Projet SAE3

Thermomètre connecté

2024

ABBASSI Yassine
EL BARAKA Seddik

Sommaire

Introduction :

Aujourd'hui, la connectivité et l'accès en temps réel de l'information figurent parmi les défis essentiels de notre société. C'est pourquoi le projet intitulé "thermomètre connecté" est un projet qui s'inscrit particulièrement dans le thème actuel.

Le projet a pour objectif de concevoir un système capable de mesurer de façon autonome et régulière la température d'un environnement. Ce thermomètre propose ainsi une réponse concrète aux défis contemporains liés à l'accès à l'information et à l'interaction entre différents systèmes.

Le rapport détaillera les différentes parties du projet de façon claire et détaillée. Cela inclut la conception du circuit électronique pour le capteur, en passant par la configuration de l'IDE Arduino, jusqu'à la finalisation de l'interface graphique pour notre page Web.

Enfin, ce rapport présentera non seulement les aspects techniques du projet,mais également les défis rencontrés et les solutions apportées. Il abordera également les perspectives d'amélioration possibles pour notre thermomètre connecté.

Cahier des charges :

Cadre :

Objectif du projet : L'objectif est de développer une solution permettant de collecter en temps réel les données de température et d'humidité depuis un capteur et de les afficher sur une page web, tout en prenant en compte les considérations économiques et environnementales. Cette solution comporte un capteur connecté et un affichage des informations via une page web.

Attentes et exigences :

Objectifs :

- Acquisition des données toutes les heures
- Affichage des données sur une page web
- Limitation de la consommation d'énergie du microcontrôleur

Equipement :

- Utilisation de l'ESP 32-S3 Reverse TFT Feather
- Utilisation du capteur SHT 41 intégré dans un circuit déjà conçu

Planification :

Livrables :

- Configuration de l'IDE Arduino
- Connexion de l'ESP 32 au réseaux
- Développement d'un serveur en python
- Développement d'une page web client
- Ajout du mode Deep sleep
- Intégration de la communication WebSocket
- Présentation en CSS

Réalisation :

A. Etude du capteur SHT 41

L'étude du capteur consistait simplement à étudier ses spécifications, notamment sa précision, son temps de réponse, sa tension d'alimentation et son interface de communication (*fig. 1*). Elle nous permettait de mieux comprendre comment a été conçue la carte électronique.

Specification

Humidity		
Typ. relative humidity accuracy	1.8	%RH
Operating relative humidity range	0 - 100	%RH
Response time (τ63%)	4	s
Calibration certificate	Factory calibration	

Temperature		
Typ. temperature accuracy	0.2	°C
Operating temperature range	-40 - 125	°C
Response time (τ63%)	2	s

Generic		
Supply voltage	1.08 - 3.6	V
Average supply current	0.4	uA
Interfaces	I²C	
Size (LxWxH)	1.5 x 1.5 x 0.5	mm³
Packaging size	2500, 10000	pcs (T&R)

Figure 1 - Spécification du concepteur

B. Etude de la carte existante

Lors de notre projet, nous avions à notre disposition la carte électronique de la précédente SAE qui a été conçue par les BUT 3 de l'année dernière. Cette carte comprend le capteur de température et d'humidité SHT 41, deux résistances de pull-up de 10kΩ ainsi qu'un condensateur de découplage de 100nF. Voici le schéma du circuit extrait de la ***Datasheet – SHT4x*** (*fig. 2*). La carte électronique conçue est un « *shield* », c'est-à-dire une carte modulaire qui se branche à toutes les broches du microcontrôleur.

Le capteur communique en I2C avec le microcontrôleur. Il sera de plus alimenté en 3.3V.

- SDA : SerialData
- SCL : SerialClock

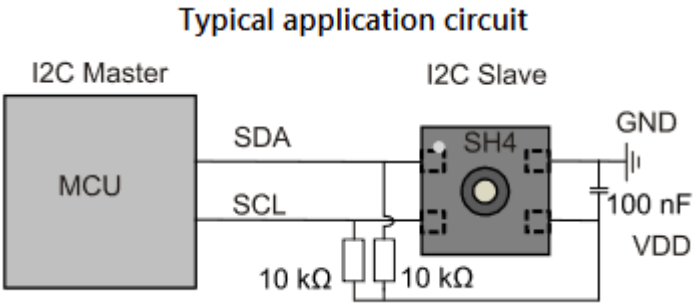
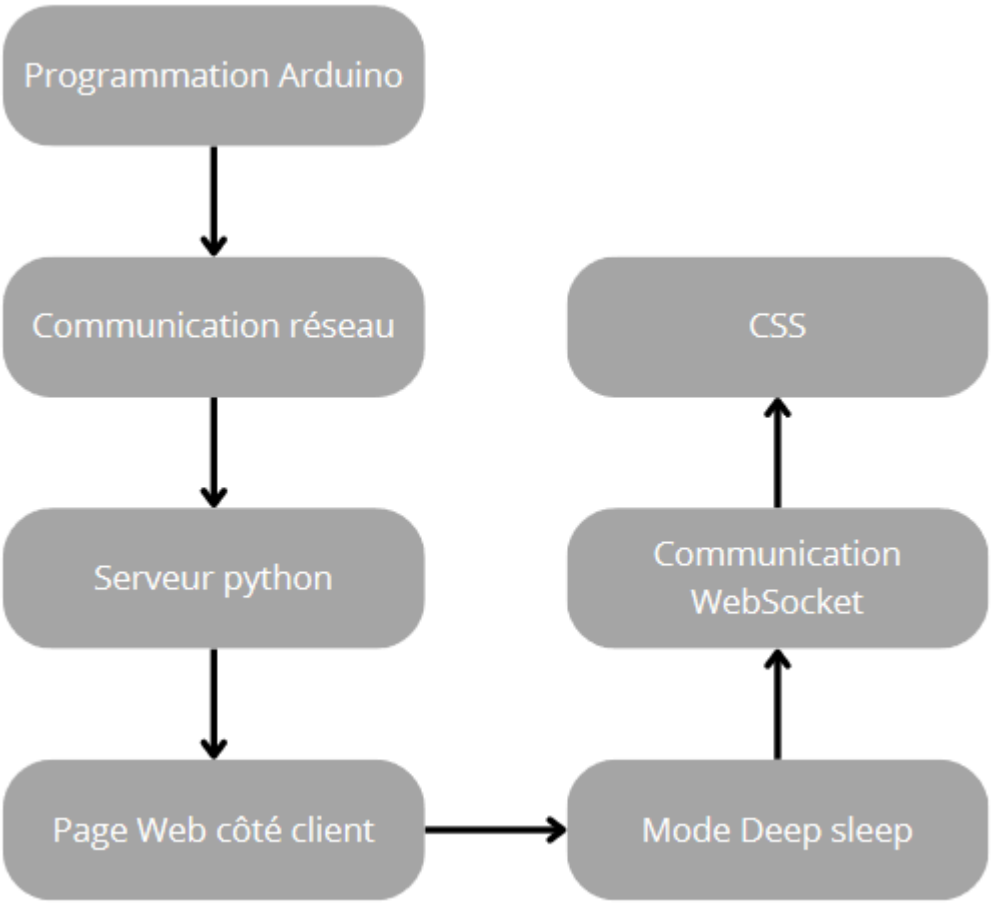


Figure 2 - Schéma de câblage du capteur

C. Gestion du projet (livrables)

Voici l'ordre dans lequel nous avons procédé :



1. Configuration de l'IDE Arduino

La configuration de l'IDE Arduino est une étape essentielle pour permettre la compilation et l'exécution de programmes sur le microcontrôleur ESP32. Cette étape comprend deux phases principales : l'ajout de l'URL du gestionnaire de cartes et l'installation du Board Support Package (BSP) spécifique à l'ESP32.

I. Configuration des préférences

Pour commencer, ouvrez l'IDE Arduino et accédez au menu File → Preferences. Dans la fenêtre des préférences, localisez le champ “Additional Board Manager URLs" et ajoutez l'***URL*** appropriée. Assurez-vous que cette URL est à jour afin de garantir une installation correcte des fichiers. Cette configuration est indispensable pour intégrer le support de l'ESP 32 dans l'environnement de développement.

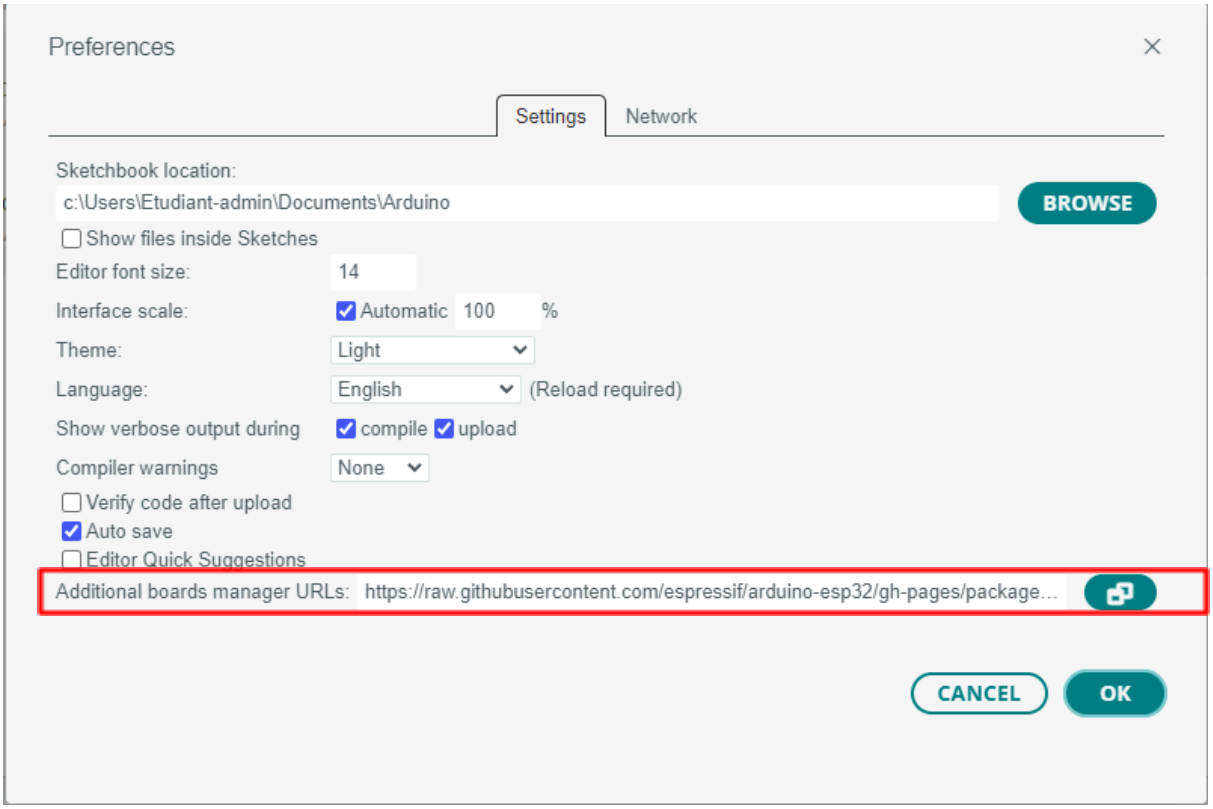


Figure 3 - Fenêtre des paramètres de préférences

II. Installation du Board Support Package (BSP)

Une fois les préférences configurées, passez à l'installation du BSP. Accédez au menu Tools → Board → Board Manager. Dans la barre de recherche, tapez "ESP32" et sélectionnez le package intitulé ESP32 by Espressif Systems. Cliquez sur Install pour télécharger et installer la version la plus récente du package. Cette opération permet à l'IDE Arduino de prendre en charge les cartes ESP32 et leurs fonctionnalités.

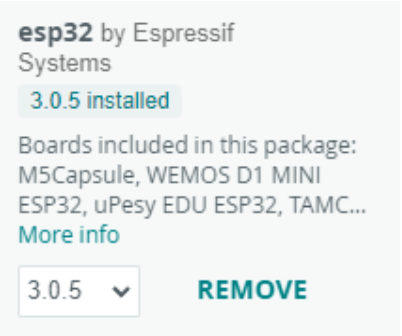


Figure 4 - BSP esp32 by Espressif

III. Configuration des ports

Une fois ces configurations terminées, il est important de noter que l'ESP 32 utilise deux ports différents, un port de compilation pour téléverser les programmes sur le microcontrôleur et un autre port de communication série pour afficher les données dans le Serial Monitor. Cela permet d'éviter tout conflit ou erreur, il faut donc s'assurer de sélectionner le port approprié en fonction de si l'on souhaite compiler un code ou afficher le terminal.

En suivant ces étapes, vous serez en mesure de configurer l'IDE Arduino de manière efficace, permettant ainsi de compiler et d'exécuter vos programmes tout en exploitant pleinement les capacités de votre microcontrôleur ESP32.

2. Communication réseau

La connexion de l'ESP 32 à un réseau est une étape clé pour établir une communication entre le microcontrôleur et d'autres appareils. Voici les étapes détaillées pour scanner les réseaux environnants et configurer une connexion.

I. Scan des réseaux Wi-Fi

Pour scanner les réseaux disponibles autour de l'ESP 32, ouvrez l'IDE Arduino et accédez à File → Examples → Examples for Adafruit ESP32-Rev-S3-Feather. Ce répertoire contient des exemples préconçus pour votre carte. Localisez et chargez l'exemple WiFiScan disponible dans Examples → Examples for YOUR BOARD NAME → WiFi → WiFiScan.

Après le chargement, une fenêtre contenant le code de l'exemple s'affichera. Ce code est conçu pour détecter les réseaux Wi-Fi alentour. Vérifiez que le nom de votre carte ESP32 apparaît bien dans la liste des cartes compatibles avant de télécharger le programme. En téléversant ce code sur votre carte, le terminal de l'IDE Arduino affichera les réseaux détectés (*fig. 5*) :

```
Scan start
Scan done
16 networks found
Nr | SSID                             | RSSI | CH | Encryption
 1 | Yassine                          |  -22 | 11 | WPA2
 2 | TP-LINK_479E                     |  -64 |  4 | WPA2
 3 | UACO2                            |  -66 |  6 | WPA2
 4 | eduroam                          |  -66 | 11 | WPA2-EAP
 5 | eduroam                          |  -67 |  6 | WPA2-EAP
 6 | UACO2                            |  -67 | 11 | WPA2
 7 | eduroam                          |  -73 |  6 | WPA2-EAP
 8 | UACO2                            |  -73 |  6 | WPA2
 9 | UACO2                            |  -74 | 11 | WPA2
10 | eduroam                          |  -75 | 11 | WPA2-EAP
11 | eduroam                          |  -79 | 11 | WPA2-EAP
```

Figure 5 - Terminal de l'IDE Arduino

II. Configuration de la connexion Wi-Fi

Pour connecter l'ESP 32 à un réseau spécifique, modifiez le code de ce ***lien*** en remplaçant les informations suivantes :

"SECRET_SSID" : le nom du réseau Wi-Fi.
"SECRET_PASS" : le mot de passe associé.

Ces paramètres sont clairement indiqués dans le code par des commentaires explicatifs. Une fois ces valeurs mises à jour, téléversez le programme sur la carte. Si ces informations ne sont pas modifiées, l'ESP 32 cherchera en vain une connexion, ce qui entraînera un échec de la configuration. En exécutant le code modifié (*fig. 6*), le terminal de l'IDE Arduino devrait confirmer une connexion réussie au réseau spécifié.

```
Attempting to connect to SSID: Yassine
.
Connected to WiFi
SSID: Yassine
IP Address: 192.168.108.142
signal strength (RSSI):-45 dBm

Starting connection to server...
connected to server
HTTP/1.1 200 OK
Server: nginx/1.18.0 (Ubuntu)
Date: Fri, 27 Sep 2024 07:54:27 GMT
Content-Type: text/html
Content-Length: 69
Last-Modified: Thu, 09 Dec 2021 17:26:22 GMT
Connection: close
ETag: "61b23c3e-45"
Accept-Ranges: bytes

This is a test of Adafruit WiFi!
If you can read this, its working :)
disconnecting from server.
```

Figure 6 - Terminal de l'IDE Arduino

Pensez à bien modifier ces informations avant de compiler le programme. Si elles ne sont pas mises à jour, l'ESP 32 recherchera indéfiniment une connexion sans jamais en trouver.

Avec ces étapes, vous serez en mesure d'établir une communication fiable entre votre ESP32 et un réseau Wi-Fi.

3. Serveur Python

L'objectif est de créer un serveur capable de recevoir les données de température et d'humidité transmises par le microcontrôleur ESP32. Pour ce faire, nous utilisons le micro-framework Flask, connu pour sa flexibilité et sa simplicité.

I. Installation de Flask

Commencez par installer Flask sur votre environnement de développement, tel que Visual Studio Code. Exécutez la commande suivante dans le terminal :

```
pip install flask
```

Ensuite, dans le fichier Python dédié au serveur, importez Flask à l'aide de la ligne de code suivante :

```
from flask import Flask
```

Créez une instance de l'application Flask avec la commande suivante :

```
app = Flask(__name__)
```

II. Création des routes

Ajoutez une route à l'application Flask pour afficher une page d'accueil présentant les données de température et d'humidité en créant un fichier index :

```
@app.route("/page/")        # Définit la route pour afficher la page web
                            # Fonction appelée quand quelqu'un accède à /page/
def page():
    return render_template("index.html")
```

Ajoutez une seconde route permettant à l'ESP 32 d'envoyer des données au serveur via les futures requêtes GET :

```
# Définit la route API qui recevra les données de l'ESP 32 via GET
@app.route("/api/", methods=['GET'])
```

Enfin, pour exécuter l'application sur votre réseau local, utilisez :

```
app.run()
```

4. Page web côté client

Pour permettre au microcontrôleur ESP32 d'envoyer les données acquises au serveur, il est nécessaire de formuler des requêtes HTTP vers l'API configurée. Dans ce cas, nous utiliserons la méthode GET, qui permet d'envoyer des requêtes avec des paramètres directement dans l'URL. Voici les étapes détaillées pour intégrer cette fonctionnalité au programme de l'ESP32.

I. Construction de l'URL et envoi des données

Dans le code Arduino, commencez par inclure la bibliothèque HTTPClient, indispensable pour gérer les requêtes HTTP :

```
#include <HTTPClient.h>
```

Pour transmettre les données de température et d'humidité au serveur, utilisez les lignes de code suivantes :

```
HTTPClient http;
String serverName = "http://192.168.159.159:5000/api/";
String serverPath =  serverName+"?temperature="+tempd +"&humidite="+hum;
http.begin(serverPath.c_str());
Serial.print("Sending URL: ");
Serial.println("http://192.168.159.122:5000/api/?temperature=...&humidite=...");
```

Ce code permet de transmettre les données vers le serveur web à travers le protocole HTTP. Il utilise la classe HTTPClient pour gérer la communication réseau. L'adresse du serveur est définie comme "http://192.168.159.159:5000/api/", et les données sont envoyées sous forme de paramètres dans l'URL (par exemple "?temperature=25&humidite=60"). Le code construit d'abord l'URL complet en combinant l'adresse du serveur avec les valeurs des capteurs, initialise une connexion HTTP avec cette URL, et affiche sur le port série l'URL formée pour faciliter le débogage.

Une fois l'URL formée et la connexion établie, la requête peut être envoyée avec les lignes suivantes : cpp Copier le code

```
http.setTimeout(1000);
int httpCode = http.GET();
```

Ces deux lignes constituent le cœur du processus de communication avec le serveur. La première instruction, `http.setTimeout(1000)`, configure un délai maximal d'une seconde pour la requête, prévenant ainsi un blocage indéfini de l'Arduino en cas de problème de connexion. La seconde instruction ,`int httpCode = http.GET()`, exécute l'envoi de la requête HTTP au serveur et récupère le code de réponse

II. Partie serveur

Dans le serveur Flask, la route /api/ est configurée pour recevoir les données envoyées par l'ESP32 via des requêtes HTTP GET. Voici l'implémentation :

```
# Fonction appelée quand l'ESP32 envoie des données
def get():
    data["temperature"] = request.args.get('temperature')                    #
Récupère la température depuis les paramètres de la requête
    data["humidite"] = request.args.get('humidite')                          #
Récupère l'humidité depuis les paramètres de la requête
    print(f"Données reçues: Temp={data['temperature']}°C, Hum={data['humidite']}%") #
Affiche les données reçues dans la console
```

Lorsqu'une requête HTTP GET est effectuée vers l'adresse "/api/" vue page 11, la fonction `get()` est déclenchée. Cette fonction extrait les valeurs de température et d'humidité depuis les paramètres de l'URL, les stocke dans un dictionnaire nommé "data" , puis les affiche dans la console du serveur sous un format lisible, tel que : "Données reçues : Temp= data°C, Hum= data%". Ce processus permet au serveur de recevoir, traiter et afficher les données transmises.

5. Le mode “deep sleep”

Certains microcontrôleurs possèdent des fonctions permettant de mettre en veille toutes ou une partie de leurs différentes fonctions et c’est notamment le cas de l’ESP 32 S3 Reverse TFT Feather avec le mode deep sleep. Dans ce mode, l'ESP 32 désactive la majorité de ses composants internes, notamment le processeur principal, les périphériques, et la mémoire RAM, tout en maintenant un circuit minimal actif pour se réveiller ou répondre à des événements programmés. C’est l’un des modes les plus économes en énergie disponible sur l’ESP32.

Pourquoi utiliser le mode deep sleep ?

1. Réduction de la consommation énergétique
   - Dans des projets alimentés par batterie, comme les capteurs IoT ou les dispositifs connectés, économiser de l’énergie est crucial pour prolonger la durée de vie de la batterie.
   - En mode actif, l’ESP32 consomme généralement entre 80 et 200 mA. En mode deep sleep, cette consommation peut chuter à environ 10 µA, soit une baisse très importante de la consommation d’énergie.

2. Optimisation des performances pour les tâches périodiques
   - Si un capteur ou un système n’a besoin de fonctionner que périodiquement (par exemple, relever une température toutes les heures), il peut rester en mode deep sleep entre chaque cycle.
   - L'ESP 32 travaillera uniquement pendant les périodes nécessaires, ce qui améliore l’efficacité énergétique.

Pour ce projet, l’ESP 32 devra se mettre en mode deep sleep entre chaque envois de données de température et d’humidité, il faut donc ajouter dans le programme Arduino un cycle de veille entre chaque envois. De plus, le réveil de l’ESP 32 se fera à l’aide d’un timer que l’on pourra ajuster à notre guise en fonction du temps de mise en veille que l’on voudra à l’aide de cette ligne :

```
//Réveil de l'ESP 32 grâce au timer
esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);
```

Le choix du temps de mise en veille s’est tourné vers 20 secondes car en dessous de ces 20 secondes, l’ESP 32 n’aura pas le temps d’envoyer ces données vers le serveur, ce qui bloquera le microcontrôleur et essayera de se connecter en vain. Par déduction, 20 secondes est le temps minimal au microcontrôleur afin qu’il puisse envoyer les données du capteur vers le serveur sans avoir de problèmes de connexion.

Ensuite, le programme principal devra se glisser entre les deux lignes de codes ci-dessous pour que l'ESP 32 puisse envoyer ses données :

```
//Réveil de l'ESP 32 grâce au timer
esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);


//Mode Deep sleep
esp_deep_sleep_start();
```

Tout le programme principal devra maintenant se faire dans la fonction `void setup()` pour pouvoir fonctionner car avec le mode deep sleep, rien ne s'exécutera dans la boucle `void loop()`. Une fois le mode deep sleep ajouter, le programme principal du projet est terminé. Après que les tests ont été réalisés en temps réel, le microcontrôleur ne consommait plus que 19 µA (*fig. 7*).

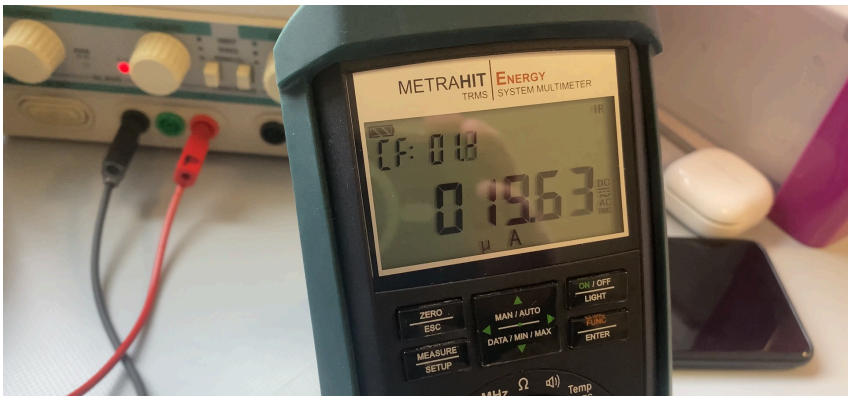


Figure 7 - Consommation du courant du microcontrôleur en mode deep sleep

6. Serveur Websocket

Le WebSocket est un protocole de communication qui permet d'établir une connexion bidirectionnelle en temps réel entre un client et un serveur. Jusqu'ici il aurait fallu faire un rafraichissement de la page toute les secondes par exemple, mais avec le websocket ce ne sera plus nécessaire, pour y parvenir :

On fait une première configuration, on crée un ensemble qui stocke tous les clients WebSocket connectés, lorsqu'un client se connecte, il est ajouté à l'ensemble CLIENTS :

```
CLIENTS = set()
```

Lorsqu'une connexion WebSocket est établie, cette fonction est appelée et s'exécute dans une tâche asynchrone dédiée. Elle commence par enregistrer la connexion dans le dictionnaire CLIENTS, puis entre dans une boucle d'écoute des messages envoyés par le client. L'utilisation du bloc `try:`/ `finally:` est essentielle car elle garantit que le client sera correctement retiré de CLIENTS en cas de perte de la connexion.

```
async def handle_websocket(websocket):
    CLIENTS.add(websocket)
    print("Nouveau client connecté")
    try:
        async for message in websocket:
            pass
    finally:
        CLIENTS.remove(websocket)
        print("Client déconnecté")
```

Cette fonction configure et démarre le serveur WebSocket. En utilisant le gestionnaire de contexte async with, elle crée un serveur écoutant sur toutes les interfaces réseau (0.0.0.0) et le port 5001. La fonction handle_websocket est définie comme gestionnaire pour toutes les nouvelles connexions. L'instruction await `asyncio.Future()` à la fin est un élément clé, car elle crée une tâche qui ne se termine jamais, assurant ainsi un fonctionnement continu du serveur.

```
async def websocket_server():
    async with websockets.serve(handle_websocket, "0.0.0.0", 5001):
        await asyncio.Future()
```

Cette fonction gère la diffusion des messages à tous les clients connectés. Elle est conçue de manière sécurisée en opérant sur une copie du dictionnaire CLIENTS afin d'éviter les problèmes liés aux modifications durant l'itération, et elle gère les erreurs d'envoi en retirant automatiquement les clients défectueux. Étant asynchrone, cette fonction permet d'envoyer des messages à plusieurs clients simultanément sans bloquer l'exécution du reste du programme.

```
async def send_to_all(message):
    for client in CLIENTS.copy():
          try:
            await client.send(message)
        except:
            CLIENTS.remove(client)
```

Dans la route /api/, la fonction get() comme expliqué page 13 fait le lien entre le protocole HTTP traditionnel (utilisé par l'ESP 32) et le protocole WebSocket. Elle reçoit les données de l'ESP 32 via des paramètres GET, les stocke dans le dictionnaire data, qui est ensuite utilisé par asyncio.run pour diffuser les informations à tous les clients WebSocket.

```
def get():
    data["temperature"] = request.args.get('temperature')
    data["humidite"] = request.args.get('humidite')
    print(f"Données reçues: Temp={data['temperature']}°C, Hum={data['humidite']}%")
    asyncio.run(send_to_all(f"{data['temperature']},{data['humidite']}"))
    return "OK"
```

La fonction main() orchestre l'ensemble du système en démarrant Flask dans un thread séparé, tandis que le serveur WebSocket s'exécute dans le thread principal. Cette séparation est essentielle, car Flask et le serveur WebSocket nécessitent des boucles d'événements distinctes pour fonctionner correctement. Le thread Flask est responsable de la gestion des requêtes HTTP traditionnelles, tandis que le thread principal s'occupe des connexions WebSocket persistantes. Cette architecture permet une communication en temps réel efficace et fiable entre l'ESP32 et plusieurs clients web simultanément, tout en minimisant la latence et optimisant l'utilisation des ressources système.

```
def main():
    Thread(target=run_flask).start()
    asyncio.run(websocket_server())
```

Voici la fonction run_flask() :

```
def run_flask():
    app.run(host='0.0.0.0', port=5000)
```

7. Présentation CSS

Cette partie vise à rendre la page plus dynamique et attrayante, encourageant ainsi l'utilisateur à y rester. Pour cela, nous utiliserons le CSS, un langage informatique dédié à la mise en forme des pages web, ici en complément de notre structure HTML sous forme de fichier CSS.

Revenons à notre fichier index mentionné à la page 11, où nous allons commencer par ajouter cette ligne de code. Elle établira le lien entre notre fichier CSS et notre fichier HTML, permettant ainsi d'appliquer les styles définis :

```
<link rel="stylesheet" href="{{ url_for('static',filename='css/style.css')}}">
```

Cette section commence par initialiser les marges et les espacements par défaut de la page. Grâce à box-sizing: border-box, les dimensions des éléments intègrent directement le padding et les bordures, ce qui simplifie le calcul de leur taille :

```
* {
  margin: 0;
  padding: 0;
  box-sizing: border-box;
}
```

Le corps de la page est configuré avec les propriétés suivantes : une police Arial ou une police de secours sans-serif, une hauteur minimale de 100vh pour occuper l'ensemble de l'écran, un padding de 20px pour espacer le contenu, un arrière-plan en dégradé défini par linear-gradient, et une couleur de texte blanche pour assurer la visibilité :

```
body {
  font-family: Arial, sans-serif;
  min-height: 100vh;
  padding: 20px;
  background: linear-gradient(to bottom, #1E56A0, #92D4F3);
  color: white;
}
```

Le titre principal est stylisé avec un centrage du texte, une capitalisation automatique pour chaque mot, une taille de police de 2.5rem pour le rendre plus visible, et une ombre légère (2px 2px) pour ajouter de la profondeur et de la dimension :

```
h1 {
  text-align: center;
  text-transform: capitalize;
  margin-bottom: 40px;
  font-size: 2.5rem;
  text-shadow: 2px 2px 4px rgba(0,0,0,0.2);
}
```

La disposition utilise CSS Grid pour créer deux colonnes de taille égale, définir un espacement de 100px entre les éléments, limiter la largeur maximale à 1750px et centrer l'ensemble horizontalement avec margin: 0 auto :

```
.dashboard {
  display: grid;
  grid-template-columns: repeat(2, 1fr);
  gap: 100px;
  max-width: 1750px;
  margin: 0 auto;
  padding: 20px;
}
```

Chaque carte utilise un fond bleu clair, une bordure fine, et des coins arrondis. Elle est centrée avec Flexbox pour un alignement vertical et horizontal, et bénéficie d'une animation de transition pour les transformations. De plus, une ombre portée ajoute de la profondeur :

```
.data-card {
  background: rgb(115, 171, 243);
  border: 2px solid rgb(209, 235, 247);
  border-radius: 15px;
  padding: 30px;
  text-align: center;
  min-height: 400px;
  display: flex;
  flex-direction: column;
  justify-content: center;
  align-items: center;
  transition: transform 0.3s ease;
  box-shadow: 0 4px 6px rgba(0,0,0,0.1);
}
```

Au survol, les cartes se soulèvent légèrement grâce à une translation verticale de -5px et renforcent leur ombre avec une profondeur (box-shadow: 0 6px 12px rgba(0,0,0,0.15)) :

```
.data-card:hover {
  transform: translateY(-5px);
  box-shadow: 0 6px 12px rgba(0,0,0,0.15);
}
```

La section des images utilise Flexbox pour la disposition, avec un espacement uniforme de 30px entre les éléments et un centrage parfait à la fois horizontalement et verticalement :

```
.image-section {
  display: flex;
  justify-content: center;
```

```
  align-items: center;
  gap: 30px;
  margin-top: 30px;
  width: 100%;
}
```

Les images bénéficient d'une taille maximale définie à 250px, de coins arrondis, et d'un effet de zoom au survol grâce à la transformation scale(1.05) :

```
.dashboard-image1,
.dashboard-image2 {
  max-width: 250px;
  height: auto;
  border-radius: 10px;
  transition: transform 0.3s ease;
}

.dashboard-image1:hover,
.dashboard-image2:hover {
  transform: scale(1.05);
}
```

Perspectives d’améliorations :

1. Optimisation énergétique
    - Mode deep sleep avancé : Ajuster dynamiquement la durée de veille selon les besoins pour réduire encore plus la consommation d’énergie.
    - Réveil par interruption externe : Utiliser des événements comme une variation rapide de température pour réveiller l'ESP 32 uniquement si nécessaire.
2. Amélioration de la carte électronique
    - Ajout de capteurs complémentaires : Intégrer des capteurs de pression ou de qualité de l'air pour enrichir les données.
3. Interface utilisateur
    - Graphiques interactifs : Visualiser l'évolution des données en temps réel avec des outils comme Chart.js.
4. Sécurité et réseau
    - Chiffrement : Sécuriser les communications via HTTPS.

Conclusion :

En conclusion, ce projet nous a permis de développer une multitude de compétences techniques en lien avec notre formation, notamment avec la conception de programmes sur des logiciels spécialisés tels que l'IDE Arduino ou Visual Studio Code, mais aussi sur de la programmation de microcontrôleur et de la communication avec différents capteurs. Nous avons aussi découvert la communication réseau via Wi-Fi afin d'envoyer des données à un serveur codé en Python. De plus, nous avons également acquis des compétences en développement Web, en concevant une interface pour afficher les données en temps réel, et en optimisant la consommation énergétique du système grâce au mode deep sleep de notre microcontrôleur. Enfin, nous avons appris à styliser et à personnaliser une page Web avec du HTML ainsi que du CSS qui ont été pour nous des compétences bonus mais importantes à nos yeux pour ce projet. En résumé, le thermomètre connecté a été le projet le plus important de notre formation jusque là et il nous a permis de nous renforcer dans le domaine de la programmation.

Nous souhaitons remercier nos professeurs pour leur aide précieuse tout au long de notre projet, de la gestion des livrables jusqu'à la réalisation du serveur en WebSocket. Vos conseils, votre patience et votre disponibilité nous ont vraiment aidé à avancer et à mieux comprendre les choses. Grâce à vous, nous avons mené ce projet à bien et avons acquis beaucoup de savoir. Merci de votre soutien.

Index :

Code final :

```
#include <WiFi.h>
#include <HTTPClient.h>
#include <WiFiClient.h>
#include <dummy.h>
#include <SHTSensor.h>
#include <Wire.h>
SHTSensor sht;

#define uS_TO_S_FACTOR 1000000ULL       //Facteur de conversion en secondes pour les
microsecondes
#define TIME_TO_SLEEP  20               //Temps pendant lequel l'ESP 32 va se mettre
en veille (en secondes)

//SSID et Mdp du réseau
const char* ssid = "YOUR_SSID";
const char* password =  "YOUR_PSSWD";

void print_wakeup_reason() {
  esp_sleep_wakeup_cause_t wakeup_reason;

  wakeup_reason = esp_sleep_get_wakeup_cause();

  switch (wakeup_reason) {
    case ESP_SLEEP_WAKEUP_EXT0:     Serial.println("Wakeup caused by external signal
using RTC_IO"); break;
    case ESP_SLEEP_WAKEUP_EXT1:     Serial.println("Wakeup caused by external signal
using RTC_CNTL"); break;
    case ESP_SLEEP_WAKEUP_TIMER:    Serial.println("Wakeup caused by timer"); break;
    case ESP_SLEEP_WAKEUP_TOUCHPAD: Serial.println("Wakeup caused by touchpad");
break;
    case ESP_SLEEP_WAKEUP_ULP:      Serial.println("Wakeup caused by ULP program");
break;
    default:                        Serial.printf("Wakeup was not caused by deep
sleep: %d\n", wakeup_reason); break;
  }
}
```

```
void setup() {
  //Print the wakeup reason for ESP32
  print_wakeup_reason();
  //réveil de l'ESP 32 grâce au timer
  esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);

  //Code de l'envoi des données
  Wire.begin();
  sht.init();
  Serial.begin(115200);
  WiFi.begin(ssid, password);

  while (WiFi.status() != WL_CONNECTED) {
    delay(1000);
    Serial.println("Connecting to WiFi..");
  }
  Serial.println("Connected to the WiFi network");

  //Lecture des valeurs
  sht.readSample();

  float tempd=sht.getTemperature();    //Valeur de la température
  float hum=sht.getHumidity();         //Valeur de l'humidité

  if ((WiFi.status() == WL_CONNECTED)) {   //Vérifie l'état actuel de la connexion

    //Affichage des données dans le terminal
    Serial.println(sht.getTemperature());
    Serial.println(sht.getHumidity());

    //Envoi des valeurs vers le serveur
    HTTPClient http;
    String serverName = "http://192.168.159.159:5000/api/";
    String serverPath =  serverName+"?temperature="+tempd +"&humidite="+hum;

    http.begin(serverPath.c_str());                  //Spécifie l'URL

    Serial.print("Sending URL: ");
    Serial.println("http://192.168.159.122:5000/api/?temperature=...&humidite=...)");

    http.setTimeout(1000);
    int httpCode = http.GET();                       //Fait une requête
```

```
    //Serial.println("Sending a request");
    //delay(1000);

    if (httpCode > 0) {                                  //Vérifie le code de retour

        String payload = http.getString();
        //Serial.println(httpCode);
        Serial.println(payload);
      }

    else {
      Serial.println("Error on HTTP request");
      Serial.println(httpCode);
    }

    http.end();                                           //Libère les ressources
  }

   Serial.println("Setup ESP32 to sleep for every " + String(TIME_TO_SLEEP) + "
Seconds");
  Serial.println("Going to sleep now");
  Serial.flush();

esp_deep_sleep_start();                              //Mode Deep sleep
}
void loop() {}
```

Code serveur :

```
from flask import Flask, render_template, request
import asyncio
import websockets

# Pour exécuter Flask dans un thread séparé
from threading import Thread

app = Flask(__name__, static_folder='static')

CLIENTS = set()

data = {"temperature": "0", "humidite": "0"}             # Dictionnaire pour stocker
les dernières valeurs reçues, initialisées à "0"
@app.route("/page/")                                        # Définit la route pour
afficher la page web

# Fonction appelée quand quelqu'un accède à /page/
def page():
    return render_template("index.html")

# Définit la route API qui recevra les données de l'ESP32 via GET
@app.route("/api/", methods=['GET'])

# Fonction appelée quand l'ESP32 envoie des données
def get():
    data["temperature"] = request.args.get('temperature')                    #
Récupère la température depuis les paramètres de la requête
    data["humidite"] = request.args.get('humidite')                          #
Récupère l'humidité depuis les paramètres de la requête
    print(f"Données reçues: Temp={data['temperature']}°C, Hum={data['humidite']}%") #
Affiche les données reçues dans la console

    asyncio.run(send_to_all(f"{data['temperature']},{data['humidite']}"))    #
Envoie les nouvelles données à tous les clients WebSocket

    return "OK"

# Fonction asynchrone pour envoyer un message à tous les clients
async def send_to_all(message):
    for client in CLIENTS.copy():                                            #
Pour chaque client connecté (on utilise .copy() pour éviter les erreurs de
modification pendant la boucle)
```

```
        try:
            await client.send(message)                                    #
Tente d'envoyer le message au client

        except:
            CLIENTS.remove(client)                                        #
Si erreur (client déconnecté), le retire de la liste


# Fonction qui gère chaque connexion WebSocket
async def handle_websocket(websocket):
    CLIENTS.add(websocket)
    # Ajoute le nouveau client à l'ensemble des clients
    print("Nouveau client connecté")
    # Affiche un message quand un client se connecte
    try:
        async for message in websocket:
            pass
            # Attend les messages du client (ici on ne fait rien avec)
    finally:
        CLIENTS.remove(websocket)
        # Retire le client de l'ensemble quand il se déconnecté
        print("Client déconnecté")
        # Affiche un message quand un client se déconnecte


async def websocket_server():
    # Fonction qui démarre le serveur WebSocket
    async with websockets.serve(handle_websocket, "0.0.0.0", 5001):
         # Démarre le serveur WebSocket sur le port 5001, accessible depuis n'importe
quelle IP
        await asyncio.Future()
        # Maintient le serveur en fonctionnement


def run_flask():
    # Fonction qui démarre le serveur Flask
    app.run(host='0.0.0.0', port=5000)
    # Démarre Flask sur le port 5000, accessible depuis n'importe quelle IP


def main():
```

```
    # Fonction principale qui démarre tout
    Thread(target=run_flask).start()
    # Démarre Flask dans un thread séparé
    asyncio.run(websocket_server())
    # Démarre le serveur WebSocket dans le thread principal


if __name__ == "__main__":
    # Si ce fichier est exécuté directement (pas importé comme module)
    main()
    # Lance la fonction principale

    app.run()
```

Code Index :

```
<!DOCTYPE html>
<html>
<head>
    <title>ESP32 Data</title>
    <link rel="stylesheet" href="{{ url_for('static', filename='css/style.css') }}">
</head>
<body>
    <h1>Données ESP32</h1>

    <div class="dashboard">
        <div class="data-card">
            <div class="data-label">Température</div>
            <div>
                <span id="temperature" class="temperature-value">-</span>
                <span class="unit">°C</span>
            </div>
        </div>

        <div class="data-card">
            <div class="data-label">Humidité</div>
            <div>
                <span id="humidite" class="humidity-value">-</span>
                <span class="unit">%</span>
            </div>
        </div>
```

```
    </div>

    <div class="image-section">
        <div class="image-container1">
                    <img  src="{{  url_for('static',  filename='images/thermo.png')  }}"
alt="Thermomètre" class="dashboard-image1">
        </div>
        <div class="image-container2">
                     <img  src="{{  url_for('static',  filename='images/nuage.png')  }}"
alt="Nuage" class="dashboard-image2">
        </div>
    </div>

    <script>
        const ws = new WebSocket('ws://localhost:5001');

        ws.onmessage = function(event) {
            const [temp, hum] = event.data.split(',');

            // Mise à jour température avec animation
            const tempElement = document.getElementById('temperature');
            tempElement.textContent = temp;
            tempElement.classList.add('value-updated');
            setTimeout(() => tempElement.classList.remove('value-updated'), 300);

            // Mise à jour humidité avec animation
            const humElement = document.getElementById('humidite');
            humElement.textContent = hum;
            humElement.classList.add('value-updated');
            setTimeout(() => humElement.classList.remove('value-updated'), 300);

            console.log("Nouvelles données:", temp, hum);
        };
    </script>
</body>
</html>
```

Code CSS:

```css
* {                                /* Sélectionne tous les éléments */
  margin: 0;                       /* Supprime toutes les marges externes */
  padding: 0;                      /* Supprime toutes les marges internes */
  box-sizing: border-box;
}

body {                              /* Style du corps de la page */
  font-family: Arial, sans-serif;
  min-height: 100vh;                /* Hauteur minimale de 100% de la vue */
  padding: 20px;
   background: linear-gradient(to bottom, #1E56A0, #92D4F3);    /* Dégradé de bleu en
arrière-plan */
  color: white;                     /* Couleur du texte en blanc */
}

h1 {                                /* Style du titre principal */
  text-align: center;               /* Centre le texte */
  text-transform: capitalize;
  margin-bottom: 40px;
  font-size: 2.5rem;
  text-shadow: 2px 2px 4px rgba(0,0,0,0.2);
}

.dashboard {                         /* Style du conteneur principal */
  display: grid;                     /* Utilise la disposition en grille */
  grid-template-columns: repeat(2, 1fr);      /* Créer 2 colonnes de même taille */
  gap: 100px;                        /* Espace entre les cartes */
  max-width: 1750px;                  /* Largeur maximale */
  margin: 0 auto;                     /* Centre horizontalement */
  padding: 20px;                      /* Espace intérieur */
}

.data-card {                          /* Style des cartes de données */
  background: rgb(115, 171, 243);
  border: 2px solid rgb(209, 235, 247);
  border-radius: 15px;
  padding: 30px;
  text-align: center;
  min-height: 400px;
  display: flex;                      /* Utilise flexbox */
  flex-direction: column;             /* Empile les éléments verticalement */
  justify-content: center;            /* Centre verticalement */
```

```
  align-items: center;                /* Centre horizontalement */
  transition: transform 0.3s ease;    /* Animation douce au survol */
  box-shadow: 0 4px 6px rgba(0,0,0,0.1);      /* Ombre portée */
}

.data-label {                         /* Style des labels (Température/Humidité) */
  font-size: 24px;
  color: white;
  margin-bottom: 20px;
  font-weight: 500;
  letter-spacing: 1px;
}

.temperature-value {                  /* Style de la valeur de température */
  font-size: 72px;
  color: white;
  font-weight: bold;
  text-shadow:  2px  2px  4px  rgba(0,0,0,0.1);       /*  Légère  ombre  pour  meilleure
lisibilité */
}

.humidity-value {                     /* Style de la valeur d'humidité */
  font-size: 72px;
  color: white;
  font-weight: bold;
  text-shadow: 2px 2px 4px rgba(0,0,0,0.1);  /* Même ombre que temperature-value */
}

.unit {                               /* Style des unités (°C et %) */
  font-size: 24px;                    /* Taille plus petite que les valeurs */
  color: white;
  margin-left: 5px;
  opacity: 0.9;                       /* Légèrement transparent */
}

@media (max-width: 768px) {           /* Styles pour les écrans mobiles (<768px) */
  .dashboard {
      grid-template-columns: 1fr;     /* Passe à une seule colonne */
  }

  .data-card {
      min-height: 250px;              /* Réduit la hauteur des cartes */
  }
```

```
  .temperature-value,
  .humidity-value {
      font-size: 3.5rem;              /* Réduit la taille des valeurs */
  }

  .data-label {
      font-size: 1.2rem;              /* Réduit la taille des labels */
  }

  .unit {
      font-size: 1.2rem;              /* Réduit la taille des unités */
  }
}

@keyframes updatePulse {               /* Définit l'animation de mise à jour */
  0% {                                 /* État initial */
      transform: scale(1);             /* Taille normale */
  }
  50% {                                /* Milieu de l'animation */
      transform: scale(1.05);          /* Agrandit légèrement (5%) */
  }
  100% {                               /* État final */
      transform: scale(1);             /* Retour à la taille normale */
  }
}

.value-updated {                       /* Classe appliquée lors de la mise à jour */
  animation: updatePulse 0.3s ease-in-out;   /* Applique l'animation pendant 0.3s */
}


.image-section {
  display: flex;
  justify-content: center;
  align-items: center;
  gap: 30px;
  margin-top: 30px;
  width: 100%;
}

.image-container1,
.image-container2 {
```

```
  display: flex;
  justify-content: center;
  align-items: center;
  flex: 1;
}

.dashboard-image1,
.dashboard-image2 {
  max-width: 250px;
  height: auto;
  border-radius: 10px;
  transition: transform 0.3s ease;
}

.dashboard-image1:hover,
.dashboard-image2:hover {
  transform: scale(1.05);
}
```