

# Compte Rendu de TP :

## UE RS BE - TP BE Ascenseur



# SOMMAIRE

<b>Introduction</b>	<b>2</b>
<b>Schéma fonctionnel</b>	<b>2</b>
<b>Schéma de câblage</b>	<b>3</b>
Partie intérieure : ascenseur	3
Partie extérieure	4
<b>Initialisation du STM32L476RG</b>	<b>4</b>
Partie extérieure	4
Partie intérieure : ascenseur	5
<b>Capteur de température &amp; d'humidité DHT22 (One Wire)</b>	<b>6</b>
<b>Capteur de luminosité Grove TSL2561 (I2C)</b>	<b>12</b>
<b>Module Wifi WeMos D1 Mini ESP8266 (UART)</b>	<b>14</b>
<b>Assemblage du projet</b>	<b>19</b>
<b>Conclusion</b>	<b>19</b>

# 1. Introduction

**Nom du projet :** Ascenseur

**Matériel :** Nous utilisons deux cartes STM32L476RG, un Groove LCD RGB Backlight V4.0 (I2C), un capteur de température & d'humidité DHT22 (OneWire), un capteur de luminosité Grove TSL2561 (I2C) et deux modules Wifi WeMos D1 Mini ESP8266 (UART).

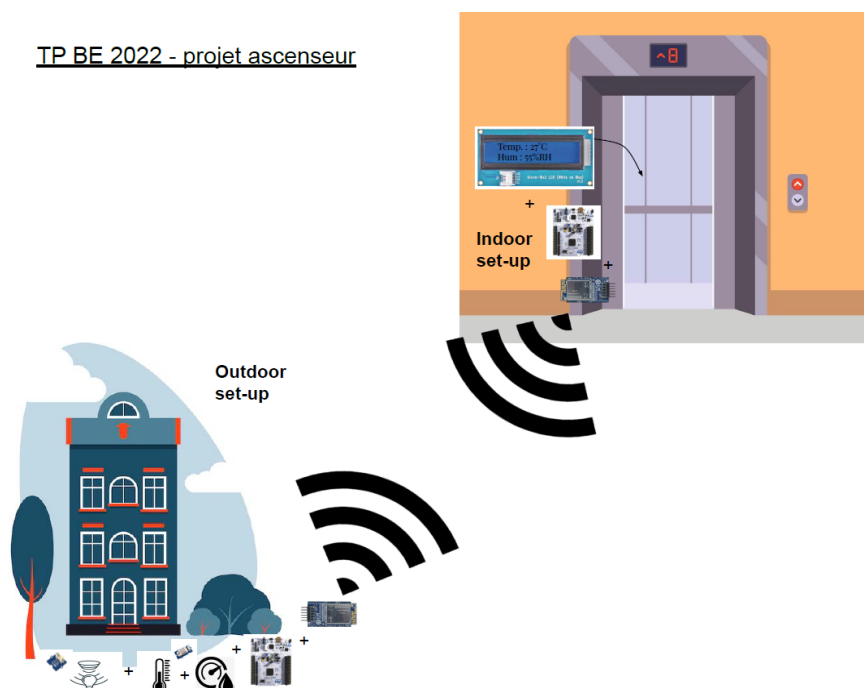
**Logiciel :** Ce TP utilisera STM32 Cube IDE V1.8.0. Tout le code utilisé pour ce projet est disponible sur Github.

**But :** L'objectif de ce TP est de communiquer la météo extérieure (température en °C, humidité de l'air en %RH et luminosité en lux) à l'utilisateur de l'ascenseur d'un immeuble via un écran LCD.

## 2. Schéma fonctionnel

Notre projet est divisé en deux parties. A l'intérieur de l'ascenseur, est installé un STM32, un module Wifi et l'écran LCD. A l'extérieur, se trouve un STM32, un module Wifi, le capteur de luminosité et le capteur de température et humidité. La partie extérieure doit communiquer via Wifi avec la partie intérieure les données mesurées par les capteurs. Ces données seront affichées par la partie intérieure sur un écran LCD.

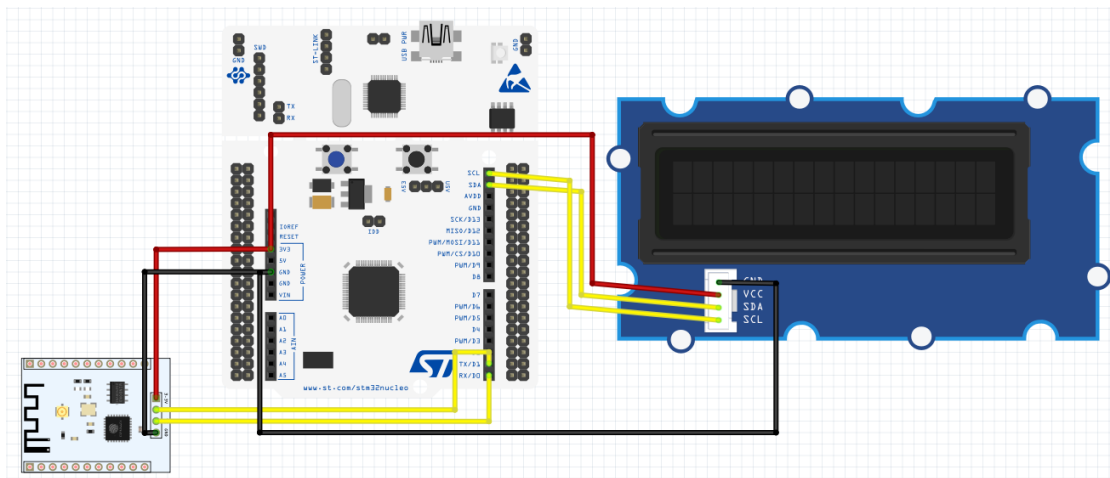
TP BE 2022 - projet ascenseur



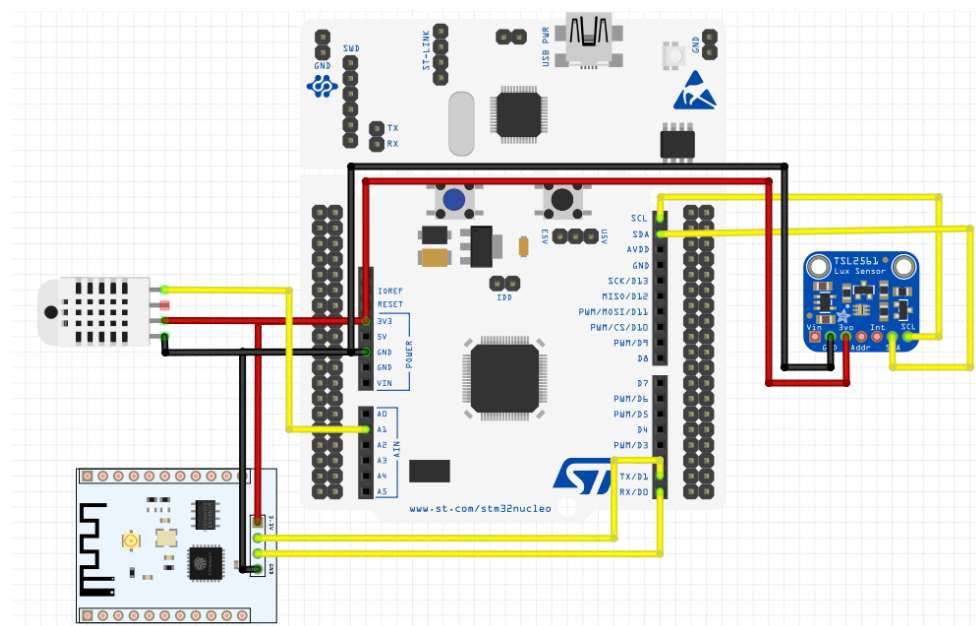
### 3. Schéma de câblage

Fritzing est utilisé pour réaliser ces schémas de câblage. Celui-ci n'offrant pas la Nucleo STM32L476RG, celle-ci a été remplacée par un équivalent. Le raccordement des Pin peut donc s'avérer inexact par rapport à la carte réellement utilisée. Nous utilisons également un Base Shield V2 de Grove, non représenté ici.

### A. Partie intérieure : ascenseur

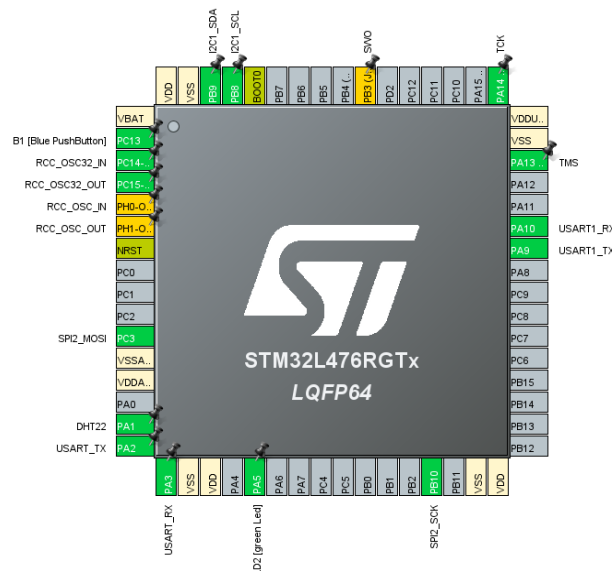


## B. Partie extérieure



## 4. Initialisation du STM32L476RG

### A. Partie extérieure



Tout d'abord notre capteur de température & d'humidité DHT22 fonctionne avec un protocole de communication One Wire. On lui attribue une broche sur le microcontrôleur qui est la broche PA1. Cette broche doit être mise à 1 au repos d'après la datasheet donc on va la paramétrer en pull-up.

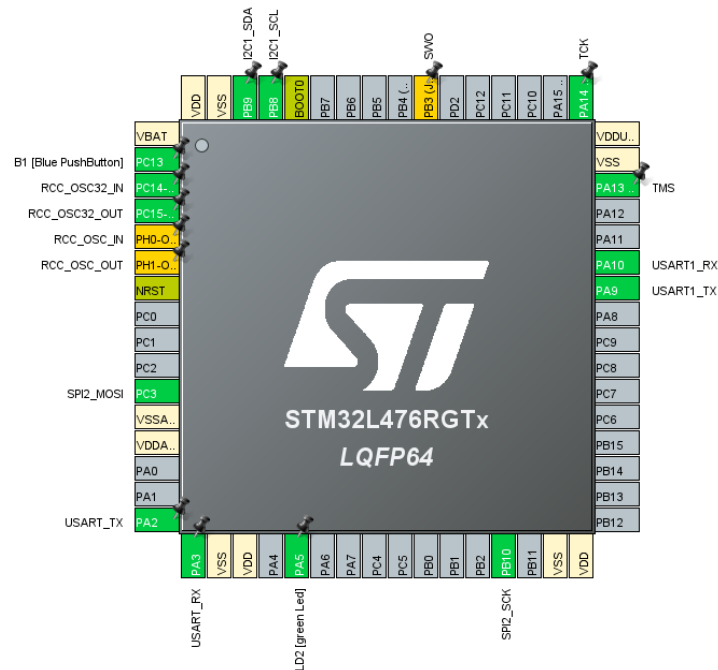
Ensuite, comme notre capteur de luminosité TSL2561 fonctionne avec un protocole de communication I2C, il faut donc paramétrer un port SDA et SCL sur le microcontrôleur pour permettre la communication avec notre capteur. On peut voir les ports sur le schéma au-dessus (PB8 & PB9).

Finalement, notre module Wifi WeMos D1 Mini ESP8266 utilise une communication UART pour dialoguer avec la carte STM32L476RG. On va donc initialiser les ports TX et RX pour permettre cette communication. On peut voir les ports sur le schéma au-dessus (PA10 & PA9). Ne pas oublier que la communication UART est asynchrone donc pour pouvoir communiquer entre 2 dispositifs, il faut qu'ils aient tous 2 le même baud rate. Dans la datasheet du module wifi on a vu qu'il a un baud rate de 74880 bauds donc nous paramétrons le baud rate du microcontrôleur à cette vitesse.

Remarque : Contrairement à la communication I2C où la broche SDA du microcontrôleur se connecte avec la broche SDA d'un capteur I2C et la broche SCL du microcontrôleur se connecte avec la broche SCL d'un capteur I2C, pour la communication UART les broches RX et TX du capteur et du microcontrôleur doivent s'entrelacer. La broche RX du capteur doit être connectée à la broche TX

du microcontrôleur et la broche TX du capteur doit être connectée à la broche RX du microcontrôleur.

## B. Partie intérieure : ascenseur



Les broches initialisées ici sont identiques à celles initialisées précédemment sauf que la broche GPIO, pour la communication One Wire avec le DHT22, a été enlevée :

Tout d'abord, notre écran LCD fonctionne avec un protocole de communication I2C. On doit donc paramétrer un port SDA et SCL sur le microcontrôleur pour permettre la communication avec l'écran LCD. On peut voir les ports sur le schéma au-dessus (PB8 & PB9).

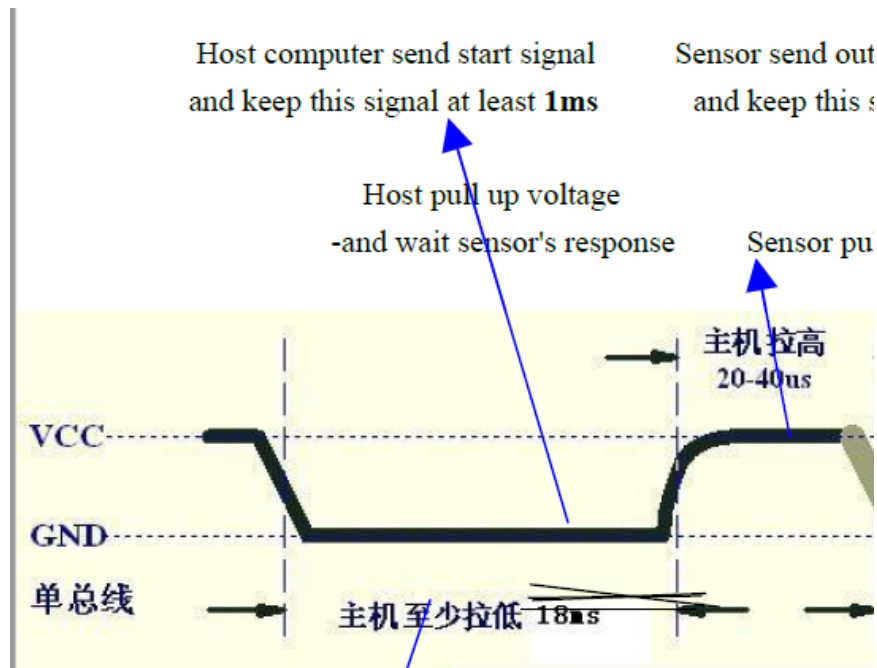
Finalement, comme précédemment, on a notre module wifi WeMos D1 Mini ESP8266 qui utilise une communication UART pour communiquer avec le microcontrôleur STM32L476RG. On va donc initialiser les ports TX et RX pour permettre cette communication. On peut voir les ports sur le schéma au-dessus (PA10 & PA9) et mettre également le baud rate du microcontrôleur à 74880 bauds.

## 5. Capteur de température & d'humidité DHT22 (One Wire)

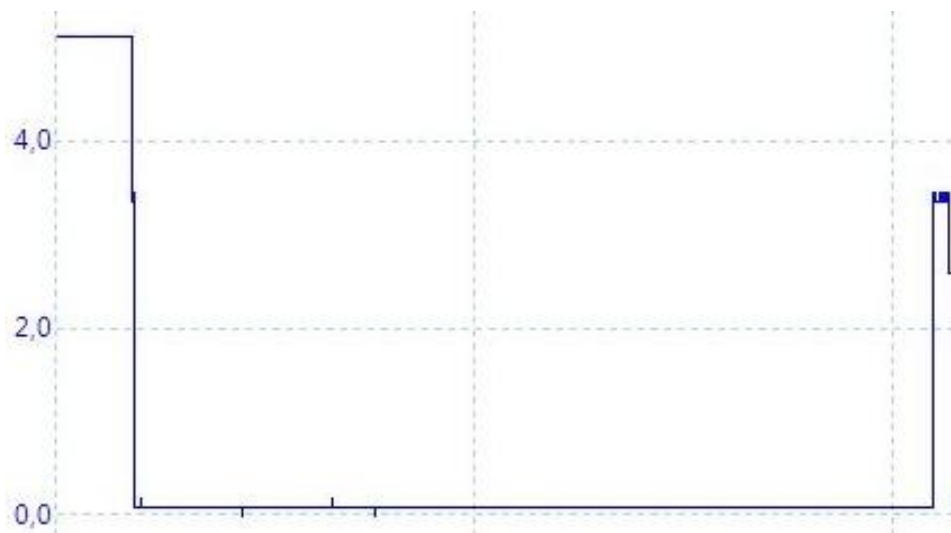
Tout d'abord, on va créer une fonction pour notre timer en microsecondes.

```
void Delay_us(uint16_t us)
{
    __HAL_TIM_SET_COUNTER(&htim2, 0);
    while(__HAL_TIM_GET_COUNTER(&htim2) < us);
}
```

Ensuite, on regarde la datasheet du DHT22 pour voir comment celui-ci s'initialise :



La datasheet nous indique qu'il faut que la broche du signal reliant le microcontrôleur et le DHT22 soit mise à 0 pendant au minimum 1 ms et au maximum 18 ms, puis doit être mise à 1 pendant 20 à 40  $\mu$ s.



Ici, si on compare notre graphe obtenu avec le picoscope avec celui de la datasheet, on voit qu'on est bien à 1 au repos, puis à 0 pendant exactement 1.2 ms et on remonte à 1 pendant 30  $\mu$ s. Par contre, on voit qu'à droite sur la photo, au

moment où on veut repasser à 0, la tension descend un peu avant de vraiment descendre à 0. Ce petit changement de hauteur est dû au PIN qu'on fait changer de sens. On le change en entrée.

Voici les fonctions que l'on a utilisé pour initialiser le DHT22 :

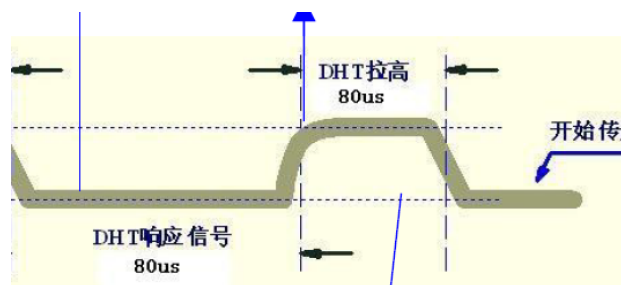
```
void Set_Pin_Output (GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin)
{
    GPIO_InitTypeDef GPIO_InitStructure = {0};
    GPIO_InitStructure.Pin = GPIO_Pin;
    GPIO_InitStructure.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOx, &GPIO_InitStructure);
}

void Set_Pin_Input (GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin)
{
    GPIO_InitTypeDef GPIO_InitStructure = {0};
    GPIO_InitStructure.Pin = GPIO_Pin;
    GPIO_InitStructure.Mode = GPIO_MODE_INPUT;
    GPIO_InitStructure.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(GPIOx, &GPIO_InitStructure);
}

void DHT22_start(void)
{
    Set_Pin_Output(DHT22_GPIO_Port, DHT22_Pin); //Mettre broche en sortie pour initialiser DHT22
    HAL_GPIO_WritePin(DHT22_GPIO_Port, DHT22_Pin, GPIO_PIN_RESET);
    Delay_us(1200);
    HAL_GPIO_WritePin(DHT22_GPIO_Port, DHT22_Pin, GPIO_PIN_SET);
    Delay_us(30);
    Set_Pin_Input(DHT22_GPIO_Port, DHT22_Pin); //Mettre broche en entrée pour recevoir réponse DHT22
}
```

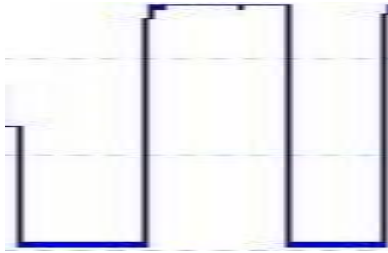
Comme on le voit, nous avons une fonction pour mettre une broche du microcontrôleur en sortie puis une autre fonction pour mettre une broche en entrée. Ensuite, nous avons une 3<sup>ème</sup> fonction qui va initialiser le DHT22 selon les instructions de sa datasheet expliquées précédemment.

Après l'initialisation du DHT22, nous avons analysé sa réponse. Il va répondre en 2 phases distinctes :



Tout d'abord, si l'initialisation a bien fonctionné alors elle va répondre au microcontrôleur par un signal qui sera à 0 pendant 80 µs puis à 1 pendant 80 µs et enfin à 0 pendant 50 µs avant d'avoir une nouvelle mise à 1 qui sera notre premier bit de donnée fourni par le DHT22.





Ici comme sur la datasheet le capteur répond par un 0 puis un 1 et on revient à 0 avant de recevoir le premier bit. Par contre, nous n'avons pas exactement les mêmes valeurs. On est d'abord à 0 pendant 73  $\mu$ s puis à 1 pendant 78  $\mu$ s d'après nos mesures contre 80  $\mu$ s d'après la datasheet puis nous sommes bien à 0 pendant 50  $\mu$ s avant de recevoir notre premier bit. On est donc assez proche des valeurs données par la datasheet.

Pour l'analyse de la réponse du capteur, on va programmer cette fonction pour la réception de la réponse du DHT22 :

```
int DHT22_Answer(void)
{
    //Je choisis toujours 10 us de plus pour les temps pour une marge d'erreur
    int temps = 0, reponse = 0;
    //verifier qu'on est bien a 0 pendant 80 us
    while((HAL_GPIO_ReadPin(DHT22_GPIO_Port, DHT22_Pin) == GPIO_PIN_RESET) && (temps < 90))
    {
        Delay_us(1);
        temps++;
    }

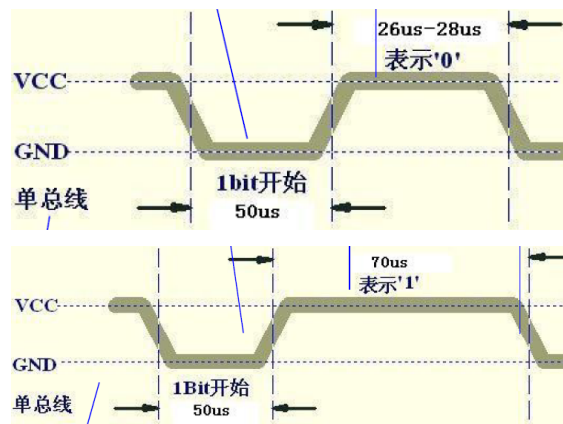
    if(temps < 90)//si on est bien rester à 0 pendant de 80 us alors OK
    {
        temps = 0;
        //verifier qu'on est bien a 1 pendant 80 us
        while((HAL_GPIO_ReadPin(DHT22_GPIO_Port, DHT22_Pin) == GPIO_PIN_SET) && (temps < 90))
        {
            Delay_us(1);
            temps++;
        }

        if(temps < 90)//si on est bien rester à 1 pendant 80 us alors OK
        {
            //début de la trame de donnée 50 us à 0 puis 1er bit de donnée
            temps = 0;

            //vérifier la reception du premier bit
            while((HAL_GPIO_ReadPin(DHT22_GPIO_Port, DHT22_Pin) == GPIO_PIN_RESET) && (temps < 60))
            {
                Delay_us(1);
                temps++;
            }
            if(temps < 60)
            {
                reponse = 1;
            }
        }
    }

    Delay_us(40);
    return reponse;
}
```

Ensuite, on va recevoir un signal qui sera une variation de mise à 1 et de mise à 0. Ce signal contiendra nos données d'humidité et de température. Le décryptage du signal se fera ensuite de la manière suivante :



Si le signal est à 1 pendant 26 à 28  $\mu\text{s}$  alors notre bit est un 0, sinon c'est que notre bit est un 1.



Voici une partie des données qu'on a reçues sur le picoscope. Comme sur la datasheet, on voit bien que la durée où on est à l'état 1 varie : quand la durée est courte on a un bit à 0 sinon on a un bit à 1.

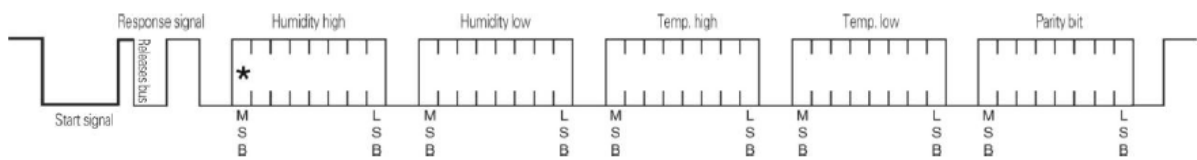
Ensuite pour décrypter ce message venant du DHT22 on va donc utiliser cette fonction :

```

void DHT22_Read_Data (uint8_t *data)
{
    int i, k;
    for (i=0;i<8;i++)
    {
        if (HAL_GPIO_ReadPin (DHT22_GPIO_Port, DHT22_Pin) == GPIO_PIN_RESET)
        {
            (*data)&= ~(1<<(7-i)); //Mettre bit à 0
            while(!(HAL_GPIO_ReadPin(GPIOA, GPIO_PIN_1)));
            Delay_us(40);
        }
        else
        {
            (*data)|= (1<<(7-i)); //Mettre bit à 1
            for (k=0;k<1000;k++)
            {
                if (HAL_GPIO_ReadPin (DHT22_GPIO_Port, DHT22_Pin) == GPIO_PIN_RESET)
                {
                    break;
                }
            }
            while(!(HAL_GPIO_ReadPin(DHT22_GPIO_Port, DHT22_Pin)));
            Delay_us(40);
        }
    }
}

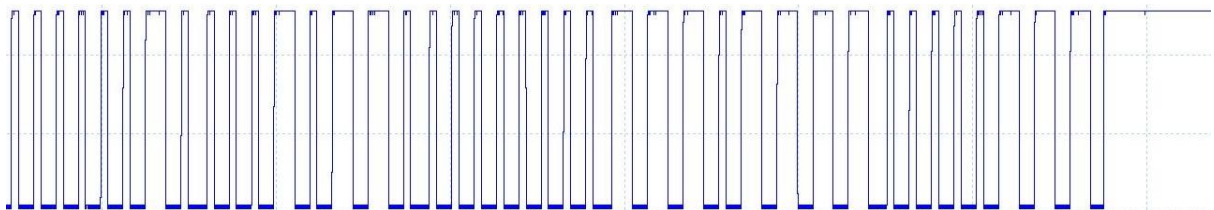
```

Enfin d'après la datasheet les données seront sous la forme suivante :



**Pic5:** AM2302 Single-bus communication protocol

On aura donc 16 bits pour la valeur de l'humidité puis 16 bits pour la valeur de la température et 8 bits de parité à la fin qui servira à vérifier que l'on n'ait pas d'erreurs de transmission. Normalement la valeur de l'octet de parité est égale à la somme des 4 octets de données précédents.



Ici sur notre résultat vu au picoscope comme sur la datasheet, on voit bien qu'on reçoit 40 bits qui seront interprétés comme expliqué précédemment .

On a ensuite programmé le code de la manière suivante pour lire les données fournies par le DHT22 :

```

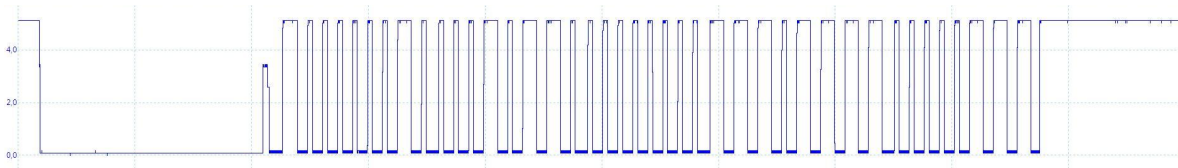
if(reponse == 1)
{
    //Commencer à acquérir les 5 différents octets
    DHT22_Read_Data(&HH);
    DHT22_Read_Data(&HL);
    DHT22_Read_Data(&TH);
    DHT22_Read_Data(&TL);
    DHT22_Read_Data(&SUM);

    check = HH + HL + TH + TL;
    if(check == SUM)//vérifier qu'il n'y a pas d'erreur dans les données reçues
    {
        //combiner 2 octets d'humidité et diviser résultat par 10 pour avoir humidité en %
        Humidite = (float) ((HH<<8) | HL) / 10;
        //combiner 2 octets de température et diviser résultat par 10 pour avoir température en °C
        Temperature = (float) ((TH<<8) | TL) / 10;
    }
}

```

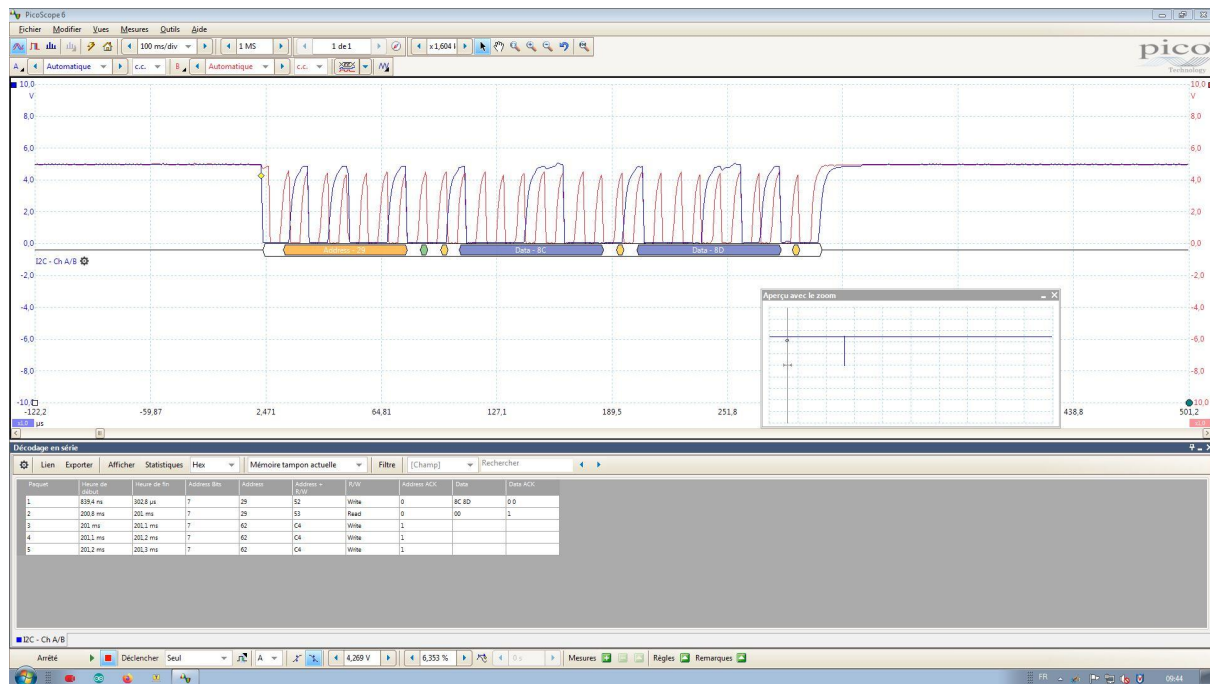
Comme expliqué, on va avoir 5 variables de 8 bits chacune qui vont prendre les valeurs des 5 octets envoyés par le DHT22 puis on vérifie qu'il n'y ait pas d'erreurs. Ensuite, on fusionne les 2 octets de température et les 2 octets d'humidité et on divise les 2 valeurs obtenues par 10 car d'après la datasheet l'indique.

Voici une photo de notre trame entière :



## 6. Capteur de luminosité Grove TSL2561 (I2C)

Ce capteur de luminosité provient du fabricant Grove et dialogue en I2C. Il est intégré au présent projet comme l'indique le schéma de câblage précédent. Premièrement, l'idée a été de réutiliser le code réalisé en TP de base pour le SHT31 (cf Github). Les variables ont ensuite été adaptées à notre nouveau capteur. Cette méthode fonctionne pour l'adresse cependant nous remarquons un problème lors de la réception des données :



Le capteur répond correctement à la fonction HAL\_I2C\_Master\_Transmit comme l'atteste la capture d'écran du picoscope. Cependant, nous obtenons un NACK (cf tableau) avec HAL\_I2C\_Master\_Receive juste après l'envoi de l'adresse. Malgré nos efforts, nous ne parviendrons jamais à réceptionner de la data avec ces fonctions HAL\_I2C. L'une des hypothèses de cet échec de notre V0 est que le format proposé par la fonction Master\_Receive ne convient pas à ce capteur notamment à cause de la condition de start répétée en milieu de trame (cf datasheet).

Nous avons ensuite décidé de travailler via un code proposé par [Scholtyssek](#). Ce code est originellement prévu pour fonctionner en dehors de Cube IDE avec un STM32F4. Nous avons quand même tenté d'utiliser ce code via Cube IDE mais de nombreux conflits entre Cube IDE et les initialisations déjà réalisées par l'auteur sont apparus. Afin de contourner le problème, nous sommes passés sur Visual Studio Code ce qui a été l'occasion de s'apercevoir que les bibliothèques d'un STM32F4 et STM32L4 sont complètement différentes et que la portabilité entre les deux modèles était donc impossible sans revoir en profondeur le code. Nous avons donc dû abandonner cette V4.

Après de nombreuses recherches, nous avons pu trouver un nouveau code correspondant à un TSL2581 sur [Waveshare](#). Ce code n'est pas prévu pour notre capteur mais l'analyse de la datasheet du TSL2581 permet de s'apercevoir de nombreuses similarités. Nous utiliserons donc ce code pour la suite du projet.

Il est composé des fichiers automatiquement générés par Cube IDE ainsi que de deux nouveaux fichiers : TSL2561.c et .h. Ces fichiers contiennent toutes les fonctions et adresses nécessaires au fonctionnement du capteur. Le fichier en .h contient donc toutes les initialisations des registres ainsi que des constantes nécessaires à la conversion de la valeur rapportée par le capteur. Toutes ses

informations sont disponibles sur la datasheet. Le fichier en .c contient lui plusieurs fonctions :

- I2C\_DEV\_Write et I2C\_DEV\_Read qui permettent d'écrire et lire sur le bus I2C
- I2C\_DEV\_init qui permet d'initialiser tous les registres
- Reload\_register qui permet de gérer les bits d'interruption et l'activation des channels
- SET\_Interrupt\_Threshold qui permet de calculer les temps d'intégration pour chaque conversion
- Read\_Channel qui permet d'interroger les registres de data du channel 1 et 0 ainsi que de calculer la valeur obtenue sur le channel grâce à une formule fournie par le constructeur
- calculateLux qui permet de convertir la valeur obtenue par le capteur en une valeur en lux

Le fichier main.c est maintenant modifié pour réaliser une demande de mesure et retourner une valeur en lux. En dehors de la fonction principale, une nouvelle fonction permettant de vérifier si une interruption est en cours (permet de surveiller la présence d'une différence significative par rapport aux registres THRESH dans la valeur trouvée en lux) est créée afin de veiller à la conformité des valeurs. Dernièrement, la fonction calculateLux puis Read\_gpio\_interrupt sont appelées en boucle pour fournir une valeur de luminosité fiable en permanence.

Nous passons maintenant aux tests et constatons que nous ne fonctionnons pas correctement. Nous vérifions donc la logique de notre code et testons de nouvelles initialisations pour de multiples registres sans succès. Nous récupérons effectivement une valeur via nos variables mais celle-ci ne varie pas après la 2e exécution de boucle. De plus, nos valeurs ne sont pas celles évoquées par la datasheet :

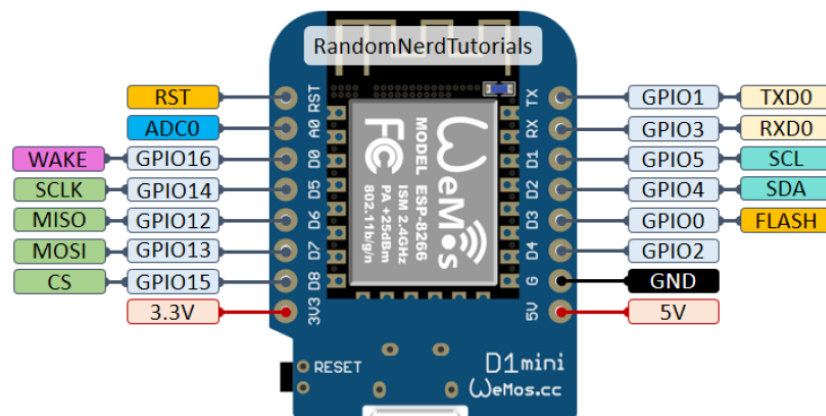
Name	Type	Value
(x)= iGain	uint16_t	2
(x)= tIntCycles	uint16_t	148
(x)= chScale0	unsigned long	134219285
(x)= chScale1	unsigned long	536969192
(x)= channel1	unsigned long	1
(x)= channel0	unsigned long	1000
(x)= temp	unsigned long	134219157
(x)= ratio1	unsigned long	0
(x)= ratio	unsigned long	536969179
(x)= lux_temp	unsigned long	536969176
(x)= b	unsigned int	805306379
(x)= m	unsigned int	2000

Malgré de nombreuses tentatives, nous n'arriverons jamais à corriger notre erreur sur notre V8. La principale hypothèse est que l'un de nos registres est mal initialisé. Le code d'origine ayant été conçu pour un capteur aux registres différents, nous avons dû transcrire tous les registres au TSL2561 ce qui est propice aux erreurs.

## 7. Module Wifi WeMos D1 Mini ESP8266 (UART)

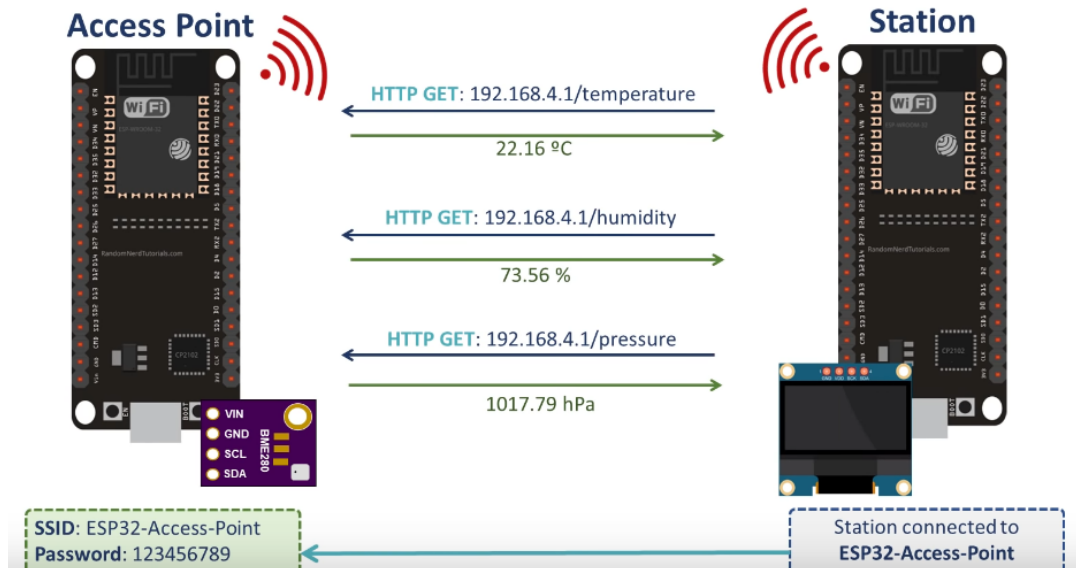
Ce module Wifi possède 16 pin dont 2 pin d'alimentation (3,3V et 5V), 1 pin de ground, 1 pin ADC (Analogue to digital converter), 1 pin RST (reset) et 11 pin GPIO (General Pin Input Output). Parmi ces 11 Pin, 2 sont dédiées de base à la communication UART (Rx et Tx) et peuvent directement être utilisées. Les autres Pin peuvent être utilisés pour de l'SPI et de l'I2C mais contrairement aux Pin UART, ces autres Pin doivent être configurés avec un programme pour pouvoir utiliser l'SPI ou l'I2C.

### ESP8266 WeMos D1 Mini Pinout



Ce module wifi peut-être utilisé de 2 manières. La première méthode est de la programmer directement avec un IDE et dans ce cas là, il est complètement indépendant. La deuxième méthode est de lui téléverser des firmware AT fournis par le constructeur et qui permettront de commander le module avec un microcontrôleur en utilisant des commandes spécifiques. Cette méthode contrairement à la première rend le module Wifi complètement dépendant à notre STM32 et l'on pourra donc dicter ses actions à la lettre. C'est donc cette deuxième méthode que l'on va choisir car dans notre projet, notre microcontrôleur doit gérer tous les capteurs et modules qui lui sont reliés pour permettre une bonne acquisition de certaines données et l'envoi à un autre microcontrôleur.

Tout d'abord, il faut savoir que le module Wifi, en partie extérieure, doit être initialisé en tant que point d'accès. Il devra créer un point d'accès Wifi qui permettra à toute personne ayant le mot de passe de s'y connecter. Ensuite, le module Wifi intérieur doit être configuré comme station. Il devra se connecter à notre module extérieur qui est en mode point d'accès pour récupérer les données acquises à l'extérieur. Voici un schéma illustrant ce principe :



Remarque : Les modules sur l'image ne correspondent pas aux nôtres et servent uniquement d'illustration explicative de la communication entre deux modules Wifi.

Pour l'initialiser on doit d'abord brancher le microcontrôleur au port USB de l'ordinateur et aussi l'alimenter car le branchement avec l'ordinateur ne l'alimente pas. Ensuite on va sur le site du constructeur où on doit télécharger leur logiciel de téléversement de firmware AT vers notre module wifi qui s'appelle "flash\_download\_tool" et on doit aussi télécharger le package des firmware AT qui s'appelle ESP8266\_NONOS\_SDK. Ensuite après avoir suivi leur guide d'initialisation du module, et après avoir fini cette initialisation on doit tester les commandes AT pour voir si le module fonctionne bien avec ce package installer. Pour tester ce module, nous devons connecter les broches UART avec un autre dispositif et envoyer des commandes AT via cette communication UART puis attendre la réponse du module Wifi. Nous n'utilisons que cette communication UART car après avoir téléversé les firmware AT, seule cette communication est disponible. La communication SPI et I2C n'est disponible que lorsque le module est indépendant.

On branche donc le module wifi avec notre STM32 en n'oubliant pas de configurer notre microcontrôleur avec un baud rate de 74 880 bauds qui est le baud rate de notre module wifi. On utilise la fonction HAL\_UART\_Transmit() pour envoyer les



commandes AT depuis le STM32 vers le module Wifi et la fonction HAL\_UART\_Receive() pour attendre la réponse du module Wifi.

Voici quelques commandes AT de base :

-**Test AT Startup** : STM32 commande = "AT" => Module wifi réponse = "OK"

-**Restart a module** : STM32 commande = "AT+RST" => Module wifi réponse = "OK"

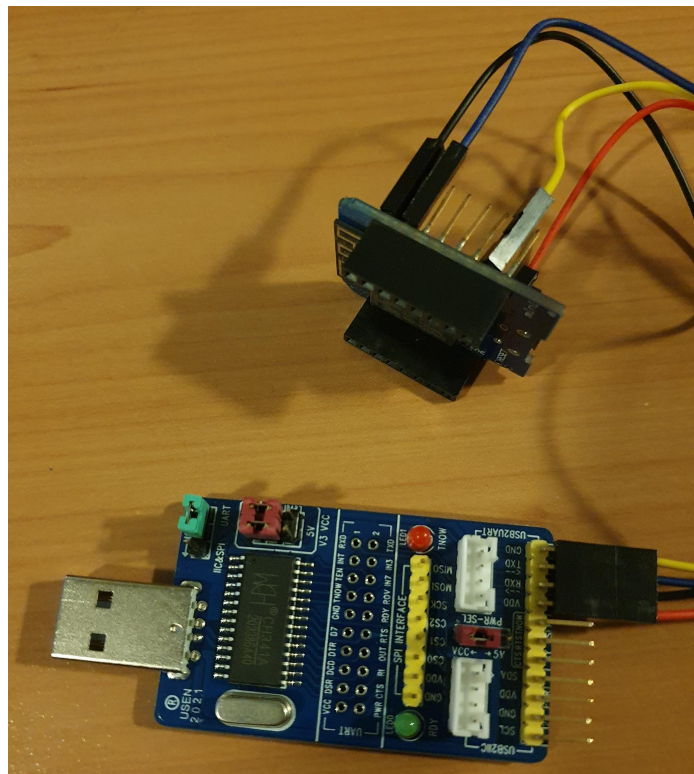
-**Check Version Information** :

STM32 commande = "AT+GMR" => Module wifi réponse :

```
<AT version info>
<SDK version info>
<compile time>
<Bin version>

OK
```

On regarde donc si le module fonctionne en envoyant la commande "AT" depuis le microcontrôleur avec la fonction HAL\_UART\_Transmit() et on attend la réponse du module avec HAL\_UART\_Receive(). On se rend compte qu'on ne reçoit jamais la réponse du module Wifi. On a donc décidé d'acheter un sniffer pour tester la communication UART du module wifi et du STM32 et ainsi voir si on n'avait pas un problème.



Ce sniffer que l'on a acheté permet de convertir l'UART, l'SPI et l'I2C vers un port USB et ainsi faire communiquer un module avec un moniteur série d'un ordinateur. Nous avons branché le STM32 au sniffer et le sniffer au port USB de

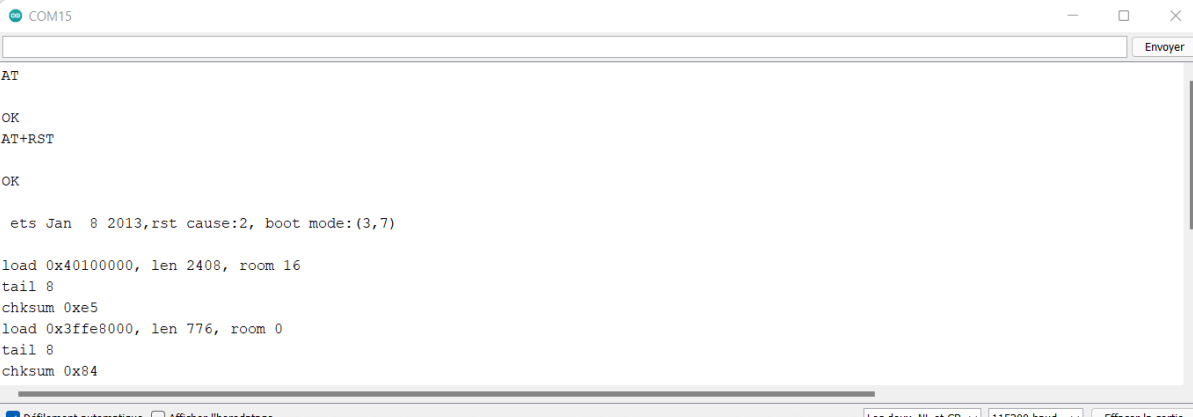
notre ordinateur et l'on voit que la communication UART entre le STM32 et le port série de notre ordinateur fonctionne bien. On a donc pu confirmer que notre sniffer et le port UART du STM32 fonctionnent très bien. On a donc ensuite branché le module Wifi au sniffer et le sniffer au port USB de notre ordinateur puis on a envoyé des commandes AT depuis notre moniteur série vers notre module wifi et comme avec le STM32, on ne reçoit toujours aucune réponse de notre module wifi.

On s'est finalement demandé si notre module wifi était en panne car il ne répondait jamais aux commandes AT. Pour tester notre module, on a donc implémenté la première méthode qui consiste à directement programmer le module Wifi et le rendant ainsi indépendant. Pour cela, nous avons programmé avec Arduino IDE et avons utilisé ce code pour créer un point d'accès avec le module.

```
1 #include <ESP8266WiFi.h>
2 #include <WiFiClient.h>
3 #include <ESP8266WebServer.h>
4
5 //Définir le nom et le mot de passe du serveur wifi
6 #ifndef APSSID
7 #define APSSID "ESP8266Brendan"
8 #define APPSK "thereisnospoon"
9 #endif
10
11 const char *ssid = APSSID;
12 const char *password = APPSK;
13
14 ESP8266WebServer server(80);
15
16 void handleRoot() {
17     //Envoyé text HTML "<h1>Tu es connecté</h1>" sur le serveur
18     server.send(200, "text/html", "<h1>Tu es connecté</h1>");
19 }
20
21 void setup() {
22     delay(1000);
23     Serial.begin(115200);
24     Serial.println();
25     Serial.print("Configuring access point...");
26
27     //Définir le module wifi en point d'accès
28     WiFi.softAP(ssid, password);
29
30     //Récupérer l'adresse IP de point point d'accès
31     IPAddress myIP = WiFi.softAPIP();
32     Serial.print("AP IP address: ");
33     Serial.println(myIP);
34
35     //Démarrer le serveur
36     server.on("/", handleRoot);
37     server.begin();
38     Serial.println("HTTP server started");
39 }
40
41 void loop() {
42     server.handleClient();
43 }
```

Après avoir téléversé ce code dans notre module Wifi, on crée bien un point d'accès avec le module qui aura un nom et un mot de passe et on envoie bien le texte html "<h1>Tu es connecté</h1>" sur un serveur. Donc notre module wifi n'est pas en panne.

On a finalement décidé d'emprunter le module wifi d'un autre groupe pour tester les commandes AT. Ce groupe avait un module WIFI ESP8266-13. Comme avec notre module WIFI, nous l'avons branché sur notre sniffer puis le sniffer sur l'ordinateur. On a ensuite ouvert le moniteur série d'Arduino IDE et envoyé une commande AT. Voici le résultat :



```
COM15
AT
OK
AT+RST
OK
ets Jan 8 2013,rst cause:2, boot mode:(3,7)

load 0x40100000, len 2408, room 16
tail 8
chksum 0xe5
load 0x3ffe8000, len 776, room 0
tail 8
chksum 0x84
```

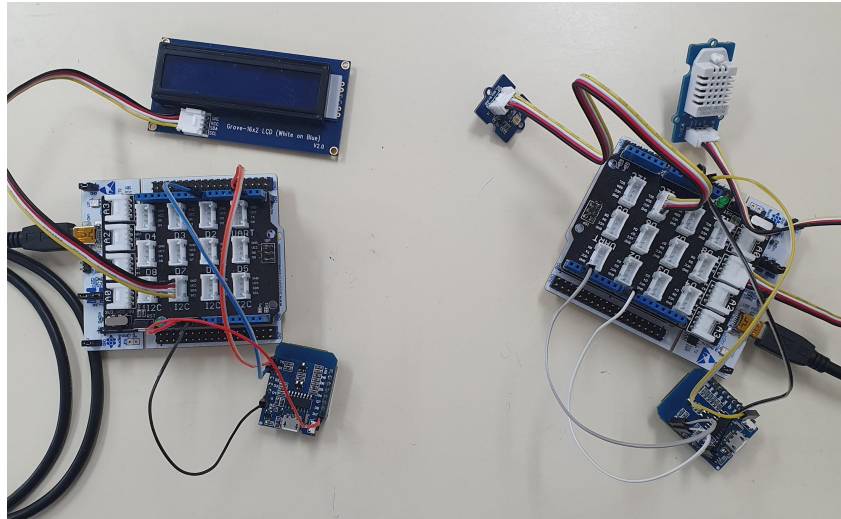
The screenshot shows the Arduino IDE serial monitor window titled 'COM15'. It displays the output of AT commands. The first command 'AT' returns 'OK'. The second command 'AT+RST' returns 'OK' followed by a series of lines indicating a reset: 'ets Jan 8 2013,rst cause:2, boot mode:(3,7)', 'load 0x40100000, len 2408, room 16', 'tail 8', 'chksum 0xe5', 'load 0x3ffe8000, len 776, room 0', 'tail 8', and 'chksum 0x84'. The bottom of the window shows settings: 'Défilement automatique' (checked), 'Afficher l'horodatage' (unchecked), 'Les deux, NL et CR' (selected), '115200 baud', and 'Effacer la sortie'.

Nous observons bien avec ce module que l'envoi d'une commande "AT" via le moniteur série engendre une réponse du module sous forme d'un "OK". L'envoi d'un "AT+RST" via le moniteur donne une série de lignes correspondant au reset du module. Ce module répond donc bien aux commandes AT contrairement au notre.

Suite à ce résultat on se demande donc pourquoi notre module wifi ne fonctionne pas avec les commandes AT. La première hypothèse est que les firmware AT téléversés dans le module wifi sont erronés ce qui est très peu probable car ce sont des packages fournis par le constructeur. La deuxième hypothèse est que les pin UART de notre module wifi ne fonctionnent pas correctement et qu'au final le module Wifi n'arrive pas à recevoir la commande ou à envoyer la réponse. Cette hypothèse est la plus probable. Après, nous avons peut-être pu manquer une étape pour faire marcher ce module, reste à savoir laquelle.

## 8. Assemblage du projet

Nous avons réalisé le montage final de notre projet :



La partie intérieure du projet est donc à gauche et comporte l'écran LCD et le module Wifi. La partie extérieure est à gauche est regroupe le module Wifi, le capteur de température/humidité et le capteur de luminosité.

Nous avons donc deux projets Cube IDE soit un par STM32 qui sont retrouvables sur Github. Le premier contient le code correspondant au module Wifi précédemment présenté ainsi que celui relatif à l'écran LCD que nous ne détaillerons pas puisque celui-ci a été réalisé lors d'une séance de TP et réutilisé lors des TP de base. Le deuxième projet rassemble les codes spécifiques aux capteurs DHT22 et TSL2561.

Évidemment, ce montage n'est absolument pas fonctionnel mais nous tenions néanmoins à le réaliser.

## 9. Conclusion

Malgré de nombreuses heures de recherche, ce TP BE s'avère un échec avec uniquement un capteur, sur notre objectif de deux, fonctionnel et un Wifi qui ne fonctionne pas.