

Projet Capteurs & réseaux

Auteurs : Oliver BELLIARD & Valérien PRIOU

Enseignant encadrant : Christophe BARES

Boîte utilisée : n°6

Table des matières

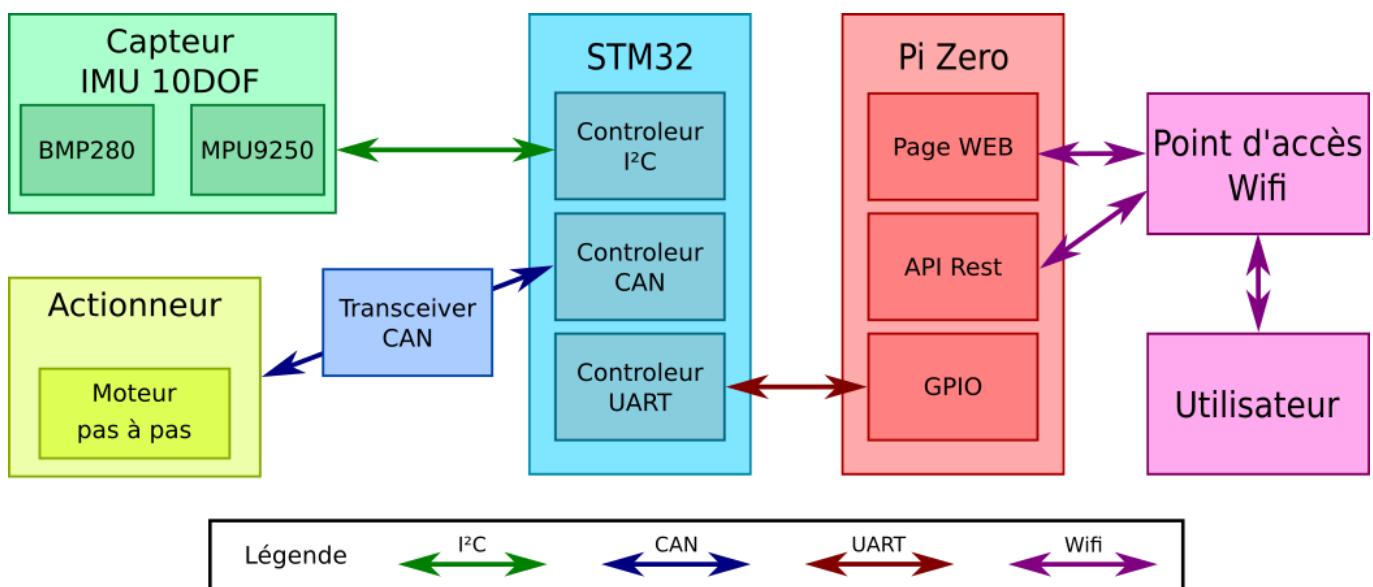
Générée automatiquement avec l'outil : [Markdown TOC generator](#). Il faut choisir l'optimisation pour GitHub.

- 1. Présentation
- 2. TP 1 - Bus I2C
 - 2.1. Capteur BMP280
 - 2.2. Setup du STM32
 - Configuration du STM32
 - Redirection du print
 - Test de la chaîne de compilation et communication UART sur USB
 - 2.3. Communication I²C
 - Primitives I²C sous STM32_HAL
 - Communication avec le BMP280
 - Identification du BMP280
 - Configuration du BMP280
 - Récupération de l'étalonnage, de la température et de la pression
 - Calcul des températures et des pressions compensées
- 3. TP2 - Interfaçage STM32 - Raspberry
 - 3.1. Mise en route du Raspberry PI Zéro
 - Préparation du Raspberry
 - Premier démarrage
 - 3.2. Port Série
 - Loopback
 - Communication avec la STM32
 - 3.3. Commande depuis Python
- 4. TP3 - Interface REST
 - 4.1. Installation du serveur Python
 - Installation
 - Premier fichier Web
 - Télécharger des fichiers depuis le Raspberry
 - Méthode manuelle
 - Script bash pour télécharger un dossier et son contenu
 - 4.2. Première page REST
 - Première route
 - Première page REST
 - Réponse JSON
 - 1re solution

- 2e solution
- Erreur 404
- 4.3. Nouvelles méthodes HTTP
 - Méthodes POST, PUT, DELETE...
 - Méthode POST
 - API CRUD
- 4.4. Et encore plus fort...
- 5. TP4 - Bus CAN
 - 5.1. Pilotage du moteur
 - 5.2. Interfaçage avec le capteur
- 6. TP5 - Intégration I²C - Serial - REST - CAN
- Aller plus loin
 - Doxygen

1. Présentation

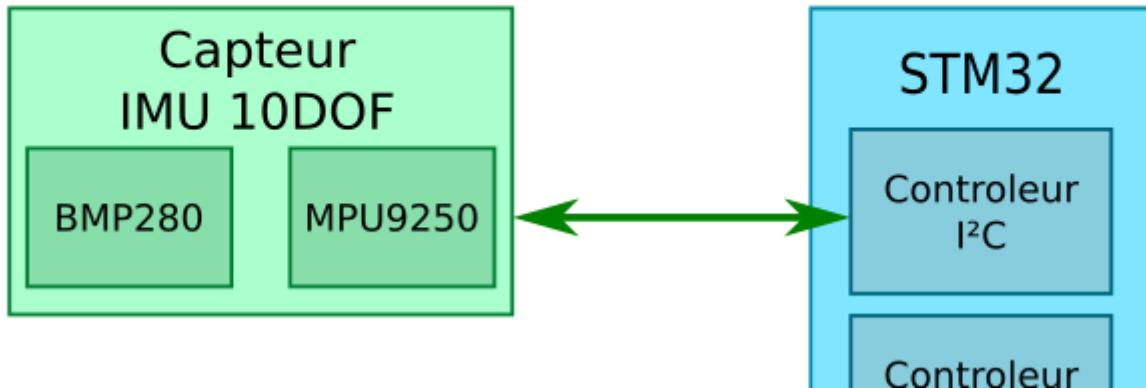
Ensemble de TP de Bus & Réseaux. Ces TP seront réalisés en C pour la partie STM32, et Python pour la partie Raspberry Pi.



L'échelonnement des TP est le suivant:

1. Interrogation des capteurs par le bus I²C
2. Interfaçage STM32 <-> Raspberry Pi
3. Interface Web sur Raspberry Pi
4. Interface API Rest & pilotage d'actionneur par bus CAN

2. TP 1 - Bus I²C



2.1. Capteur BMP280

À partir de la datasheet du [BMP280](#), on identifie les éléments suivants:

- les adresses I²C possibles pour ce composant :**

Ce composant dispose de deux adresses sur 7 bits 111011x disponibles en slave mode: 0x76 et 0x77. On peut changer cette adresse en mettant à 0 ou 1 le bit "SDO".

- le registre et la valeur permettant d'identifier ce composant :**

Le registre permettant d'identifier ce composant est à l'adresse 0xD0, il doit contenir un chip_id à la valeur "0x58".

- le registre et la valeur permettant de placer le composant en mode normal :**

Le mode normal peut être activé en modifiant le registre de control à l'adresse 0xF4, c'est un registre de 2 bits qu'il faut mettre à la valeur "11".

- les registres contenant l'étalonnage du composant :**

Les registres d'étalonnage du composant sont situés aux adresses "0x88, ..., 0xA1"

- les registres contenant la température (ainsi que le format) :**

Les registres contenant les valeurs de température sont présents aux adresses "0xFA, 0xFB, 0xFC". La température est codée sur 20 bits, d'où la nécessité d'avoir plusieurs adresses. FA le MSB, FB le LSB et FC le XLSB.

- les registres contenant la pression (ainsi que le format) :**

Les registres contenant les valeurs de pression sont présents aux adresses "0xF7, 0xF8, 0xF9". La pression est codée sur 20 bits, d'où la nécessité d'avoir plusieurs adresses. F7 le MSB, F8 le LSB et F9 le XLSB.

- les fonctions permettant le calcul de la température et de la pression compensées, en format entier 32 bits :**

Les fonctions permettant les calculs compensés de la température et de la pression sont données dans la datasheet page 22. On doit se servir de coefficients de compensation qui sont détaillés dans le code qui y est donné :

```
// Returns temperature in DegC, resolution is 0.01 DegC. Output value of
// "5123" equals 51.23 DegC.
```

```

// t_fine carries fine temperature as global value
BMP280_S32_t t_fine;
BMP280_S32_t bmp280_compensate_T_int32(BMP280_S32_t adc_T)
{
    BMP280_S32_t var1, var2, T;
    var1 = (((adc_T>>3) - ((BMP280_S32_t)dig_T1<<1))) *
((BMP280_S32_t)dig_T2)) >> 11;
    var2 = (((((adc_T>>4) - ((BMP280_S32_t)dig_T1)) * ((adc_T>>4) -
((BMP280_S32_t)dig_T1))) >> 12) *
((BMP280_S32_t)dig_T3)) >> 14;
    t_fine = var1 + var2;
    T = (t_fine * 5 + 128) >> 8;
    return T;
}

// Returns pressure in Pa as unsigned 32 bit integer in Q24.8 format (24
integer bits and 8 fractional bits).
// Output value of "24674867" represents 24674867/256 = 96386.2 Pa =
963.862 hPa
BMP280_U32_t bmp280_compensate_P_int64(BMP280_S32_t adc_P)
{
    BMP280_S64_t var1, var2, p;
    var1 = ((BMP280_S64_t)t_fine) - 128000;
    var2 = var1 * var1 * (BMP280_S64_t)dig_P6;
    var2 = var2 + ((var1*(BMP280_S64_t)dig_P5)<<17);
    var2 = var2 + (((BMP280_S64_t)dig_P4)<<35);
    var1 = ((var1 * var1 * (BMP280_S64_t)dig_P3)>>8) + ((var1 *
(BMP280_S64_t)dig_P2)<<12);
    var1 = (((((BMP280_S64_t)1)<<47)+var1))*((BMP280_S64_t)dig_P1)>>33;

    if (var1 == 0)
    {
        return 0; // avoid exception caused by division by zero
    }

    p = 1048576-adc_P;
    p = (((p<<31)-var2)*3125)/var1;
    var1 = (((BMP280_S64_t)dig_P9) * (p>>13) * (p>>13)) >> 25;
    var2 = (((BMP280_S64_t)dig_P8) * p) >> 19;
    p = ((p + var1 + var2) >> 8) + (((BMP280_S64_t)dig_P7)<<4);
    return (BMP280_U32_t)p;
}

```

2.2. Setup du STM32

Configuration du STM32

Sous STM32CubeIDE, mettre en place une configuration adaptée à votre carte de développement STM32.

Pour ce TP, vous aurez besoin des connections suivantes:

- d'une liaison I²C. Si possible, on utilisera les broches compatibles avec l'empreinte arduino (broches PB8 et PB9) ([Doc nucleo 64](#))
- d'une UART sur USB (UART2 sur les broches PA2 et PA3) ([Doc nucleo 64](#))
- d'une liaison UART indépendante pour la communication avec le Raspberry (TP2)
- d'une liaison CAN (TP4)

Prenez soin de bien choisir les ports associés à chacun de ces bus.

Redirection du print

Afin de pouvoir facilement déboguer votre programme STM32, faites en sorte que la fonction printf renvoie bien ses chaînes de caractères sur la liaison UART sur USB, en ajoutant le code suivant au fichier [stm32f4xx_hal_msp.c](#):

```
/* USER CODE BEGIN PV */
extern UART_HandleTypeDef huart2;
/* USER CODE END PV */

/* USER CODE BEGIN Macro */
#ifndef __GNUC__ /* With GCC/RAISONANCE, small printf (option LD Linker->Libraries->Small printf set to 'Yes') calls __io_putchar() */
#define PUTCHAR_PROTOTYPE int __io_putchar(int ch)
#else
#define PUTCHAR_PROTOTYPE int fputc(int ch, FILE *f)
#endif /* __GNUC__ */
/* USER CODE END Macro */

/* USER CODE BEGIN 1 */
/**
 * @brief Retargets the C library printf function to the USART.
 * @param None
 * @retval None
 */
PUTCHAR_PROTOTYPE
{
    /* Place your implementation of fputc here */
    /* e.g. write a character to the USART2 and Loop until the end of
transmission */
    HAL_UART_Transmit(&huart2, (uint8_t *)&ch, 1, 0xFFFF);

    return ch;
}
/* USER CODE END 1 */
```

Il est aussi possible de réécrire la fonction [__io_putchar](#) en reprenant son prototype, c'est la méthode que nous avons choisi :

```
/* Private user code -----
---*/
```

```

/* USER CODE BEGIN 0 */
/**
 * @brief Transmit a character over UART.
 * @param ch: Character to transmit.
 * @retval int: The transmitted character.
 */
int __io_putchar(int ch)
{
    HAL_UART_Transmit(&huart2, (uint8_t *)&ch, 1, 0xFFFF);
    return ch;
}

/* USER CODE END 0 */

```

Cette approche permet de ne pas aller modifier d'autres fichiers et de centraliser cette unique fonction de comportement de la fonction `printf` dans le `main.c`.

Test de la chaîne de compilation et communication UART sur USB

Testez ce printf avec un programme de type echo.

Pour voir le retour de la liaison série USB (USART2) nous utilisons le programme "minicom" sur linux (Ubuntu) qui permet de tout gérer depuis un terminal. Une fois le paquet installé, il suffit de lancer la communication avec la commande suivante sur un terminal :

```
minicom -D dev/ttyACM0
```

Ici notre périphérique est le `ttyACM0` mais il peut varier selon les postes ou l'appareil depuis lequel il est lancé. Une fois lancée la commande et la carte redémarrée, on observe sur notre terminal :

```

Welcome to minicom 2.9

OPTIONS: I18n
Port /dev/ttyACM0, 09:36:39

Press CTRL-A Z for help on special keys

==== TP Capteurs & Réseaux ====

```

la ligne `==== TP Capteurs & Réseaux ===` correspond au premier `printf` de notre programme, on sait alors que notre redirection du flux de la fonction a bien été pris en compte.

2.3. Communication I²C

Primitives I²C sous STM32_HAL

L'API HAL (Hardware Abstraction Layer) fournit par ST propose entre autres 2 primitives permettant d'interagir avec le bus I²C en mode Master:

- `HAL_StatusTypeDef HAL_I2C_Master_Transmit(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size, uint32_t Timeout)`
- `HAL_StatusTypeDef HAL_I2C_Master_Receive(I2C_HandleTypeDef *hi2c, uint16_t DevAddress, uint8_t *pData, uint16_t Size, uint32_t Timeout)`

où:

- `I2C_HandleTypeDef hi2c`: structure stockant les informations du contrôleur I²C
- `uint16_t DevAddress`: adresse I²C du périphérique Slave avec lequel on souhaite interagir.
- `uint8_t *pData`: buffer de données
- `uint16_t Size`: taille du buffer de données
- `uint32_t Timeout`: peut prendre la valeur `HAL_MAX_DELAY`

Communication avec le BMP280

Identification du BMP280

L'identification du BMP280 consiste en la lecture du registre ID.

En I²C, la lecture se déroule de la manière suivante :

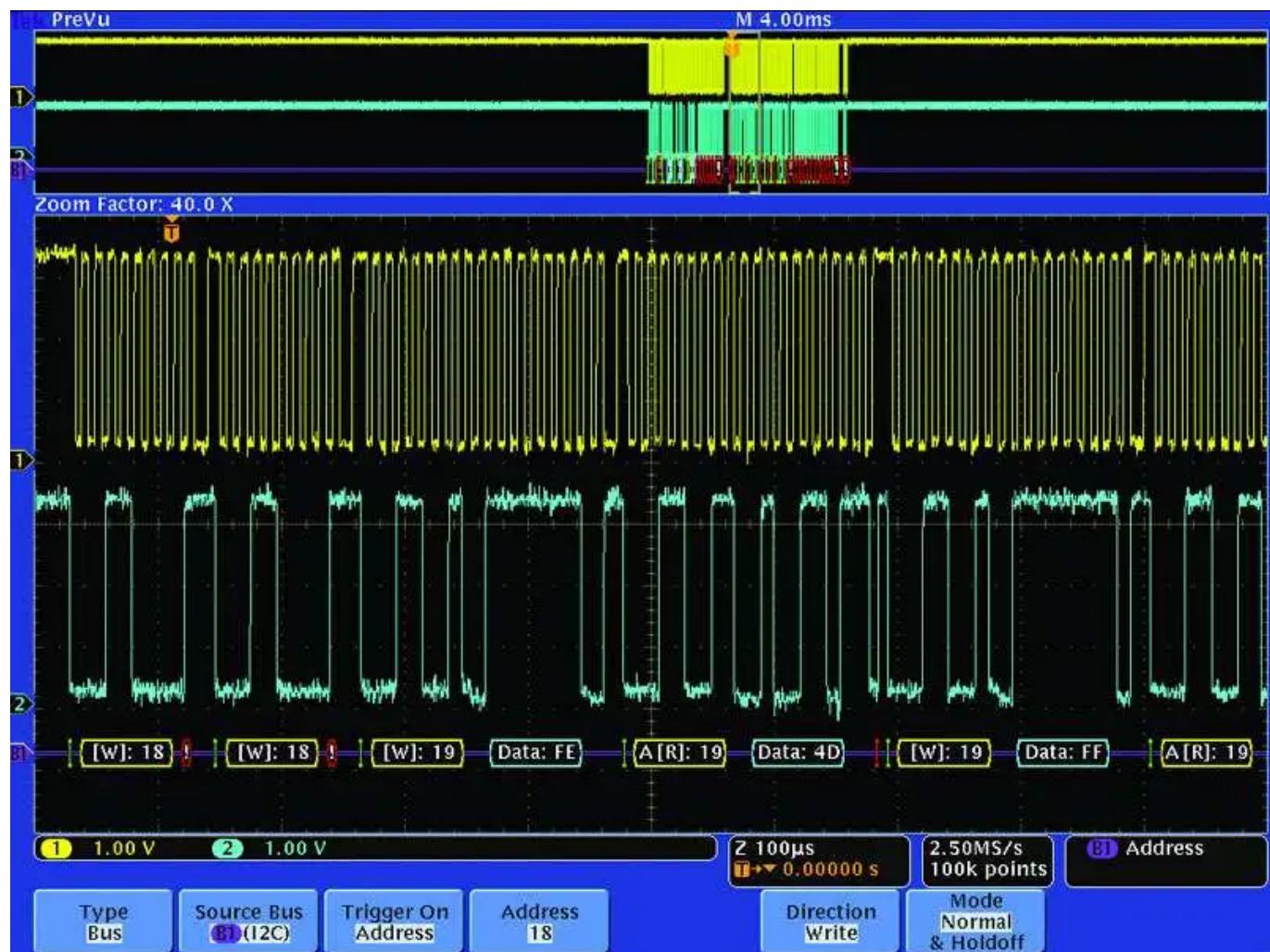
1. envoyer l'adresse du registre ID
2. recevoir 1 octet correspondant au contenu du registre

Vérifiez que le contenu du registre correspond bien à la datasheet :

Lorsque nous faisons une requête I²C à notre capteur, nous recevons bien `0x58`, l'ID du chip.

Vérifiez à l'oscilloscope que la forme des trames I²C est conforme :

N'ayant pas eu le temps de visualiser nos signaux à l'oscilloscope, nous avons emprunté cette capture d'écran prise avec un oscilloscope comme ceux présent à l'ENSEA de [ce site](#)



On observe bien le signal d'orloge en jaune et la trame de données en cyan.

Configuration du BMP280

Avant de pouvoir faire une mesure, il faut configurer le BMP280.

Pour commencer, nous allons utiliser la configuration suivante: mode normal, Pressure oversampling x16, Temperature oversampling x2

En I²C, l'écriture dans un registre se déroule de la manière suivante:

1. envoyer l'adresse du registre à écrire, suivi de la valeur du registre
2. si on reçoit immédiatement, la valeur reçue sera la nouvelle valeur du registre

Vérifiez que la configuration a bien été écrite dans le registre.

Récupération de l'étalonnage, de la température et de la pression

Récupérez en une fois le contenu des registres qui contiennent l'étalonnage du BMP280.

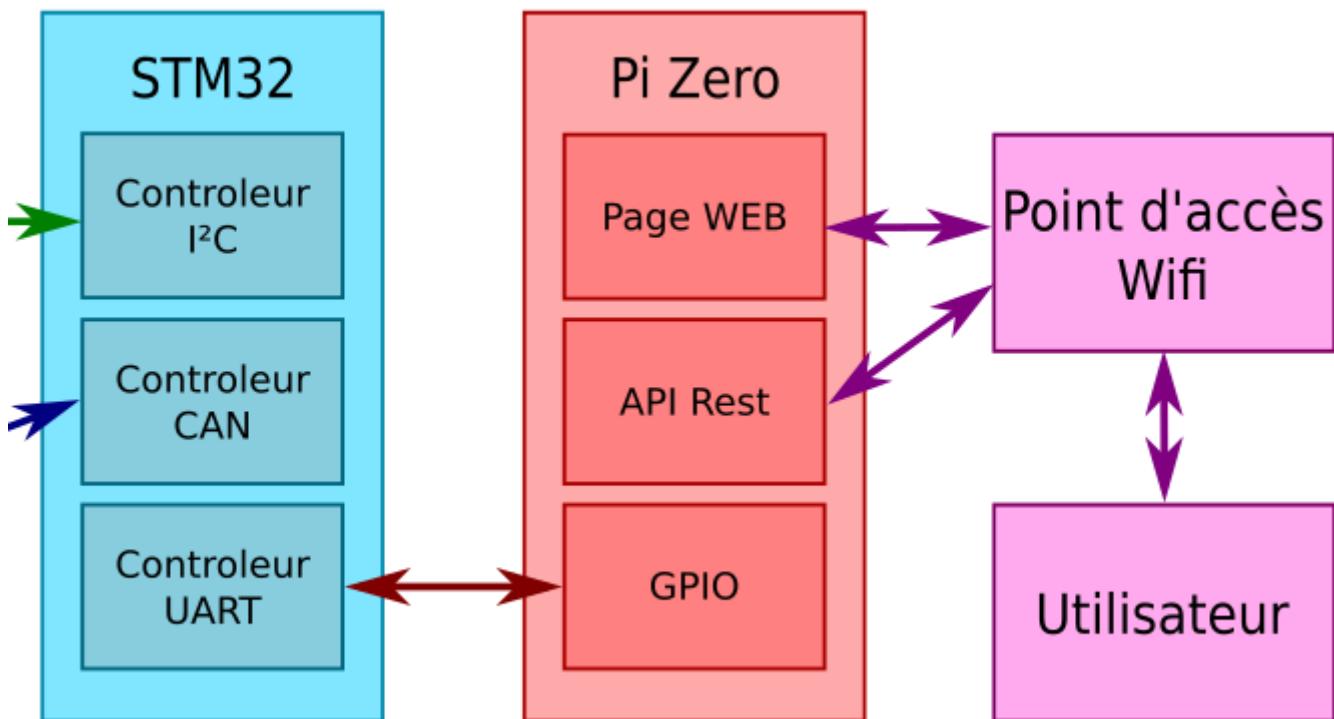
Dans la boucle infinie du STM32, récupérez les valeurs de la température et de la pression. Envoyez sur le port série le valeurs 32 bit non compensées de la pression de la température.

Calcul des températures et des pression compensées

Retrouvez dans la datasheet du STM32 le code permettant de compenser la température et la pression à l'aide des valeurs de l'étalonnage au format entier 32 bits (on utilisera pas les flottants pour des problèmes de performance).

Transmettez sur le port série les valeurs compensés de température et de pression sous un format lisible.

3. TP2 - Interfaçage STM32 - Raspberry



3.1. Mise en route du Raspberry PI Zéro

Le Pi Zero est suffisamment peu puissant pour être alimenté par le port USB de l'ordinateur.

Préparation du Raspberry

Téléchargez l'image "Raspberry Pi OS (32-bit) Lite" et installez la sur la carte SD, disponible à cette adresse : <https://www.raspberrypi.org/downloads/raspberry-pi-os/>.

Pour l'installation sur la carte SD, nous avons utilisé : Rpi_Imager: <https://www.raspberrypi.com/software/>

Rpi_imager va nous permettre de choisir l'image, de la télécharger et de la configurer.

Configuration réseau du routeur utilisé en TP :

SSID : ESE_Bus_Network

Password : ilovelinux

Pour activer le port série sur connecteur GPIO, sur la partition boot, on a modifié le fichier config.txt pour ajouter à la fin les lignes suivantes :

```
enable_uart=1  
dtoverlay=disable-bt
```

Premier démarrage

On installe la carte SD dans le Raspberry puis on branche l'alimentation.

On peut alors se connecter à notre Raspberry via SSH en utilisant ces IDs :

```
utilisateur : voese  
mdp : voese  
IP : 192.168.88.235
```

Pour nous connecter en SSH au Raspberry nous utilisons la commande :

```
ssh voese@192.168.88.235
```

Comment le Raspberry a obtenu son adresse IP ?

Le Raspberry Pi a obtenu son adresse IP via le réseau Wi-Fi auquel il s'est connecté. Celui-ci est de la forme "192.168.88.XXX" avec les "X" définis par l'ordre auquel on est arrivé sur le réseau par rapport aux autres élèves de la classe.

On doit ensuite effectuer des modifications dans le fichier config.txt pour ajouter à la fin du fichier les lignes suivantes :

```
enable_uart=1  
dtoverlay=disable-bt
```

Ensuite, dans la fichier cmdline.txt on doit retirer l'option "console=serial0,115200".

Pour vérifier si le port série de la Raspberry fonctionne bien, on branche d'abord son TX et RX en loopback et on utilise la commande "minicom -D /dev/ttyAMA0" pour voir si les caractères qu'on rentre nous sont renvoyés, ce qui est vrai dans notre cas donc le port série de la Raspberry est correctement configuré. Pour voir l'attribution de notre arduino nous avons utilisé ce site : <https://pinout.xyz/>

On peut maintenant brancher les pins TX/RX de notre Raspberry sur le UART1 du STM32 et notre capteur en I2C sur la STM32. On pense au passage à ne pas oublier de ramener la masse commune à la Raspberry.

3.2. Port Série

Loopback

À fin de tester le bon fonctionnement du port série de notre Raspberry Pi, nous avons mis en place une communication en loopback électriquement. Pour cela, nous avons branché le port série du Raspberry en boucle : RX sur TX.

Nous avons ensuite utilisé le logiciel **minicom**, mis en place précédemment sur le Raspberry, pour tester le port série. On lance le logiciel avec la ligne suivante, comme expliqué plus haut :

```
minicom -D /dev/ttyAMA0
```

Une fois dans **minicom**, il faut configurer le port série en pressant **CTRL+A** suivi de **0**.

Pensez à désactiver le contrôle de flux matériel (on n'utilise pas les lignes **RTS/CTS**).

Après, il suffit d'écrire quelques lettres au clavier pour vérifier le bon fonctionnement. Quand elles s'affichent, on sait que le loopback fonctionne.

Nous pouvons enfin appuyer sur **CTRL+A** puis **Q** pour quitter **minicom**.

Communication avec la STM32

! Attention : pour que le port UART utilisé pour la communication avec le PC (UART over USB) puisse fonctionner, les pins PA2 et PA3 ne sont pas par défaut connectés aux borniers CN9 et CN10. (C'est possible en jouant avec le fer à souder : doc Nucleo 64 page 27).

C'est pourquoi nous devons utiliser un 2e port UART sur le STM32, qui servira à la communication avec le Raspberry Pi (comme indiqué lors du TP1). Nous avons modifié notre fonction **printf** pour qu'elle envoie des caractères sur les 2 ports série en même temps et pouvoir debugger nos envois de données avec un terminal en écoute de la liaison série.

```
/** 
 * @brief Transmit a character over UART.
 * @param ch: Character to transmit.
 * @retval int: The transmitted character.
 */
int __io_putchar(int ch)
{
    HAL_UART_Transmit(&huart2, (uint8_t *)&ch, 1, 0xFFFF);
    HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, 0xFFFF);

    return ch;
}
```

Le protocole de communication entre le Raspberry et la STM32 est le suivant:

Requête du RPi	Réponse du STM	Commentaire
GET_T	T=+12.50_C	Température compensée sur 10 caractères
GET_P	P=102300Pa	Pression compensée sur 10 caractères
SET_K=1234	SET_K=OK	Fixe le coefficient K (en 1/100e)
GET_K	K=12.34000	Coefficient K sur 10 caractères
GET_A	A=125.7000	Angle sur 10 caractères

Les valeurs compensées de P et de T pourront être remplacées par les valeurs brutes hexadécimales sur 5 caractères (20 bits) suivis d'un "H", par exemple : T=7F54B2H

Ce protocole a été implémenté dans le STM32.

Une des difficultés rencontrées lors du développement de la reconnaissance des commandes reçues par la liaison série a été d'utiliser la fonction `strcmp` à la place de la fonction `strncmp` qui prend en compte la longueur et le contenu de deux chaînes de caractères lors de la comparaison.

Après nous avons branché le STM32 sur le Raspberry en prenant soin de croiser les signaux RX et TX. Les 2 fonctionnent en 3,3 V donc aucune adaptation de niveau n'est nécessaire.

Tous nos tests pour ce protocole ont été effectués depuis le Raspberry à l'aide de `minicom`, en envoyant des ordres manuellement et en vérifiant les valeurs renvoyées par le STM32. C'est à ce moment-là que nous avons remarqué que nos fonctions de calibrage n'étaillaient pas correctement nos capteurs. En revanche, les commandes sont reconnues et renvoient des données dans le format spécifié.

3.3. Commande depuis Python

Finalement pour que notre Raspberry puisse héberger une interface REST nous devons dans un premier installer dessus python 3 et plusieurs bibliothèques :

```
sudo apt update  
sudo apt install python3-pip
```

Installez ensuite la bibliothèque pyserial:

```
pip3 install pyserial
```

À partir de là, la bibliothèque est accessible après avoir effectué un : `import serial`

Plus d'infos sur : <https://pyserial.readthedocs.io/en/latest/shortintro.html>

À partir de là, nous pouvons réaliser un script en Python3 qui permet de communiquer avec le STM32. C'est ce que nous avons fait conjointement avec notre interface REST qui utilise cette fonctionnalité.

4. TP3 - Interface REST

4.1. Installation du serveur Python

Installation

On crée notre propre utilisateur avec les commandes :

```
sudo adduser <username>  
sudo usermod -aG sudo <username>
```

```
sudo usermod -aG dialout <username>
```

Notre nouvel utilisateur est :

```
utilisateur : vo  
mdp : voese
```

Nous avons créé un environnement Python pour notre projet avec les commandes :

```
python3 -m venv REST_VENV
```

Puis, pour activer l'environnement nous utilisons la commande :

```
source REST_VENV/bin/activate
```

Cependant, par soucis de simplicité, nous utiliseront le wrapper de venv : [pipenv](#).

Pour cela on installe le package :

```
sudo apt install pipenv
```

Puis on crée un environnement virtuel Python 3 :

```
pipenv --python 3
```

Pour activer l'environnement virtuel que nous venons de créer il suffit d'utiliser la commande :

```
pipenv shell
```

Ensuite il suffira de lancer la commande :

```
pipenv install -r requirements.txt
```

Pour installer les paquets requis contenus dans le fichier [requirements.txt](#) créé précédemment.

Premier fichier Web

On crée un fichier hello.py dans lequel avons placé le code suivant:

```
from flask import Flask
app = Flask(__name__)

@app.route('/')
def hello_world():
    return 'Hello, World!\n'
```

C'est notre nouveau serveur web! On peut le lancer avec:

```
pi@raspberrypi:~/server $ FLASK_APP=hello.py flask run
```

On peut tester votre nouveau serveur avec la commande curl (dans un 2e terminal):

```
pi@raspberrypi:~/server $ curl http://127.0.0.1:5000 -s -D -
```

- **-s** : Cette option active le "mode silencieux" pour supprimer les messages de progression ou d'erreur standard. Cela permet d'afficher uniquement la réponse.
- **-D -** : Cette option demande à curl d'afficher les en-têtes de la réponse HTTP (comme le code de statut, les en-têtes de type de contenu, etc.) au lieu de les enregistrer dans un fichier. Le - indique que les en-têtes seront affichés dans la sortie standard (le terminal).

Le problème est que votre serveur ne fonctionne pour le moment que sur la loopback. Cela est résolu avec :

```
pi@raspberrypi:~/server $ FLASK_APP=hello.py FLASK_ENV=development flask run --host 0.0.0.0
```

La constante **FLASK_ENV=development** permet de lancer un mode debug. À partir de maintenant, on peut tester notre serveur web avec un navigateur.

Une fois la commande lancée, le serveur se démarre :

```
(REST_server) vo@voese:~/REST_server $ FLASK_APP=hello.py
FLASK_ENV=development flask run --host 0.0.0.0
 * Serving Flask app 'hello.py'
 * Debug mode: off
WARNING: This is a development server. Do not use it in a production
deployment. Use a production WSGI server instead.
 * Running on all addresses (0.0.0.0)
 * Running on http://127.0.0.1:5000
 * Running on http://192.168.88.207:5000
Press CTRL+C to quit
```

On nous y explique qu'on peut maintenant se connecter au serveur depuis le localhost (<http://127.0.0.1:5000>) ou depuis une adresse "externe", ce qui nous intéresse ici car on cherche à tester notre serveur depuis mon PC personnel.

Depuis mon navigateur la connexion fonctionne bien et on obtient :

```
<html>
  <head>
    <meta http-equiv="content-type" content="text/html; charset=UTF-8">
  </head>
  <body>Welcome to 3ESE API!</body>
</html>
```

Qui s'affiche comme ci dessous :

Welcome to 3ESE API!

Côté serveur, cette ligne apparaît :

```
192.168.88.222 - - [25/Oct/2024 08:37:18] "GET / HTTP/1.1" 200 -
```

Signifiant que mon ordinateur s'est bien connecté à notre serveur REST.

Télécharger des fichiers depuis le Raspberry

Méthode manuelle

On peut profiter de la connection en SSH pour télécharger des fichiers pour notre contrôle de version. Il suffit d'utiliser le protocole SFTP (SSH File Transfer System) avec la commande suivante :

```
sftp vo@192.168.88.235
```

On rappelle qu'ici **vo** est notre utilisateur de **192.168.88.235** l'ip du Raspberry. Ensuite, pour télécharger le dossier de notre projet on utilise la commande :

```
get -r <nom_du_dossier>/
```

Le téléchargement du dossier et son contenu se télécharge alors dans le dossier où pointe notre terminal local. Nous avons trouvé [ce tuto](#) qui nous a permis de comprendre rapidement son fonctionnement avec la documentation à travers la commande **man sftp**.

Script bash pour télécharger un dossier et son contenu

Pour nous simplifier la tâche de gérer les versions et sauvegarder notre travail effectué sur le Raspberry, nous avons rédigés un script bash permettant de télécharger l'ensemble du dossier contenant notre serveur REST. Le script se présente comme ce ci :

```
# Warning: you need the expect command: sudo apt install expect

set timeout -1
spawn sftp vo@192.168.88.235
expect "password:"
send -- "voese\n"
expect "sftp>"
send -- "get -r REST_server/\n"
expect "sftp>"
send -- "bye\n"
```

En résumé, il accède au Raspberry via SFTP, ils s'identifie pour avoir accès à la machine et télécharge le dossier **REST_server**. Il est à noté que c'est une façon qui n'est pas très sécurisée du à la présence des identifiants dans le script.

4.2. Première page REST

Première route

Nous avons ajouté les lignes suivantes au fichier **hello.py**:

```
welcome = "Welcome to 3ESE API!"

@app.route('/api/welcome/')
def api_welcome():
    return welcome

@app.route('/api/welcome/<int:index>')
def api_welcome_index(index):
    return welcome[index]
```

Quel est le rôle du décorateur `@app.route`?

Le décorateur `@app.route` est utilisé dans le framework Flask (ou un autre framework web similaire) pour définir les routes d'une application web, c'est-à-dire les URL qui vont être associées à des fonctions spécifiques.

Le décorateur `@app.route('/api/welcome/')` lie l'URL `/api/welcome/` à la fonction `api_welcome`. Quand un utilisateur accède à cette URL, Flask exécute la fonction `api_welcome`, qui renvoie la chaîne de caractères "`Welcome to 3ESE API!`".

Quel est le rôle du fragment `<int:index>`?

Le fragment `<int:index>` dans `@app.route('/api/welcome/<int:index>')` permet de capturer une partie variable de l'URL, ici un entier (int) nommé index.

Lorsque cette URL est appelée avec un entier en tant que suffixe, comme `/api/welcome/5`, l'entier 5 sera

extrait de l'URL et passé à la fonction `api_welcome_index` sous forme de paramètre index.

Dans `api_welcome_index`, cet entier index est utilisé pour accéder au caractère de la chaîne welcome à l'indice spécifié.

Pour pouvoir prétendre être RESTful, notre serveur va devoir:

- répondre sous forme JSON.
- différencier les méthodes HTTP.

C'est ce que nous allons voir maintenant.

Première page REST

Réponse JSON

Un module JSON est disponible dans la librairie standard de python:

<https://docs.python.org/3/library/json.html> Le plus simple pour générer du JSON est d'utiliser la fonction `json.dumps()` sur un objet Python. On peut par exemple remplacer la dernière ligne de la fonction `api_welcome_index` par:

```
return json.dumps({"index": index, "val": welcome[index]})
```

Evidemment sans oublier le "import json" en début de fichier.

```
import json
```

Est-ce suffisant pour dire que la réponse est bien du JSON?

Non, car la page n'a pas encore été "JSONifiée", en Flask, pour indiquer explicitement que le contenu est de type JSON, il est préférable d'utiliser `jsonify`, qui va non seulement convertir les données en JSON mais aussi définir l'en-tête `Content-Type` à `application/json`, ce qui aide les clients à reconnaître le format de la réponse.

On peut observer en particulier les entêtes de la réponse: sous Firefox ou Chrome il faut ouvrir les outils de développement (F12), en sélectionnant l'onglet "réseau" et en rechargeant la page. On peut alors normalement trouver l'entête de réponse Content-Type: ce n'est effectivement pas du JSON!

1re solution

Il faut modifier la réponse renvoyée par flask, car la réponse par défaut est sous format HTML. On doit alors ajouter au contenu du return des entêtes personnalisées sous forme d'un dictionnaire:

```
return json.dumps({"index": index, "val": welcome[index]}), {"Content-Type": "application/json"}
```

À partir de maintenant la réponse est bien en JSON, et Firefox nous présente le résultat de manière différente (Chrome aussi, mais c'est moins visible).

2e solution

L'utilisation de json avec flask étant très fréquente, une fonction jsonify() existe dans la bibliothèque. Elle est accessible après un "from flask import jsonify". Cette fonction gère à la fois la conversion en json et l'ajout de l'entête.

Nous avons donc modifié notre code avant de le tester, dans le but d'implémenter jsonify() en écrivant notre code de cette façon :

```
return jsonify({"message": 'Hello, World!'})
```

nous obtenons bel et bien un résultat en json si nous le vérifions sur firefox.

Erreur 404

Il arrive souvent que les URL demandées soient fausses, il faut donc que notre serveur puisse renvoyer une erreur 404.

En créant un fichier vide en .html, en le nommant `page_not_found.html` et en copiant-collant le contenu du fichier suivant : `page_not_found.html` dedans, nous avons créé un template vers lequel nous sommes redirigés lorsque l'url entrée est fausse. Il est nécessaire de créer ce fichier dans un répertoire qu'on vient de créer et de nommer `templates`, cette nomenclature étant imposée par flask.

En ajoutant les lignes suivantes à notre hello.py, nous pouvons maintenant orienter le client vers la page d'erreur voulue :

```
@app.errorhandler(404)
def page_not_found(error):
    return render_template('page_not_found.html'), 404
```

On modifie donc notre fonction `api_welcome_index` pour implémenter cette fonction sur notre page d'accueil, celle-ci devient utile si l'index entré n'est pas correct, on utilise par ailleurs une fonction fournie par Flask à cet effet : `abort(404)`

D'autres méthodes auraient pu être utilisées, notamment `redirect` avec `url_for`.

4.3. Nouvelles méthodes HTTP

Méthodes POST, PUT, DELETE...

Pour être encore un peu plus RESTful, notre application doit gérer plusieurs méthodes (verb) HTTP

Méthode POST

Pour essayer une autre méthode on peut utiliser l'utilitaire `curl` sur linux (par exemple directement depuis le raspberry).

Cette méthode est utilisable si on installe curl au préalable à l'aide de cette ligne de commande :

```
sudo apt-get install curl
```

Une fois que curl est disponible on peut l'utiliser de la façon suivante :

```
curl -X POST http://192.168.88.235/api/welcome/14
```

On peut aussi utiliser l'extension RESTED de Firefox, c'est l'option qu'on a retenu car plus intuitive.

Notre Raspberry nous renvoie l'erreur 405. En effet, nous n'avons pas encore ajouté la liste des méthodes acceptées à notre route, on peut remédier à ce problème en écrivant notre route de cette façon :

```
@app.route('/api/welcome/int:index', methods=['GET','POST'])
```

Mais ajouter la méthode ne suffit pas, il faut aussi que notre fonction réagisse correctement à cette méthode. Dans le cas d'un POST, le corps de la requête contient des informations qui doivent être fournies à notre serveur REST.

A l'aide de l'extension RESTED de Firefox on teste la fonction suivante :

```
@app.route('/api/request/', methods=['GET', 'POST'])
@app.route('/api/request/<path>', methods=['GET', 'POST'])
def api_request(path=None):
    resp = {
        "method": request.method,
        "url" : request.url,
        "path" : path,
        "args": request.args,
        "headers": dict(request.headers),
    }
    if request.method == 'POST':
        resp["POST"] = {
            "data" : request.get_json(),
        }
    return jsonify(resp)
```

On arrive à peupler les champs `args` et `data` en effectuant une requête POST. En effet, dans la fonction fournie le champ "data" se remplit seulement si la requête reçue est un POST.

API CRUD

Pour que notre serveur soit "rested", il faudrait que celui-ci puisse supporter les requêtes suivantes :

CRUD	Method	Path	Action
Create	POST	welcome/	Change sentence
Retreive	GET	welcome/	Return sentence
Retreive	GET	welcome/x	Return letter x
Update	PUT	welcome/x	Insert new word at position x
Update	PATCH	welcome/x	Change letter at position x
Delete	DELETE	welcome/x	Delete letter at position x
Delete	DELETE	welcome/	Delete sentence

Pour chaque action, l'échange de donnée devrait se faire en JSON, et si une action ne renvoie rien, alors le code status de la réponse devrait être modifié. Par exemple, le POST doit retourner un [202 No Content](#).

Pour les codes de succès HTTP: <https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html#sec10.2>

Manquant de temps pour compléter chaque action correctement, nous avons décidé de pleinement supporter les requêtes GET et POST seulement.

4.4. Et encore plus fort...

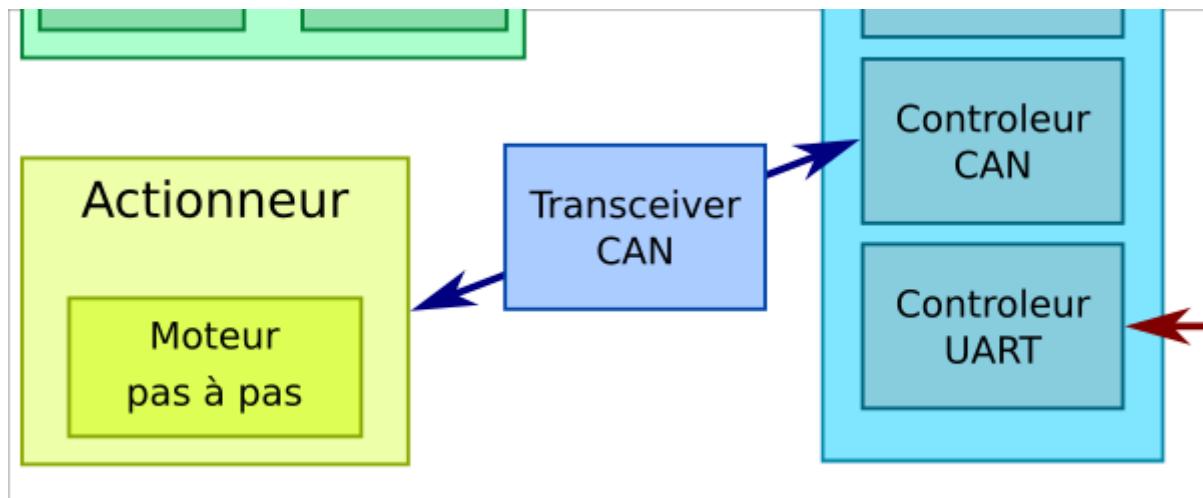
Le code écrit avec Flask pour créer une API REST est rapide, mais finalement il y a encore beaucoup de redondance...

Et donc, il y a encore plus fort: FASTAPI, <https://fastapi.tiangolo.com/>

En plus d'accélérer encore plus l'écriture du code, FASTAPI est auto-documenté... Pour améliorer encore plus notre serveur, une bonne initiative aurait été de réécrire notre code avec FASTAPI, et aller voir la page [/docs](#).

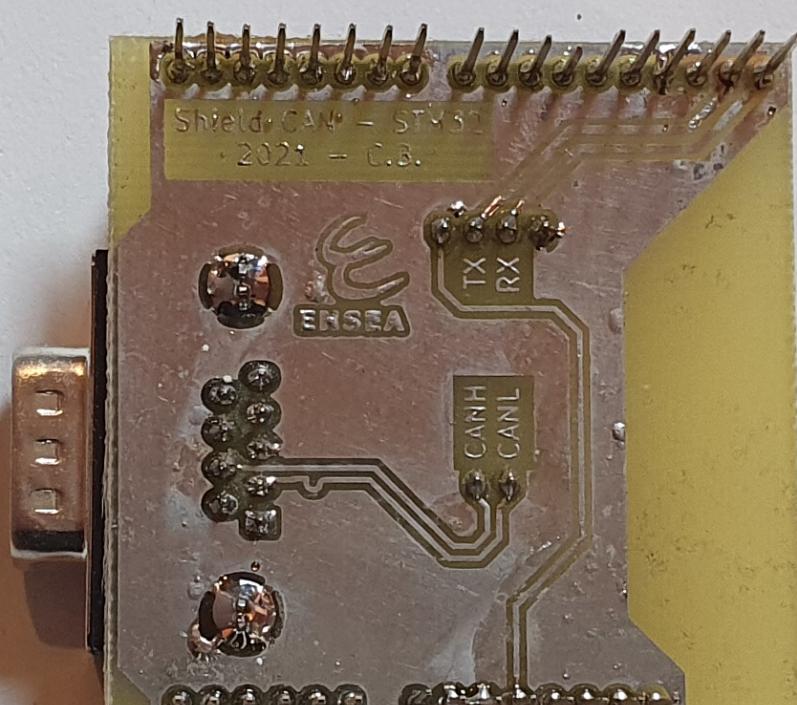
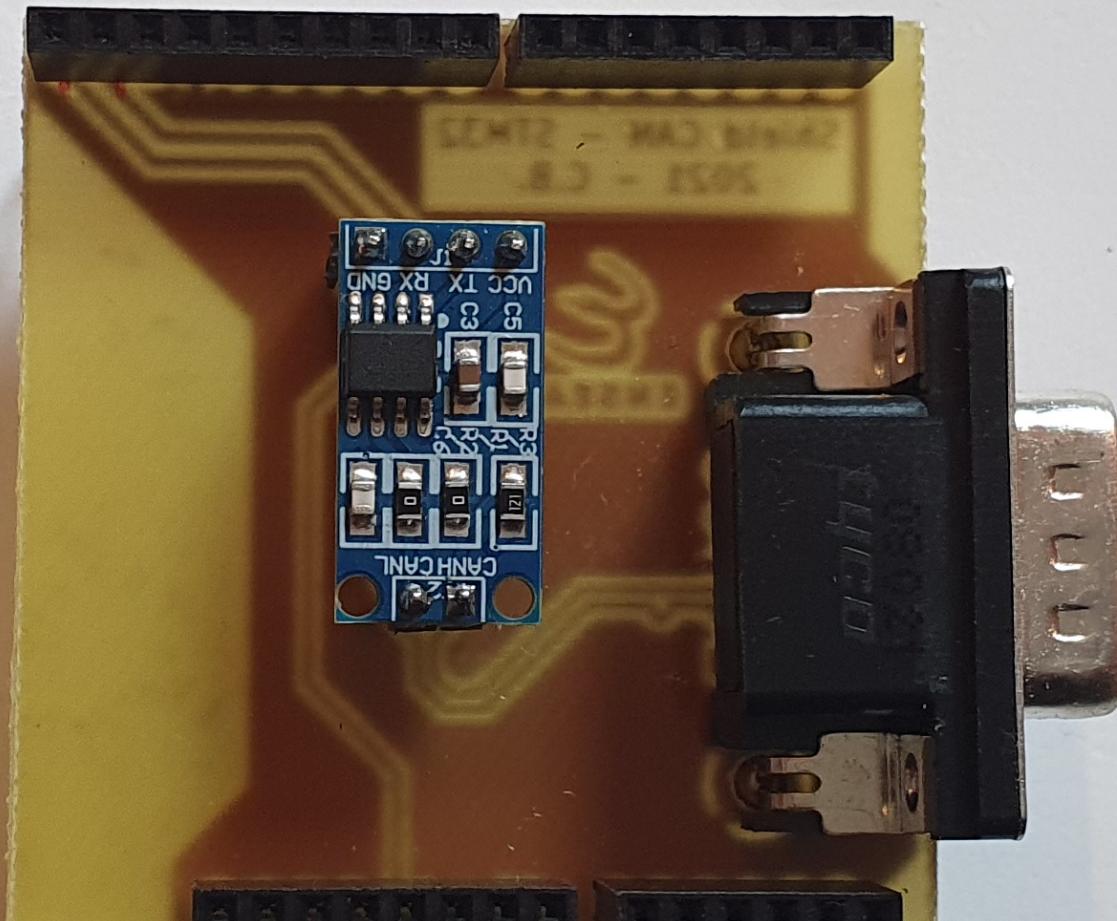
5. TP4 - Bus CAN

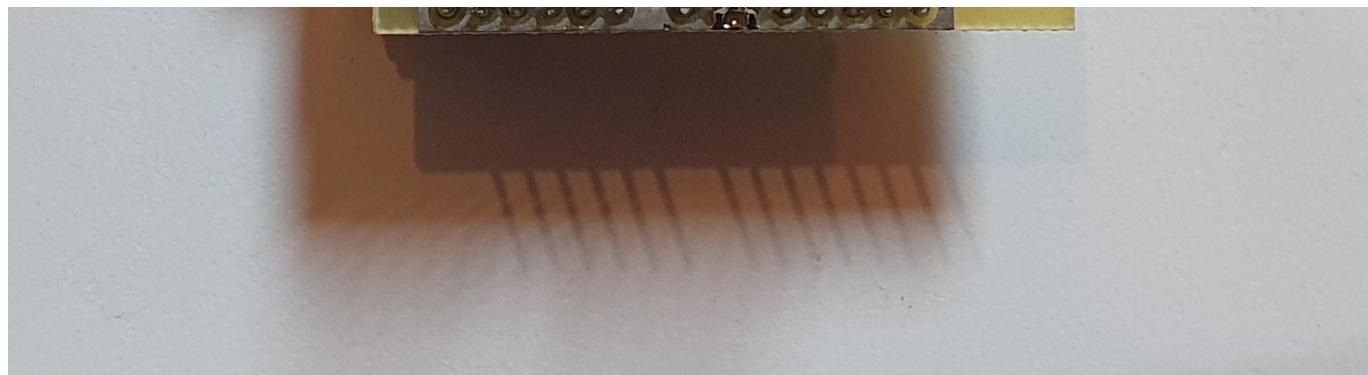
Objectif: Développement d'une API Rest et mise en place d'un périphérique sur bus CAN



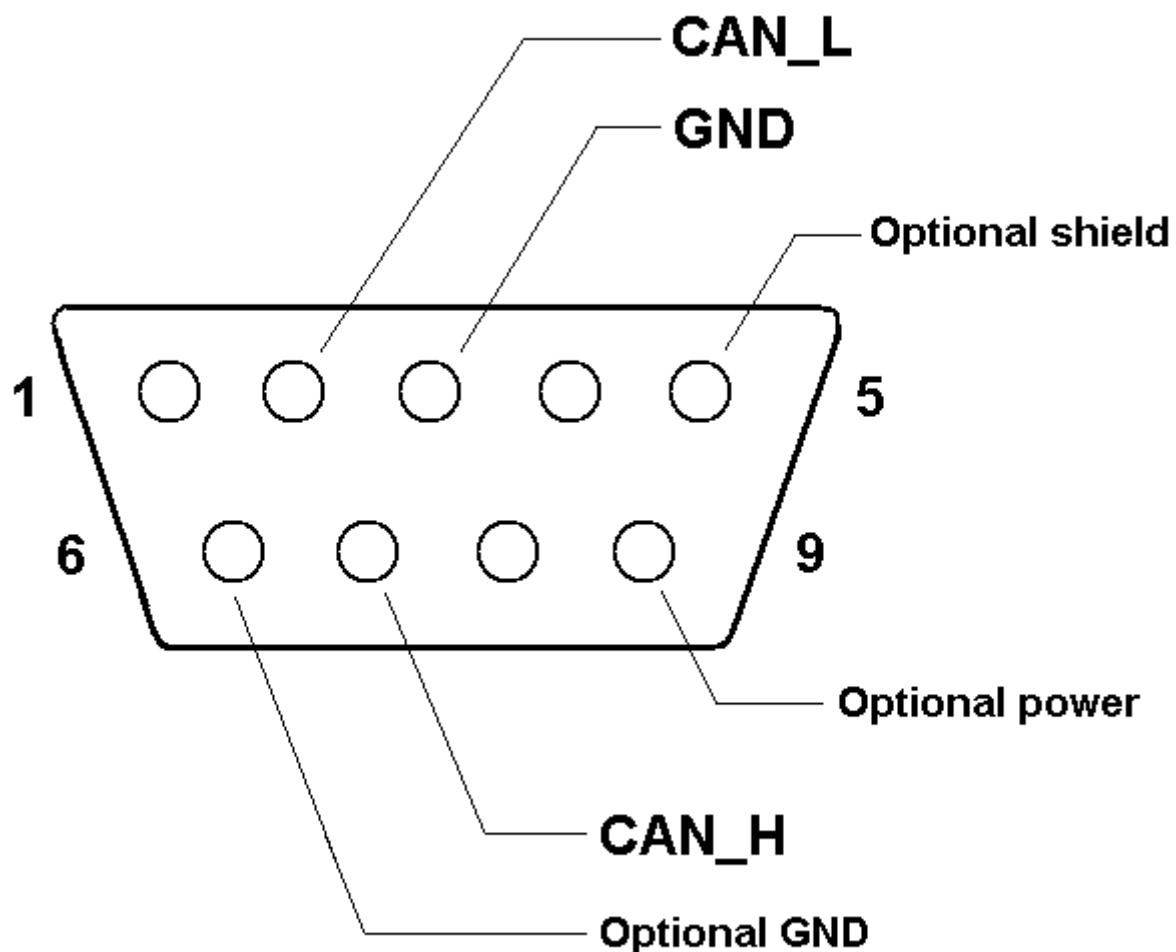
Notre carte STM32L476 est équipée d'un contrôleur CAN intégré. Pour pouvoir l'utiliser, il faut lui adjoindre un Transceiver CAN. Ce rôle est donné à un [TJA1050](#). Ce composant est alimenté en 5V, mais possède des Entrées / Sorties compatibles en 3,3 V.

Afin de faciliter sa mise en œuvre, ce composant a été installé sur une carte fille (shield) au format Arduino, qui peut donc se brancher sur notre carte nucleo 64.





Ce shield possède un connecteur subd9, qui nous permet de relier notre STM32 à notre moteur à l'aide d'un câble au format CAN. Pour rappel, le connecteur se broche de cette façon :



This is a male connector viewed from the connector side, or a female connector viewed from the soldering side.

Seules les broches 2, 3 et 7 sont utilisés sur les câbles à notre disposition.

On voit notamment que les lignes CANL et CANH ont été routées en tant que paire différentielle, et qu'une boucle a été ajouté à la ligne CANL pour la mettre à la même longueur que la ligne CANH. Sans quoi nous risquions d'avoir des interférences de par la différence de longueur entre ces deux pistes, ce qui est aurait pu compromettre toute la communication CAN.

Ce bus CAN va être utilisé pour communiquer avec un module de moteur pas-à-pas. Celui-ci s'alimente en 12v. L'ensemble des informations nécessaires pour utiliser ce module est disponible dans ce document: <https://moodle.ensea.fr/mod/resource/view.php?id=1921>

La carte moteur tolère qu'une vitesse CAN de 500kbit/s. Nous avons effectué les réglages nécessaires sur CubeMX pour atteindre cette vitesse en utilisant le calculateur suivant : <http://www.bittiming.can-wiki.info/>

Par ailleurs, les pins du shield qui étaient reliés au CAN tombaient sur l'emplacement qu'on avait d'abord alloué pour la communication I2C sur la STM32, on a donc dû allouer des pins différents en conséquence.

5.1. Pilotage du moteur

Pour Piloter ce moteur nous avons cherché à lui faire exécuter plusieurs fonctions en utilisant les primitives HAL suivantes:

```
HAL_StatusTypeDef HAL_CAN_Start (CAN_HandleTypeDef * hcan)
```

pour activer le module CAN et

```
HAL_StatusTypeDef HAL_CAN_AddTxMessage (CAN_HandleTypeDef * hcan,  
CAN_TxHeaderTypeDef * pHeader, uint8_t aData[], uint32_t * pTxMailbox)
```

pour envoyer un message, où:

`CAN_HandleTypeDef * hcan` pointe vers la structure stockant les infos du contrôleur CAN.

`CAN_TxHeaderTypeDef * pHeader` pointe vers la structure contenant les infos du header de la trame CAN à envoyer.

`uint8_t aData[] buffer` contient les données à envoyer et.

`uint32_t * pTxMailbox` pointe vers la boîte au lettre de transmission.

La variable `hcan` est définie par CubeMX, donc ce sera `hcan1`.

La variable `pHeader` est une structure contenant les champs suivants, que l'on remplit avant de faire appel à `HAL_CAN_AddTxMessage` :

.`StdId` contient le message ID quand celui-ci est standard (11 bits)

.`ExtId` contient le message ID quand celui-ci est étendu (29 bits)

.`IDE` définit si la trame est standard (CAN_ID_STD) ou étendue (CAN_ID_EXT)

.`RTR` définit si la trame est du type standard (CAN_RTR_DATA) ou RTR (CAN_RTR_REMOTE) (voir le cours)

.`DLC` est un entier représentant la taille des données à transmettre (entre 0 et 8)

.`TransmitGlobal` dispositif permettant de mesurer les temps de réponse du bus CAN, qu'on utilisera pas.

Le fixer à DISABLE

Dans un premier temps nous initialisons la communication CAN :

```

#define TRUE 1
#define FALSE 0

int logs; // Boolean to choose to display logs or not

/**
 * @brief Initialises the CAN communication
 */
void CAN_Init()
{
    HAL_StatusTypeDef status;
    logs = FALSE;

    status = HAL_CAN_Start(&hcan1);

    switch (status)
    {
        case HAL_OK:
            if (logs == TRUE) printf("CAN started successfully.\r\n");
            break;
        case HAL_ERROR:
            if (logs == TRUE) printf("Error: CAN start failed.\r\n");
            Error_Handler(); // Optional: Go to error handler
            break;
        case HAL_BUSY:
            if (logs == TRUE) printf("Warning: CAN is busy. Retry later.\r\n");
            // Optional: add retry logic if desired
            break;
        case HAL_TIMEOUT:
            if (logs == TRUE) printf("Error: CAN start timed out.\r\n");
            Error_Handler(); // Optional: Go to error handler
            break;
        default:
            if (logs == TRUE) printf("Unknown status returned from
HAL_CAN_Start.\r\n");
            Error_Handler(); // Optional: Go to error handler
            break;
    }
}

```

Lors de notre initialisation, nous nous assurons de gérer les différentes erreurs qui peuvent arriver pour rendre notre driver plus robuste.

Nous avons ensuite mis au point la fonction permettant d'envoyer nos messages en CAN :

```

/**
 * @brief Sends a CAN message with retry logic.
 *
 * This function attempts to send a message over the CAN bus to a specified
 * message ID (`msg_id`). If the CAN bus is busy, it will retry sending up
 * to

```

```
* a maximum number of attempts (`maxRetries`). In case of any other error
* (such as timeout or general error), the function will call
`Error_Handler()`
* to manage the failure.
*
* @param uint8_t* aData      Pointer to the data buffer containing the
message to send.
*                               The data should be in the form of an array of
`uint8_t`.
* @param uint32_t size       Size of the data in bytes (must match the Data
Length Code (DLC)
*                               field in the CAN frame).
* @param uint32_t msg_id     CAN message identifier (11-bit standard ID)
that defines the
*                               destination or type of the message being sent.
*
* @retval None
*/
void CAN_Send(uint8_t * aData, uint32_t size, uint32_t msg_id)
{
    HAL_StatusTypeDef status;
    CAN_TxHeaderTypeDef header;
    uint32_t txMailbox;
    int retryCount = 0;
    const int maxRetries = 5;

    // Initialiser le header
    header.StdId = msg_id;
    header.IDE = CAN_ID_STD;
    header.RTR = CAN_RTR_DATA;
    header.DLC = size;
    header.TransmitGlobalTime = DISABLE;

    // Pointer vers les variables locales
    CAN_TxHeaderTypeDef *pHeader = &header;
    uint32_t *pTxMailbox = &txMailbox;

    // Attempt to add the CAN message to the transmission mailbox with
retry logic
    do {
        status = HAL_CAN_AddTxMessage(&hcan1, pHeader, aData, pTxMailbox);

        switch (status)
        {
            case HAL_OK:
                if (logs == TRUE)
                {
                    printf("CAN message ");
                    for (int i = 0; i<size; i++)
                        printf(" 0x%X", aData[i]);
                    printf(" sent successfully to  0x%X.\r\n", (unsigned
int)msg_id);
                }
        }
        return; // Exit the function if the message was sent
    }
```

successfully

```

        case HAL_BUSY:
            retryCount++;
            if (logs == TRUE) printf("Warning: CAN bus is busy, retrying
(%d/%d)...\\r\\n", retryCount, maxRetries);
            HAL_Delay(10); // Optional: Add a small delay between retries
            break;

        case HAL_ERROR:
            if (logs == TRUE) printf("Error: Failed to send CAN
message.\\r\\n");
            Error_Handler(); // Optional: Go to error handler for critical
failure
            return;

        case HAL_TIMEOUT:
            if (logs == TRUE) printf("Error: CAN message send timed
out.\\r\\n");
            Error_Handler(); // Optional: Go to error handler for timeout
            return;

        default:
            if (logs == TRUE) printf("Unknown status returned from
HAL_CAN_AddTxMessage.\\r\\n");
            Error_Handler(); // Optional: Handle unexpected status
            return;
    }

} while (status == HAL_BUSY && retryCount < maxRetries);

if (retryCount == maxRetries)
{
    if (logs == TRUE) printf("Error: Exceeded maximum retries for CAN
message send.\\r\\n");
    Error_Handler(); // Optional: Go to error handler after max
retries
}
}

```

La gestion des priorités sur le bus CAN implique qu'il n'est pas garanti qu'un message puisse être envoyé immédiatement. Pour gérer cela, la fonction `HAL_StatusTypeDef` utilise un mécanisme de boîtes aux lettres : le message est placé dans une boîte aux lettres, où il reste en attente d'être transmis dès que le bus devient disponible. Cette fonction retourne le numéro de la boîte aux lettres via l'argument `pTxMailbox`, ce qui permet de vérifier à tout moment l'état de l'envoi du message à l'aide de la fonction `HAL_CAN_IsTxMessagePending`. Il est également possible d'annuler un message en attente avec la fonction `HAL_CAN_AbortTxRequest`.

C'est pour prendre en compte cette caractéristique des communications CAN que notre fonction `CAN_Send` essaie d'envoyer le message plusieurs fois jusqu'à ce que le nombre d'essais maximum soit atteint, ici `maxRetries = 5`, ou bien le message est envoyé avec succès.

5.2. Interfaçage avec le capteur

En reprenant le code des TP précédents, nous avons réussi à faire évoluer le curseur du moteur proportionnellement aux valeurs de température et de pression récupérées par notre capteur associé.

6. TP5 - Intégration I²C - Serial - REST - CAN

Objectif: Faire marcher ensemble les TP 1, 2, 3 et 4

Cahier des charges:

- Mesures de température et de pression sur I²C par le STM32
- Communication série entre la STM32 et le Raspberry PI zero: implémentation du protocole proposé au TP2.4
- API REST sur le Raspberry:

CRUD	Method	Path	Action
Create	POST	temp/	Retrieve new temperature
Create	POST	pres/	Retrieve new pressure
Retreive	GET	temp/	Return all previous temperatures
Retreive	GET	temp/x	Return temperature #x
Retrieve	GET	pres/	Return all previous pressures
Retrieve	GET	pres/x	Return pressure #x
Retrieve	GET	scale/	Return scale (K)
Retrieve	GET	angle/	Return angle (temp x scale)
Update	POST	scale/x	Change scale (K) for x
Delete	DELETE	temp/x	Delete temperature #x
Delete	DELETE	pres/x	Delete pressure #x

Les valeurs seront stockées dans des variables globales du serveur python.

Toutes les appels doivent générer une demande entre le Raspberry et la STM32.

Serveur REST sur Raspberry PI

```
'''
```

```
This Flask application exposes a RESTful API for interfacing with a BMP280 sensor connected to an STM32 microcontroller. It manages:
- Temperature: Reading and storing temperature values.
- Pressure: Reading and storing pressure values.
```

```
'''  
  
import serial  
  
import json  
from flask import Flask  
  
from flask import jsonify  
from flask import render_template  
from flask import abort  
from flask import request  
  
app = Flask(__name__)  
  
#####  
# Core application pages #  
#####  
@app.route('/')  
def hello_world():  
    return jsonify({"message": 'Hello, World!'})  
  
welcome = "Welcome to 3ESE API!"  
  
@app.route('/api/welcome/')  
def api_welcome():  
    if request.method == 'GET' :  
        return jsonify({"phrase" : welcome})  
  
@app.route('/api/welcome/<int:index>', methods=['GET', 'POST', 'DELETE'])  
def api_welcome_index(index):  
    if index < 0 or index >= len(welcome):  
        abort(404)  
  
    if request.method == 'GET' :  
        return jsonify({"index" : index, "value" : welcome[index]})  
  
    elif request.method == 'POST' :  
  
        #return jsonify({"index": index, "val": welcome[index]})  
  
@app.errorhandler(404)  
def page_not_found(error):  
    return render_template('page_not_found.html'), 404  
  
@app.route('/api/request/', methods=['GET', 'POST'])  
@app.route('/api/request/<path>', methods=['GET', 'POST'])  
def api_request(path=None):  
    resp = {  
        "method": request.method,  
        "url" : request.url,  
        "path" : path,  
        "args": request.args,  
        "headers": dict(request.headers),  
        /
```

```
    }
    if request.method == 'POST':
        resp["POST"] = {
            "data" : request.get_json(),
        }
    return jsonify(resp)

#####
# Communication with the STM32 #
#####

SERIAL_BUFFER_SIZE = 10

tab_T = [] # Array for temperatures
tab_P = [] # Array for pressures

# To test with another device
ser = serial.Serial("/dev/ttyACM0", 115200, timeout=1)
#ser = serial.Serial("/dev/ttyAMA0", 115200, timeout=1)
ser.reset_output_buffer()
ser.reset_input_buffer()

def fill_serial_buffer(msg:str):
    if msg.__len__ < SERIAL_BUFFER_SIZE-2:
        return f"msg{' '*msg.__len__ - SERIAL_BUFFER_SIZE-2}"

    elif msg.__len__ > SERIAL_BUFFER_SIZE-2:
        return msg[:SERIAL_BUFFER_SIZE-2]
    else:
        return msg

# Temperature endpoint
@app.route('/api/temp/', methods=['GET', 'POST'])
def api_temp():
    ser.reset_output_buffer()
    ser.reset_input_buffer()
    resp = {
        "method": request.method,
        "url": request.url,
        "args": request.args,
        "headers": dict(request.headers),
    }
    if request.method == 'POST':
        ser.write(b'GET_T  ') # Sends to the STM32 that we want to
perform a GET_T
        tempo = ser.readline().decode() # Retrieve the value sent by the
STM32
        tab_T.append(tempo[:-9]) # Remove '\r\n' and add it to the array
        return jsonify(tab_T[-1]) # Return the last value
    if request.method == 'GET':
        return jsonify(tab_T) # Return the entire temperature array

@app.route('/api/temp/<int:index>', methods=['GET', 'DELETE'])
def api_temp_index(index=None):
    resp = {
```

```
"method": request.method,
"url": request.url,
"index": index,
"args": request.args,
"headers": dict(request.headers),
}
if request.method == 'GET':
    if index < len(tab_T):
        return jsonify(tab_T[index]) # Retrieve the value from the
array at the index
    else:
        return jsonify("error: index out of range")
if request.method == 'DELETE':
    if index < len(tab_T):
        return jsonify(f"The value {tab_T.pop(index)} has been
removed") # Remove value from the array
    else:
        return jsonify("error: index out of range")

# Pressure endpoint
@app.route('/api/pres/', methods=['GET', 'POST'])
def api_pres():
    ser.reset_output_buffer()
    ser.reset_input_buffer()
    resp = {
        "method": request.method,
        "url": request.url,
        "args": request.args,
        "headers": dict(request.headers),
    }
    if request.method == 'POST':
        ser.write(b'GET_P ') # Sends to the STM32 that we want to
perform a GET_P
        tempo = ser.readline().decode() # Retrieve the value sent by the
STM32
        tab_P.append(tempo[:20]) # Remove '\r\n' and add it to the array
        return jsonify(tab_P[-1]) # Return the last value
    if request.method == 'GET':
        return jsonify(tab_P) # Return the entire pressure array

@app.route('/api/pres/<int:index>', methods=['GET', 'DELETE'])
def api_pres_index(index=None):
    resp = {
        "method": request.method,
        "url": request.url,
        "index": index,
        "args": request.args,
        "headers": dict(request.headers),
    }
    if request.method == 'GET':
        if index < len(tab_P):
            return jsonify(tab_P[index]) # Retrieve the value from the
array at the index
        else:
            /
    
```

```
        return jsonify("error: index out of range")
if request.method == 'DELETE':
    if index < len(tab_P):
        return jsonify(f"The value {tab_P.pop(index)} has been
removed") # Remove value from the array
    else:
        return jsonify("error: index out of range")
```

Aller plus loin

Doxxygen

Nous pouvons installer Doxygen et faire une première documentation (<https://doxygen.nl/index.html>)

To launch Doxygen GUI: **doxywizard** must be typed in the terminal.

Nos premières documentations sont accessibles sur le chemin [Documentation/html/index.html](#).

Pandoc

Il est également possible de générer un pdf à partir d'un fichier MarkDown, utile pour un **README .md**, avec l'outil **Pandoc** qui utilise les librairies de **TEX Live**. Pour mettre le tout en place il suffit d'exécuter les commandes :

```
sudo apt install pandoc
sudo apt install texlive
```

Après l'intallation. Une simple commande :

```
pandoc README.md -o README.pdf
```

Suffit pour générer un pdf de notre **README .md**.