



INSTITUT SUPÉRIEUR  
INDUSTRIEL DE BRUXELLES  
I.S.I.B.

HE2B - MASTER 2

INTERNET OF THINGS

---

## Manipulations LoRa, LoRaWAN

---

*Auteurs :*

Michael GOBERT  
Ibrahim MBOULA

*Professeur :*

G. LE VAILLANT

26 décembre 2023

# Table des matières

<b>1 Émission/réception LoRa</b>	<b>1</b>
1.1 Mesures émission et réception . . . . .	1
1.1.1 Équipements . . . . .	1
1.1.2 Mesures . . . . .	1
1.1.3 Conditions de test . . . . .	1
1.1.4 Protocole d'expérience . . . . .	1
1.1.5 Résultats . . . . .	2
1.2 Fréquence, Chirp Spread, Spreading Factor . . . . .	4
1.2.1 Fréquence centrale et fréquences disponibles . . . . .	4
1.2.2 Chirp Spread Spectrum & Spreading Factor . . . . .	4
<b>2 Réseau de capteurs</b>	<b>6</b>
2.1 Montage et instrumentation des capteurs . . . . .	6
2.1.1 Capteur humidité H25K5A . . . . .	6
2.1.2 Capteur luminosité TEPT5700 . . . . .	6
2.1.3 Capteur température LMT84 . . . . .	7
2.1.4 Envoie des données . . . . .	7
2.2 Traitement des données dans le cloud . . . . .	7
2.3 Application de visualisation . . . . .	7
2.3.1 Broker MQTT de TTN . . . . .	8
2.3.2 Procédure de récupération des données . . . . .	8
2.3.3 Persistence des données et stockage . . . . .	9
2.3.4 Visualisation des données . . . . .	9
2.4 Analyser et optimiser les données envoyées . . . . .	9
2.4.1 Taille des paquets . . . . .	9
2.4.2 AirTime, Fréquence, SNR, RSSI . . . . .	9
2.4.3 Fair-Use Policy . . . . .	10
2.5 Consommation électrique du noeud . . . . .	10
2.6 Mécanismes de sécurité des données . . . . .	10
<b>3 Conclusion</b>	<b>10</b>
<b>Appendices</b>	<b>I</b>
<b>A Emission/ réception LoRa</b>	<b>I</b>
A.1 Matériel . . . . .	I
A.2 Fréquence centrale et fréquence dans le monde . . . . .	I
A.3 Résultats . . . . .	II
A.3.1 Différence checksum . . . . .	II
A.3.2 Comparaison SF7 à SF12 . . . . .	III
<b>B Configuration STM32 WL55JC1 &amp; MKRWAN1300</b>	<b>III</b>
B.1 Configuration STM32 Nucleo WL55JC1 . . . . .	III
B.2 MKRWAN1300 . . . . .	VIII
B.2.1 Code Arduino . . . . .	XI

B.3	Visualisation des données . . . . .	XII
B.4	Payload Formatter . . . . .	XIII
B.5	Decoded payload . . . . .	XIV
B.6	AirTime-Fréquence-SF-SNR-RSSI STM32 WL55JC1 . . . . .	XIV
B.7	AirTime-Fréquence-SF-SNR-RSSI MKRWAN1300 . . . . .	XV
B.8	Consommation électrique du noeud . . . . .	XV

## Table des figures

1	Comparaison Packet-RSSI pour tous les étages . . . . .	2
2	Comparaison RSSI pour tous les étages . . . . .	3
3	Comparaison SNR pour tous les étages . . . . .	3
4	Paquets manquants . . . . .	4
5	Architecture du réseau . . . . .	6
6	Matériel nécessaire à l'émission et réception LoRa . . . . .	I
7	Fréquences utilisées par LoRa dans le monde . . . . .	I
8	Déférence checksum . . . . .	II
9	comparaison sf7 et sf12 . . . . .	III
10	Configuration du MiddleWare . . . . .	IV
11	Configuration de JTAG and Trace . . . . .	IV
12	Force rejoin after reboot . . . . .	V
13	configStmConnexion . . . . .	V
14	tempSensorSysSensor . . . . .	VI
15	modifAdc10bit . . . . .	VI
16	syssensorH . . . . .	VII
17	syssensorC . . . . .	VII
18	loraAppDecla . . . . .	VII
19	loraAppBuffer . . . . .	VIII
20	ttnFormatterStm32 . . . . .	VIII
21	firstConfigCodeArduino . . . . .	IX
22	progFirstConfig . . . . .	IX
23	configTtnArduino . . . . .	X
24	configTtnArduino2 . . . . .	XI
25	Instrumentation code MKRWAN . . . . .	XI
26	connexion code MKRWAN . . . . .	XI
27	envoi message MKRWAN . . . . .	XII
28	Graphique température 20 dernières valeurs en temps réel . . . . .	XII
29	Graphique humidité en temps réel . . . . .	XIII
30	Payload Formatter . . . . .	XIII
31	Decoded payload . . . . .	XIV
32	AirTime-Fréquence-SF-SNR-RSSI STM32 WL55JC1 . . . . .	XIV
33	AirTime-Fréquence-SF-SNR-RSSI MKRWAN1300 . . . . .	XV
34	Consommation électrique du noeud . . . . .	XV

# 1 Émission/réception LoRa

## 1.1 Mesures émission et réception

Nous allons utiliser deux shields LoRa de la marque Dragino qui permettront l'envoi et la réception des paquets. Ceux-ci seront reliés à un raspberry Pi 3 B+ et un raspberry Pi 4. Afin de gérer l'émission et la réception de trames personnalisées (nouveau payload) nous avons adaptés le fichier main.c du projet dragino\_lora\_app.

### 1.1.1 Équipements

Nous pouvons retrouver le module du shield LoRa de dragino à la figure 6a et le raspberry Pi 3B+ à la figure 6b.

### 1.1.2 Mesures

Nous allons tester le fait que tous les paquets soient réceptionnés, que nous recevons les paquets dans le bon ordre d'émission, nous pourrons ainsi compter le nombre de paquets perdus.

Nous allons vérifier si les paquets ont été altérés grâce à une checksum de caractères ASCII faites à l'émission et à la réception.

Nous pourrons également mesurer la puissance de réception grâce au RSSI et dans un dernier temps analyser le ratio de bruit du signal (SNR).

Notre payload aura la forme suivante :

#### Payload :

- id : S'assurer que nous recevons bien uniquement nos paquets
- nPaquet : Vérifier l'ordre de réception et le nombre de paquet perdus
- dataMessage : Un message
- checksum(ascii) : intégrité du message
- RSSI : Puissance de réception du message
- SNR : Signal Noise Ratio

### 1.1.3 Conditions de test

Nous fixons le spreading factor à 7 (SF7), nous utiliserons également la modulation Chirp Spread Spectrum(CSS) et nous travaillerons à la fréquence de 868.1 MHz.

### 1.1.4 Protocole d'expérience

Nous aurons le module de réception qui se trouvera au laboratoire LARAS (3ème étage). Notre second module d'émission se trouvera à différents étages du bâtiment. Dans un premier temps, nous nous trouverons à 5m de notre module de réception ensuite à 20m. Après nous descendrons au second étage, au premier, au rez-de-chaussée, au niveau de la cafétéria(-1) et ensuite au -2.

Nous ferons également une 2ème expérience mais cette fois-ci en changeant le spreading factor à SF12 et en nous situant au -2 du bâtiment, nous pourrons observer les différences au niveau du RSSI, du SNR et du nombre de paquets perdus.

### 1.1.5 Résultats

Cent mesures ont été prises à chaque niveau du bâtiment de l'école. L'objectif est d'évaluer le comportement des paramètres des paquets reçus (Packet RSSI, RSSI, SNR et des paquets perdus), notamment en termes de pertes potentielles. Les graphiques inclus dans cette section illustrent ces observations et fournissent un aperçu détaillé de la performance des communications dans différentes conditions.

#### Packet RSSI

Le Packet RSSI est une mesure spécifiquement la force du signal au moment de la réception d'un paquet de données particulier.

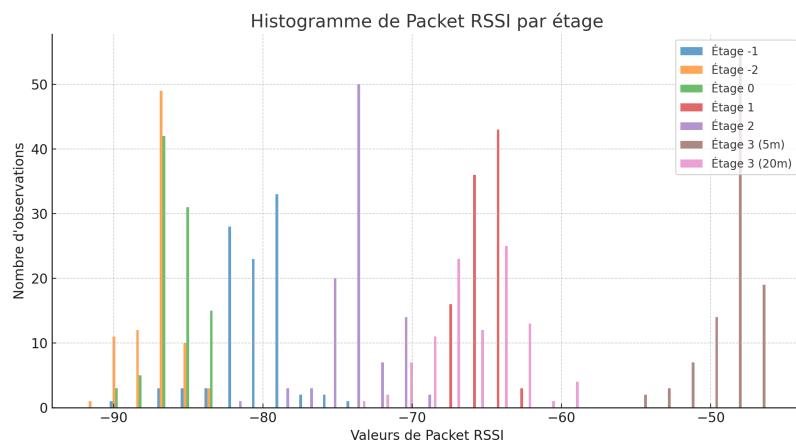


FIGURE 1 – Comparaison Packet-RSSI pour tous les étages

Ce graphe montre la distribution des valeurs des Paquets RSSI pour chaque étage, les étages supérieurs tendent à avoir des valeurs de plus en plus élevées. Lorsque l'émetteur et le récepteur sont proches, la valeur du signal est encore plus élevée. Les étages inférieurs présentent des valeurs plus faibles, suggérant un signal plus faible.

#### RSSI

Le RSSI (Received Signal Strength Indication) est une mesure de la puissance du signal radio reçu d'un émetteur. Il indique la force du signal à l'arrivée, mesurée en décibels (dBm).

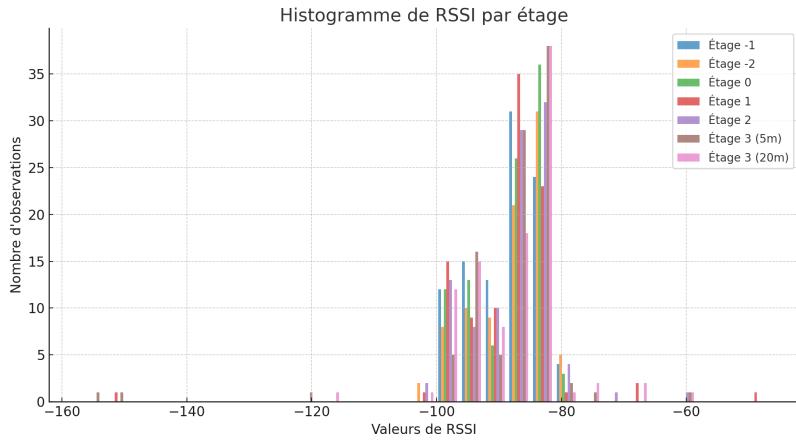


FIGURE 2 – Comparaison RSSI pour tous les étages

Les valeurs du RSSI sont assez similaires sur la plupart des étages. Il y a peu de variation significative dû au fait que nous ne changeons pas la puissance d'émission, ce qui montre qu'il est relativement stable indépendamment de l'emplacement dans le bâtiment.

## SNR

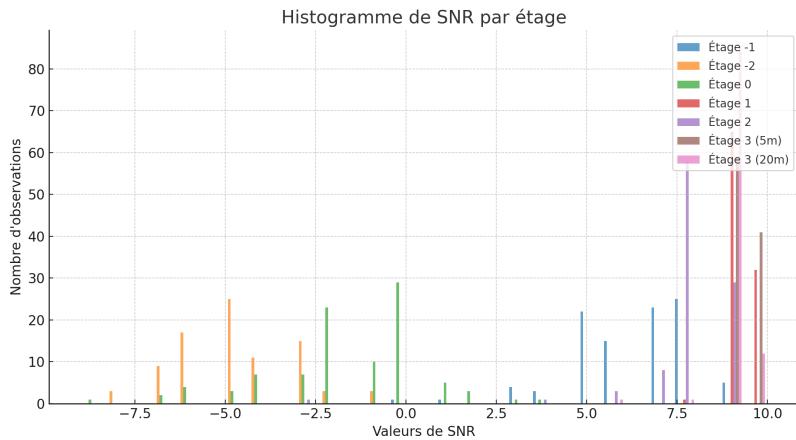


FIGURE 3 – Comparaison SNR pour tous les étages

Les étages supérieurs, en particulier l'étage 3, montrent des valeurs de SNR plus élevées, indiquant un meilleur rapport signal sur bruit. Les étages inférieurs, présentent des valeurs plus faibles, ce qui peut être le résultat d'un environnement plus bruyant ou d'un signal plus faible. Un SNR élevé indique une meilleure qualité de transmission, avec un signal nettement plus fort que le bruit, menant à une communication plus claire et plus fiable.

## Paquets erronés

Les paquets erronés peuvent correspondre soit à un paquet qui n'est pas arrivé où bien à un paquet avec une mauvaise checksum.

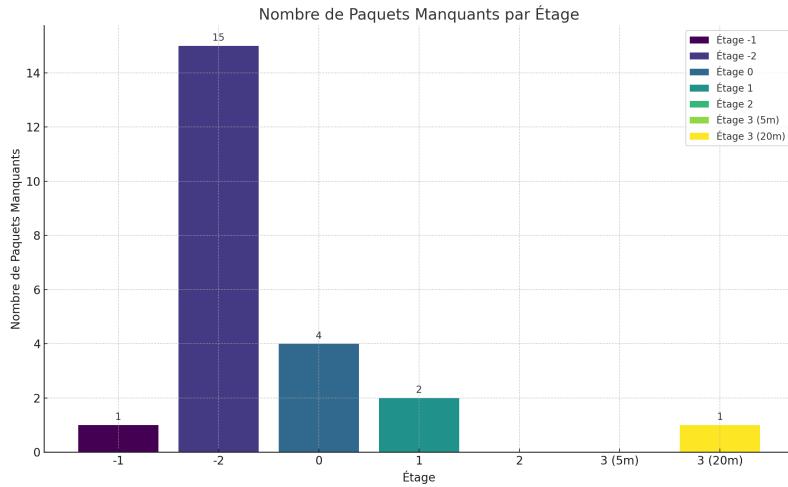


FIGURE 4 – Paquets manquants

Nous pouvons constater que nous avons plus de pertes de paquets lorsque nous nous éloignons du récepteur en dehors de l'étage -1 où nous n'avons eu qu'une perte.

Le graphique 8 montre qu'il n'y a qu'une différence entre la checksum calculée et reçue, uniquement à l'étage -2.

### Comparaison sf7 et sf12 à l'étage -2

La figure 9 montre que le changement de SF7 à SF12 n'entraîne pas de grandes modifications dans la qualité de réception du signal à l'étage -2. Nous pouvons juste constater que nous avons reçu plus de messages mais ceux-ci sont tout de même corrompu en partie.

## 1.2 Fréquence, Chirp Spread, Spreading Factor

### 1.2.1 Fréquence centrale et fréquences disponibles

La fréquence centrale que nous utiliserons dans notre cas est de 868.1MHz. Une autre fréquence libre également disponible que nous pouvons utiliser pour LoRa est 433MHz. Nous pouvons voir quelles sont les autres bande de fréquence disponibles dans le monde pour LoRa à la figure 7.

### 1.2.2 Chirp Spread Spectrum & Spreading Factor

#### 1) Chirp Spread Spectrum

Le Chirp Spread Spectrum (CSS) est une technique d'étalement du spectre qui utilise des impulsions chirp à modulation de fréquence linéaire à large bande pour coder des informations. Un chirp est un signal sinusoïdal dont la fréquence augmente ou diminue avec le temps.

## **Vue d'ensemble :**

- Le CSS utilise toute la largeur de bande allouée pour diffuser un signal, ce qui le rend résistant au bruit du canal.
- Les chirps, utilisant une large bande du spectre, rendent le CSS résistant à l'évanouissement par trajets multiples, même à faible puissance.
- Résiste à l'effet Doppler

### **Avantages et inconvénients de l'étalement de spectre chirp**

Avantages :

- Faible latence
- Performances précises et résistance aux interférences radio
- Consommation d'énergie efficace

Inconvénients :

- Vulnérabilité aux attaques, compromettant la confidentialité des données.
- Excès de bande passante nécessaire.

## **2) Spreading Factor :**

LoRa repose sur la technologie Chirp Spread Spectrum (CSS), utilisant des chirps (ou symboles) comme support des données. Le facteur d'étalement contrôle le taux de chirp et, par conséquent, la vitesse de transmission des données. Des facteurs d'étalement plus bas entraînent des chirps plus rapides, augmentant le taux de transmission, tandis que chaque augmentation du facteur d'étalement divise la vitesse de balayage du chirp par deux, réduisant ainsi le taux de transmission.

Cependant, des facteurs d'étalement plus faibles réduisent la portée des transmissions LoRa en diminuant le gain de traitement et en augmentant le débit binaire. La modification du facteur d'étalement permet au réseau de réguler le débit pour chaque appareil, mais cela se fait au détriment de la portée.

### **Influence des facteurs d'étalement**

La modulation LoRa comporte un total de 6 facteurs d'étalement, de SF7 à SF12. Les facteurs d'étalement influencent le débit de données, le temps d'utilisation, l'autonomie de la batterie et la sensibilité du récepteur.

### **Distance**

Des facteurs d'étalement plus importants signifient un gain de traitement plus important, de sorte qu'un signal modulé avec un facteur d'étalement plus important peut être reçu avec moins d'erreurs qu'un signal avec un facteur d'étalement plus faible, et donc parcourir une plus grande distance.

### **AirTime**

Par rapport à un facteur d'étalement plus faible, l'envoi d'une quantité fixe de données (charge utile) avec un facteur d'étalement plus élevé et une largeur de bande fixe nécessite un temps de propagation plus long.

## 2 Réseau de capteurs

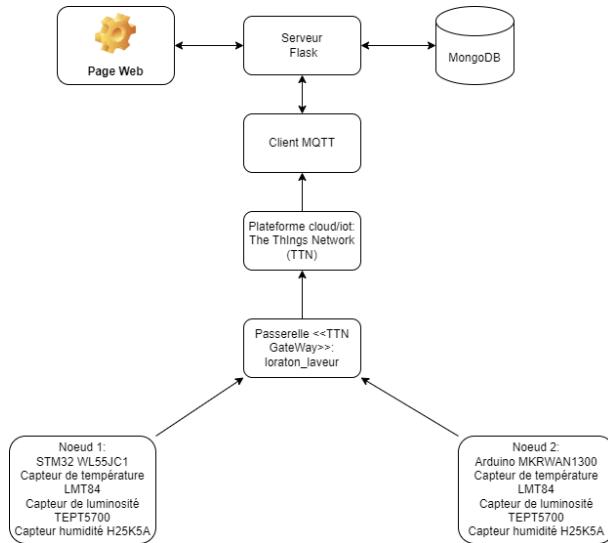


FIGURE 5 – Architecture du réseau

### 2.1 Montage et instrumentation des capteurs

#### 2.1.1 Capteur humidité H25K5A

Ce capteur nous donne en sortie une valeur analogue. Afin de déterminer l'humidité nous possédons une table reprenant dans les colonnes la température ainsi que le pourcentage d'humidité dans les lignes. Chaque case représente une valeur de résistance. Nous allons donc devoir déterminer cette valeur de résistance afin d'en conclure le pourcentage d'humidité. Nous calculons cette valeur de résistance grâce à un pont diviseur de tension.

$$R_1 = R_2 * \left( \frac{(V_{in})}{(V_{out})} - 1 \right) \quad (1)$$

où

- $V_{in}=3,3V$
- $V_{out}=(\text{ValueRead}/1024)*V_{in}$  ;
- $R_2=1M\Omega$

Une fois que nous avons la valeur de  $R_1$  nous pouvons extrapoler la pourcentage d'humidité en fonction de la température.

#### 2.1.2 Capteur luminosité TEPT5700

Ce capteur est un phototransistor NPN qui nous donne également une valeur analogue. Comme pour le capteur d'humidité nous devons lire cette valeur sur une pin analogue (PB1 sur STM32 et A2 sur MKRWAN). Lorsque nous lisons cette valeur sur une pin analogue nous utilisons en fait l'ADC interne des cartes. Nous allons donc diviser la valeur lue par 1024 (ADC 10 bit sur le MKRWAN, ADC 10 bit défini sur le STM32) pour ensuite la multiplier par 100 afin d'avoir notre valeur sous forme de pourcentage.

Nous devons également relier une résistance de  $39k\Omega$  en série au capteur afin de limiter le courant.

$$light = \frac{(light)}{(1024)} * 100 \quad (2)$$

### 2.1.3 Capteur température LMT84

Nous avons un dernier capteur qui est une capteur de température (thermistance) nous donnant également une sortie analogique (le capteur délivre une tension inversement proportionnelle à la température). Afin de déterminer la valeur de température nous devons faire le calcul suivant :

$$temperature = \frac{(temperature_{read})}{(1024)} * V_{in} \quad (3)$$

$$temperature = (-183.4862 * temperature) + 186.697247 \quad (4)$$

Les valeurs de multiplication et d'addition ont été déterminées à l'aide du graphe "Output Voltage vs Temperature" dans la datasheet.

Nous avons également une capacité de bypass de  $820nF$  (condensateur de découplage, destiné à réduire le couplage entre signal et alimentation) qui relie la pin VDD à la pin GND.

### 2.1.4 Envoie des données

#### STM32 WL55JC1 :

Nous devons dans un premier temps créer nos variables de nos capteurs dans sys\_sensor.h, ensuite dans le fichier sys\_sensor.c nous allons faire les instrumentations des capteurs, valeurs que nous rajouterons dans la structure sensor\_data. Nous pourrons par après, dans le fichier lora\_app.c, ajouter ces capteurs dans le AppBuffer que nous enverrons. Nous aurons deux bytes pour le capteur d'humidité, un byte pour la température et encore deux bytes pour le capteur de luminosité. Nous pouvons retrouver les différentes étapes aux figures 16, 17, 19.

#### Arduino MKRWAN1300 :

A nouveau nous faisons l'instrumentation des capteurs(25) comme présenté dans cette section, ensuite nous allons stocker les différentes valeurs dans un String, String que nous enverrons(26 et 27) et qui sera parser dans le payload Formatter.

## 2.2 Traitement des données dans le cloud

Pour le traitement des données dans le cloud nous allons devoir créer un Custom Javascript Formatter afin de décoder nos payloads spécifiques à nos modules. Nous en créons un pour notre STM32(30a) et un autre pour notre MKRWAN1300(30b).

## 2.3 Application de visualisation

L'application réalisé a pour objectif de visualiser les données envoyées par nos noeuds. Elle suit une architecture à trois niveaux, comprenant un serveur Flask, une base de

données MongoDB et un client. Les données en temps réel sont récupérées à partir de TTN grâce à un broker MQTT, ce qui nous permet de surveiller les données au fur et à mesure qu'elles arrivent.

### 2.3.1 Broker MQTT de TTN

Les noeuds utilisent *LoRaWAN*, qui est un protocole de communication longue portée, pour envoyer leurs données des capteurs à la passerelle *TTN* située à l'école. La plateforme cloud *TTN* rend ces données accessibles via l'interface *Live Data*.

Pour exploiter ces données en dehors de *TTN*, nous utilisons le broker *MQTT* de *TTN*. On commence par s'abonner aux topics *MQTT* (s'abonner à un flux de donnée) correspondant à nos nœuds en créant une clé API via la section *Intégrations -> MQTT* de *TTN*. Dans notre application développée, nous utilisons `client.subscribe("#")` pour nous abonner à tous les topics disponibles sur le broker *MQTT*. Cela permet à notre client de recevoir tous les messages publiés par nos nœuds '*Arduino MKRWAN*' et '*STM*', facilitant ainsi leur traitement et leur visualisation dans notre application.

### 2.3.2 Procédure de récupération des données

Tout d'abord il faut installer la librairie *paho.mqtt*

#### — Configuration du Client MQTT :

```
client = mqtt.Client()
```

Configuration des identifiants avec `client.username_pw_set(app_id, app_key)`. Ces informations sont disponibles dans intégration -> MQTT

#### — Connexion au Broker MQTT :

```
client.connect("eu1.cloud.thethings.network", 1883, 60)
```

Cette ligne établit une connexion avec le broker MQTT fourni par TTN. Le nombre 1883 représente le port standard utilisé pour les connexions MQTT, c'est le port sur lequel le client MQTT tente de se connecter au broker. Le nombre 60, représente un paramètre qui permet de maintenir la connexion ouverte entre le client et le broker, il envoie régulièrement un ping pour confirmer que la connexion est toujours active.

#### — Abonnement aux Topics MQTT :

```
client.subscribe("#")
```

Cela souscrit à tous les messages de tous les topics, permettant de recevoir n'importe quelle donnée publiée.

#### — Réception et Traitement des Messages :

```
def on_message(client, userdata, message):
    # Traitement du message
```

Cette fonction est appelée à chaque fois qu'un message est reçu. Elle décode le message, extrait les données, et traite les données en fonction de l'identifiant du nœud, pour mieux les stocker.

### 2.3.3 Persistence des données et stockage

Avoir une base de données est essentielle dans notre architecture pour pallier l'absence de stockage de données à long terme dans le *cloud TTN*. L'utilisation de *MongoDB* offre une solution pour la persistance des données, permettant ainsi d'archiver l'historique des mesures et de faciliter leur traitement ultérieur.

L'application utilise MongoClient pour se connecter à MongoDB. Les données traitées par les fonctions *process\_stm\_payload* et *process\_arduino\_payload* sont insérées dans des collections spécifiques via *collection\_arduino.update\_one()* et *collection\_stm.update\_one()*, en utilisant un identifiant unique pour chaque entrée. Cette approche garantit non seulement que nos données sont sauvegardées de manière durable mais aussi qu'elles sont uniques et facilement récupérables pour des analyses.

### 2.3.4 Visualisation des données

Pour visualiser les données dans notre application, nous utilisons une combinaison de *Flask*, *MongoDB*, et *Chart.js*. *Flask* sert de pont entre *MongoDB* et notre interface utilisateur web. Les routes *Flask* comme `/data` et `/data/stm/last/<int:count>` récupèrent les données de *MongoDB* et les retournent au format JSON.

Dans le front-end, des fonctions JavaScript comme `fetchMeasures()` récupèrent ces données JSON et les passent à `updateChart()`, qui utilise *Chart.js* pour les afficher sous forme de graphiques interactifs. Les graphiques sont configurés pour se mettre à jour toutes les 10 secondes, permettant ainsi aux utilisateurs de bénéficier d'une visualisation dynamique qui reflète l'évolution instantanée des données capturées par les noeuds.

De plus, l'application offre aux utilisateurs la capacité de filtrer les données selon des critères spécifiques tels que la date ou une quantité déterminée de mesures, et de choisir entre différents types de représentations graphiques ( barres ou courbes). Nous pouvons retrouver les figures 28 et 29 en annexes qui représentent les résultats de l'API.

## 2.4 Analyser et optimiser les données envoyées

Nous enverrons dans le payload uniquement les bytes correspondants aux valeurs que nous aurons mesurés sur nos différents capteurs.

### 2.4.1 Taille des paquets

Dans les deux cas nous n'envoyons que les valeurs des capteurs de sorte à minimiser la taille des paquets envoyés. Pour le STM32(31a), nous avons deux bytes pour l'humidité, 1 byte pour la température et 2 bytes pour la luminosité. Pour le MKRWAN1300, chaque valeur est codée sur le même nombre de bytes(31b).

### 2.4.2 AirTime, Fréquence, SNR, RSSI

L'airTime reste le même étant donné que notre payload ne change pas mais il est légèrement supérieur pour le MKRWAN par rapport au STM32. La fréquence fluctue légèrement car Lora détermine quelle fréquence est la moins encombrée. Nos SNR restent positifs mais il varient constamment et les RSSI varient également.

#### **STM32 WL55JC1 :**

Les différents facteurs sont repris aux figures 32a et 32b.

#### **Arduino MKRWAN1300 :**

Les différents facteurs sont repris aux figures 33a et 33b.

#### **2.4.3 Fair-Use Policy**

Nous avons droit à 30s par jour par noeud d'air time.

Pour le STM32, nous avons un airtime de 0,051sec donc nous pourrons envoyer 588 messages par jour.

Pour le MKRWAN1300, étant donné que l'airtime est de 0.067sec plus ou moins nous pourrons envoyer 447 messages sur la journée.

### **2.5 Consommation électrique du noeud**

Pour cette partie, nous avons travaillé en groupe avec les électroniciens, nous aurons donc les mêmes captures pour la figure 34a et 34b.

Nous avons un grand pic de consommation de courant lorsque le STM32 se "réveille". Nous avons ensuite un second pic plus faible lors de la lecture des capteurs et enfin nous pouvons constater une durée de consommation moins importantes lors de l'envoie des données, durée qui varie selon le nombre de bytes à transmettre.

### **2.6 Mécanismes de sécurité des données**

Nous devons récupérer le DevEUI de notre module. Ensuite, nous allons sur TTN où nous créons un nouveau device dans notre application. Nous définissons sur TTN un joinEUI unique pour notre module (nous le donnerons à notre module par la suite). Dans le nouveau device créé sur TTN, nous donnons le devEUI et nous pouvons à présent générer une appKey que nous renseignerons à notre module afin qu'il puisse communiquer au travers de ce device créé sur TTN.

La clé d'application (AppKey) n'est connue que du dispositif et de l'application. Les appareils activés dynamiquement (OTAA) utilisent la clé d'application (AppKey) pour dériver les deux clés de session pendant la procédure d'activation.

## **3 Conclusion**

La première partie de ce projet a permis une exploration approfondie de la modulation LoRa, en se concentrant sur des aspects clés comme le RSSI, le SNR et le facteur d'étalement. Des mesures ont été effectuées pour observer ces paramètres.

La suite du projet consiste à la mise en œuvre d'une variété de capteurs et l'acquisition de données via des microcontrôleurs. En utilisant le protocole LoRaWAN, nous avons assuré une transmission des données vers une passerelle, puis vers le cloud. L'emploi de MQTT a ensuite simplifié la récupération et le stockage des données dans une base de données pour enfin les afficher sur une API.

# Appendices

## A Emission/ réception LoRa

### A.1 Matériel



(a) Shield LoRa Dragino



(b) RaspberryPi 3B+

FIGURE 6 – Matériel nécessaire à l'émission et réception LoRa

### A.2 Fréquence centrale et fréquence dans le monde

Region	The Lora-based Frequency
Europe	863-870 MHz
	433 MHz
US	902-928 MHz
	470-510 MHz
China	779-787 MHz
	915-928 MHz
Australian	865-867 MHz
Indian	433 MHz
Asia	915 MHz
North America	915 MHz

FIGURE 7 – Fréquences utilisées par LoRa dans le monde

## A.3 Résultats

### A.3.1 Différence checksum

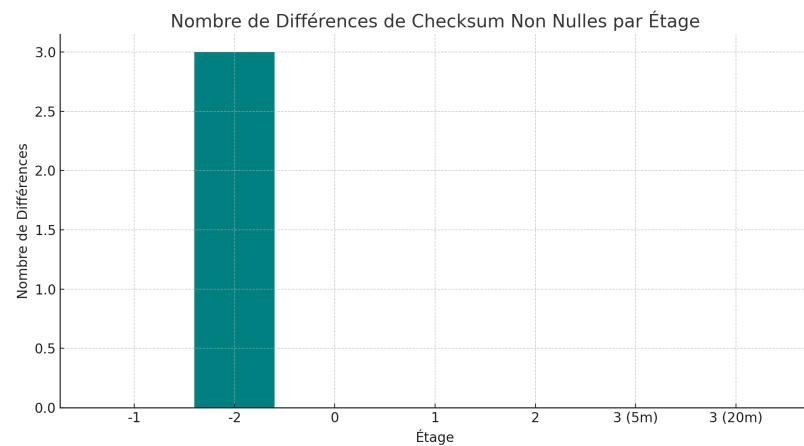


FIGURE 8 – Différence checksum

### A.3.2 Comparaison SF7 à SF12

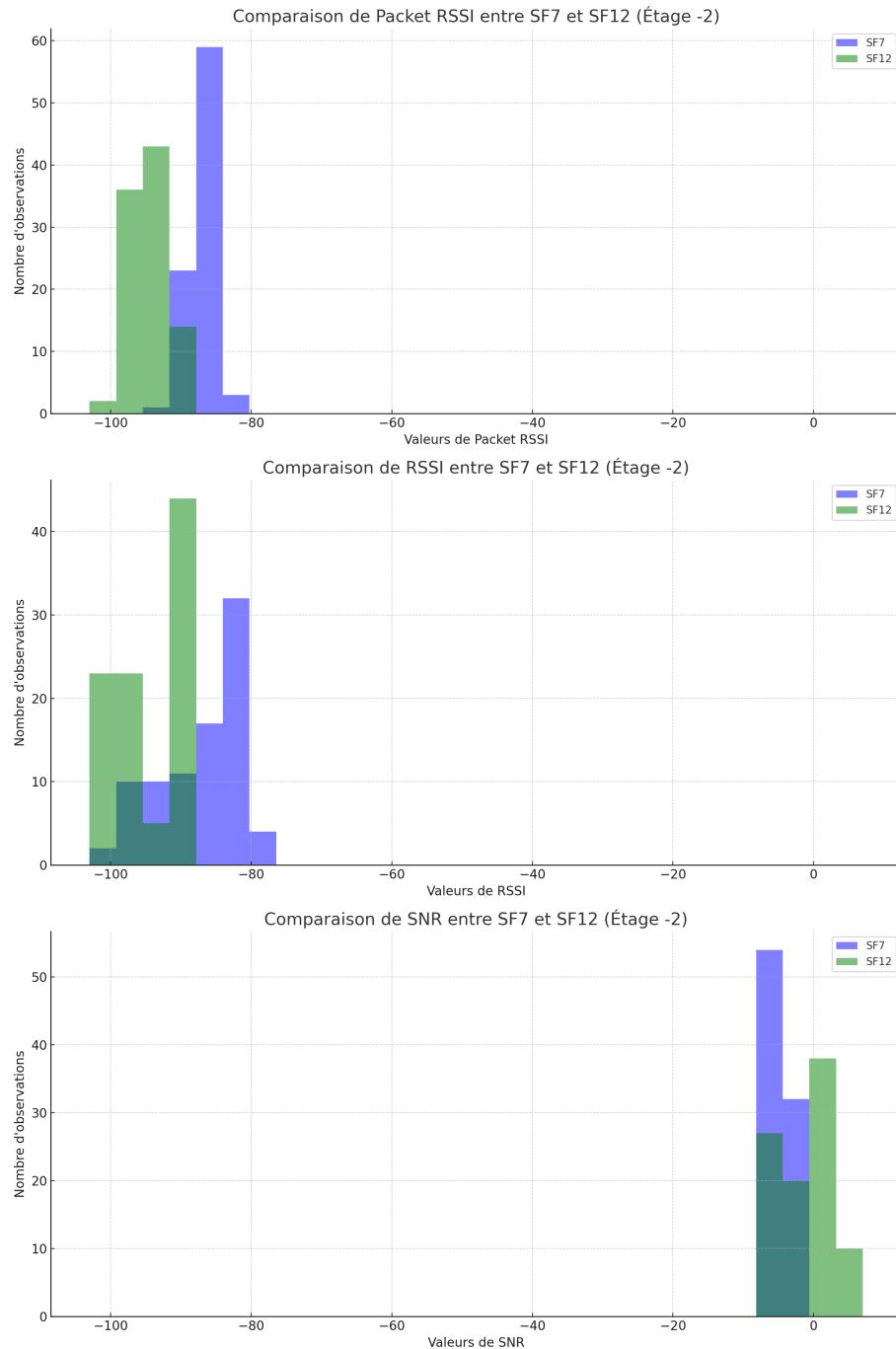


FIGURE 9 – comparaison sf7 et sf12

## B Configuration STM32 WL55JC1 & MKRWAN1300

### B.1 Configuration STM32 Nucleo WL55JC1

Configuration du MiddleWare sur STM32 Nucleo WL55JC1

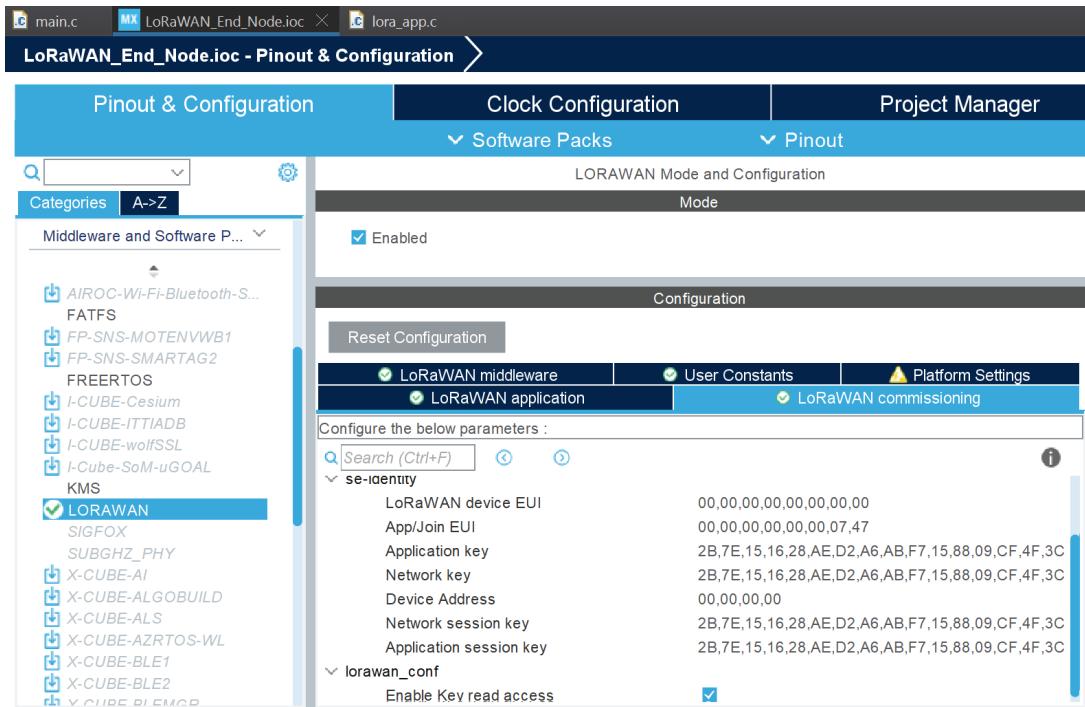


FIGURE 10 – Configuration du MiddleWare

Configuration de JTAG and Trace dans debuger défini en serialWire

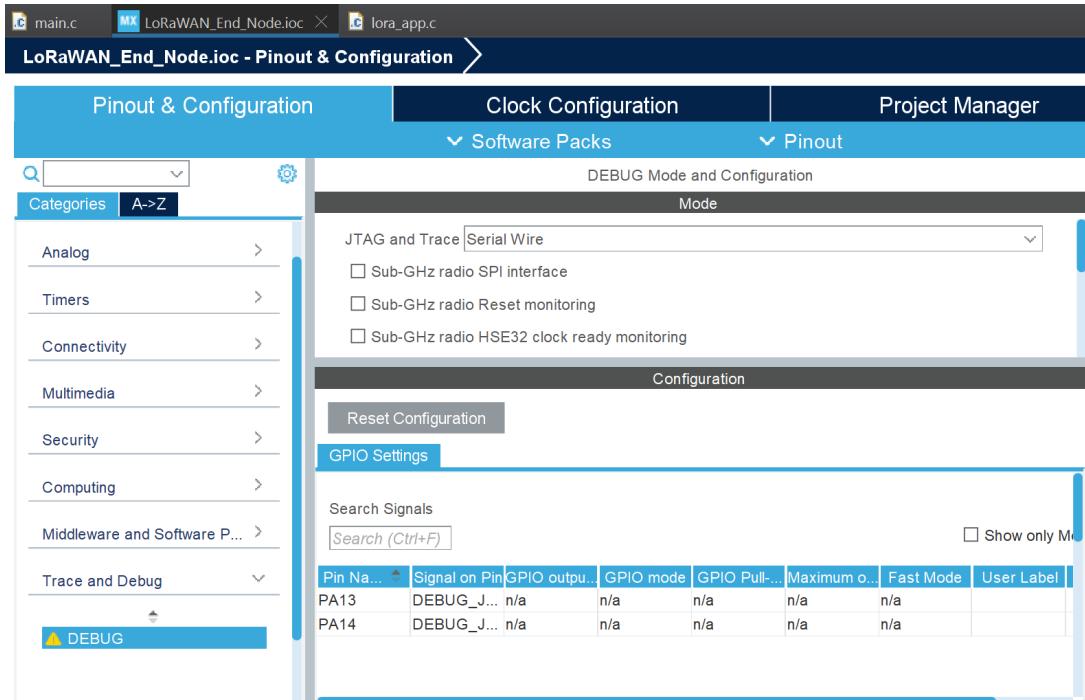


FIGURE 11 – Configuration de JTAG and Trace

CubeMX -> LORAWAN -> LoRaWAN application, activer "Force rejoin after reboot"

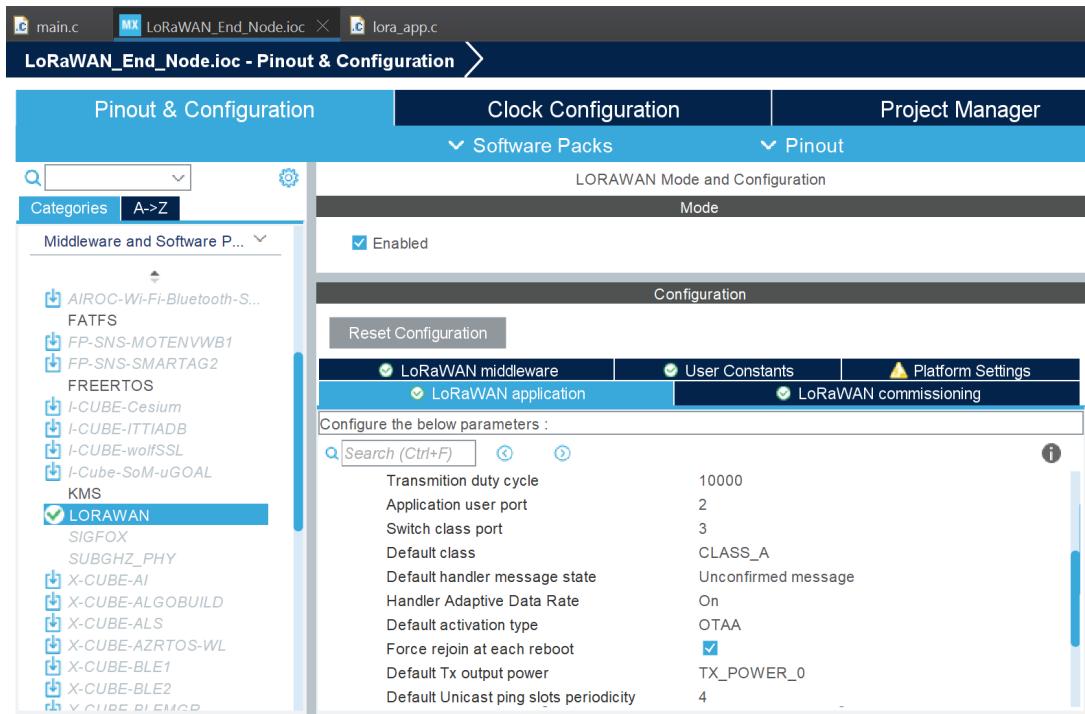


FIGURE 12 – Force rejoin after reboot

Le downgrade de version est à faire afin d'éviter des refus de connexion, si nous nous connectons avec le même id que précédemment la connexion sera refusée, c'est pourquoi nous choisissons cette version qui n'implémente pas cela.

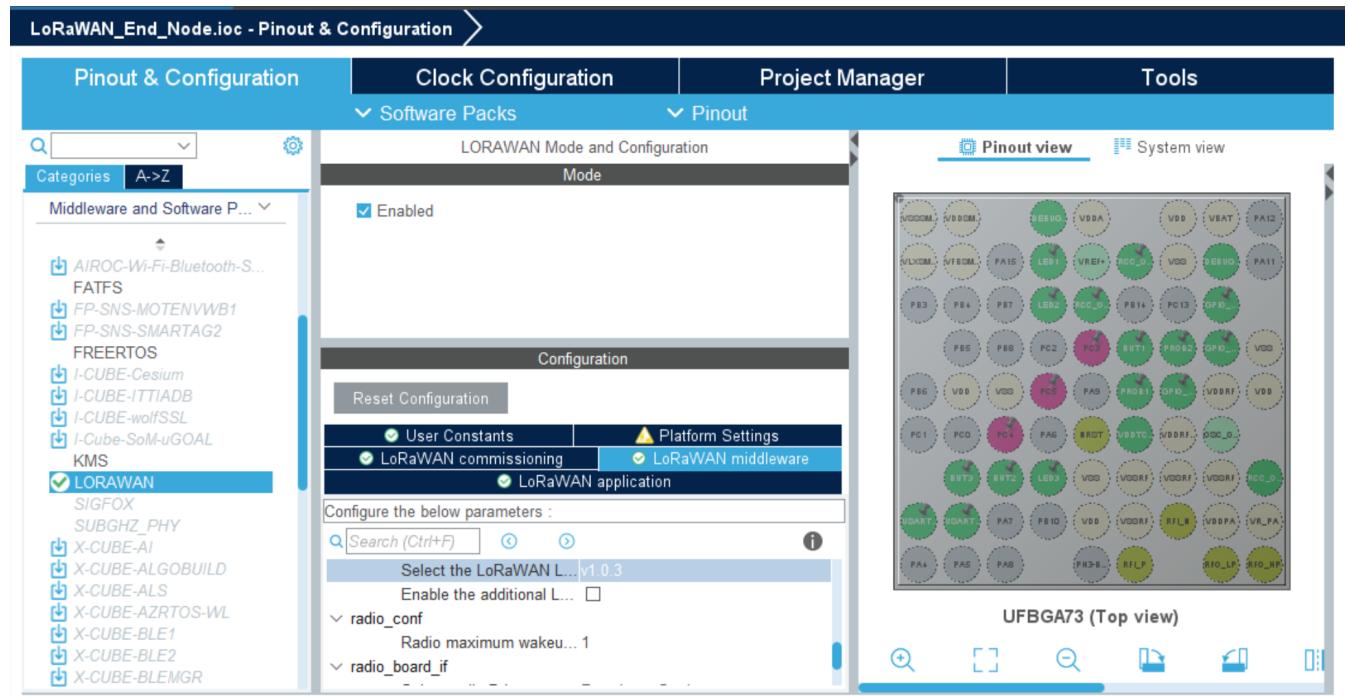


FIGURE 13 – configStmConnexion

Connexion sur le STM32

- PB1 - A0 (adc IN5) Luminosité
- PB2-A1 (adcIN4) Température
- PA10-A2 (adc IN6) Humidité

Code :

dans adc\_if.c on supprime static de la fonction read channels  
dans sys\_sensors

```
TEMPERATURE_Value = ADC_ReadChannels(ADC_CHANNEL_4);
TEMPERATURE_Value = (TEMPERATURE_Value/1023)*3.3; // Tension proportionnelle de la température, 1023 car adc sur 10 bit
TEMPERATURE_Value = (-183.4862*TEMPERATURE_Value)+186.697247; // Conversion en température
```

FIGURE 14 – tempSensorSysSensor

On divise par 1024 car adc sur 10 bit (normalement 12 bit resolution mais on la config pour 10bit)

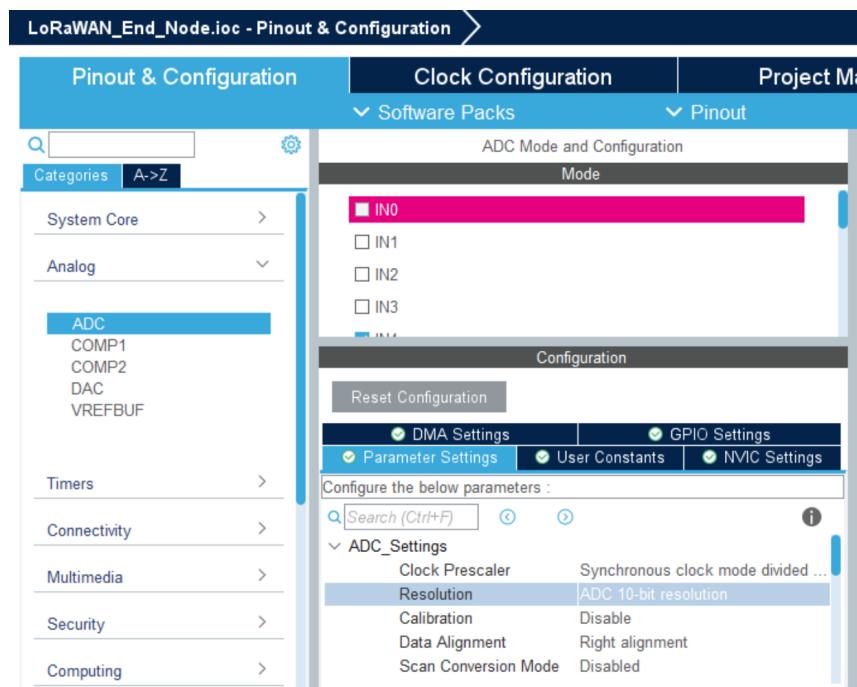


FIGURE 15 – modifAdc10bit

Data light :  
dans sys\_sensors.h on ajoute

```

38 @typedef struct
39 {
40     float pressure;           /*!< in mbar */
41     float temperature;        /*!< in degC */
42     float humidity;          /*!< in % */
43     int32_t latitude;         /*!< latitude converted to binary */
44     int32_t longitude;        /*!< longitude converted to binary */
45     int16_t altitudeGps;      /*!< in m */
46     int16_t altitudeBar;      /*!< in m * 10 */
47     /* more may be added */
48     /* USER CODE BEGIN sensor_t */
49     float lightM;
50     /* USER CODE END sensor_t */
51 } sensor_t;

```

FIGURE 16 – syssensorH

ensuite dans sys\_sensors.c

```

147
148 #else
149     TEMPERATURE_Value = ADC_ReadChannels(ADC_CHANNEL_4); //PB2 A1 adc_in4
150     TEMPERATURE_Value = (TEMPERATURE_Value/1024)*3.3; // Tension proportionnelle de la température, 1023 car adc sur 10 bit
151     TEMPERATURE_Value = (-183.4862*TEMPERATURE_Value)+186.697247; // Conversion en température
152
153     HUMIDITY_Value = ADC_ReadChannels(ADC_CHANNEL_6); //PA10 A2 adc_in6
154     float Vout = (HUMIDITY_Value/1024)*3.3; // Tension proportionnelle de la température, 1023 car adc sur 10 bit
155     float R2=1000000;
156     float Vin=3.3;
157     float R1 = R2*((Vin/Vout)-1);
158     HUMIDITY_Value = R1;
159
160     LIGHTM_Value = ADC_ReadChannels(ADC_CHANNEL_5); //PB1 A0 adc_in5
161     LIGHTM_Value = (LIGHTM_Value/1024)*100; // Tension proportionnelle de la température, 1023 car adc sur 10 bit
162
163
164
165 #endif /* SENSOR_ENABLED */
166
167     sensor_data->humidity    = HUMIDITY_Value;
168     sensor_data->temperature  = TEMPERATURE_Value;
169     sensor_data->lightM      = LIGHTM_Value;
170
171     sensor_data->latitude   = (int32_t)((STSOP_LATTITUDE * MAX_GPS_POS) / 90);
172     sensor_data->longitude  = (int32_t)((STSOP_LONGITUDE * MAX_GPS_POS) / 180);
173
174     return 0;
175     /* USER CODE END EnvSensors_Read */
176 }

```

FIGURE 17 – syssensorC

ensuite dans lora\_app, on déclare une variable où on va mettre la value de notre light

```

564
565     #else
566         uint16_t pressure = 0;
567         int16_t temperature = 0;
568         uint16_t humidity = 0;
569         uint32_t i = 0;
570         int32_t latitude = 0;
571         int32_t longitude = 0;
572         uint16_t altitudeGps = 0;
573         uint16_t lightMich=0;
574     #endif /* CAYENNE_LPP */

```

FIGURE 18 – loraAppDecla

dans loraApp, on va ajouter la valeur dans le buffer

```
597 #else /* not CAYENNE_LPP */
598     humidity = (uint16_t)(sensor_data.humidity * 10); /* in %*10 */
599     temperature = (int16_t)(sensor_data.temperature);
600     lightMich = (uint16_t)(sensor_data.lightM); /* in hPa / 10 */
601
602     AppData.Buffer[i++] = AppLedStateOn;
603     AppData.Buffer[i++] = (uint8_t)((lightMich >> 8) & 0xFF);
604     AppData.Buffer[i++] = (uint8_t)(lightMich & 0xFF);
605     AppData.Buffer[i++] = (uint8_t)(temperature & 0xFF);
606     AppData.Buffer[i++] = (uint8_t)((humidity >> 8) & 0xFF);
607     AppData.Buffer[i++] = (uint8_t)(humidity & 0xFF);
608
609     if ((LmHandlerParams.ActiveRegion == LORAMAC_REGION_US915) || (LmHandlerParams.ActiveRegion == LORAMAC_REGION_AU915)
610         || (LmHandlerParams.ActiveRegion == LORAMAC_REGION_AS923))
611     {
612         AppData.Buffer[i++] = 0;
613         AppData.Buffer[i++] = 0;
614         AppData.Buffer[i++] = 0;
615         AppData.Buffer[i++] = 0;
616     }
}
```

FIGURE 19 – loraAppBuffer

Ensuite pour bien afficher les bonnes valeurs dans ttn il faut faire un custom payload formatter, on copie le repository formatter et on modifie de sorte à lire les bons bytes pour chaque capteur.

```
    data.lightMich = {
        displayName: 'Light ambient',
        unit: '%',
        value: ((input.bytes[0] << 8) + input.bytes[1])
    };
    data.temperature = {
        displayName: 'Internal temperature',
        unit: '°C',
        value: input.bytes[2] & 0x80 ? input.bytes[2] - 0x100 : input.bytes[2]
    };
    data.humidity = {
        displayName: 'R humidity',
        unit: 'Ohm',
        value: ((input.bytes[3] << 8) + input.bytes[4]) /10
    };
}
```

FIGURE 20 – ttnFormatterStm32

## B.2 MKRWAN1300

Bilbiothèque pour envoie data :

<https://github.com/arduino-libraries/MKRWAN/blob/master/examples/LoraSendAndReceive/LoraSendAndReceive.ino>

Dans un premier temps, nous devons lancer le code FirstConfig de la librairie "MKRWAN" de sorte à récupérer le device UID de notre carte. Celui-ci sera nécessaire pour l'ajout du device dans les devices de notre application TTN.

```

FirstConfiguration
  Serial.println("Failed to start module");
  while (1) {}

  Serial.print("Your module version is: ");
  Serial.println(modem.getVersion());
  if (modem.getVersion() != ARDUINO_FW_VERSION) {
    Serial.println("Please make sure that the latest modem firmware is installed.");
    Serial.println("To update the firmware upload the 'MKRWANFWUpdate_standalone.ino' sketch.");
  }
  Serial.print("Your device EUI is: ");
  Serial.println(modem.getDevEUI());

  int mode = 0;
  while (mode != 1 && mode != 2) {
    Serial.println("Are you connecting via OTAA (1) or ABP (2)?");
    while (!Serial.available());
    mode = Serial.read();
    mode = Serial.readStringUntil("\n").toInt();
  }

  int connected = 0;
  if (mode == 1) {
    Serial.println("Enter your APP EUI");
    while (!Serial.available());
    appEui = Serial.readStringUntil("\n");

    Serial.println("Enter your APP KEY");
    while (!Serial.available());
    appKey = Serial.readStringUntil("\n");

    appKey.trim();
    appEui.trim();

    connected = modem.joinOTAA(appEui, appKey);
  } else if (mode == 2) {
    Serial.println("Enter your Device Address");
    while (!Serial.available());
    devAddr = Serial.readStringUntil("\n");
  }

Arduino : FAST_MULTI_PAGE_WRITE
Modem   : CAN_GCCONFIG_MEMORY_BUFFER
Process flash
Done in 0.448 seconds

Write 19180 bytes to flash (500 pages)
-----| 500B (300/300 pages)
Done in 0.167 seconds

Verify 19180 bytes of flash with checksum.
Verify successful
Done in 0.016 seconds
CPU reset.

  
```

FIGURE 21 – firstConfigCodeArduino

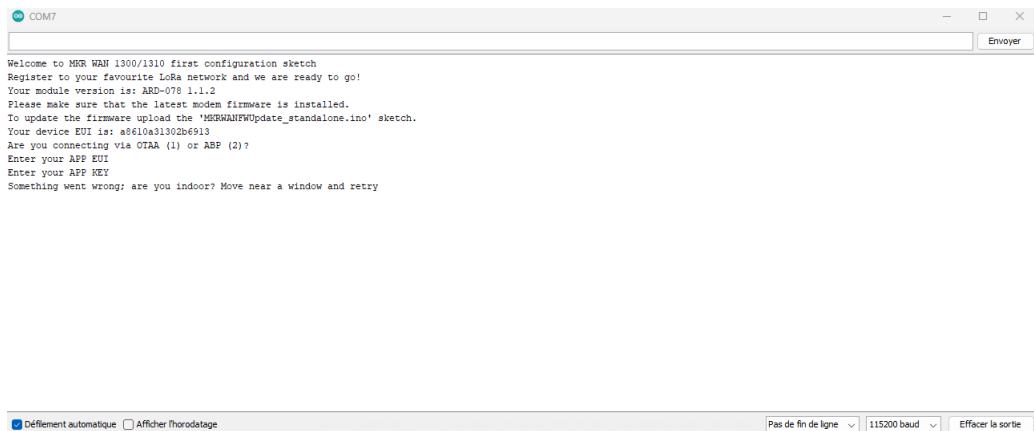


FIGURE 22 – progFirstConfig

On récupère le DevEUI du code du serial monitor, pour créer le noeud sur TTN  
On choisit appEUI sur TTN, à présent le noeud est créé sur TTN  
(Dans le noeud, general setting -> Join settings-> enable Resets joins)

Dans serial monitor,  
On choisit OTAA  
On entre appEUI On entre le appKey auto-généré sur TTN

## Register end device

Does your end device have a LoRaWAN® Device Identification QR Code? Scan it to speed up onboarding.

[Scan end device QR code](#)

[Device registration help](#)

### End device type

Input method [?](#)

- Select the end device in the LoRaWAN Device Repository  
 Enter end device specifics manually

End device brand [?](#) \* Model [?](#) \* Hardware Ver. [?](#) \* Firmware Ver. [?](#) \* Profile (Region) \*

Arduino SA		▼	Arduino MKR WAN...		▼	1.0		▼	1.2.3		▼	EU_863_870		▼
------------	--	---	--------------------	--	---	-----	--	---	-------	--	---	------------	--	---

#### Arduino MKR WAN 1310

LoRaWAN Specification 1.0.2, RP001 Regional Parameters 1.0.2, Over the air activation (OTAA), Class A



The Arduino MKR WAN 1310 is a development board that provides a practical and cost-effective solution to add LoRaWAN® connectivity for projects requiring long-range, low-power wireless communication. Sensors and actuators can be connected to the board through the analog, digital, UART, SPI, and I2C pins. The MKR WAN 1310 comes complete with an ATECC508 secure element, a battery charger, 2MByte SPI Flash, and power consumption as low as 104 uA.

[Product website](#)

Frequency plan [?](#) \*

Europe 863-870 MHz (SF9 for RX2 - recommended)

### Provisioning information

JoinEUI [?](#) \*

00 00 00 00 00 00 07 48

[Confirm](#)

To continue, please enter the JoinEUI of the end device so we can determine onboarding options

FIGURE 23 – configTtnArduino

**Provisioning information**

**JoinEUI** ⓘ \*

Reset

This end device can be registered on the network

**DevEUI** ⓘ \*

Generate 1/50 used

**AppKey** ⓘ \*

Generate

**End device ID** ⓘ \*

This value is automatically prefilled using the DevEUI

**After registration**

View registered end device

Register another end device of this type

**Register end device**

FIGURE 24 – configTtnArduino2

### B.2.1 Code Arduino

```

float mesureHumidite(){// Fonction pour la mesure de l'humidité
    // Déclaration des variables propres à la mesure de l'humidité
    float Vin = Vbroche; // Tension d'entrée du pont diviseur de tension
    float valeurAnalogiqueHumidite = analogRead(brocheAnalogiqueHumidite); // Va
    float Vout = (valeurAnalogiqueHumidite/1023)*Vbroche; // Valeur de la tensio
    float R1 = R2*((Vin/Vout)-1); // Valeur de la résistance liée à la source du
    Serial.print("Résistance humidité : ");
    Serial.println(R1);
    return R1;
}

float mesureLumiere(){// Fonction pour la mesure de la Lumière
    float valeurAnalogiqueLumiere = analogRead(brocheAnalogiqueLumiere); // Vale
    float RatioLumiere = (valeurAnalogiqueLumiere/1023)*100; // Pourcentage d'éc
    Serial.print("% Lumière : ");
    Serial.println(RatioLumiere);
    return RatioLumiere;
}

float mesureTemperature(){ // Fonction pour la mesure de la Température
    float valeurAnalogiqueTemperature = analogRead(brocheAnalogiqueTemperature);
    float Vt = (valeurAnalogiqueTemperature/1023)*Vbroche; // Tension proportion
    float T = (-183.4862*Vt)+186.697247; // Conversion en température
    Serial.print("T : ");
    Serial.println(T);
    return T;
}

```

FIGURE 25 – Instrumentation code MKRWAN

```

int connected = modem.joinOTAA(appEui, appKey);
if (!connected) {
    Serial.println("Something went wrong; are you
        while (1) {}

// Set poll interval to 60 secs.
modem.minPollInterval(60);

```

FIGURE 26 – connexion code MKRWAN

```

void loop() {
    int temp=mesureTemperature();
    int humidity=mesureHumidite();
    int light=mesureLumiere();

    String data_send = String(temp) + "," + String(humidity) + "," + String(light);

    int err;
    modem.beginPacket(); // Début de l'envoie
    modem.print(data_send);
    err = modem.endPacket(true);

    if (err > 0) {
        Serial.println("Message sent correctly!");
    } else {
        Serial.println("Error sending message :(");
    }

    delay(120000); // 2min
}

```

FIGURE 27 – envoi message MKRWAN

### B.3 Visualisation des données

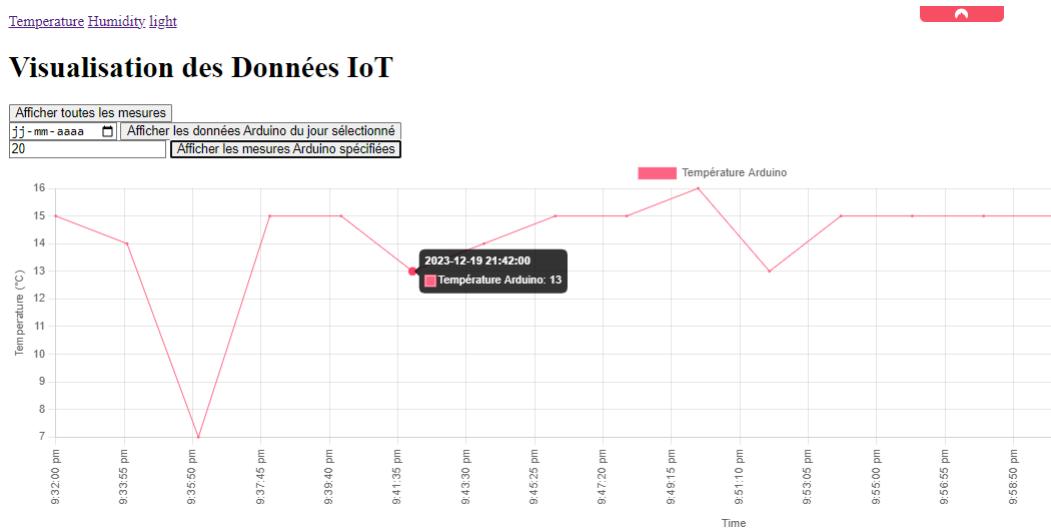


FIGURE 28 – Graphique température 20 dernières valeurs en temps réel

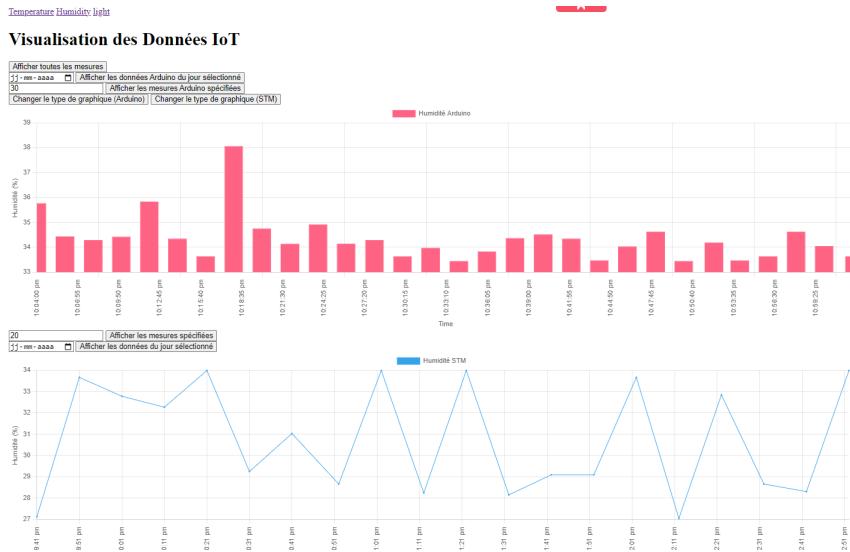


FIGURE 29 – Graphique humidité en temps réel

## B.4 Payload Formatter

**Setup**

Formatter type\*

Formatter code\*

```

1 function decodeUplink(input) {
2     var data = {};
3     switch (input.fPort) {
4         case 2:
5             data.lightAmbient = {
6                 displayName: 'Light ambient',
7                 unit: '%',
8                 value: ((input.bytes[0] << 8) + input.bytes[1])
9             };
10            data.temperature = {
11                displayName: 'Temperature',
12                unit: '°C',
13                value: input.bytes[2] & 0x80 ? input.bytes[2] - 0x100 : input.bytes[2]
14            };
15            data.humidity = {
16                displayName: 'R humidity',
17                unit: 'Ohm',
18                value: ((input.bytes[3] << 8) + input.bytes[4]) / 10
19            };
20        return {
21            data: data
22        };
23    default:
24        return {
25            errors: ['Unknown FPort']
26        };
27    }
28}

```

**Setup**

Formatter type\*

Formatter code\*

```

1 function Decoder(bytes, port) {
2     var decodedString = String.fromCharCode.apply(null, bytes);
3     var values = decodedString.split(',');
4
5     return {
6         temperature: parseFloat(values[0]),
7         humidity: parseFloat(values[1]),
8         light: parseFloat(values[2])
9     };
10}

```

(a) Payload Formatter STM32      (b) Payload Formatter MKRWAN1300

FIGURE 30 – Payload Formatter

## B.5 Decoded payload

```
"decoded_payload": {  
    "humidity": {  
        "displayName": "R humidity",  
        "unit": "Ohm",  
        "value": 1140.9  
    },  
    "lightMich": {  
        "displayName": "Light ambient",  
        "unit": "%",  
        "value": 0  
    },  
    "temperature": {  
        "displayName": "Internal temperature",  
        "unit": "°C",  
        "value": 19  
    }  
}
```

(a) Decoded payload STM32 WL55JC1

```
"decoded_payload": {  
    "humidity": 28228.57,  
    "light": 22,  
    "temperature": 18  
},  
..
```

(b) Decoded payload MKRWAN1300

FIGURE 31 – Decoded payload

## B.6 AirTime-Frequency-SF-SNR-RSSI STM32 WL55JC1

```
"settings": {  
    "data_rate": {  
        "lora": {  
            "bandwidth": 125000,  
            "spreading_factor": 7,  
            "coding_rate": "4/5"  
        }  
    },  
    "frequency": "867100000",  
    "timestamp": 2813566955,  
    "time": "2023-12-19T21:06:26.751312Z"  
},  
"received_at": "2023-12-19T21:06:26.785576857Z",  
"consumed_airtime": "0.051456s",
```

(a) AirTime-Frequency-SF STM32 WL55JC1

```
"rx_metadata": [  
    {  
        "gateway_ids": {  
            "gateway_id": "kerlink-irisib",  
            "eui": "7276FF0039030828"  
        },  
        "time": "2023-12-19T21:08:46.858793Z",  
        "timestamp": 2953674347,  
        "rssi": -51,  
        "channel_rssi": -51,  
        "snr": 6.8,  
    }]
```

(b) SNR-RSSI STM32 WL55JC1

FIGURE 32 – AirTime-Frequency-SF-SNR-RSSI STM32 WL55JC1

## B.7 AirTime-Fréquence-SF-SNR-RSSI MKRWAN1300

```

"settings": {
  "data_rate": {
    "lora": {
      "bandwidth": 125000,
      "spreading_factor": 7,
      "coding_rate": "4/5"
    }
  },
  "frequency": "867500000",
  "timestamp": 892211507,
  "time": "2023-12-19T21:46:00.364610Z"
},
"received_at": "2023-12-19T21:46:00.390549834Z",
"confirmed": true,
"consumed_airtime": "0.061696s",

```

(a) AirTime-Fréquence-SF MKRWAN1300

```

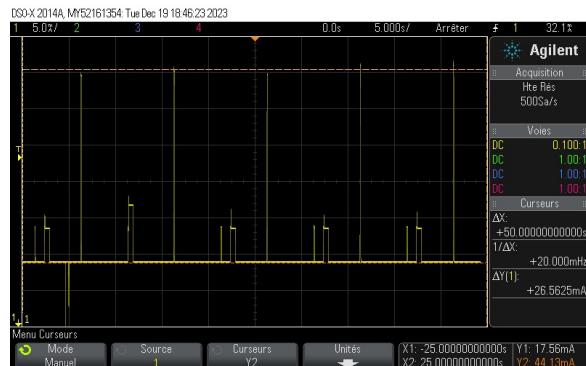
"rx_metadata": [
  {
    "gateway_ids": {
      "gateway_id": "kerlink-irisib",
      "eui": "7276FF0039030828"
    },
    "time": "2023-12-19T21:46:00.364610Z",
    "timestamp": 892211507,
    "rssi": -63,
    "channel_rssi": -63,
    "snr": 7
  }
]

```

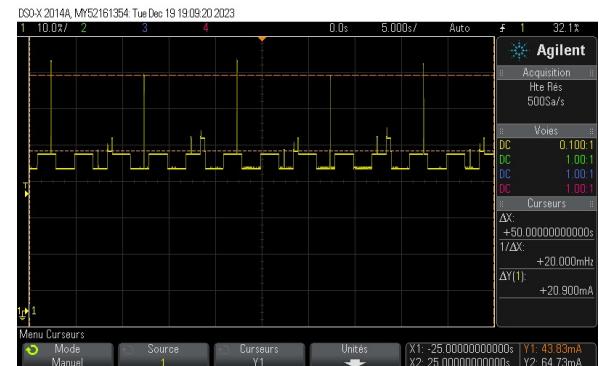
(b) SNR-RSSI MKRWAN1300

FIGURE 33 – AirTime-Fréquence-SF-SNR-RSSI MKRWAN1300

## B.8 Consommation électrique du noeud



(a) Consommation électrique du noeud STM32 pour 6 bytes envoyés



(b) Consommation électrique du noeud STM32 pour 50 bytes envoyés

FIGURE 34 – Consommation électrique du noeud