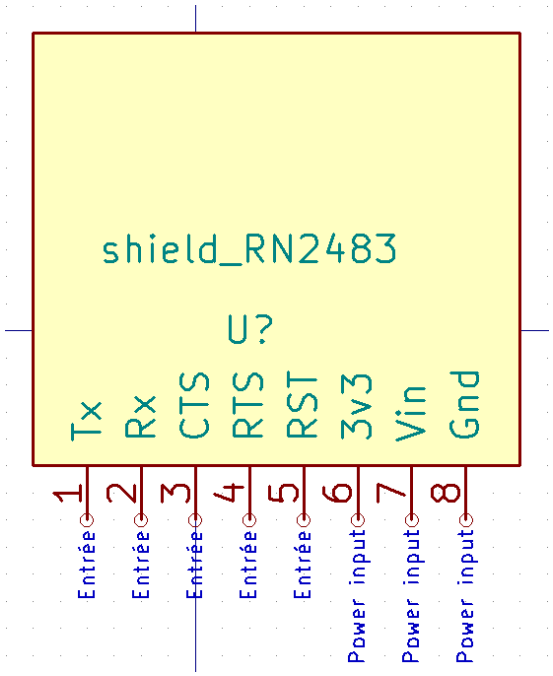
Grove Gas Sensor with MQ-9
http://wiki.seeedstudio.com/Grove-Gas_Sensor-MQ9/

LM741 datasheet
http://www.ti.com/lit/ds/symlink/lm741.pdf

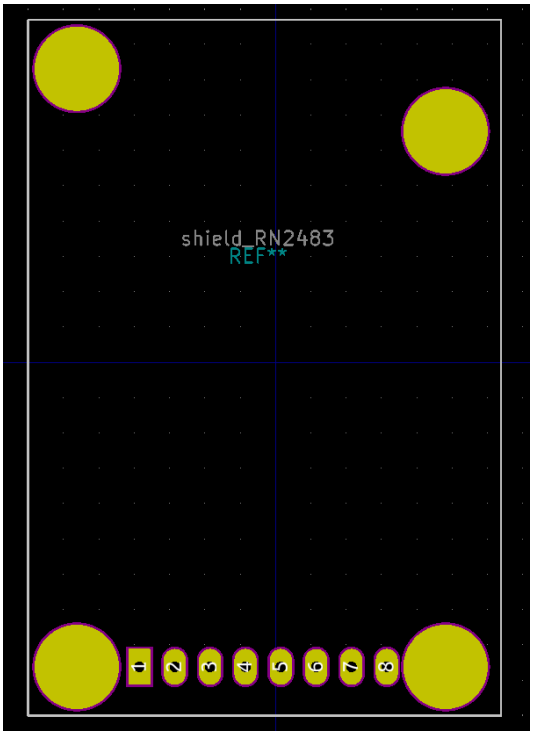
Puce RN2483 datasheet
http://ww1.microchip.com/downloads/en/DeviceDoc/50002346C.pdf

RN2483 breakout (bare board)
https://www.tindie.com/products/DrAzzy/rn2483-breakout-bare-board/

Symbole et Empreinte de la puce LoRa RN2483 :

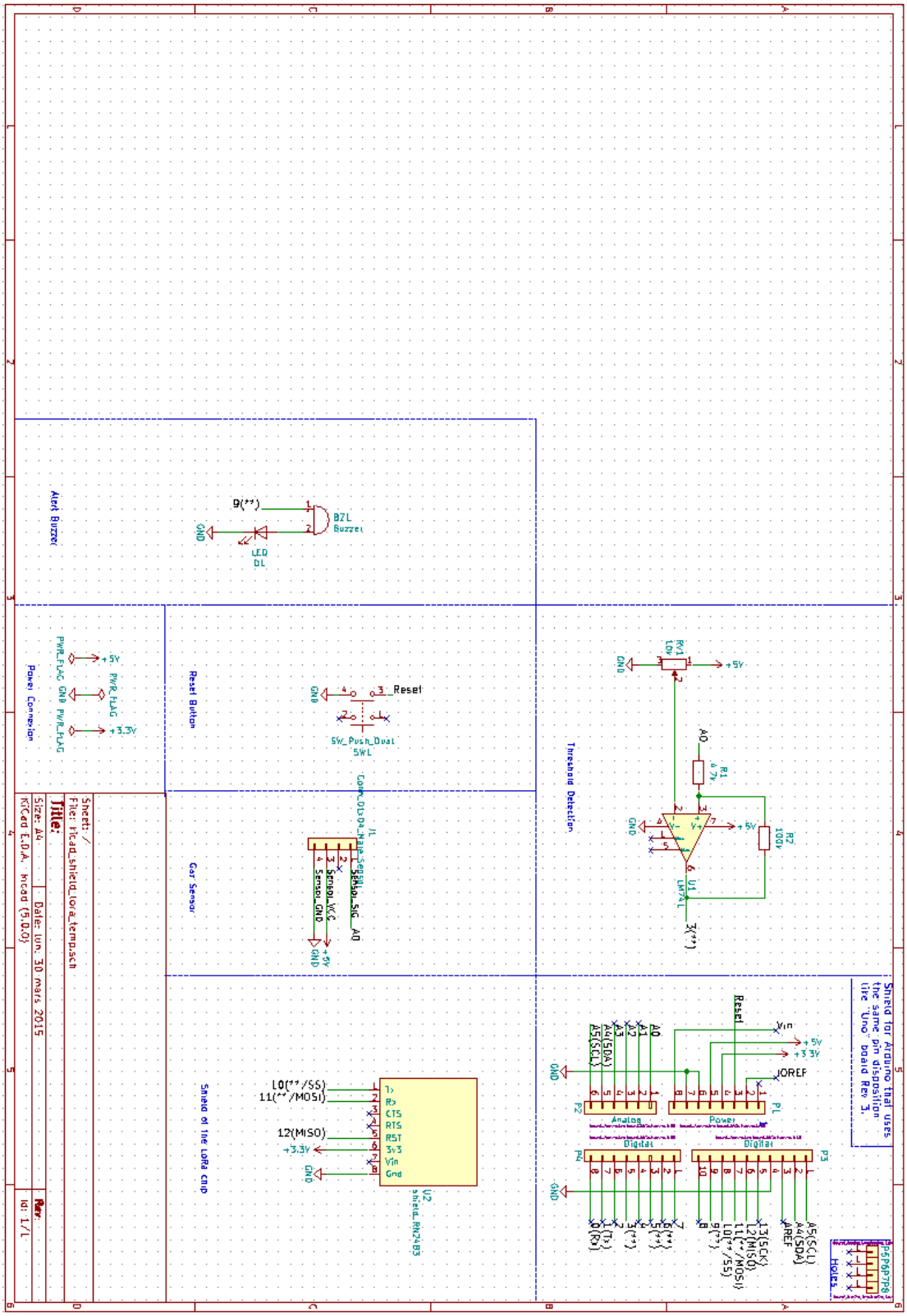


Symbole de la puce RN2483



Empreinte de la puce RN2483

Montage du shield Arduino:



Liste des Empreintes :