



DIGICHRONE®
COLLABORATIVE WAY FOR INNOVATIVE PROJECTS



Réseaux
et Télécoms

iut Nord Franche-Comté

SYNCHRONISEUR SOLAIRE

VERSION
2.0



Rapport de stage

9 janvier - 3 mars 2023

Présenté par

Yassine EL HAMIOUI

🌐 Stephane GIVRON - Responsable Pédagogique

🌐 Francois SPIES - Responsable Professionnel

📍 4 Place Tharradin 25200 MONTBELIARD

📍 21 rue marceau petit 21340 NOLAY

1. Simulation du Capteur de mesure de courant

1.1 Solution alternative au capteur SCT013

Dans le cadre de ce projet, l'objectif était de récupérer des mesures de courant avec le capteur SCT013, cependant le matériel s'est avéré incompatible avec la récupération des mesures de courant. J'ai donc opté pour une simulation du capteur (dans l'attente de réception du bon capteur), avec l'outil socat sur un système Raspberry Pi. Tout d'abord, l'outil socat permet d'établir une communication entre deux terminaux virtuels (pty) via un lien série. J'ai donc développé un script Python appelé "simu_socat_primo.py" pour lire les valeurs simulées du SCT013 à partir du port série virtuel.

1.2 Configuration du port série avec socat

La première étape consiste à l'installer et le configurer afin de créer une paire de ports série virtuels.

Cela se fait en exécutant la commande suivante dans un terminal :

"socat -d -d pty,raw,echo=0,link=/dev/ttySCT pty,raw,echo=0"

```
root@raspberrypi:/home/pi/sct013/finale# socat -d -d pty,raw,echo=0,link=/dev/ttySCT pty,raw,echo=0
2023/06/10 19:03:14 socat[8568] N PTY is /dev/pts/1
2023/06/10 19:03:14 socat[8568] N PTY is /dev/pts/2
2023/06/10 19:03:14 socat[8568] N starting data transfer loop with FDs [5,5] and [7,7]
```

Cette commande crée une paire de ports série virtuels, par exemple dans mon cas **/dev/pts/1** et **/dev/pts/2**, qui sont liés l'un à l'autre.

1.3 Génération de valeur aléatoires

J'ai développé une commande shell qui génère des données aléatoires (qui pourrait refléter des mesure réaliste, plus ou moins) et les envoie en continu à notre port virtuel socat, avec un délai d'une seconde entre chaque envoi.

```
while true; do echo "scale=1; $((($RANDOM % 200)) / 10" | bc > /dev/pts/1; sleep 1; done
```

1.4.1 Le script de simulation du capteur

Le script développé est une réadaptation du script de capture de mesure de courant, qui utilise la bibliothèque pigpio pour la communication GPIO et la bibliothèque serial pour la communication série. Le port série '/dev/ttyS0' a été configuré dans le script pour lire les valeurs du SCT013.

Le code Python du script comprend les étapes suivantes :

- Configuration du port série pour communiquer avec le port virtuel créé par socat
- Boucle principale pour la lecture continue des mesures simulées
- Conversion des données flottante
- Affichage de la mesure de courant avec une précision de deux chiffres après la virgule
- Si besoin, affichage des problèmes avec un retour erreur

Le script est disponible à ce lien github [gregegrgeg](#) qui donne ce résultat :

```
root@raspberrypi:/home/pi/sct013/finale# python3 simu_socat_primo.py
Mesure de courant : 11.600 A
Mesure de courant : 1.200 A
Mesure de courant : 15.900 A
Mesure de courant : 8.000 A
Mesure de courant : 8.600 A
Mesure de courant : 1.400 A
Mesure de courant : 19.500 A
```

1.4.2 Explication du code “simu_socat_primo.py”

Je vais procéder à l’explication de chaque ligne du code :

Les deux modules `time` et `serial` sont importés pour utiliser les fonctions liées à la gestion du temps et à la communication série.

```
import time
import serial
```

Une instance de la classe `Serial` est créée et assignée à la variable `ser`. Cela configure la communication série avec le port `/dev/pts/2` à une vitesse de 38400 bauds.

```
ser = serial.Serial('/dev/pts/2', 38400)
```

Ceci est une boucle infinie qui permet de continuer la lecture des données en continu.

La fonction `readline()` est utilisée pour lire une ligne de données du port série. `decode()` convertit les octets lus en une chaîne de caractères et `strip()` supprime les espaces blancs supplémentaires. La ligne lue est ensuite assignée à la variable `data`.

```
while True:
    # Lecture des mesures
    data = ser.readline().decode().strip()
```

Cette condition vérifie si des données ont été reçues. Si `data` n'est pas une chaîne vide, les instructions à l'intérieur de ce bloc sont exécutées.

```
if data:
    try:
        # On convertit la donnée en float pour pouvoir avoir des données après la virgule
        mesure = float(data)

        courant = mesure

        # J'affiche la mesure reçue sur le port
        print("Mesure de courant : {:.3f} A".format(courant))

    except ValueError:
        # Si problème
        print("Erreur de conversion des données")
```

Dans ce bloc `try-except`, la donnée `data` est convertie en un nombre flottant à l'aide de `float(data)`. Si la conversion réussit, la valeur est assignée à la variable `mesure` et la mesure de courant est calculée. Ensuite, la mesure de courant est affichée avec une précision de trois chiffres après la virgule. Si une erreur de valeur se produit lors de la conversion en flottant, une exception `ValueError` est levée et le message "Erreur de conversion des données" est affiché.

Après chaque lecture et traitement des données, le programme attend une seconde avant de lire les prochaines données. Cela permet de ralentir le rythme de lecture et de laisser le temps aux nouvelles données d'arriver.

```
time.sleep(1)
```

2.1 Pourquoi le précédent capteur ne fonctionnait

Le SCT013 50A/1V fournit une tension de sortie proportionnelle au courant, tandis que le SCT013 100A/50mA fournit un courant de sortie proportionnel au courant mesuré.



Cependant, il est important de prendre en compte la plage de mesure du SCT013 100A/50mA qui est de 0 à 50 mA. Si le courant à mesurer dépasse cette plage, il faudra utiliser un capteur avec une plage de mesure plus élevée. La documentation ce lien nous fourni les informations nécessaire :

RPICT3T1 & RPIZ_CT3T1

RPICT3V1 & RPIZ CT3V1

RPIC7V1 all versions

Figure 2 : Page de calibrage pour chaque carte

2.2 Préparation du microcontrôleur

Afin de manipuler les données entrantes mesurées par le capteur, nous avons besoin d'un microcontrôleur, donc le RaspberryPi Zéro dans notre cas, le capteur SCT013 ainsi que la carte CT3TA qui sert d'intermédiaire. Le capteur en ma possession était branché sur le port ct1 de la carte relié au RpiZ.

Avant de commencer il est essentiel de configurer la lecture en série dès port disponible, à ce lien nous avons toutes les indications nécessaires :

http://lechacal.com/wiki/index.php?title=Howto_setup_Raspbian_for_serial_read

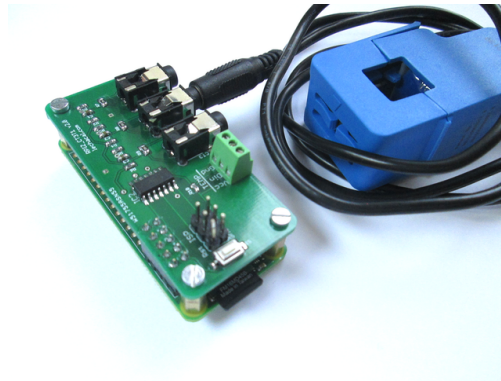


Figure 3 : Image issue du site lechacal.com

Cette image nous présente le matériel nécessaire à la mesure des données : Le capteur SCT013 est l'objet en bleu, qui dans cet exemple est branché sur le ct2 de la carte CT3T1, qui son tour est connecté par des pins au RPIZ, qui nous sert de microcontrôleur.

Comme indiqué dans le chapitre “**4.2.4 Capteur de mesure de courant**” il faut faire passer un seul fil puisque le neutre ne doit pas passer dans la bobine, sinon le champ magnétique va être annulé par le neutre. Mon tuteur M.Spies m'a donc fabriqué une multiprise compatible, qui possédait donc les fils dénudés avec 2 fils positifs et un négatifs.

2.3.1 Développement du script

Anciennement, j'avais développé un code qui utilisait des bibliothèques telles que pi gpio etc, qui se sont avérées assez inutiles, j'ai donc renoncé et recommencé de zéro.

Le code Python du script comprend les étapes suivantes :

- J'importe les bibliothèques nécessaires (serial et time)
- J'établis une connexion série avec le capteur
- Lecture de la réponse du capteur
- Lecture de la valeur des données du courant mesuré
- En dernier j'affiche le code

2.3.2 Explication du code "simu_socat_primo.py"

Je vais procéder à de chaque ligne du code :

Les bibliothèques serial et time sont importées pour permettre la communication série et gérer les délais respectivement.

```
# Importation des bibliothèques nécessaires (serial et time)
import serial
import time
```

Une instance de la classe Serial est créée pour établir la connexion série. Le paramètre /dev/ttyAMA0 spécifie le port série à utiliser, et 38400 est la vitesse de communication en bauds. J'ai ajouté en commentaire une commande de test du port "stty -echo -F /dev/ttyAMA0 raw speed 38400".

```
# J'établis une connexion série avec le capteur
primo = serial.Serial('/dev/ttyAMA0', 38400) #Je vérifie avec stty -echo -F /dev/ttyAMA0 raw speed 38400 et cat /dev/ttyAMA0
#Lecture de la réponse du capteur
```

Cette fonction renvoie le caractère 'c' au RPIZ_CT3T1 pour demander la mesure de courant. Elle lit ensuite la réponse du capteur, la décode et la convertit en un nombre à virgule flottante représentant la mesure de courant. La valeur est divisée par 1000 pour convertir le courant de milliampères en ampères. Enfin, la fonction renvoie cette valeur de courant.

```
def lecture_courant():
    primo.write(b'c')
    reponse = primo.readline().decode().strip()
    ct1 = float(reponse[0:5].replace(' ', ''))/1000
    return ct1
```

Il est possible d'utiliser les 3 entrées de la carte, par exemple si l'on souhaite ajouter d'autre capteur, il nous suffit d'ajouter par exemple

```
ct2 = float(reponse[5:10].replace(' ', ''))/1000
ct3 = float(reponse[10:15].replace(' ', ''))/1000
```

puis lors du return : return ct1, ct2, ct3.

La boucle while True est utilisée pour effectuer une lecture continue du courant. À chaque itération, la fonction lecture_courant() est appelée pour récupérer la mesure de courant. La valeur est ensuite affichée avec une précision de deux décimales à l'aide de l'expression de formatage {Amp_courant:.2f}. Enfin, la boucle fait une pause de 1 seconde avant de passer à l'itération suivante.

```
while True:
    Amp_courant = lecture_courant()
    # Affichage du courant mesuré avec deux chiffres après la virgule
    print(f'Le courant mesuré est : {Amp_courant:.2f} A')
    time.sleep(1)
```

2.3.3 Principale problème rencontré avec le programme

Lors des premiers test après avoir développé le code, je n'ai pas réussi à l'exécuter correctement, je n'ai pas réussi à faire de screen, mais j'ai tout de même réussi à exporter l'erreur :

```
root@raspberrypi:/home/pi/sct013# python3 ki.py
```

Traceback (most recent call last):

File "/home/pi/sct013/ki.py", line 24, in <module>

Amp_courant = lecture_courant()

File "/home/pi/sct013/ki.py", line 15, in lire_courant

ct1 = float(reponse[0:5])/1000

ValueError: could not convert string to float: '11 25'

Cette erreur nous indique que c'est due à une tentative de conversion d'une chaîne de caractères en nombre à virgule flottante (float), mais la chaîne de caractères contenait des espaces ou d'autres caractères indésirables qui ne peuvent pas être convertis en un nombre.

J'ai donc utilisé la méthode "replace()" afin de supprimer les espaces avant la conversion en float

2.3.4 Résultat et conclusion

En résumé le capteur de courant récupère bel et bien les mesure de courant, j'ai pu le tester donc sur la multiprise fabriqué par mon tuteur, je pouvais brancher ou débrancher une source d'énergie, les mesures changeaient en fonction de cela.

```
root@raspberrypi:/home/pi/sct013# python3 codemarche.py
Le courant mesuré est : 1.19 A
Le courant mesuré est : 1.18 A
Le courant mesuré est : 1.19 A
Le courant mesuré est : 1.11 A
```

Par exemple, dans ce test, lorsque c'est 1.19 A ou 1.18 A, le chargeur de mon pc était branché, et lorsque celui-ci est débranché, les mesures descendent à 1.11 A.

Tous les codes sont disponibles dans mon github à ce lien :

“ https://github.com/Yasko0x000/Projet_de_Stage_Synchroniseur_Solaire ”

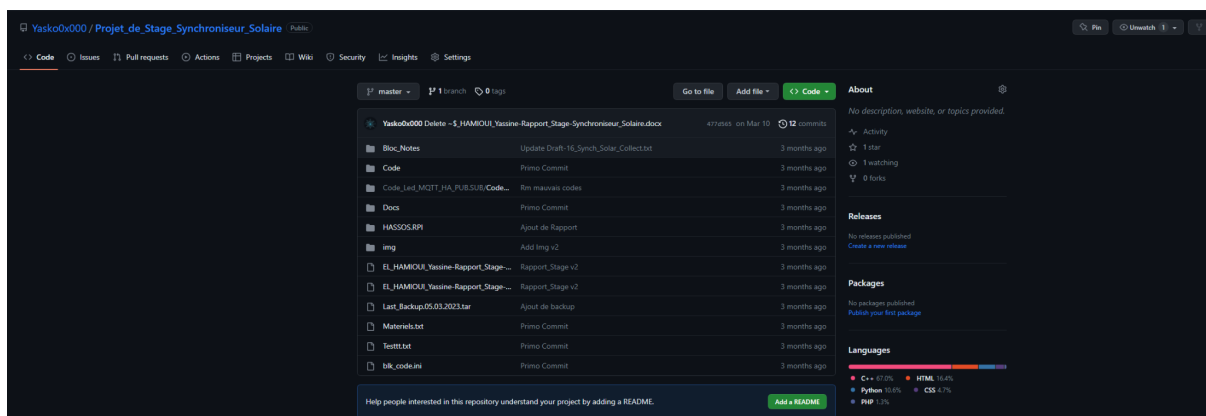


Figure 4 : GitHub du projet de stage

3. Prise connecté Meross x Matter

3.1 Nouvelle prise connecté compatible Matter

La prise connectée Meross 315 Matter a été testée et intégrée avec succès dans Home Assistant. Sa principale différence avec la prise Meross 310, réside dans les normes de communications utilisées. C'est à dire, la prise MSS 315 Matter est compatible avec la norme **Matter** qui est un protocole de communication basé sur IP et qui vise à devenir le protocol standard de la communication entre les appareils connectés

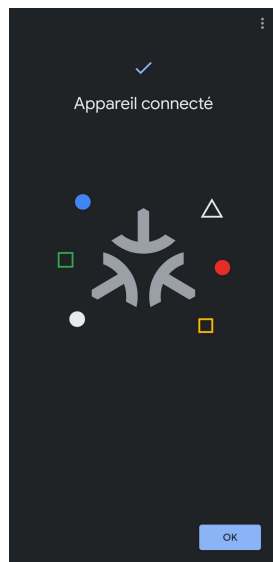


Figure 5 : Connexion à la prise réussit



Figure 6 : Consommation en temps réel

Tout d'abord, j'ai configuré la prise MSS315 depuis l'application meross, afin de l'intégrer au réseau domotique par wifi, une fois testé et configuré l'intégration de la prise dans Home Assistant à été très rapide, en effet, contrairement à la prise MSS310, la prise Matter ne nécessite pas une intégration spécifique à Meross, la détection de l'appareil à été donc automatique par Home Assistant.

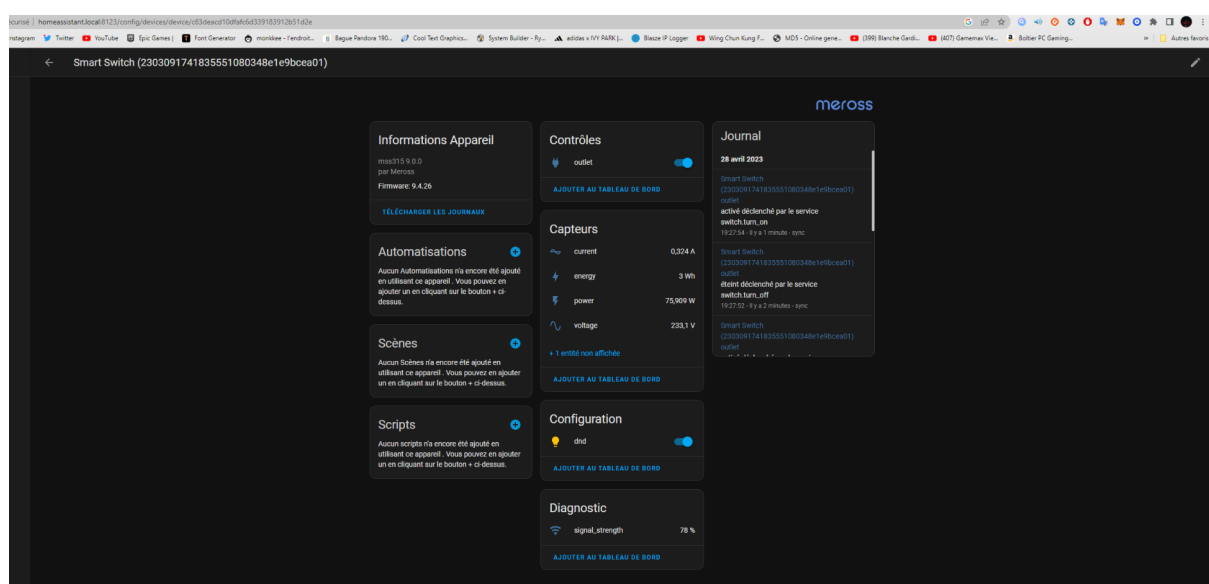


Figure 7 : Intégration Meross Matter sur Home Assistant

4. Simulateur de panneaux solaire

4.1 Rappel du contexte du projet

Pour rappel, le synchroniseur doit mesurer la consommation générale d'une habitation et la production des panneaux solaires. Pour cela, je dois simuler un panneau solaire, mon professeur m'as donc fournit un direction à suivre :

```
Bonjour Yassine,  
  
Merci pour ton retour.  
  
Voici ce que je propose que tu fasses, il te faut un simulateur de panneaux solaires, en réalisant 3 jours de production type :  
  
Un jour de production sans nuage : de 8h à 10h=1000 W, de 10h à 12h=2000W, de 12h à 14h=3000W, de 14 à 16h=2000W et de 16h à 18h=1000W  
Un jour nuageux de production : de 8h à 10h=500 W, de 10h à 12h=1000W, de 12h à 14h=1500W, de 14 à 16h=1000W et de 16h à 18h=500W  
Un jour pluvieux de production : de 8h à 10h=100 W, de 10h à 12h=500W, de 12h à 14h=100W, de 14 à 16h=500W et de 16h à 18h=100W  
  
En gros, il te faut une fonction donnant ce type de résultat : production = onduleurVirtuel(typeEnsoleillement, heure)  
typeEnsoleillement=0, 1 ou 2  
  
Tu peux prendre encore un peu de temps, car je ne peux pas placer de soutenance/démonstration avant le 31 mai.  
  
Bien cordialement,
```

Figure 8 : Échange avec le conseil de M. Spies

Par ce mail, on comprend donc que je dois simuler un panneaux solaire, qui peut calculer la production d'énergie sur une période de trois jours différents. Chaque jour est décrit en termes d'ensoleillement, et la production d'énergie est donnée pour différentes plages horaires.

Par la suite, je dois créer une fonction qui peut calculer la production d'énergie en fonction du jour 0, 1 ou 2 et enfin la retourner.

4.2.1 Développement du simulateur de panneaux solaire

Dans un premier temps j'ai réfléchis et commencé le développement d'un script, qui avait une structure de principalement "if" et "else", c'est à dire, prenons par exemple le jour 0 :

```
if typeEnsoleillement == 0:  
    if heure >= 8 and heure < 10:  
        return 1000  
    elif heure >= 10 and heure < 12:  
        return 2000  
    elif heure >= 12 and heure < 14:  
        return 3000
```

Ainsi de suite...

J'ai trouvé cependant une meilleure solution, j'ai donc opté pour l'utilisation d'un "dictionnaire" afin de stocker les valeurs de production d'énergie en fonction du type d'ensoleillement et en fonction de l'heure. Par la suite, je relie chaque type d'ensoleillement à une clé, dans mon cas 0 pour un jour sans nuage, 1 pour un jour nuageux ou 2 pour un jour pluvieux.

4.2.2 Analyse du programme de simulation du panneau solaire

Analysons maintenant dans les détails le programme complet .

Le module time est importé pour pouvoir utiliser la fonction time.localtime().tm_hour qui permet d'obtenir l'heure actuelle.

```
1 import time
```

La fonction onduleurVirtuel prend deux paramètres : typeEnsoleillement qui représente le type d'ensoleillement (0, 1 ou 2) et heure qui représente l'heure actuelle. À l'intérieur de la fonction, un dictionnaire production_par_type est défini pour stocker les valeurs de production d'énergie par type d'ensoleillement et par heure. La fonction vérifie si le type d'ensoleillement donné est présent dans le dictionnaire, puis renvoie la valeur de production d'énergie correspondante pour l'heure spécifiée. Si le type d'ensoleillement n'est pas trouvé dans le dictionnaire, la fonction renvoie 0.

```
def onduleurVirtuel(typeEnsoleillement, heure):
    production_par_type = {
        #1h 2h 3h 4h 5h 6h 7h 8h 9h 10h 11h 12h 13h 14h 15h 16h 17h 18h 19h 20h 21h 22h 23h 24h
        0: [0, 0, 0, 0, 0, 0, 0, 0, 1000, 1000, 2000, 2000, 3000, 3000, 2000, 2000, 1000, 1000, 750, 750, 500, 500, 250, 250, 210], #Un jour de production sans nuage
        1: [0, 0, 0, 0, 0, 0, 0, 0, 500, 500, 1000, 1000, 1500, 1500, 1000, 1000, 500, 500, 250, 250, 100, 100, 50, 50, 10], #Un jour nuageux de production
        2: [0, 0, 0, 0, 0, 0, 0, 0, 100, 100, 500, 500, 100, 100, 500, 500, 100, 100, 50, 50, 25, 25, 10, 10, 5] #Un jour pluvieux de production
    }

    if typeEnsoleillement in production_par_type:
        return production_par_type[typeEnsoleillement][heure]
    else:
        return 0
```

Cette partie du code utilise une boucle while pour demander à l'utilisateur de saisir le type d'ensoleillement (un entier entre 0 et 2). La saisie de l'utilisateur est convertie en entier à l'aide de int(type_ensoleillement). Si la saisie n'est pas un entier valide ou si elle est en dehors de la plage attendue, un message d'erreur est affiché et la variable type_ensoleillement est réinitialisée à None, ce qui demande à l'utilisateur de saisir une nouvelle valeur.

```
16 type_ensoleillement = None # Stock du type d'ensoleillement (pour la question)
17
18 while type_ensoleillement is None:
19     type_ensoleillement = input("Entrez le type d'ensoleillement (entier entre 0 et 2) : ")
20     try:
21         type_ensoleillement = int(type_ensoleillement)
22         if type_ensoleillement < 0 or type_ensoleillement > 2:
23             print("Type d'ensoleillement invalide. Veuillez entrer un entier entre 0 et 2.")
24             type_ensoleillement = None
25     except ValueError:
26         print("Type d'ensoleillement invalide. Veuillez entrer un entier entre 0 et 2.")
27         type_ensoleillement = None
```

Dans cette boucle infinie, l'heure actuelle est obtenue en utilisant time.localtime().tm_hour et stockée dans la variable heure_actuelle. Ensuite, la fonction onduleurVirtuel est appelée avec les paramètres type_ensoleillement et heure_actuelle pour obtenir la production d'énergie correspondante, qui est stockée dans la variable production. Enfin, l'heure actuelle et la production d'énergie sont affichées à l'écran, suivies d'une pause de 10 secondes avant de recommencer la boucle pour obtenir la nouvelle production d'énergie.

```
29 while True:
30     heure_actuelle = time.localtime().tm_hour
31     production = onduleurVirtuel(type_ensoleillement, heure_actuelle)
32     print("Heure :", heure_actuelle, "Production d'énergie actuelle :", production, "W")
33
34     time.sleep(10)
```

Ce processus se répète indéfiniment, simulant ainsi la production d'énergie d'un onduleur virtuel en fonction de l'heure et du type d'ensoleillement spécifiés.

5. Conclusion

Au cours de cette deuxième session de stage, j'ai pu poursuivre mes travaux sur le projet en focalisant sur les éléments clés nécessaires à sa réussite. Bien que le projet ne soit pas encore totalement achevé, j'ai pu aborder les parties essentielles de celui-ci.

Tout d'abord, j'ai réussi à simuler un capteur de mesure de courant, ensuite, une fois le problème de compatibilité résolu, j'ai pu adapter cette simulation à un capteur de mesure de courant réel, ce qui m'a offert l'opportunité de mettre en pratique mes connaissances et de résoudre des problèmes techniques.

Par la suite, j'ai testé et intégré avec succès une prise connectée compatible avec la norme Matter. Enfin, j'ai également réalisé une simulation d'un panneau solaire en prenant en compte des conditions spécifiques à certains jours.

Bien que le projet ne soit pas encore totalement finalisé, les avancées réalisées jusqu'à présent sont prometteuses. Je suis très reconnaissant envers mon superviseur M. Spies pour son soutien et son encadrement tout au long de ce stage.