



L3 EEA-REL

Bureau d'étude :

Réalisation d'une ruche connectée

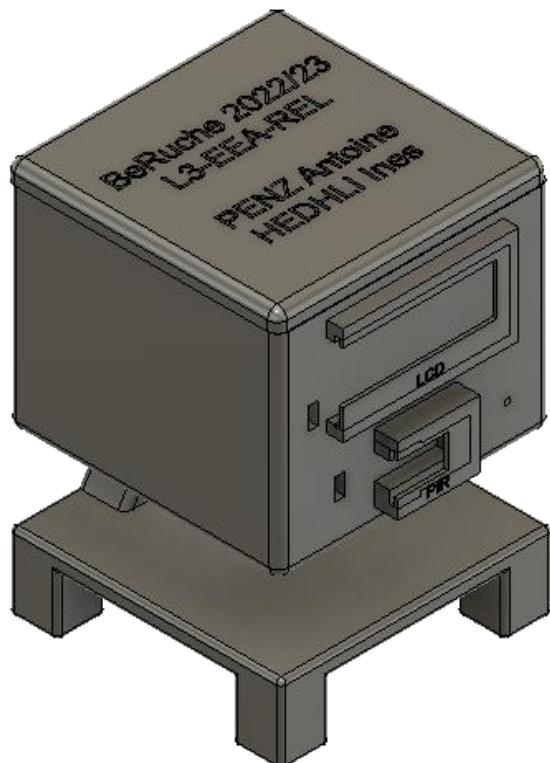


Table des matières

Présentation du sujet	3
Fonctionnement de la ruche	4
Capteur PIR.....	5
Relecture de la tension de la batterie	7
Jauge de contrainte et amplificateur HX711.....	8
Capteur température et humidité DHT11.....	14
Capteur température et humidité SHT31.....	19
Ecran LCD 16x2	26
Emetteur récepteur LoRa	27
Fonctionnement de la ruche avec tous les capteurs.....	42
Réception des données LoRa et affichage sur une page internet.....	45
Logiciel et langage de programmation utilisé sur le Raspberry pi	46
Réceptions des message LoRa & base de données	46
Serveur Web.....	49
Lancement des applications	52

Présentation du sujet

Le cahier des charges du projet final était de réussir à instrumenter une ruche et d'afficher ces informations sur un site internet pour qu'un apiculteur puisse regarder en temps réelle l'état de ça ruche.

Dès la première séance une multitude de capteur et de carte électronique nous à étais fournie :

- Deux capteurs de température et d'humidité
 - Un capteur de préséance PIR
 - Un écran LCD
 - Plusieurs jauge de contrainte, ainsi qu'un amplificateur pour jauge de contrainte
 - Plusieurs paire d'émetteur/récepteur LoRa différent
 - Une batterie au lithium avec son module de recharge
- Ainsi que plusieurs cartes arduino's avec leurs sheild grove et une carte Raspberry pi

Avec tous ce matériel en main nous avons pu créer le schéma fonctionnel de la ruche et du Raspberry pi.

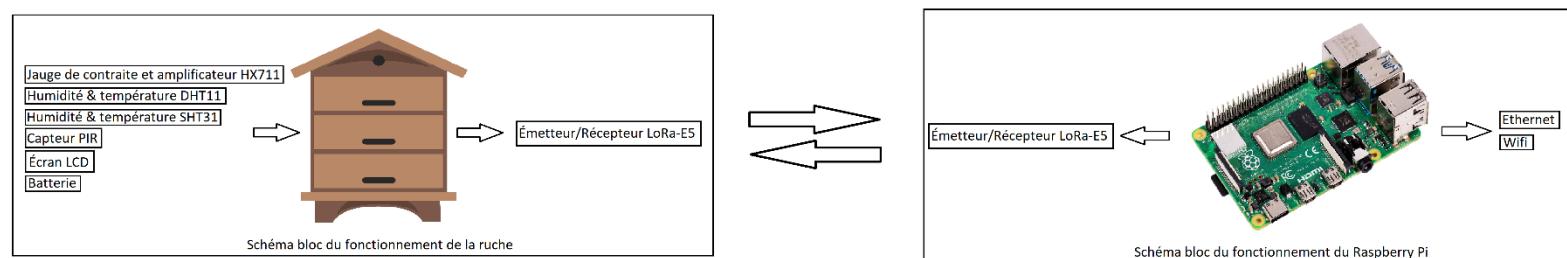


Figure 1 : schéma bloc du système complet de la ruche

Le fonctionnement du système complet de ce projet à était coupé en deux parties.

La première sera l'acquisition des données de la ruche via un arduino et l'émission de celle-ci à un Raspberry pi via des émetteurs/récepteurs LoRa.

La deuxième partie sera l'analyse et le stockage des données sur un raspberry pi pour ensuite pouvoir les afficher sur une page web.

Comme nous savions que la partie LoRa et la réalisation de la page web allait nous prendre beaucoup de temps, nous avons décidé dès le début de nous répartir le projet en deux. Ines s'occupera de la partie arduino avec tous les capteurs de la ruche, alors qu'Antoine s'occupera de la liaison LoRa et la réalisation de la page web hébergé sur le raspberry pi.

Fonctionnement de la ruche

Comme expliquer précédemment le rôle de la ruche connectée sera de lire les données des différents capteurs via un arduino et d'emmèter ces valeurs sans fil via un émetteur récepteur radio LoRa.

En plus du code final de la ruche, nous avons pu développer des multitudes de code de test pour les différents capteur et fonction pour que l'on puisse valider leurs utilisations.

Tous ces codes peuvent être retrouver sur le Github du projet :
« <https://github.com/Penzanto/L3-EEA-BE-Ruche-Connectee/> »

Dans un premier temps nous allons nous intéresser à chaque capteur de la ruche, puis à la fin nous regarderons le schéma complet de la ruche ainsi que le code final de la ruche connectée.

Capteur PIR

Ce capteur est le seul capteur de toutes la ruche à ne pas avoir de code exemple tellement son utilisation est simple.

Ce capteur est en réalité un capteur de présence infrarouge, au lieu d'analyser le mouvement d'une personne dans le spectre de couleur que nous utilisons. Ce capteur utilise le spectre infrarouge et donc utilise les variations de température pour détecter un mouvement. Cela veut dire pour nous qu'il ne détectera pas un arbre qui pourrait bouger devant le capteur à cause du vent. Mais il détectera bien une personne qui passe devant la ruche ! Ceci pourrait être utile pour détecter d'éventuelle voleur de mielle ou de ruche.



Figure 2 : Recto/Verso du capteur PIR

Ce capteur possède deux potentiomètres. Un pour régler la sensibilité de détection du capteur, puis le second est utilisé pour modifier le temps On de la sortie tous ou rien du capteur. Cette valeur peut varier de 3 secondes jusqu'à 130 secondes.

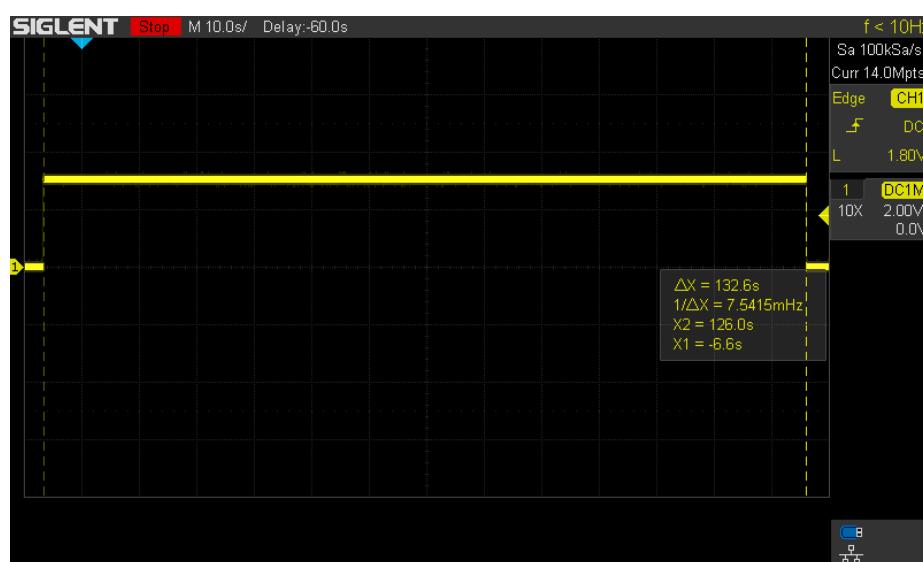


Figure 3 : Sortie TOR du capteur avec le Ton le plus élevé

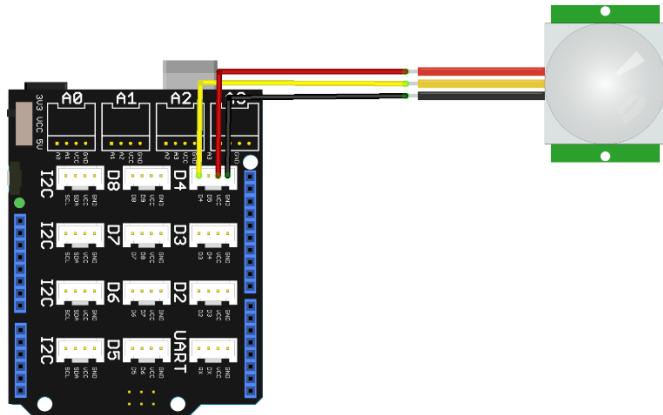


Figure 4 : Schéma de câblage du capteur au shield arduino Grove

```
#define outputPIR 4 //Liaison du numéro du pin de la carte arduino à un nom compréhensible

void setup() {
  pinMode(outputPIR, INPUT); //Déclaration du pin PIR en entrée

  Serial.begin(9600); //Démarrage du port série de la carte arduino à une vitesse de 9600baud/seconde
}

void loop() {
  Serial.println("Valeur capteur PIR :" + String(digitalRead(outputPIR))); //Écriture de la valeur lue par la carte arduino sur le port série

  delay(1000); //Délais d'attente d'une seconde
}
```

Figure 5 : Code arduino du capteur PIR

```
01:36:13.305 -> Valeur capteur PIR :0
01:36:14.275 -> Valeur capteur PIR :0
01:36:15.315 -> Valeur capteur PIR :0
01:36:16.310 -> Valeur capteur PIR :0
01:36:17.281 -> Valeur capteur PIR :1
01:36:18.288 -> Valeur capteur PIR :1
01:36:19.315 -> Valeur capteur PIR :1
01:36:20.295 -> Valeur capteur PIR :0
01:36:21.318 -> Valeur capteur PIR :0
01:36:22.315 -> Valeur capteur PIR :0
```

Figure 6 : Message de sortie du code arduino sur le port série

Relecture de la tension de la batterie

Ce code peut-être retrouver sur notre Github sous le nom « Lecture_tensionBatRuche.ino »

Ce morceau de code sera responsable de la lecture de la tension de la batterie qui se trouve dans la ruche pour notifier l'utilisateur d'un éventuel problème si elle venait à avoir une tension trop faible.

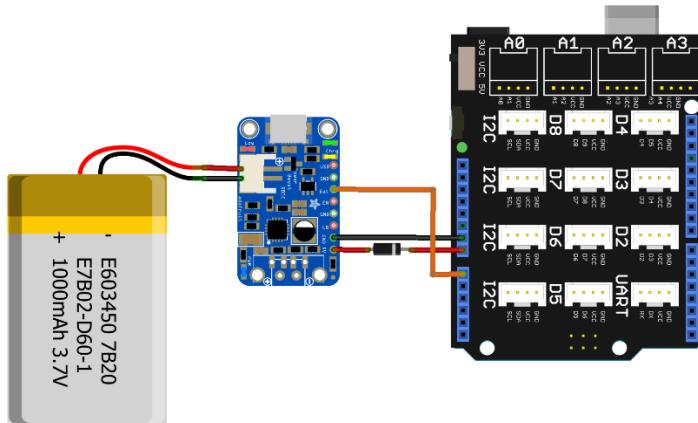


Figure 7 : Circuit de câblage de la batterie à la ruche

On remarquera l'utilisation d'une diode sur la sortie du régulateur de la batterie. Cette diode est présente pour éviter d'alimenté la sortie du régulateur de la batterie lorsque l'arduino est brancher sur une autre source d'alimentation (via le port usb par exemple lors de la reprogrammation de celui-ci).

Nous avons aussi directement câblé la tension de sortie de la batterie sur l'entrée analogique de l'arduino. Ceci vient du fait que l'entrée analogique de l'arduino peut monter jusqu'à 5V et que la tension de la batterie ne peut pas dépasser 4.2V lorsqu'elle est pleine.

Cependant nous avons eu un problème lorsque nous alimentons l'arduino avec seulement la batterie. En effet la pin Vin ou nous avons brancher la sortie 5V du régulateur de tension de la batterie n'est sensé recevoir que des tensions entre 7 et 12V.

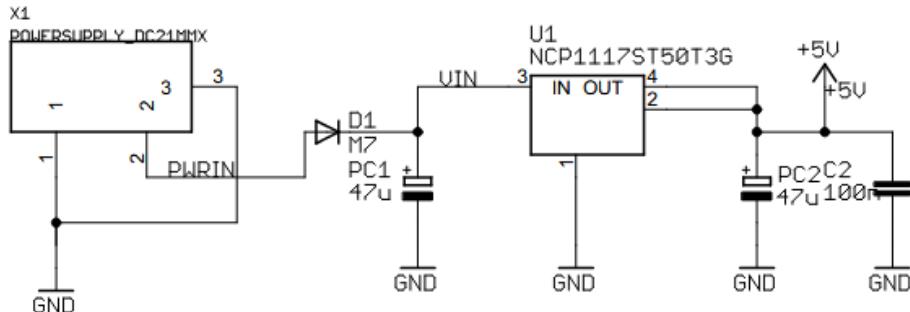


Figure 8 : Circuit de génération du +5V à partir du Vin sur la carte arduino

Comme nous rentrons en 5V en amont du régulateur de tension linéaire « NCP1117ST50T3G » il lui est impossible de générer la même tension en sortie, il y aura forcément une chute de tension entre l'entrée et la sortie sur ces types de convertisseur.

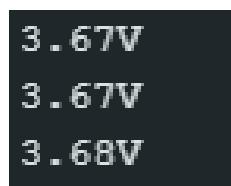
C'est pour cela que nous, nous retrouvons avec seulement 3.5V sur l'alimentation général 5V de l'arduino et donc de tous les autres capteurs par la même occasion.

Cette tension d'alimentation est bien-sûr beaucoup trop faible pour pouvoir faire fonctionner correctement la ruche. Il faudrait rajouter un convertisseur DC-DC boost ou élévateur pour remonter la tension de sortie du régulateur de la batterie à une tension utilisable par l'arduino de minimum 7V.

malheureusement nous n'avons pas eu le temps d'implémenter cette solution à notre projet. Du coup pour palier à ce problème, même si la batterie est branchée sur la ruche il faudra qu'une autre source d'alimentation soit présent pour que celle-ci fonctionne. Dans les essais qui vont suivre nous avons toujours eu la prise usb de branchée sur la carte arduino ce qui lui permettait de fonctionner correctement.

```
void setup() {  
    //initialisation de la communication série port com  
    Serial.begin(9600);  
}  
  
void loop() {  
    //lecture et conversion de la valeur lue en tension  
    float voltage = analogRead(A0) * (5.0 / 1023.0); //l'arduino comprend un ADC 10bits avec une valeur max de 5V  
  
    //écriture de la valeur lue  
    Serial.println(String(voltage) + "V");  
  
    delay(500);  
}
```

Figure 9 : Code arduino de la lecture de la tension de la batterie



3.67V
3.67V
3.68V

Figure 10 : Valeur rendu sur le port série de l'arduino

Remarque : Comme l'arduino possède un convertisseur analogique digital sur 10bits avec comme tension de référence 5V. l'équation pour retrouver la tension lue sur le pin analogique est le suivant :

$$Tension_{Lu} = Val_{ADC} * \frac{5}{2^{10} - 1}$$

Jauge de contrainte et amplificateur HX711

Pour mesurer le poids de la ruche nous utilisons une jauge de contrainte à base d'un point de Wheatstone et d'un circuit intégrer qui nous fait l'amplificateur et le codage série des données pour nous faciliter l'utilisation de cette jauge.

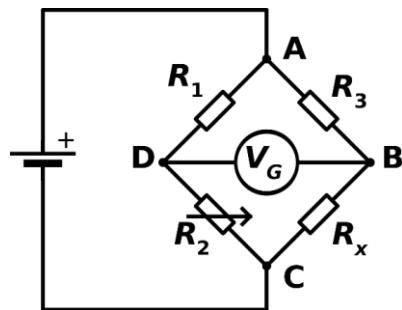


Figure 11 : Exemple de fonctionnement d'un pont de Wheatstone

Une jauge de contrainte fonctionne en faisant varier la valeur d'une des résistances du pont de Wheatstone (dans notre exemple R_2). Pour créer une tension V_g variable en fonction de la déformation de la jauge de contrainte.



Figure 12 : placement de la jauge de contrainte dans la maquette de la ruche

La maquette de la ruche à était réalisée pour nous aider à calibrer la jauge de contrainte. La jauge est placée d'une façon à avoir tout le poids de la ruche de chaque coté pour avoir une déformation bien linéaire en fonction de la masse appliquée à celle-ci.

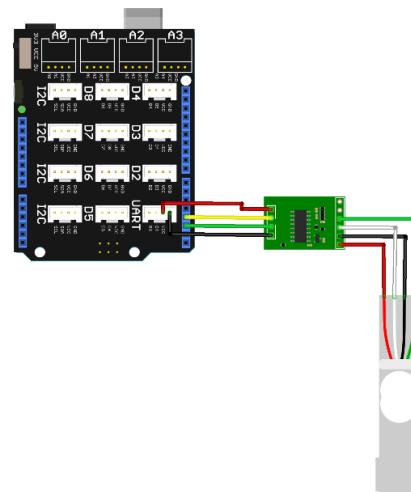


Figure 13 : Schéma de câblage de la jauge de contrainte et de son amplificateur

Le rôle de l'amplificateur va être de rendre les petites variations de tension au niveau de la jauge de contrainte lisible par l'arduino. Le circuit intégrer HX711 réalise cela en venant digitaliser la tension de la jauge de contrainte. Ceci nous permet aussi de réduire les erreurs de lecture de la tension au niveau de l'arduino dans le cas où nous aurions eu un environnement pollué par des interférences électromagnétiques.

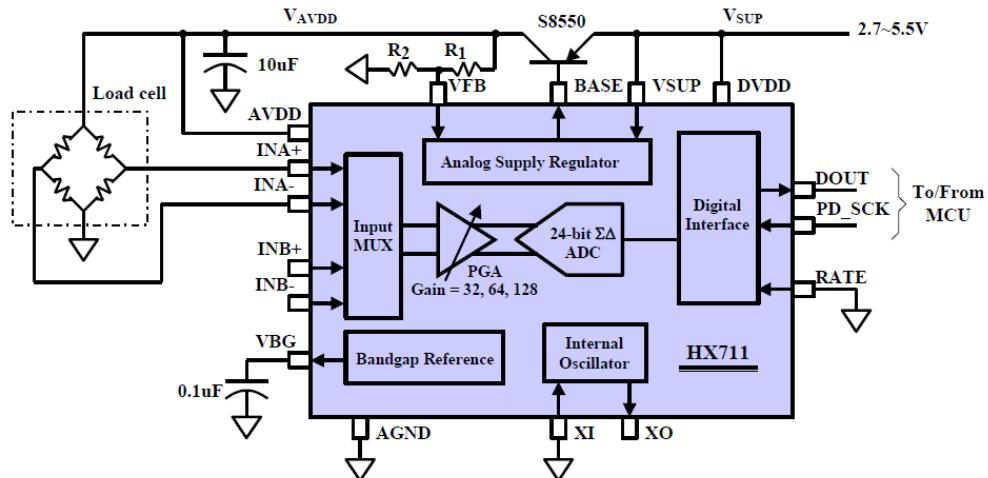


Figure 14 : Câblage et schéma bloc interne de fonctionnement du circuit HX711

Dans le circuit intégré HX711 on retrouve de gauche à droite, un multiplexeur pour pouvoir choisir qu'elle entrée de jauge de contrainte doit être digitalisé (dans notre cas on utilise que l'entrée A). Nous retrouvons ensuite un amplificateur à gain sélectionnable entre 32, 64 et 128. Ensuite nous retrouvons un ADC 24bits sigma delta et enfin le circuit digitaliseur.

Pour pouvoir communiquer avec ce circuit intégrer il nous faut envoyer avec la carte arduino des pulses de clock sur l'entrée « PD_SCK ». Par exemple si on veut sélectionner le gain 128 pour l'entrée A il nous faudra envoyer 25 pulses sur l'entrée PD_SCK.

PD_SCK Pulses	Input channel	Gain
25	A	128
26	B	32
27	A	64

Figure 15 : Sélection du gain du CI HX711 (extrait de la datasheet)

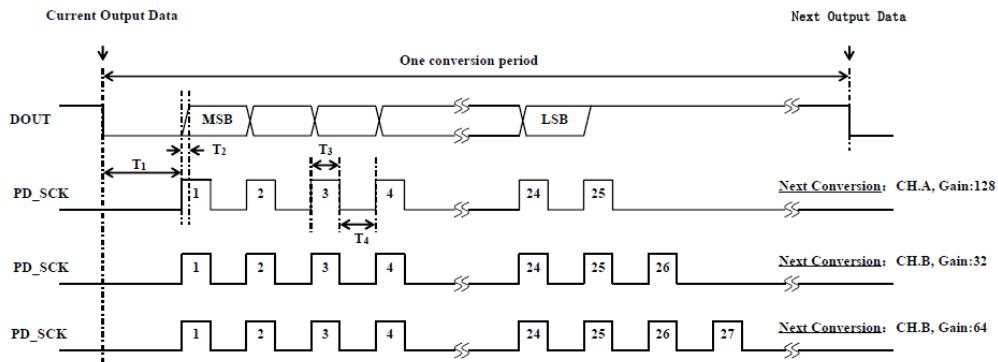


Figure 16 : Communication série complète du composant HX711

Ce circuit intégrer fonctionne en fait en deux temps. Dans un premier temps on vient envoyer les pulses qui indique qu'elle entrée doit être lu avec le gain que l'on souhaite. Un fois cela fait on regarde l'état de la pin Dout, si ce pin est à l'état haut cela veut dire que le circuit intégré est en train de faire les conversions nécessaires pour redonner la bonne valeur de la jauge de contrainte. Une fois que la pin Dout passe à l'état bas on peut envoyer les X pulses que l'on souhaite pour la conversion suivante, cependant ces fronts montants d'horloge indique aussi au circuit intégrer quand est-ce qu'il doit changer sa valeur de sortie et passer à la valeur suivant (valeur de la jauge de contrainte digitalisé sur 25bits). Dans notre cas comme on veut faire qu'une seule lecture on va juste générer 25 pulsations en entrée de PD_SCK du HX711 et lire les sorties Dout sur tous les fronts descendants de notre PD_SCK comme une liaison série classique.

Symbol	Note	MIN	TYP	MAX	Unit
T ₁	DOUT falling edge to PD_SCK rising edge	0.1			µs
T ₂	PD_SCK rising edge to DOUT data ready			0.1	µs
T ₃	PD_SCK high time	0.2	1	50	µs
T ₄	PD_SCK low time	0.2	1		µs

Figure 17 : Timming des clocks demandé par la datasheet du HX711

Tous ces temps de clock sont extrêmement faibles (de l'ordre de la µs). Cependant la datasheet nous donne un exemple de code en C ou il suffit de passer la pin PD_SCK à l'état haut puis juste après à l'état bas pour générer les impulsions de 1µs.

```

//declaration des pin du capteur
#define data_HX711 2
#define clk_HX711 3

void setup() {
    //initialisation des pin du pc
    pinMode(clk_HX711, OUTPUT);
    pinMode(data_HX711, INPUT);
    //initialisation de la communication serie
    Serial.begin(9600);
}

void loop() {
    //variable local
    uint32_t valeur_HX711 = 0;
    uint8_t erreur_HX711 =0;

    //lecture de la valeur
    lectureHX711(&valeur_HX711, &erreur_HX711, clk_HX711, data_HX711);

    //verification de l erreur de la lecture
    if(erreur_HX711 != 1)
    {
        //correction de la valeur rendu par le capteur
        Serial.println("Masse = " + String((valeur_HX711*0.00215525)-611.284) + "g");
    }
    else
    {
        //ecriture d un message d erreur
        Serial.println("erreur lors de la lecture");
    }

    delay(100);
    // //mise en veille du capteur pendant 3 secondes
    // miseEnVeilleHX711(clk_HX711);
    // delay(3000);
}

void lectureHX711(uint32_t *valeur_HX711, uint8_t *erreur_HX711, uint8_t pinClk_HX711, uint8_t pinData_HX711)
{
    /*
        Fonction de lecture d une valeur du capteur HX711

        Args:
            valeur_HX711 (uint32_t): valeur brut lu par le capteur
            erreur_HX711 (uint8_t): flag d erreur de lecture de la valeur du capteur (true veut dire qu il y a eu une erreur durant la lecture)
            pinClk_HX711 (uint8_t): numero du pin de la clock du capteur sur le pc
            pinData_HX711 (uint8_t): numero du pin de la data du capteur sur le pc
    */

    //variable local
    uint16_t timeOut =0;

    //mise a zero des pointeurs
    *valeur_HX711 =0;
    *erreur_HX711 =0;

    //envoie de la premiere trame de demande de conversion avec un gain de 128
    for(int i =0; i<25; i++)
    {
        digitalWrite(pinClk_HX711, HIGH);
        digitalWrite(pinClk_HX711, LOW);
    }

    //attente de la conversion du capteur avec un timeout de securitee
    timeOut =0;
    while(digitalRead(pinData_HX711)!=0 & timeOut < 1000)
    {
        delay(10);
        timeOut = timeOut +10;
    }

    //verification que le capteur a bien fini de faire ca conversion
    if(digitalRead(pinData_HX711)==0)
    {
        *valeur_HX711 = 0;
        //envoie des trame de clock pour relire la data du capteur
        for(int i =0; i<25; i++)
        {
            digitalWrite(pinClk_HX711, HIGH);
            *valeur_HX711 = *valeur_HX711 << 1 | digitalRead(pinData_HX711);
            digitalWrite(pinClk_HX711, LOW);
        }
        else
        {
            *erreur_HX711 =1;
        }
    }
}

void miseEnVeilleHX711(uint8_t pinClk_HX711)
{
    /*
        Fonction de mise en veille du capteur HX711

        Args:
            pinClk_HX711 (uint8_t): numero du pin de la clock du capteur sur le pc
    */

    digitalWrite(pinClk_HX711, HIGH);
}

```

Figure 18 : code arduino pour venir lire les valeurs de la jauge de contrainte et mettre ne
vieille le circuit intégrer HX711

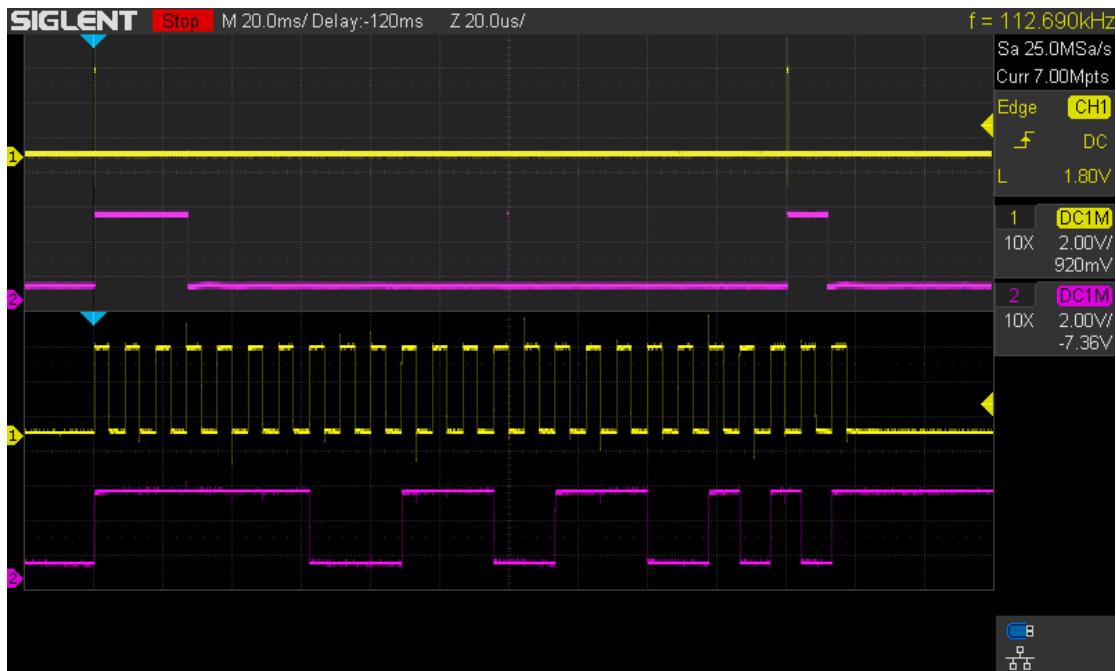


Figure 19 : Exemple de communication entre l'arduino et le circuit intégrer HX711

Une fois ce nouveau code chargé dans l'arduino on retrouve sur le port série la valeur du poids que l'on place sur la ruche (le poids de la ruche à était taré de la jauge de contrainte). Cependant avant ça nous avons demandé à l'arduino de nous sortir les valeurs brutes de la jauge de contrainte avant compensation. Ceci nous a permis de tracer la réponse du système en fonction de différents poids étalons que nous avons placés sur la ruche.

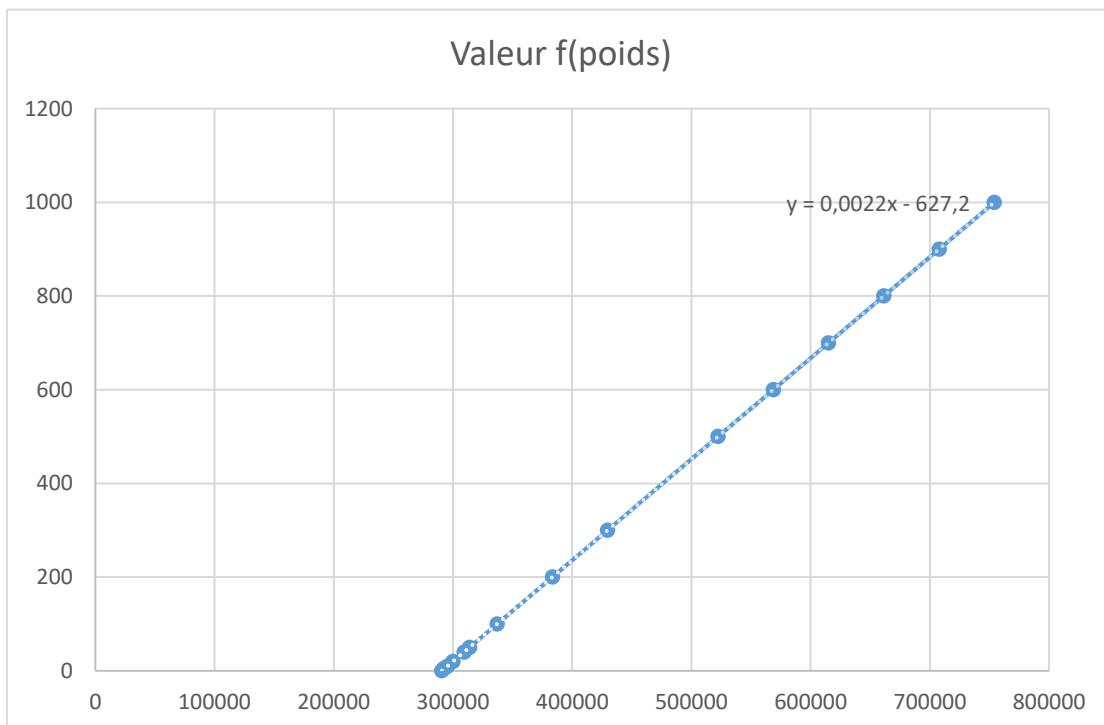


Figure 20 : étalonnage de la jauge de contrainte

On voit directement que la fonction résultante est une droite de la forme $Ax+B$ avec $A = 0.0022$ et $B = -627.2$. Une fois ces valeurs chargées dans l'arduino on peut retrouver le poids que l'on pose sur la ruche dans le port série.

Dans notre cas comme nous avons une maquette imprimée en 3D et une jauge de contrainte de 10Kg. Nous avons décidé de prendre un gain de 128 (gain réglable dans le composant HX711) car nous ne placerons que des masses faibles pour éviter d'endommager la maquette. Cependant même avec une jauge de contrainte de 10Kg nous arrivons à lire des masses de 5g avec une précision de $+/-0.5g$.

Capteur température et humidité DHT11

Pour mesurer la température et l'humidité à l'intérieur de la ruche, il nous a été fourni un capteur de température DHT11. Ce capteur peut mesurer des températures allant de 0°C jusqu'à 50°C avec une précision de $+/-2^{\circ}\text{C}$ et une humidité relative de 20% jusqu'à 90% avec une précision de $+/-5\%$.

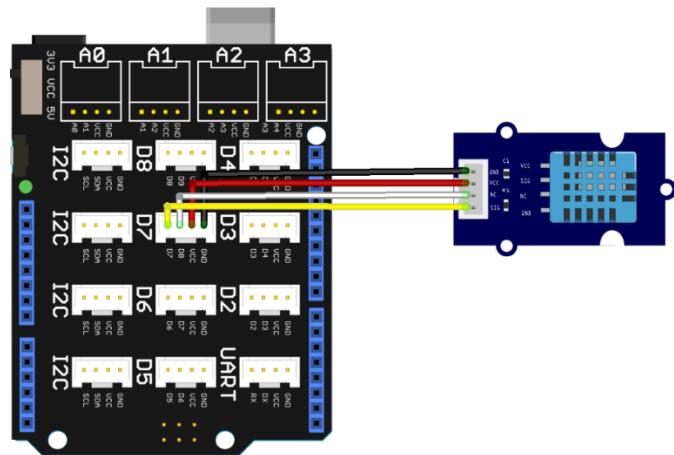


Figure 21 : Schéma électrique du câblage du capteur DHT11

Même si le capteur Grove DHT11 possède un connecteur à quatre pins, seulement trois d'entre eux sont utilisés pour faire fonctionner ce capteur. Ce capteur fonctionne sur un protocole de communication appelé « One wire », il utilise seulement un fil de communication pour recevoir et envoyer des informations au microcontrôleur.

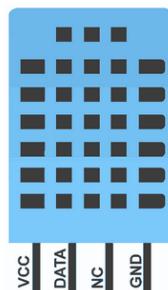


Figure 22 : Pinout du capteur DHT11

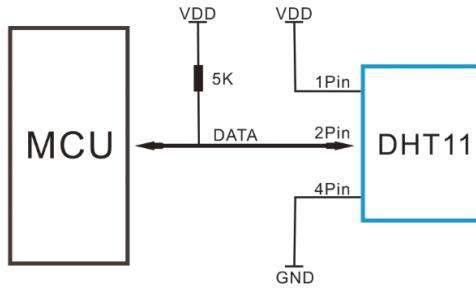


Figure 23 : Câblage recommandé par la datasheet du composant

La résistance de pull-up de $5\text{K}\Omega$ est déjà comprise sur le pcb du module Grove. Ce capteur s'alimente et communique en 5V donc il parfaitement adapter à une utilisation sur un arduino.

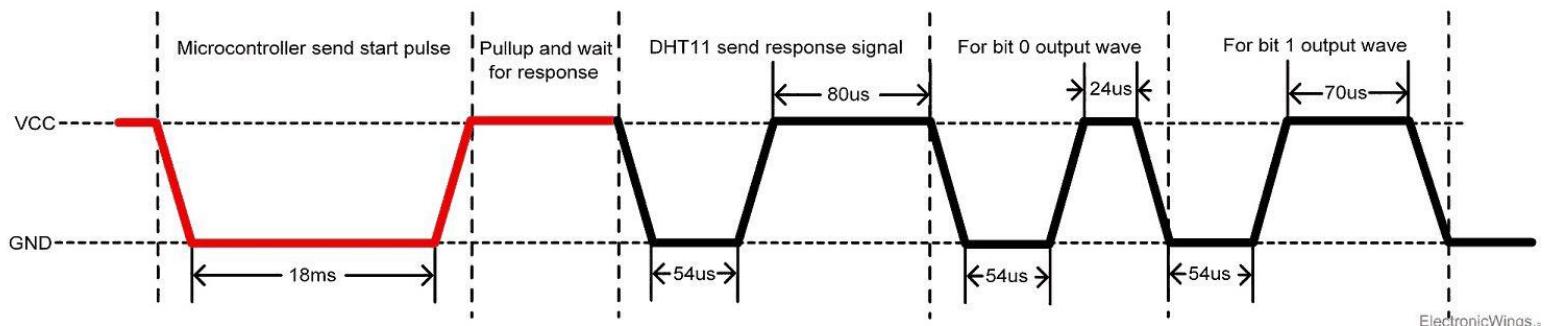


Figure 24 : Communication entre l'arduino et le DHT11 pour faire une demande de mesure

Ce capteur est très simple d'utilisation du point de vue de son protocole de communication. Pour recevoir les informations de température et d'humidité, il suffit que le microcontrôleur passe à l'état bas pendant 18ms la pin de donnée du capteur DHT11. Une fois cela fait le capteur DHT11 va envoyer une impulsion basse de $54\mu\text{s}$ puis haute de $80\mu\text{s}$ pour informer qu'il a bien reçu la roquette et qu'il va envoyer sur son port série les informations d'humidité et puis de température.

Comme dit précédemment le DHT11 redonne les informations de température et d'humidité sous forme d'information sérialisé. Que ce soit un « 0 » ou un « 1 », la trame commence toujours de la même façon. Le capteur DHT11 va passer à l'état bas sa pin « data » pendant $50\mu\text{s}$, puis c'est seulement le temp haut qui va définir si la donnée est un « 1 » ou un « 0 ». Dans le cas d'un « 0 » le temp haut sera entre 26 et $28\mu\text{s}$ et pour un « 1 » ce temps sera de $70\mu\text{s}$.

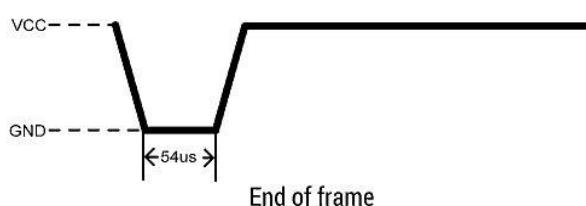


Figure 25 : Fin de communication du capteur DHT11

Pour signaler la fin de la communication, le capteur fait une dernière impulsion basse de $54\mu\text{s}$ et laisse la ligne des datas remonter à l'état haut.

Une fois toutes les trames reçues il nous faut pouvoir les décoder pour retrouver les informations de la température et de l'humidité.

Com single wire two way DHT11				
8bits	8bits	8bits	8bits	8bits
humidite(%) partie entière	humidite(%) partie decimal	Temperature(°C) partie entière	Temperature(°C) partie decimal	CRC = dernier 8bits du calcul : humid entier + humid decimal + temp entier + temp decimal

Figure 26 : Répartition des données

Cette communication série est une communication sur 40bits avec les 16 premiers étant la valeur de l'humidité suivie de 16 bits pour la température et 8 bits de CRC pour vérifier la bonne réception des données.

En soit le système de communication n'est pas très compliqué, cependant la relecture des signaux à la μ s près est beaucoup plus compliqué à mettre en place. L'une des solutions aurait été d'utiliser un timer avec une interruption pour cadencer la relecture des informations. Cependant par manque de temps et de place dans la SRAM du microcontrôleur, nous avons préféré utiliser une librairie déjà toutes faite pour l'utilisation de ce capteur (librairie utilisé "Grove Temperature And Humidity").

```
//ajout des lib utilise
#include "DHT.h" //librairie "Grove Temperature And Humidity" pour l utilisation du capteur DHT11

//definition du pin utilise par le capteur DHT11
#define pin_DHT11 7

//initialisation du pin et du type de capteur utilise
DHT dht(pin_DHT11, DHT11);

void setup() {
    //initialisaiton de la liaison serie via le port usb
    Serial.begin(9600);

    //initialisation de la librairie DHT
    dht.begin();
}

void loop() {
    //variable local
    float humTemp_DHT11[2] = {0};

    //lecture de la valeur d humidite et de temperature
    dht.readTempAndHumidity(humTemp_DHT11);

    //ecriture des valeurs
    Serial.println("");
    Serial.println("Humidity: " + String(humTemp_DHT11[0]) + "%");
    Serial.println("Temperature: " + String(humTemp_DHT11[1]) + "C");

    delay(1000);
}
```

Figure 27 : Code arduino utiliser pour le fonctionnement du capteur DHT11

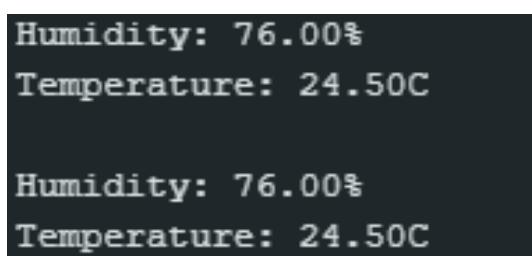


Figure 28 : résultat du code arduino

Ce code vient juste lire la valeur de la température et de l'humidité du capteur DHT11 toutes les secondes.

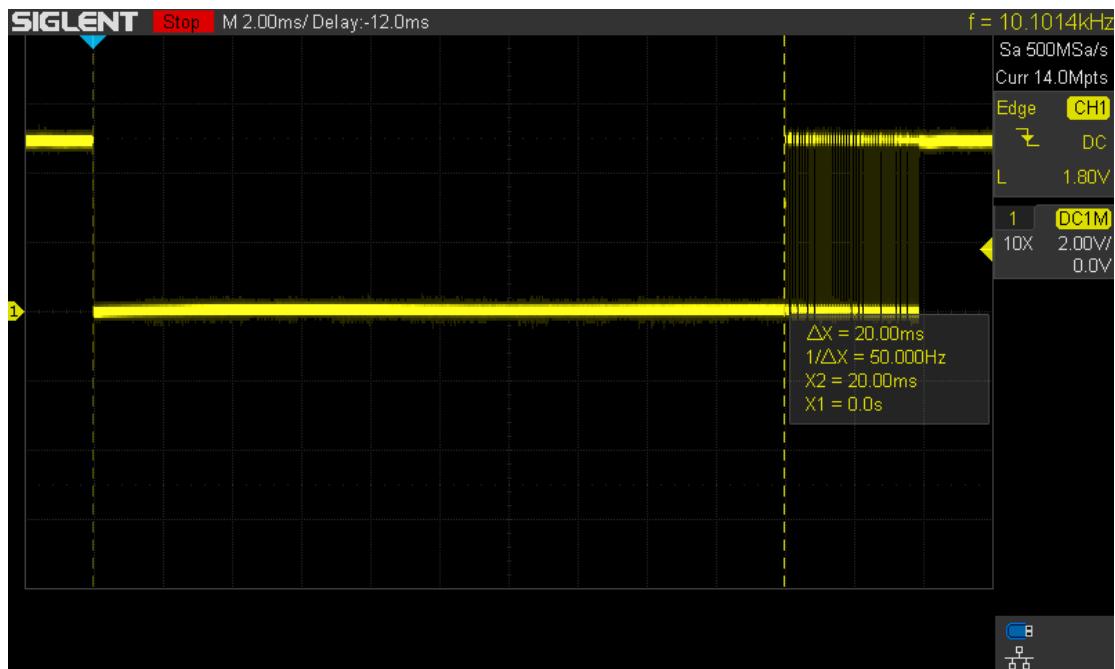


Figure 29 : Vérification de la pulse de 18ms de l'arduino

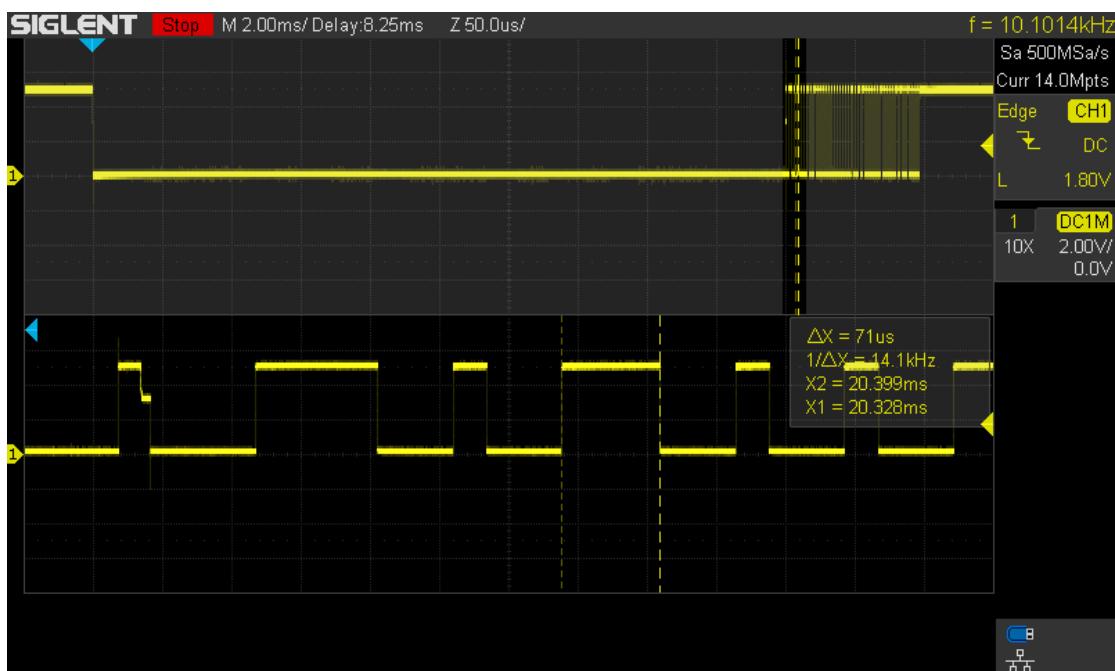


Figure 30 : Ton de la pulse du DHT11 pour un bit « 1 »

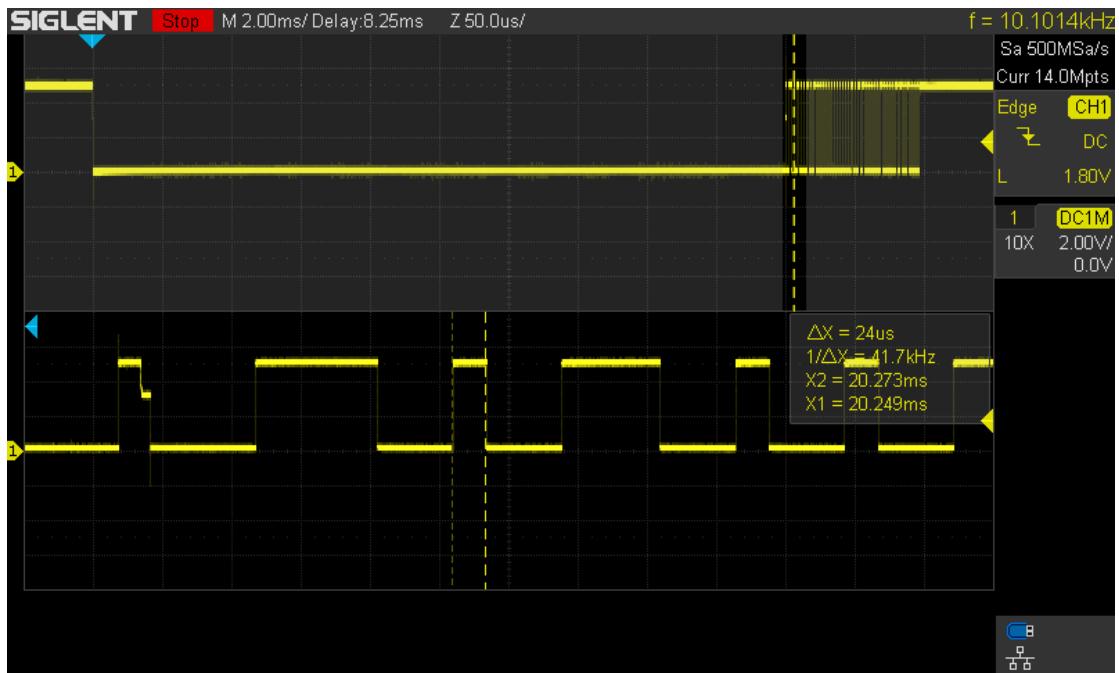


Figure 31 : Ton de la pulse du DHT11 pour un bit « 0 »

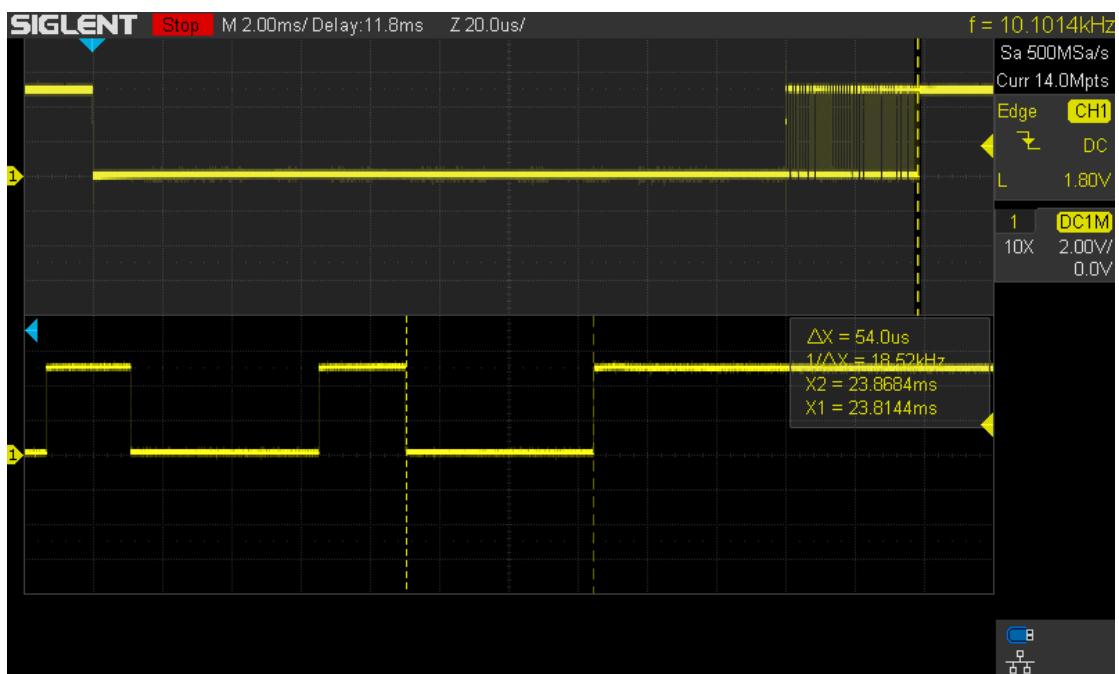


Figure 32 : Toff de la dernière pulse du DHT11 pour un indiquer la fin de la communication

Capteur température et humidité SHT31

Le capteur SHT31 est aussi un capteur de température et d'humidité comme le capteur DHT11. Cependant ce dernier utilise un système de communication bien plus complexe basé sur de l'I2C (« Inter Integrated Circuit ») qui nous permet de régler certain registre à l'intérieur du capteur.

Ce capteur peut mesurer des températures allant de -40°C jusqu'à 125°C avec une précision de +/-0.3°C et une humidité relative de 0% jusqu'à 100% avec une précision de +/-2%.

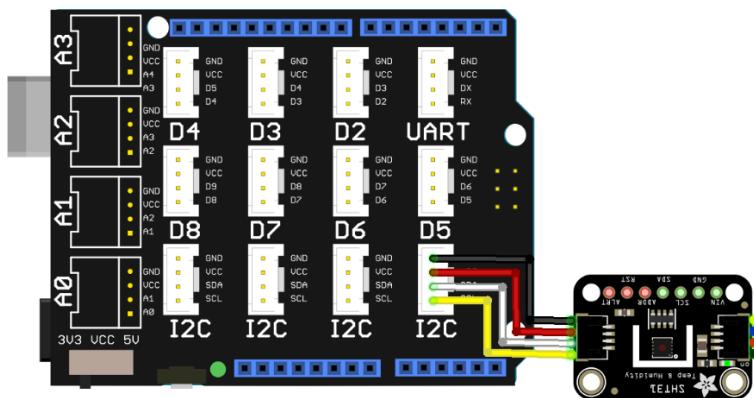


Figure 33 : Câblage électrique du capteur SHT31

Comme le shield grove dispose d'un port I2C nous utiliserons ces ports pour faciliter le câblage des différents appareil I2C. une fois câbler sur le shield Grove les deux ligne I2C « SCL » et « SDA » se retrouve rerouter sur le port « SCL » (pin18) et « SDA » (pin17) de la carte arduino. Il est aussi possible de souder deux pins de la carte grove pour rerouter ces deux fils vers les pins « A4 » et « A5 » de la carte arduino où se trouve la deuxième paire de fils I2C (sachant que l'arduino uno R3 ne possède qu'un seul driver I2C cette deuxième paire I2C est juste mis en parallèle à la première paire placer au pin17 et 18 de la carte arduino).

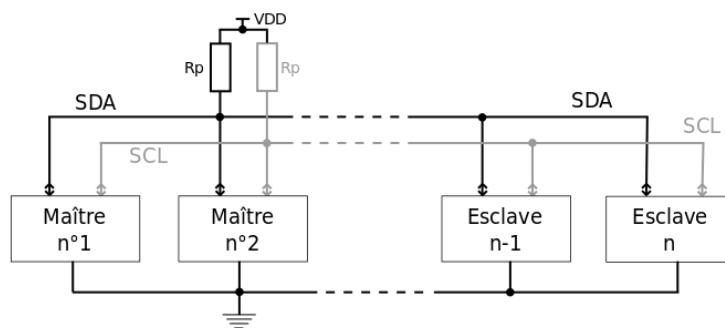


Figure 34 : Exemple de schéma de liaison I2C entre plusieurs maîtres/esclaves

La liaison I2C est une liaison série synchrone bidirectionnel en half-duplex. Cela veut dire que n'importe qui peut parler à n'importe qu'elle autre appareil connecter au réseau I2C

(théoriquement non vrai car un maître ne peut pas discuter avec un autre maître et idem entre esclave) du moment qu'un seul appareil communique en même temps.

Comme tous les maîtres/esclaves sont brancher en parallèle, n'importe qui sur le réseau I2C reçoit les messages des autres qui ne lui sont pas destinée. Pour éviter cela un système d'adressé I2C a été mis en place. Chaque composant de la chaîne I2C a sa propre adresse I2C qui est envoyée en même temps que le message pour savoir qui doit recevoir le message sur la chaîne I2C.

pour exemple le capteur SHT31 à le choix entre deux adresses différentes :

SHT3x-DIS	I2C Address in Hex. representation	Condition
I2C address A	0x44 (default)	ADDR (pin 2) connected to VSS
I2C address B	0x45	ADDR (pin 2) connected to VDD

Figure 35 : différente adresse I2C possible pour le capteur SHT31

Soit l'adresse 0x44 si la pin 2 est reliée à la masse ou l'adresse 0x45 si elle est reliée au +VCC

Une dernière spécificité de la liaison I2C est que seuls les maîtres peuvent discuter avec les esclaves. Les esclaves ont seulement le droit de répondre à leurs maîtres quand le maître leur autorise.

Maintenant pour le fonctionnement d'envoi de commande ou de donnée sur le bus I2C.

En premier il faut envoyer les 7 bits d'adresse + 1 bit pour décrire si on veut écrire ou lire des données dans le registre de l'esclave (0 pour l'écriture et 1 pour la lecture).

Comme sur une liaison I2C il y a 1 bit d'acknowledgment tous les 8 bits. Il faut ajouter ce bit (le maître laisse la ligne SDA retomber au +VCC via les résistances de pull-up et l'esclave doit faire retomber cette ligne pour indiquer qu'il a bien reçu les 8 derniers bits).

ensuite on peut envoyer les XX bits de données et de commandes que l'on voulait envoyer sans oublier le bit acknowledgement tous les 8 bits.

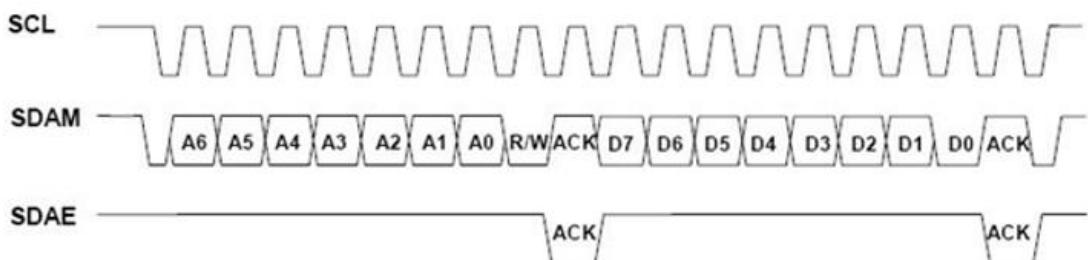


Figure 36 : Exemple de communication I2C

Remarque : SDAM représente les actions du maître sur la ligne SDA. Et la ligne SDAE représente les actions de l'esclave sur la ligne SDA.

Maintenant que nous savons comment fonctionne la liaison I2C nous allons regarder le fonctionnement des registres de la puce SHT31.

Condition		Hex. code	
Repeatability	Clock stretching	MSB	LSB
High	enabled	0x2C	06
Medium			0D
Low			10
High	disabled	0x24	00
Medium			0B
Low			16

Figure 37 : Registre du capteur pour faire une demande de mesure

Pour lancer une simple lecture de température et d'humidité il suffit d'envoyer la commande 0x2400. Ceci signifiera au capteur de faire une lecture de température et d'humidité avec la plus grande précision et sans « Clock stretching » (le Clock stretching viens mettre à 0 la clock de la liaison I2C le temps que le capteur fasse sa convention. Comme nous avons d'autre appareil sur la bus I2C nous avons décidé de ne pas utiliser cette technique).

La trame I2C devra donc ressembler à :

0x44	0	0	0x24	0	0x00	0
Adres se I2c	Commande Write	acknowled ge	8 premiers bits de la commande	acknowled ge	8 derniers bits de la commande	acknowled ge

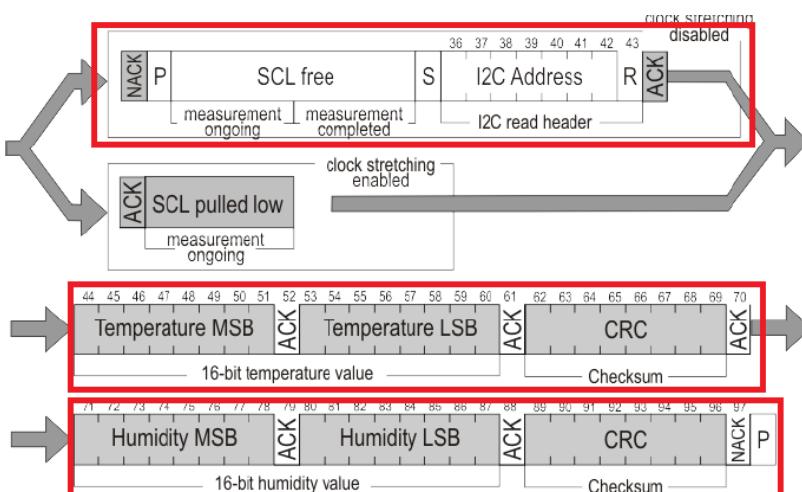


Figure 38 : Réponse du capteur sur le bus I2C

Pour réceptionner il suffira de faire une demande de lecture sur l'adresse I2C du capteur SHT31. Celui-ci nous rendra ensuite un message sur 48 comportant la température sur 16 bits suivie d'un CRC sur 8bits pour la température, puis de nouveau un message de 16 bits pour l'humidité avec le CRC de l'humidité sur 8 bits aussi.



Figure 39 : Demande de lecture de la température et l'humidité

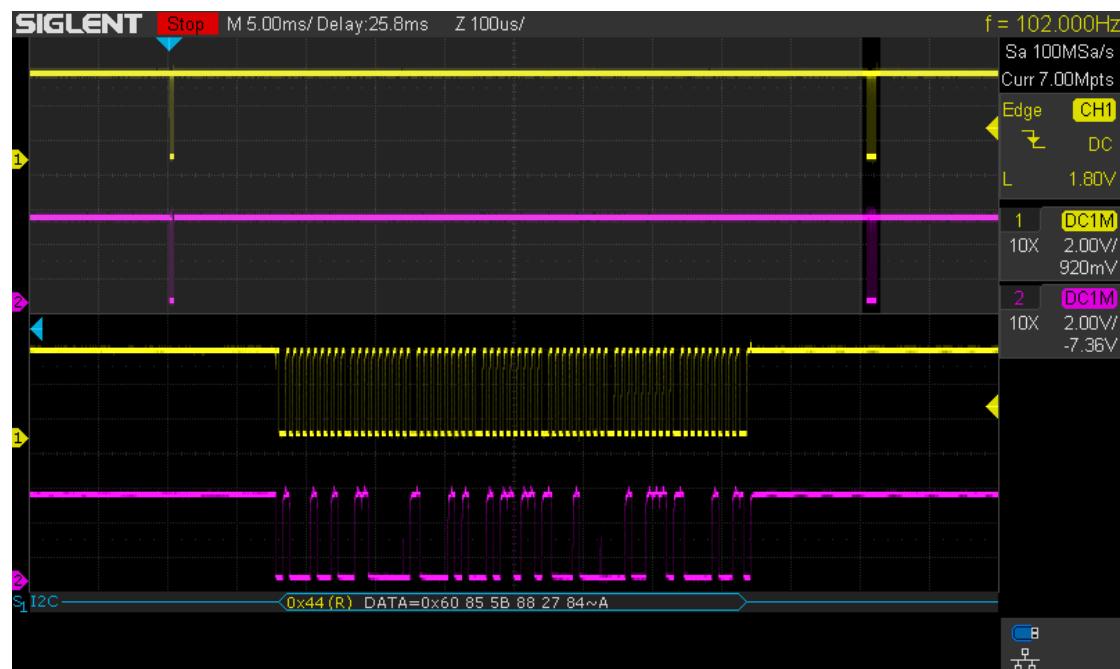


Figure 40 : Réponse du capteur SHT31

Ch1 : représente la clock « SCL », Ch2 : représente les données « SDA »

Maintenant il nous reste à calculer les valeurs de température et d'humidité selon les données que le capteur nous a retourné. Selon la datasheet :

$$T^{\circ}C = -45 + 175 \times \frac{Val_{I2C}T}{2^{16} - 1}$$

$$Humidité relative = 100 \times \frac{Val_{I2C}RH}{2^{16} - 1}$$

Soit dans notre exemple :

Val_I2C de la température = 0x6085 = 24709

Val_I2C de l'humidité = 0x8827 = 34855

$$T^{\circ}C = -45 + 175 \times \frac{24709}{2^{16} - 1} = 20.98^{\circ}C$$

$$Humidité relative = 100 \times \frac{34855}{2^{16} - 1} = 53.18\%$$

Pour finir nous ne rentrerons pas trop dans les détails, mais le calcul du CRC à étais mis en place dans le code arduino pour vérifier que les valeurs lues sur le port I2C correspondait bien aux bonnes valeurs ou non. Si le CRC reçu est bien identique au CRC recalculé alors on stocke les valeurs en mémoire de l'arduino, sinon on ne stocke pas la valeur et on monte un drapeau pour indiquer que la valeur n'est pas bonne.

Property	Value
Name	CRC-8
Width	8 bit
Protected data	read and/or write data
Polynomial	$0x31 (x^8 + x^5 + x^4 + 1)$
Initialization	0xFF
Reflect input	False
Reflect output	False
Final XOR	0x00
Examples	CRC (0xBEEF) = 0x92

Figure 41 : Paramètre de la datasheet pour recalculer le CRC

```

uint8_t sht31_crc8(uint16_t data)
{
    /*
     * Fonction de calcul du CRC du capteur SHT31
     *
     * Args:
     * data (uint16_t): data sur laquel on veut calculer le crc
     */
    //variable local
    const uint8_t pol =0x31; //polynome du crc utilise par la SHT31
    uint8_t crc= 0xFF; //valeur initial du crc

    //decoupage en deux du calcul du crc
    for (int j=0; j<2; j++)
    {
        if(j==0){
            crc = crc ^ data>>8; //prise en compte que des 8 premiers bits
        }
        else{
            crc = crc ^ data & 0xFF; //prise en compte que des 8 derniers bits
        }

        //decalage binaire et xor du polynome
        for (int i=0; i<8; i++)
        {
            if(crc & 0x80) //si msb == 1
            {
                crc = (crc << 1) ^ pol;
            }
            else
            {
                crc = crc << 1;
            }
        }
    }
    return crc;
}

```

Figure 42 : Fonction arduino créer pour recalculer le CRC

```

//initialisation des libs
#include <Wire.h>

//declaration des definition
#define addr_SHT31 0x44

void setup() {
  Serial.begin(9600);
  Wire.begin();
}

void loop() {
  //variable local
  float temp_SHT31 = 0;
  float hum_SHT31 = 0;
  uint8_t Error = 0;

  sht31_lectureTempEtHumid(addr_SHT31, &temp_SHT31, &hum_SHT31, &Error);

  if(Error == 0)
  {
    Serial.println("");
    Serial.println("Temperature :" + String(temp_SHT31) + "C");
    Serial.println("Humidite :" + String(hum_SHT31) + "%");
  }
  else
  {
    Serial.println("");
    Serial.println(" /\\" ERREUR LORS DE LA LECTURE DES VALEURS /\\"");
  }

  delay(1000);
}

//-----
//-----Fonction-----
void sht31_lectureTempEtHumid(int addrI2C_SHT31, float *temp_SHT31, float *hum_SHT31, uint8_t *Error)
{
  /*
  Fonction de recuperation de la temperature et de l humidite du capteur SHT31 via I2C

  Args:
  addrI2C_SHT31 (int): adresse I2C du capteur SHT31
  temp_SHT31 (float): pointeur de la valeur de la temperature du capteur SHT31
  hum_SHT31 (float): pointeur de la valeur de la humidite du capteur SHT31
  Error (uint8_t): pointeur de variable d erreur. si True alors erreur lors de la reception des information du capteur
  */

  //variable local
  //valeur binaire rendu par le capteur SHT31
  uint16_t tempbin_SHT31 = 0;
  uint16_t crcTempRecu_SHT31 = 0;
  uint16_t humbin_SHT31 = 0;
  uint16_t crcHumRecu_SHT31 = 0;

  //pointeur
  //valeur de temperature et humid convertie en °c et %
  *temp_SHT31 = 0;
  *hum_SHT31 = 0;
  //variable d erreur
  *Error = 0;

  //demande de mesure de temperature et d humid
  Wire.beginTransmission(addrI2C_SHT31);
  Wire.write(0x24);
  Wire.write(0x00);
  Wire.endTransmission();

  //delais de mesure
  delay(50); //delay min 13ms

  //lecture mesure
  Wire.requestFrom(addrI2C_SHT31, 6);
  if(Wire.available() >=6)
  {
    tempbin_SHT31 = Wire.read() << 8 | Wire.read();
    crcTempRecu_SHT31 = Wire.read();
    humbin_SHT31 = Wire.read() << 8 | Wire.read();
    crcHumRecu_SHT31 = Wire.read();
  }
  else
  {
    *Error = 1;
  }

  //verification des CRC
  if(crcTempRecu_SHT31 == sht31_crc8(tempbin_SHT31) && crcHumRecu_SHT31 == sht31_crc8(humbin_SHT31) && Error !=1)
  {
    *temp_SHT31 = -45+175*(tempbin_SHT31/65535.0); //calcul de la temperature
    *hum_SHT31 = 100*(humbin_SHT31/65535.0); //calcul de l humidite
  }
  else
  {
    *Error = 1;
  }
}

int8_t sht31_crc8(uint16_t data)
{
  /*
  Fonction de calcul du CRC du capteur SHT31

  Args:
  data (uint16_t): data sur laquel on veut calculer le crc
  */

  //variable local
  const uint8_t pol = 0x31; //polynome du crc utilise par la SHT31
  uint16_t crc = 0xFF; //valeur initial du crc

  //decoupage en deux du calcul du crc
  for (int j=0; j<2; j++)
  {
    if(j==0){
      crc = crc ^ data>>8; //prise en compte que des 8 premiers bits
    }
    else{
      crc = crc ^ data & 0xFF; //prise en compte que des 8 derniers bits
    }

    //decalage binaire et xor du polynome
    for (int i=0; i<8; i++)
    {
      if(crc & 0x80) //si msb == 1
      {
        crc = (crc << 1) ^ pol;
      }
      else
      {
        crc = crc << 1;
      }
    }
  }
  return crc;
}

```

Figure 43 : Programme de lecture des valeurs du Capteur SHT31

Ecran LCD 16x2

Dans cette partie nous regarderons la gestion de l'écran LCD qui nous permet d'afficher toutes les valeurs en direct sur la ruche.

Cet écran est l'écran LCD monochrome de chez Grove, il possède un affichage de 16*2 caractères et fonctionne en I2C. malheureusement comme Grove ne donne pas de datasheet ou d'information sur les registres utiliser par cette écran appart une librairie arduino. Nous sommes obligées d'utiliser leurs librairies I2C sans pouvoir faire d'analyse plus poussé.

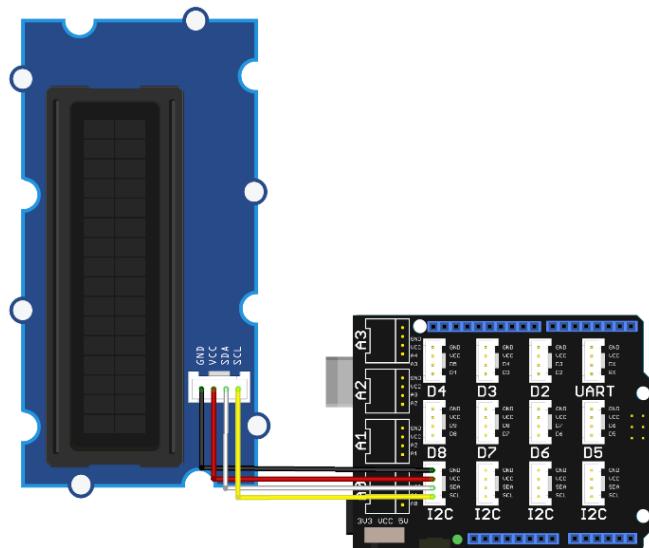


Figure 44 : Schéma de câblage de l'écran à la carte arduino

```
#include "rgb_lcd.h" //librairie "Grove-LCD RGB Backlight" pour l utilisation du l ecran LCD grove
rgb_lcd lcd; //initialisation de la librairie pour l ecran LCD

void setup() {
  lcd.begin(16, 2); //initialisation de l ecran LCD avec 16 colonnes et 2 lignes
}

void loop() {
  lcd.setCursor(0, 0); //place le curseur sur le premier caractere de la premiere ligne
  lcd.print("BE Projet :"); //ecriture d un message

  lcd.setCursor(0, 1); //place le curseur sur le premier caractere de la deuxieme ligne
  lcd.print("Ruche connectee"); //ecriture d un message
}
```

Figure 45 : Code arduino pour l'utilisation de l'écran LCD

Remarque : nous avons utilisé la librairies "Grove-LCD RGB Backlight" pour la gestion de l'écran LCD.



Figure 46 : résultat sur l'affichage de la ruche

Emetteur récepteur LoRa

Pour cette partie nous avions le choix entre 3 modules LoRa différents :

Nom	Grove LoRa Radio 433Mhz	Grove LoRa Radio 868Mhz	Grove LoRa-E5
Fréquence	433Mhz	868Mhz	868Mhz/915Mhz
Modulation	LoRa®, (G)FSK, (G)MSK and BPSK	LoRa®, (G)FSK, (G)MSK and BPSK	LoRa®, (G)FSK, (G)MSK and BPSK
Puissance RF	20dBm (100mW)	20dBm (100mW)	20dBm (100mW)
Interface	UART	UART	UART
Tension alimentation	5V/3.3V	5V/3.3V	5V/3.3V
Autre			Ajout d'un PHY Ethernet pour pouvoir utiliser le réseau LoRaWAN

Nous, nous sommes rendus compte que les modules « Grove LoRa Radio 433Mhz » et « LoRa Radio 433Mhz » étaient identiques car il utilisait le même module en interne, le module LoRa « RFM98 ». Ce module LoRa est un module très utilisés qui fonctionne en SPI, cependant Grove a rajouté un microcontrôleur AVR sur leurs PCB pour pouvoir convertir les signaux UART que l'on envoie de l'Arduino en signaux SPI. Cependant comme pour leurs écrans ils ont fourni aucune datasheet et surtout aucune information sur leurs façons de transformer les commandes SPI en UART. Ceci nous empêche d'utiliser ces deux modules.

autrement qu'avec leurs librairies arduino. Cependant comme il nous faut utiliser un module LoRa avec notre rasapberry pi pour pouvoir réceptionner les informations de la ruche, ceci nous empêche d'utiliser ces deux modules jusqu'à ce que Grove crée une librairie pour raspeberry pi ou donne une datasheet détailler des commandes à envoyer en UART.

Cependant si nous voulions vraiment utiliser le module LoRa « RFM98 » il est totalement possible d'utiliser une équivalence aux deux modules Grove sans leurs microcontrôleur AVR et rentrer directement les informations en SPI du module « RFM98 » dans le raspeberry pi.

Cependant comme nous n'avions pas de tel module, nous sommes passé sur l'utilisation du module Grove LoRa-E5 qui lui ne comporte que la puce LoRa-E5 car elle est totalement autonome avec sa propre liaison UART en entrée.

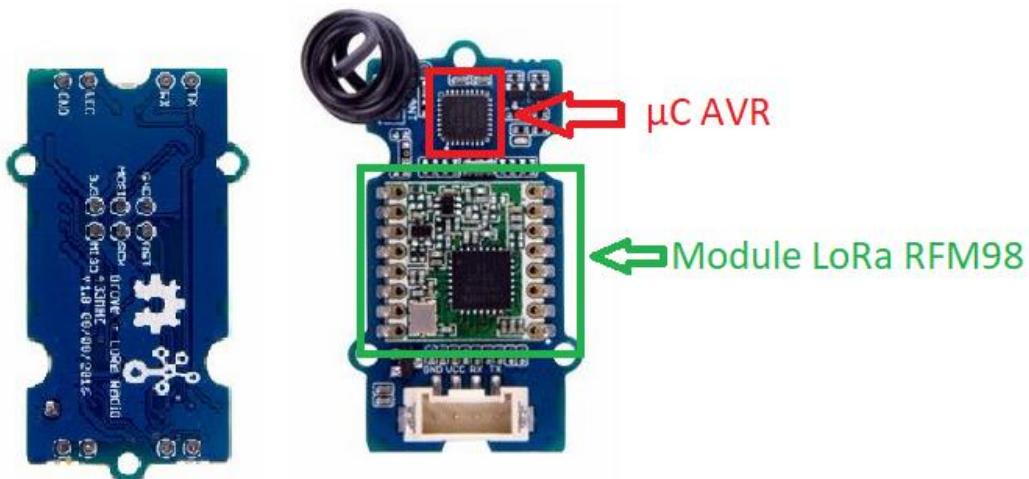


Figure 47 : Recto/Verso des module Grove LoRA Radio 433/868Mhz



Figure 48 : Recto/Verso du module Grove Lora-E5

Cependant que l'on utilise le Module LoRa-E5 ou le module Lora 433/86Mhz cela ne change strictement rien sur leurs câblages dans la ruche.

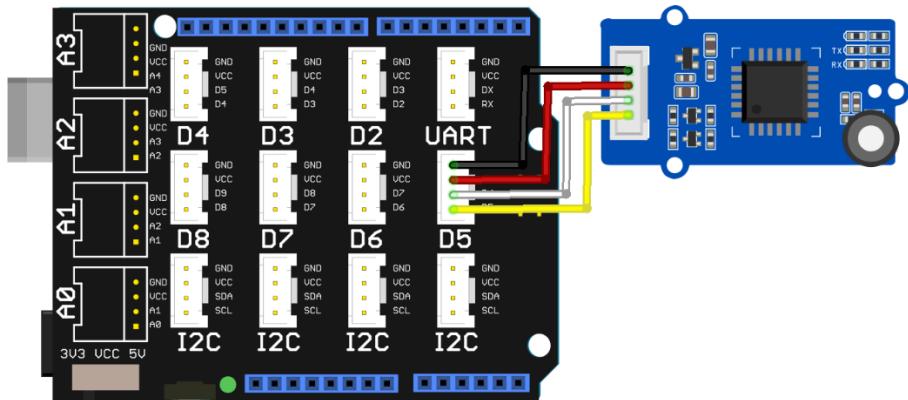


Figure 49 : Cablage du module LoRa sur la carte arduino

Comme la carte arduino ne possède qu'un seul port série UART utilisé pour la programmation ainsi que la liaison série. Nous avons décidé de câbler le module LoRa sur deux autres pins de la carte arduino et nous utilisons la librairie « SoftwareSerial.h » inclue dans arduino pour pouvoir créer un port série virtuel sur n'importe quel pin du microcontrôleur arduino.

Malgré le fait que nous ne pouvions pas utiliser les deux modules Grove LoRA Radio 433/868Mhz, nous avions quand-même fait un code de test pour vérifier leurs bons fonctionnements. Comme nous ne pouvons pas utiliser ces modules en dehors de l'écosystème arduino, notre code utilise deux arduino avec un arduino qui sera un émetteur et un autre qui sera utilisé en tant que récepteur.

```

//ajout des librairies utile
#include <SoftwareSerial.h> //librairie pour la creation du port serie virtuel
#include <RH_RF95.h> //librairie utiliser pour communiquer avec le module LoRa (nom lib: Grove - LoRa Radio 433Mhz 868Mhz)

//reglage des pins du module LoRa sur la carte arduino
#define rxPin 5
#define txPin 6
SoftwareSerial LoRaPin (rxPin, txPin); //creation d un port serie virtuel car la carte arduino ne possede pas assez de port serie
RH_RF95 rf95(LoRaPin); //envoie du port serie virtuel a la librairie du module loRa RH_RF95 lib

//reglage des parametres de la ruche
#define codeSystème 0x2C33 //cle utiliser pour reconnaître notre message parmis tous les autres message a la frequence 868/433Mhz
#define codeRuche 0x0 //indice de la ruche
#define SHT31_Temp 0x0 //Code du capteur SHT31 en mode temperature
#define SHT31_Humid 0x1 //Code du capteur SHT31 en mode humidite
#define DHT11_Temp 0x2 //Code du capteur DHT11 en mode temperature
#define DHT11_Humid 0x3 //Code du capteur DHT11 en mode humidite
#define HC_SR04 0x4 //Code du capteur ultrason HC-SR04
#define Masse 0x5 //Code du capteur de masse

void setup() {
    Serial.begin(9600); //rule la frequence de communication du port serie de l arduino pour communiquer avec l ordinateur

    //initialisation du module LoRa
    if (!rf95.init())
    {
        Serial.println("Erreur lors de l initialisation du module LoRa"); //message d erreur
        while(1);
    }
    Serial.println("Module LoRa initialisee"); //message de confirmation du bon fonctionnement du module LoRa

    //reglage des differents parametre de l emeteur
    rf95.setTxPower(5, false); //rule la puissance d emission, va de 5 a 20dBm
    rf95.setModemConfig(1); //modifie le type d emission "Bw500Cr45Sf128" ie: Bw = 500 kHz, Cr = 4/5, Sf = 128chips/symbol, CRC on. Fast+short range
    rf95.setFrequency(868.0); //rule la frequence du module LoRa (ecrire ici 434 pour le module 434Mhz)
}

void loop() {
    envoieLoRa868_433(codeSystème, codeRuche, SHT31_Humid, 0xAA); //fonction d emission des donnees
    delay(5000); //delais d attente pour ne pas trop poluer les frequences radio le temp des tests
}

void envoieLoRa868_433(uint16_t cleSystème, uint8_t idRuche, uint8_t typeCapteur, uint16_t valeurCapteur)
{
    /*
    Fonction d emission d une valeur lue par l un des capteurs de la ruche

    Args:
        cleSystème (uint16_t): cle utilisee pour dissocier les messages du projet ruche des autres messages sur la frequence 868/433Mhz
        idRuche (uint8_t): numerotation de la ruche, utiliser pour dissocier plusieurs ruche entre elle sur la meme frequence
        typeCapteur (uint8_t): numerotation du capteur utiliser ie savoir si c est un capteur de temperature ou de poid etc...
        valeurCapteur (uint16_t): valeur lu par le capteur
    */

    //Creation et remplissage du buffer d emission
    uint8_t data[6];
    data[0] = cleSystème >>8;
    data[1] = cleSystème & 0xFF;
    data[2] = idRuche;
    data[3] = typeCapteur;
    data[4] = valeurCapteur>>8;
    data[5] = valeurCapteur & 0xFF;

    rf95.send(data, sizeof(data)); //envoie du message via le module LoRa
    rf95.waitPacketSent(); //fonction d attente de la fin d emission du module LoRa
}

```

Figure 50 : Code arduino émetteur pour le module Grove LoRA Radio 433/868Mhz

```

//ajout des librairies utile
#include <SoftwareSerial.h> //librairie pour la creation du port serie virtuel
#include <RH_RF95.h> //librairie utiliser pour communiquer avec le module LoRa (nom lib: Grove - LoRa Radio 433Mhz 868Mhz)

//reglage des pins du module LoRa sur la carte arduino
#define rxPin 5
#define txPin 6
SoftwareSerial LoRaPin (rxPin, txPin); //creation d un port serie virtuel car la carte arduino ne possede pas assez de port serie
RH_RF95 rf95(LoRaPin); //envoie du port serie virtuel a la librairie du module loRa RH_RF95 lib

//reglage des parametre de la ruche
#define cleSysteme_Valide 0x2C33

void setup()
{
    Serial.begin(9600); //regle la frequence de communication du port serie de l arduino pour communiquer avec l ordinateur

    //initialisation du module LoRa
    if (!rf95.init())
    {
        Serial.println("Erreur lors de l initialisation du module LoRa"); //message d erreur
        while(1);
    }
    Serial.println("Module LoRa initialisee"); //message de confirmation du bon fonctionnement du module LoRa

    //reglage des differents parametre de l emetteur
    rf95.setModemConfig(1); //modifie le type de reception "Bw500Cr45Sf128" ie: Bw = 500 kHz, Cr = 4/5, Sf = 128chips/symbol, CRC on, Fast+short range
    rf95.setFrequency(868.0); //regle la frequence du module LoRa (ecrire ici 434 pour le module 434Mhz)
}

void loop()
{
    //variable local utilisee
    uint16_t cleSysteme =0;
    uint8_t idRuche =0;
    uint8_t typeCapteur =0;
    uint16_t valeurCapteur =0;

    receptionLoRa868_433(10000, &cleSysteme, &idRuche, &typeCapteur, &valeurCapteur); //fonction de reception des informations

    if(cleSysteme == cleSysteme_Valide) //verification de la cle emise par la ruche
    {
        //affichage de toutes les valeurs envoyee par la ruche
        Serial.println("Nouveau message recu: ");
        Serial.print("cleSysteme: ");
        Serial.println(cleSysteme);
        Serial.print("idRuche: ");
        Serial.println(idRuche);
        Serial.print("typeCapteur: ");
        Serial.println(typeCapteur);
        Serial.print("valeurCapteur: ");
        Serial.println(valeurCapteur);
        Serial.println("");
    }
    else
    {
        //affichage d un message d erreur
        Serial.println("Mauvaise cle recu");
        Serial.println("");
    }
}

void receptionLoRa868_433(uint16_t timeOut, uint16_t *cleSysteme, uint8_t *idRuche, uint8_t *typeCapteur, uint16_t *valeurCapteur)
{
    /*
        Fonction de reception et de conditionnement des information envoyee par la ruche

        Args:
            timeOut (uint16_t): temps d attente de la reception d un message LoRa
            cleSysteme (uint16_t): cle utilisee pour dissocier les messages du projet ruche des autres messages sur la frequence 868/433Mhz
            idRuche (uint8_t): numerotation de la ruche, utiliser pour dissocier plusieurs ruches entre elles sur la meme frequence
            typeCapteur (uint8_t): numerotation du capteur utilisee pour savoir si c est un capteur de temperature ou de poid etc...
            valeurCapteur (uint16_t): valeur lu par le capteur
    */

    //creation d un buffer de reception des messages
    uint8_t buf[RH_RF95_MAX_MESSAGE_LEN];
    uint8_t len = sizeof(buf);

    rf95.waitForAvailableTimeOut(timeOut); //attend qu'un message soit disponible jusqu'au timeOut

    if(rf95.recv(buf, &len)) //si un message est disponible dans le buffer
    {
        if(len == 6) //si le message fait bien 6 octets
        {
            //recuperation des differentes information dans le bon format
            *cleSysteme = buf[0]<<8 | buf[1];
            *idRuche = buf[2];
            *typeCapteur = buf[3];
            *valeurCapteur = buf[4]<<8 | buf[5];
        }
    }
}

```

Figure 51 : Code arduino recepteur module Grove LoRA Radio 433/868Mhz

Remarque : ces deux codes utilisent tous les deux la librairie « Grove - LoRa Radio 433Mhz 868Mhz »

Tous les codes de la liaison LoRa que ce soit les deux codes précédents pour l'exemple d'utilisation des modules Grove LoRA Radio 433/868Mhz. Ou les codes arduino et raspberry pi qui utilise le module Grove Lora-E5 utilise la messagerie de communication LoRa créer pour ce projet.

Clé système		Indice de la ruche	Type de capteur	Valeur Capteur		
2 octets		1 octet	1 octet	4 octets		
Clé système		Indice de la ruche	Type de capteur	Valeur capteur		
cle utiliser: 0x2C33		ruche de test: 0x0	SHT31 (Temp): 0x0		ieee 754	
Msb	Lsb	ruche du projet: 0x1	SHT31 (Humid): 0x1	Msb		Lsb
			DHT11 (Temp): 0x2			
			DHT11 (humid): 0x3			
			PIR (prox): 0x4			
			HX711 (Masse): 0x5			
			Vbat (tension batterie): 0x6			

Figure 52 : Messagerie LoRa créer pour ce projet

Cette messagerie à était créée pour que plusieurs ruches avec chaque une des capteurs différents puisse discuter avec le raspberry pi et que les informations soit parfaitement bien affiché sur la page web.

La messagerie se sépare en 4 grande parties. La première est la « clé système » qui est codé en dur dans tous les programmes utilisant un module LoRa. Cette clé est utilisée pour reconnaître notre infrastructure LoRa sans fil des autre émetteur radio à proximité. Ceci nous permet d'être sûr de décoder les bonnes informations. La deuxième partie est « l'indice de la ruche », cet indice nous permet de savoir qu'elle ruche envoie les informations. La troisièmes partie et le type de capteur chaque information de chaque capteur à un indice différent ce qui nous permet de savoir qu'elle information de qu'elle capteur nous recevons. La dernière partie la plus importante sont les informations envoyer par la ruche. Ces informations sont codées sous 4 octets et plus précisément en IEEE-754 simple précision, ce qui nous permet de pouvoir envoyer des informations sur la plage $+-6.8 \times 10^{38}$. Au début nous encodions simplement les informations en binaires sur 16 bits, mais nous avions eu des problèmes quand nous avions voulu envoyer des nombres négatifs donc nous sommes passé en IEE 754 simple précision.

Figure 53 : Code arduino de l'émission avec le module Grove LoRa-E5

Figure 54 : Code arduino de la réception avec le module Grove LoRa-E5

Figure 55 : Code python de la réception avec le module Grove LoRa-E5 pour le raspberry pi
BENZ Antoine - HEUDI Ineo

Figure 56 : Code python de l'émission avec le module Grove LoRa-E5 pour le raspberry pi

Même si théoriquement les codes de réception LoRa-E5 pour arduino et émission LoRa-E5 pour le raspberry pi ne sont pas utilisé. Ils ont quand même été créer dans l'éventualité ou des groupes de BE voudrais les utiliser.

Exemple d'émission avec l'arduino et réception des message LoRa avec le Raspberry pi

```
lancement de l application

+INFO: Input timeout
+MODE: TEST

+INFO: Input timeout
+TEST: STOP

+INFO: Input timeout
+LW: LDRO, ON

+INFO: Input timeout
+TEST: RFCFG F:868000000, SF12, BW125K, TXPR:12, RXPR:15, POW:14dBm, CRC:ON, IQ:OFF, NET:OFF

initialisation du module LoRa OK

+INFO: Input timeout
+TEST: TXLRPKT "2C330101C7F1207E"
+TEST: TX DONE
```

Figure 57 : Emission LoRa du coté arduino

```
pi@raspberrypi:~ $ cd FTP/files/python/
pi@raspberrypi:~/FTP/files/python $ python3 TxLoRa-E5.py

+INFO: Input timeout
+MODE: TEST

+INFO: Input timeout
+TEST: STOP

+INFO: Input timeout
+LW: LDRO, ON

+INFO: Input timeout
+TEST: RFCFG F:868000000, SF12, BW125K, TXPR:12, RXPR:15, POW:14dBm, CRC:ON, IQ:OFF, NET:OFF

initialisation du module LoRa OK

+INFO: Input timeout
+TEST: TXLRPKT "2C330001C7F1207E"
+TEST: TX DONE

+INFO: Input timeout
+TEST: TXLRPKT "2C330001C7F1207E"
+TEST: TX DONE
```

Figure 58 : Réception LoRa du coté Raspberry pi

Pour le moment le programme de réception est juste un programme de test. Le programme final sur le raspberry pi permet de stocker les informations dans une base de données pour pouvoir les afficher par la suite sur la page web.

4.36.5 TX LoRa Packet

After enter test mode, user could send LoRa packet through "AT+TEST=TXLRPKT" sub-command.

The command format is like below:

AT+TEST=TXLRPKT, "HEX STRING"

Command sequence to send LoRa packet:

```
// Set test mode
AT+MODE=TEST
// Query test mode, check RF configuration
AT+TEST=?
// Set RF Configuration
AT+TEST=RFCFG,[FREQUENCY],[SF],[BANDWIDTH],[TXPR],[RXPR],[POW],[CRC],[IQ],[NET]
// Send HEX format packet
AT+TEST=TXLRPKT, "HEX String"
eg:AT+TEST=TXLRPKT, "00 AA 11 BB 22 CC"
// Send TEXT format packet
AT+TEST=TXLRSTR, "TEXT"
eg:AT+TEST=TXLRSTR, "LoRaWAN Modem"

Return:
+TEST: TXLRPKT "404EA99000800A00089F6E770959"
+TEST: TXLRSTR "LoRaWAN Modem"
+TEST: TX DONE
```

Figure 59 : Exemple de commande utiliser pour mettre le module LoRa en mode émission

Le module LoRa-E5 fonctionne par commande AT envoyer via le port UART. Les commande AT sont des commandes en chaine de caractère utiliser pour paramétrer et communiquer avec des appareilles. Dans cet extrait de la datasheet on montre un exemple pour régler le module LoRa pour envoyer des informations avec la modulation LoRa sans utiliser le réseau LoRaWan (réseau ouvert dédier à l'utilisation de module LoRa).

En effet le plus grand avantage du module LoRa-E5 est l'intégration d'un PHY ethernet qui lui permet de se connecter au réseau LoRaWan. Le but du projet était de se connecter à ce réseau et de ressortir les informations de la ruche sur internet via une plateforme du type « thethingsnetwork » cependant nous, nous sommes très vite rendu compte que la couverture des balises LoRaWan en France étais bien moins rependu que ce que l'on imaginait.



Figure 60 : nombre de balise LoRa au l'alentours de Toulouse

Quand nous avions fait nos tests préliminaires il y à quelque mois de la connexion au réseau LoRaWan il n'y avait que 3 balises sur toutes la périphérie de Toulouse. Cependant comme il est très facile de créer sa propre balise LoRa et de la rajouter au réseau LoRaWan d'autre balise se sont rajouter au réseau.

Ceci explique pourquoi nous avons préféré rester sur une typologie peer to peer (pair-à-pair) sur notre connexion LoRa.

Cependant que l'on utilise le réseau LoRaWan ou non, la modulation RF du module LoRa-E5 reste inchangé. Cette modulation est une modulation très spécifique au LoRa qui s'appelle « Chirp Spread Spectrum ». Cette modulation est la clé de la popularité des modules LoRa car elle est extrêmement robuste et au module LoRa de traverser de grande distance sans perte de donnée.

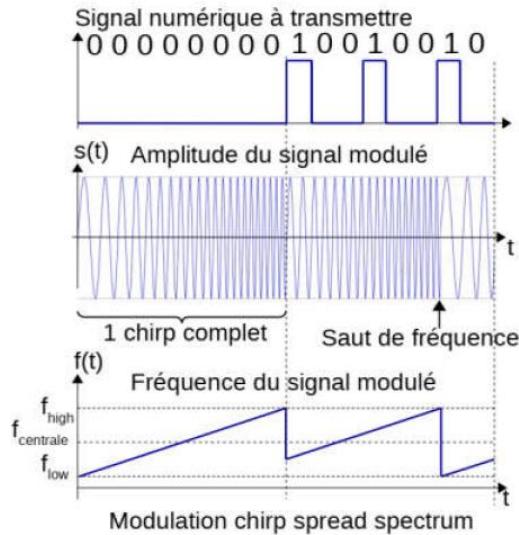


Figure 61 : exemple de la modulation Chirp Spread Spectrum

La Modulation Chirp Spread Spectrum fonctionne par paquet de « Chirp », sachant qu'un Chirp représente 8 bits d'informations. Le signal utilisé pour la modulation est un signal avec une fréquence toujours croissante entre un f_{low} et un f_{high} . A chaque émission de chirp on calcule la valeur décimale des 8bits à envoyer et on fait un rapport entre cette valeur et la fréquence f_{high} . Cette nouvelle fréquence sera la fréquence de début d'émission du nouveau chirp. Bien-sûr durant l'émission du chirp on continue de faire croître la fréquence jusqu'à atteindre la f_{high} . A ce moment fait passer la fréquence de modulation à f_{low} tous en continuant d'augmenter avec le temps la fréquence de modulation jusqu'à la fin du chirp. Une fois fini on reprend la même séquence pour le chirp suivant.

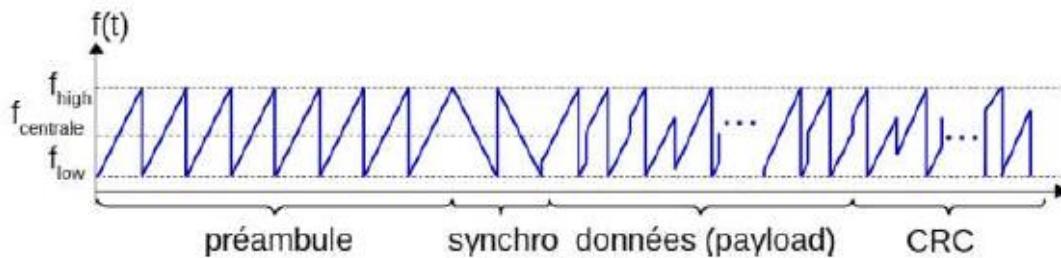


Figure 62 : Trame LoRa

Les trames LoRa sont toujours constituées de la façon suivante.
un préambule de 8 chirps, 2 chirp inversé de synchronisation puis enfin la payload suivie d'un CRC si celui-ci a été activé dans les paramètres de l'émetteur.

Pour rendre la modulation LoRa encore plus robuste il est possible de diminuer le temps d'un chirp et donc d'augmenter les changes que le message soit reçu par le récepteur. Cependant en faisant cela on diminuera forcément la vitesse de transmission des données.

Le protocole LoRa propose 6 différents étalements de spectre possible :

DataRate	Configuration	Indicative physical bit rate [bit/s]
0	LoRa: SF12 / 125 kHz	250
1	LoRa: SF11 / 125 kHz	440
2	LoRa: SF10 / 125 kHz	980
3	LoRa: SF9 / 125 kHz	1760
4	LoRa: SF8 / 125 kHz	3125
5	LoRa: SF7 / 125 kHz	5470
6	LoRa: SF7 / 250 kHz	11000

Figure 63 : différents étalements de spectre possible

En début de ce BE pour vérifier que nos trames LoRa étaient bien émises pour notre émetteur LoRa et pour débugger notre code, nous avions utiliser des modules usb SDR (software defined radio) pas très chère permettant de regarder le spectre des fréquences pour vérifier l'étalement en fréquence de la modulation LoRa.

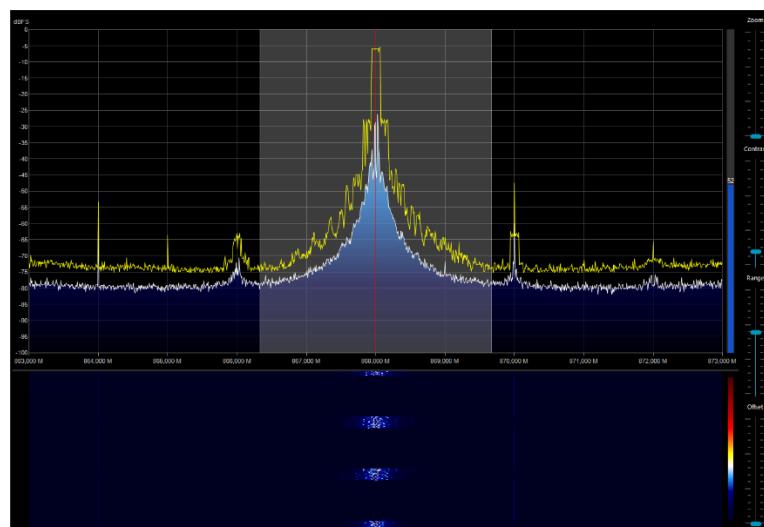


Figure 64 : étalement spectrale LoRa avec SF12

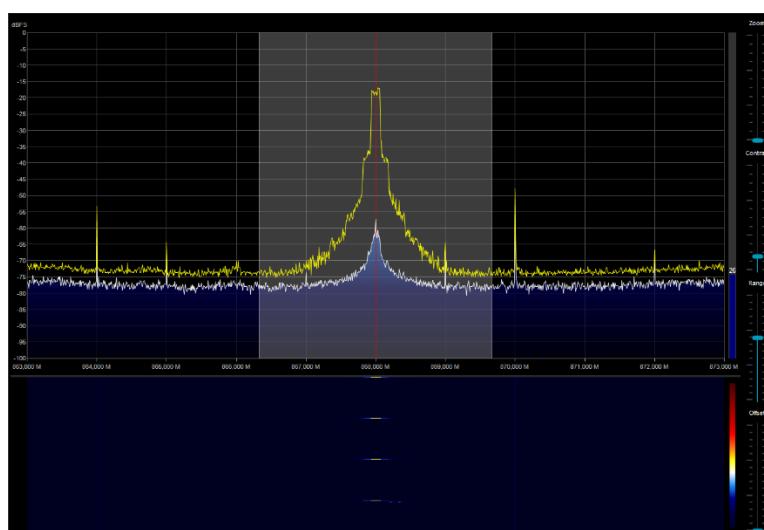


Figure 65 : étalement spectrale LoRa avec SF7

Logiciel utilisé « AIRSPY SDR# » avec un SDR « HackRF One » à une fréquence d'échantillonnage de 10Mhz.

On peut remarquer dans la représentation « waterfall » (la fenêtre inférieur) que plus on diminue le SF plus la transmission de donnée se rapide et donc consommera moins d'énergie.



Figure 66 : Exemple de chirp LoRa en temporelle

Le logiciel utilisé « GNU radio » avec un « HackRF one » à étais utiliser pour extraire les trames temporelles du module LoRa.

Remarque : il existe plusieurs GitHub qui propose des modules pour le logiciel gnu radio qui normalement devrait permettre de décoder les binaire des trame LoRa et même d'émettre ces propres message LoRa avec un HackRF one. Cependant nous n'avons jamais pu faire fonctionner ces modules.

Fonctionnement de la ruche avec tous les capteurs

Ce programme est le programme complet de la ruche avec tous les capteurs de câbler.

La liste des composants électronique présent sur la ruche est la suivante :

- Une carte arduino une R3
- Un shield Grove pour connecter tous les capteurs à l'arduino
- Température & humidité extérieur Grove « SHT31 »
- Température & humidité intérieur Grove « DHT11 »
- Capteur de présence infrarouge Grove PIR
- Une jauge de contrainte avec son amplificateur « HX711 »
- Un écran LCD « Grove-16*2 LCD (Black on Yellow) »
- Un module émetteur/récepteur Grove « LoRa-E5 »
- Une batterie lithium 3.7V 1000mAh avec son module de charge et décharge Grove

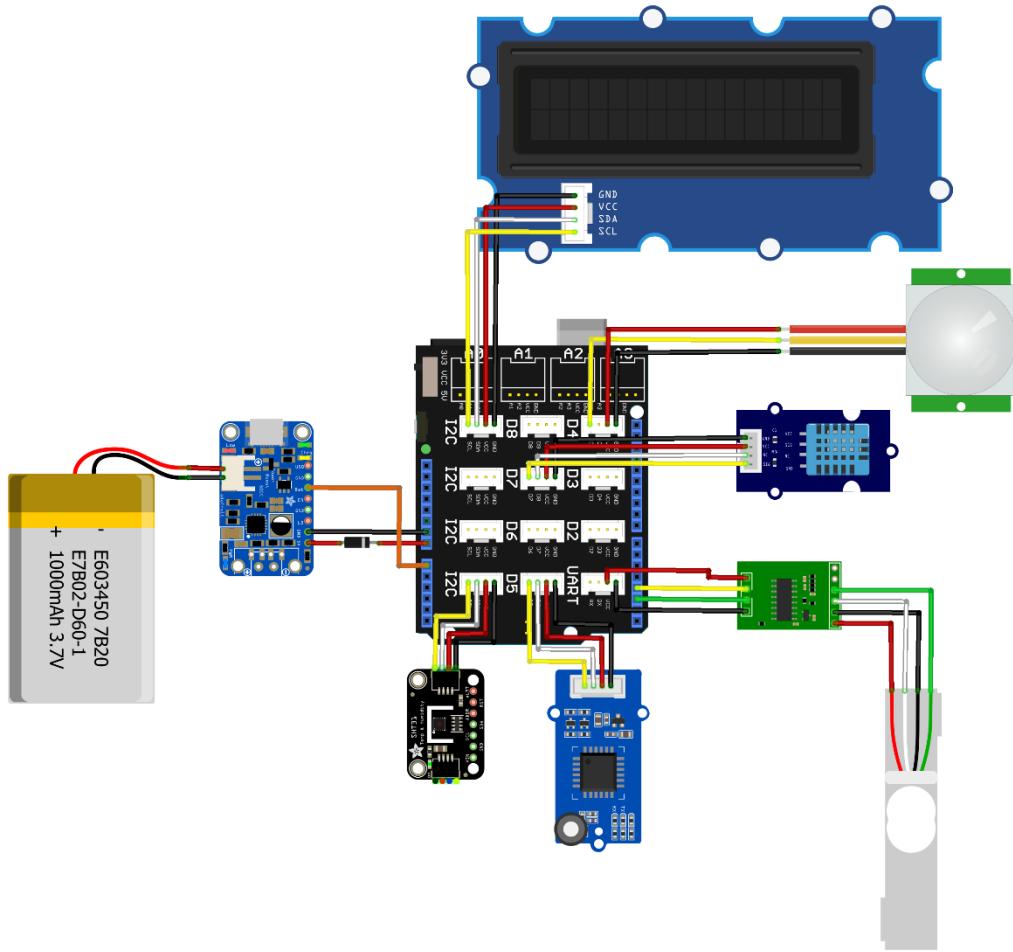


Figure 67 : Schéma électrique équivalent à celui de la ruche avec tous les capteurs

En raison de la taille importante du programme (environ 700 lignes) il est impossible de le mettre entièrement dans le rapport. C'est pourquoi à la place nous utiliserons un organigramme à la place.

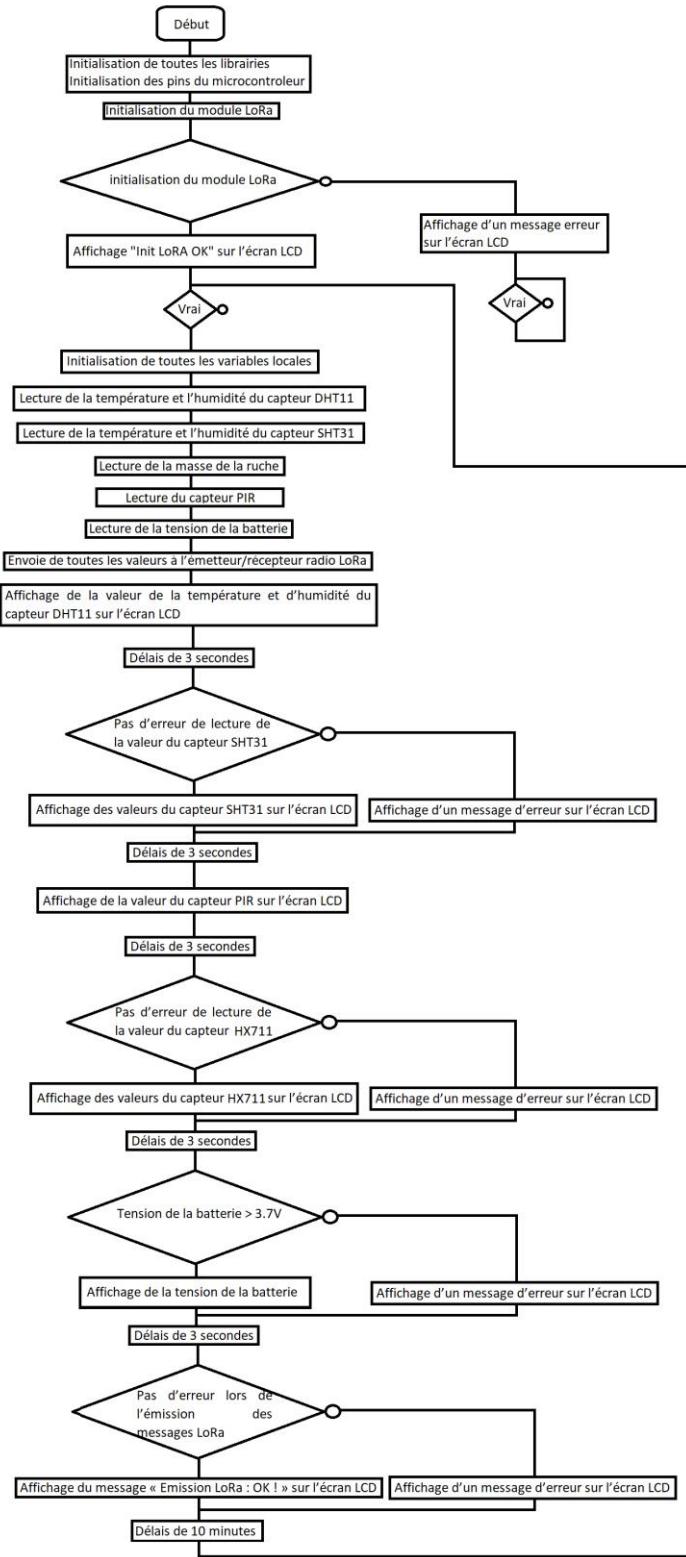


Figure 68 : Organigramme du code final de la ruche.

Ce code final vient regrouper chaque partie indépendante vu jusqu'à présent en un seul grand code. Comme chaque un des exemples de code précédent on étais sous forme de

fonction, le code final ne fait en fait que des appels de ces fonction précédemment réalisé.

Le code arduino entier est disponible sur le Github du projet au lien suivant :

«<https://github.com/Penzanto/L3-EEA-BE-Ruche-Connectee/tree/main/CodeArduinoRuche>»

Pour faire un rapide résumé du code final arduino de la ruche.

Lorsque l'arduino démarre, il place le module LoRa dans le bon mode de fonctionnement (mode émission, puissance d'émission, largeur d'empietement en fréquence etc...). Il vient vérifier que le module n'a pas rendu de code d'erreur lors de cette période sinon on bloque le programme dans une boucle infinie tous en affichant un message d'erreur sur l'écran LCD. Si tout c'est bien dérouler lors de la phase d'initialisation de l'arduino on commence la boucle principale du programme qui consiste à lire en boucle les valeurs des capteurs toutes les 10 minutes en les renvoyant au raspberry pi via le module LoRa. On vient aussi afficher les valeurs lues sur l'écran LCD avec un délais de 3 secondes entre chaque affichage après avoir envoyer toutes les valeurs via le module LoRa.

Réception des données LoRa et affichage sur une page internet

Comme cette partie ne fais pas spécialement partie des enseignements de la L3 EEA REL, nous survolerons surtout le fonctionnement du Raspberry pi et du fonctionnement de la page internet.

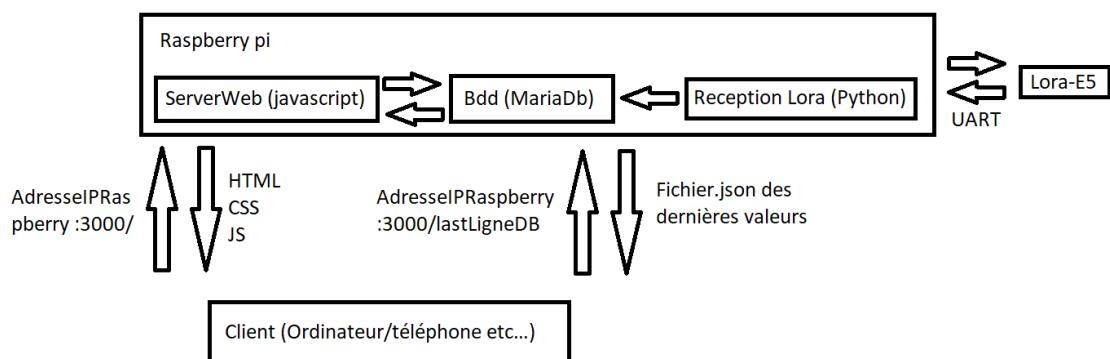


Figure 69 : schéma bloc du fonctionnement du Raspberry pi

Le Raspberry est la pièce qui fait la liaison entre les données envoyées en LoRa depuis la ruche et la page web qui est hôte sur le Raspberry pi.

Pour que tout fonctionne il faut donc que le Raspberry pi ait accès à une connexion internet, mais aussi qu'il soit suffisamment proche des ruches pour recevoir les informations LoRa.

Logiciel et langage de programmation utilisé sur le Raspberry pi

Comme logiciel et langage de programmation nous avons utilisé :

- Putty (sous windows) : pour se connecter à distance au raspberry via SSH sans avoir besoin d'écran ni de clavier.
 - FileZilla (sous windows) : pour pouvoir envoyer des fichiers directement via une liaison ethernet en FTP sans devoir brancher une clé usb pour tester des codes sur le raspberry (nativement le FTP n'est pas installé sur le raspberry, il faudra l'installer manuellement).
 - MariaDB : MariaDB est un type de base de données tel que mysql.
 - Node.js : pour pouvoir exécuter des programmes en Javascript sur le Raspberry.
 - Python3 : pour pouvoir utiliser le langage de programmation python.

Réceptions des message LoRa & base de données

Comme nous avons déjà pu expliquer comment fonctionne la partie LoRa dans la ruche il est inutile de la réexpliquer ici.
la seule différence c'est que le Raspberry pi ne fonctionne pas en 5V mais en 3.3V comparé à l'arduino. Cependant comme le module LoRa-E5 fonctionne très bien en 5V et en 3.3V ce changement ne change rien au fonctionnement du module LoRa.

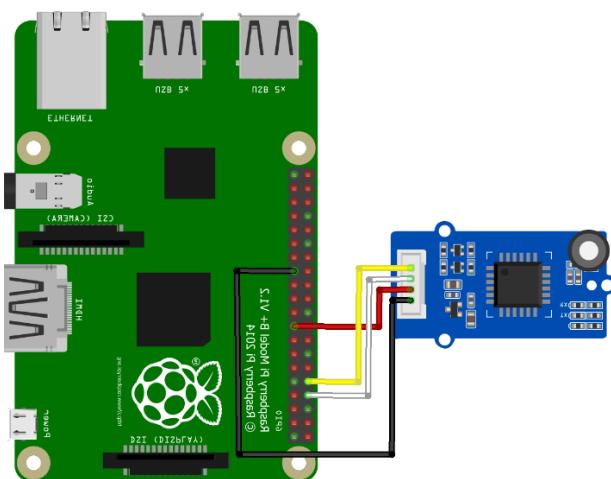


Figure 70 : Câblage du module LoRa sur le Raspberry pi

Ce programme python reçoit les messages LoRa vérifie leurs validité (bonne taille et bonne clé de ruche). Extrait l'information du capteur de la trame LoRa attend que les autres trames des autres capteurs soit reçus. Une fois tous cela fait, il va automatiquement rajouter toutes ces valeurs avec un horodatage dans la base de données.

DateHeure	Ruche	TempInter	HumidInter	TempExter	HumiExter	Masse	Proxi	Vbat
2023-05-06 02:21:00	0	40.00	90.00	25.00	70.00	100	1	4.20
2023-05-06 04:42:54	1	23.00	75.00	23.81	61.84	-130	0	3.92
2023-05-06 04:43:33	1	23.00	75.00	23.80	61.87	-130	0	3.92
2023-05-06 04:44:12	1	23.00	75.00	23.78	61.92	2356	0	3.91
2023-05-06 04:44:51	1	23.00	75.00	23.81	61.86	-130	0	3.92
2023-05-06 04:45:30	1	23.10	76.00	23.83	61.90	12	0	3.92
2023-05-06 04:46:09	1	23.10	75.00	23.80	61.78	12	0	3.92
2023-05-06 04:51:01	1	23.20	75.00	23.92	61.75	13	1	3.91
2023-05-06 14:23:10	1	21.90	76.00	22.53	65.56	14	0	3.88
2023-05-06 14:33:49	1	22.00	79.00	22.78	65.36	13	0	3.88
2023-05-06 14:44:28	1	22.20	78.00	22.90	63.51	13	0	3.88
2023-05-06 14:55:06	1	22.60	77.00	23.01	63.31	13	0	3.88
2023-05-06 15:05:45	1	22.70	77.00	23.09	63.05	13	0	3.88
2023-05-06 15:16:24	1	22.90	77.00	23.16	63.06	13	0	3.89
2023-05-06 15:27:03	1	23.10	76.00	23.32	64.80	13	0	3.88
2023-05-06 15:37:42	1	23.20	76.00	23.56	71.73	12	0	3.88

Figure 71 : Exemple de valeurs stocker dans la base de données

```
    if (isRoot) {
        if (parent == null) {
            return this;
        } else {
            return parent;
        }
    } else {
        return parent;
    }
}

public void setParent(Node parent) {
    this.parent = parent;
}

public Node getParent() {
    return parent;
}

public Node getLeft() {
    return left;
}

public Node getRight() {
    return right;
}

public void setLeft(Node left) {
    this.left = left;
}

public void setRight(Node right) {
    this.right = right;
}

public int getDepth() {
    return depth;
}

public void setDepth(int depth) {
    this.depth = depth;
}

public String toString() {
    return String.format("%s(%d)", super.toString(), value);
}

public String printInOrder() {
    if (left != null) {
        left.printInOrder();
    }
    System.out.println(value);
    if (right != null) {
        right.printInOrder();
    }
}

public String printPreOrder() {
    System.out.println(value);
    if (left != null) {
        left.printPreOrder();
    }
    if (right != null) {
        right.printPreOrder();
    }
}

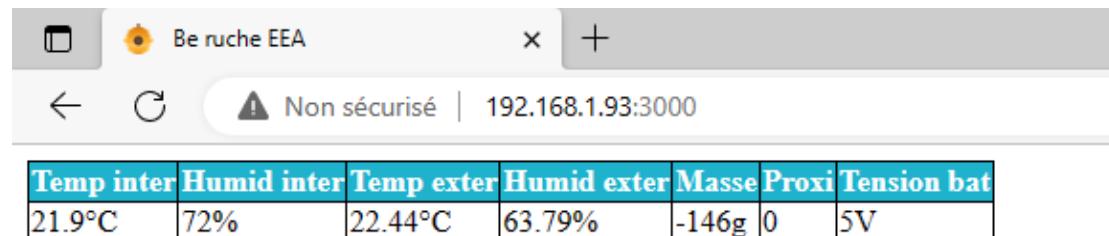
public String printPostOrder() {
    if (left != null) {
        left.printPostOrder();
    }
    if (right != null) {
        right.printPostOrder();
    }
    System.out.println(value);
}
```

Figure 72 : Algorithme de réception des message LoRa

Serveur Web

Le serveur web qui est hôte sur le raspberry pi fonctionne exactement comme ce qui est décrit dans la figure 69. Un client vient faire une requête de connexion au serveur web en se connectant à l'adresse « <http://192.168.1.93:3000/> » sachant que l'adresse « 192.168.1.93 » est l'adresse IP du Raspberry pi dans mon réseau local.

une fois la requête faite, le serveur web va répondre en envoyant les fichiers HTML, CSS et Javascript de la page web hôte chez le client.



A screenshot of a web browser window. The title bar says "Be ruche EEA". The address bar shows "Non sécurisé | 192.168.1.93:3000". The main content is a table with the following data:

Temp inter	Humid inter	Temp exter	Humid exter	Masse	Proxi	Tension bat
21.9°C	72%	22.44°C	63.79%	-146g	0	5V

Figure 73 : Page web hôte chez le client

Une fois la page ouverte, le code Javascript du client va venir faire des requêtes à l'adresse « <http://192.168.1.93:3000/lastLigneDB> » toutes les secondes pour recevoir la dernière ligne de la base de données sous forme de fichier .json qui comporte les dernières valeurs reçues de la ruche. Une fois ces informations reçues il va venir automatiquement les afficher dans le tableau.



A screenshot of a web browser window. The title bar says "192.168.1.93:3000/lastLigneDB". The address bar shows "Non sécurisé | 192.168.1.93:3000/lastLigneDB". The main content is a JSON object:

```
{"DateHeure": "2023-05-28T23:26:11.000Z", "Ruche": 1, "TempInter": 21.9, "HumidInter": 72, "TempExter": 22.44, "HumidExter": 63.79, "Masse": -146, "Proxi": 0, "Vbat": 5}
```

Figure 74 : exemple de fichier .json reçu

```

//ajout du framework mariadb
const mariadb = require('mariadb')

//ajout du framework fastify
const fastify = require('fastify')({ logger: true })

//ajout du framework fileSystem
const fs = require('fs')

//ajout du framework path
const path = require('path')


// cree la pool de la base de donnee
const pool = mariadb.createPool({
  host: 'localhost',
  port: '3306',
  user: 'userBERUCHE',
  password: 'BERUCHE'
});

//option du midlman fastify/static utiliser pour faire le transfer de fichier entre le server et le client
fastify.register(require('@fastify/static'),
{
  root: path.join(__dirname, 'SiteWeb'),
  prefix: '/',
  constraints: {}
})

//reglage de "Access-Control-Allow-Origin" (autorisation a tous le monde de faire de requette au serveur)
fastify.register(require('@fastify/cors'), {
  origin: '*'
})


// Start the server
fastify.listen({ port: 3000, host: '::' }, function (err, address)
{
  if (err)
  {
    fastify.log.error(err)
    process.exit(1)
  }
});

// Declare la route principale
fastify.get('/', async (req, reply) =>
{
  return reply.sendFile('PageWeb.html')
});

//Declare une nouvelle route secondaire
fastify.get('/lastLigneDB', async (req, reply) =>
{
  await lastValDataBase().then(function(data)
  {
    reply.header('Content-Type', 'application/json')
    return reply.send(data)
  })
});

//-----Fonction de maraiDB-----
async function lastValDataBase()
{
  /*
   * Fonction de recuperation de la dernière ligne de valeur dans la base de donnee mariadb
   *
   * return:
   *   lastRow(obj): retourne l'objet de la dernière ligne de la base de donnee
   */
  
  //connection a la db
  let conn = await pool.getConnection();

  //demande de valeur via requet sql
  conn.query("USE BERUCHE");
  const result = await conn.query("SELECT * FROM Mesure ORDER BY DateHeure DESC LIMIT 1");

  //conversion des valeurs
  const lastRow = result[0]

  //deconnection de la db
  conn.release();

  return lastRow;
}

```

Figure 75 : code du serveur web

```

window.onload=Function() {
    //fonction de recuperation et d'affichage des derniere valeurs recu de la ruche
    async function fetchData()
    {
        /*
            fonction de recuperation et d'affichage des valeurs des dernieres valeurs recu par la bdd
        */

        //recuperation des valeurs
        let url = 'http://192.168.1.93:3000/lastligneDB'
        const data = await fetch(url).then(res => res.json())

        //affichage des valeurs
        document.getElementById('TempInter').innerHTML = data.TempInter + "°C"
        document.getElementById('HumidInter').innerHTML = data.HumidInter + "%"
        document.getElementById('TempExter').innerHTML = data.TempExter + "°C"
        document.getElementById('HumidExter').innerHTML = data.HumidExter + "%"
        document.getElementById('Masse').innerHTML = data.Masse + "g"
        document.getElementById('Proxi').innerHTML = data.Proxi
        document.getElementById('Vbat').innerHTML = data.Vbat + "V"
    }

    fetchData()
    //repete la fonction FetchData() toutes les secondes
    setInterval(function() {fetchData();}, 1000);
}

```

Figure 76 : fichier javascript du client

```

<!DOCTYPE html>
<html>
    <head>
        <link href="style.css" rel="stylesheet" type="text/css"> <!-- link le fichier css -->
        <script src="clientScripts.js"></script> <!-- link le fichier javascript -->
        <link rel="icon" type="image/x-icon" href="favicon.ico"> <!-- link de l'image du site -->
    </head>
    <body>
        <table>
            <tr>
                <th>Temp inter</th>
                <th>Humid inter</th>
                <th>Temp exter</th>
                <th>Humid exter</th>
                <th>Masse</th>
                <th>Proxi</th>
                <th>Tension bat</th>
            </tr>
            <tr>
                <td id="TempInter">text</td>
                <td id="HumidInter">text</td>
                <td id="TempExter">text</td>
                <td id="HumidExter">text</td>
                <td id="Masse">text</td>
                <td id="Proxi">text</td>
                <td id="Vbat">text</td>
            </tr>
        </table>
    </body>
</html>

```

Figure 77 : Fichier HTML du client

```

body {
    background-color: rgb(255, 255, 255);
}

/* Dessin de la ruche */
table, th, td {
    border: 1px solid black;
    border-collapse: collapse;
}

th {
    background-color: #1eb2cc;
    color: white;
}

```

Figure 78 : Fichier CSS du client

Lancement des applications

Identifiant raspberry pi : pi

Mot de passe : BEruche

```
pi@raspberrypi:~ $ cd FTP/files/python/
pi@raspberrypi:~/FTP/files/python $ python3 InterfaceLoRaBdd.py

+INFO: Input timeout
+MODE: TEST

+INFO: Input timeout
+TEST: STOP

+INFO: Input timeout
+LW: LDRO, ON

+INFO: Input timeout
+TEST: RFCFG F:868000000, SF12, BW125K, TXPR:12, RXPR:15, POW:14dBm, CRC:ON, IQ:OFF, NET:OFF

+INFO: Input timeout
+TEST: RXLRPKT

initialisation du programme OK

Tous les valeurs recu sont:
Date :2023-05-31 02:10:31
idRuche: 1
TempInter: 25.299999237060547°C
HumidInter: 71.0%
TempExter: 24.21759796142578°C
HumidExter: 58.05142593383789%
Proxi: 5.877471754111438e-39
Masse: -137.46063232421875g
Vbat: OV
```

Figure 79 : Lancement du code Python de réception LoRa

```
pi@raspberrypi:~ $ cd FTP/files/ServerWeb/
pi@raspberrypi:~/FTP/files/ServerWeb $ npm run start

> serverweb@1.0.0 start
> node index.js

{"level":30,"time":1685490766604,"pid":6355,"hostname":"raspberrypi","msg":"Server listening at http://[::]:3000"}
```

Figure 80 : Lancement du ServerWeb

Une fois ces deux programmes lancer la ruche et le raspberry pi sont totalement autonome.