

TP Bus et Réseaux - TP 1 à 5

Site: [MOODLE ENSEA](#)

Cours: 2425-S9-ESE_4-Bus et réseaux industriels

Livre: TP Bus et Réseaux - TP 1 à 5

Imprimé par: Oliver BELLIARD ABREU (3A)

Date: vendredi 18 octobre 2024, 08:18

Table des matières

1. Présentation

2. TP 1 - Bus I2C

- 2.1. Capteur BMP280
- 2.2. Setup du STM32
- 2.3. Communication I²C
- 2.4. Interfaçage de l'accéléromètre

3. TP2 - Interfaçage STM32 - Raspberry

- 3.1. Mise en route du Raspberry Pi Zéro
- 3.2. Port Série
- 3.3. Commande depuis Python

4. TP3 - Interface REST

- 4.1. Installation du serveur Python
- 4.2. Première page REST
- 4.3. Nouvelles méthodes HTTP
- 4.4. Et encore plus fort...

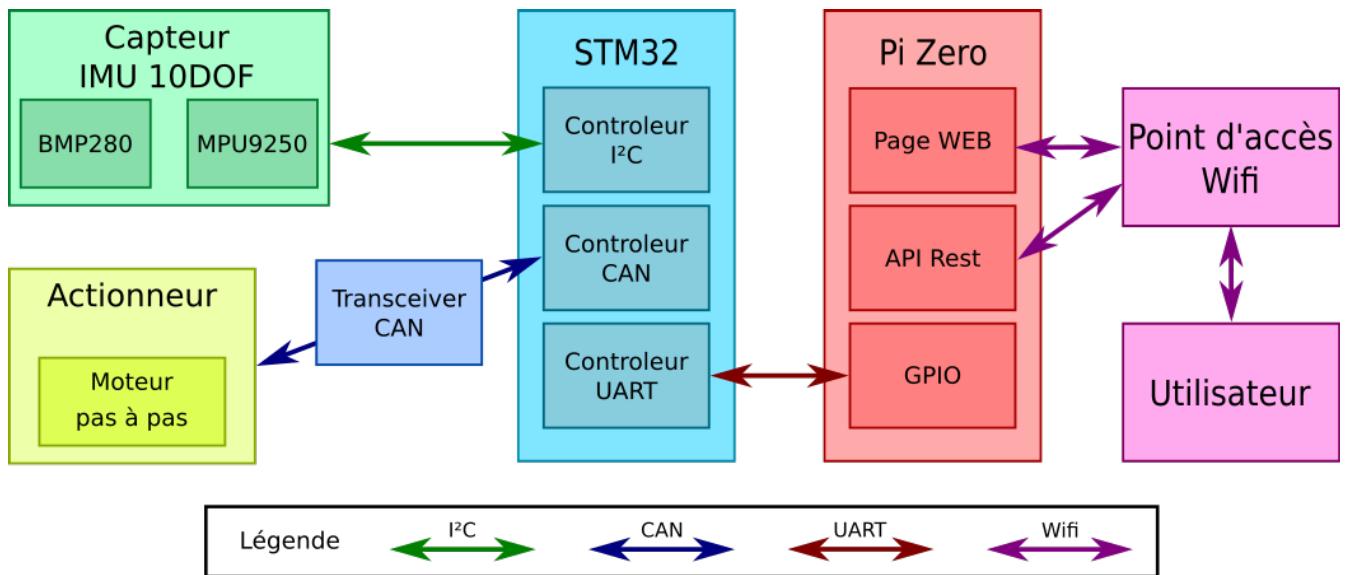
5. TP4 - Bus CAN

- 5.1. Pilotage du moteur
- 5.2. Interfaçage avec le capteur

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

1. Présentation

Le but de cette série de TP est de mettre en place l'ensemble des composants suivant:



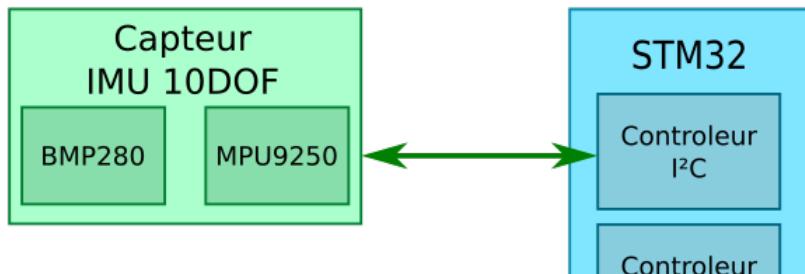
Ces TP seront réalisés en C pour la partie STM32, et Python pour la partie Raspberry Pi.

l'échelonnement des TP sera 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

Objectif: Interfacer un STM32 avec des capteurs I²C



La première étape est de mettre en place la communication entre le microcontrôleur et les capteurs (température, pression, accéléromètre...) via le bus I²C.

Le capteur comporte 2 composants I²C, qui partagent le même bus. Le STM32 jouera le rôle de Master sur le bus.

Le code du STM32 sera écrit en langage C, en utilisant la bibliothèque HAL.

2.1. Capteur BMP280

Mise en œuvre du BMP280

Le BMP280 est un capteur de pression et température développé par Bosch ([page produit](#)).

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

1. les adresses I²C possibles pour ce composant.
2. le registre et la valeur permettant d'identifier ce composant
3. le registre et la valeur permettant de placer le composant en mode normal
4. les registres contenant l'étalonnage du composant
5. les registres contenant la température (ainsi que le format)
6. les registres contenant la pression (ainsi que le format)
7. les fonctions permettant le calcul de la température et de la pression compensées, en format entier 32 bits.

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 */
```

Test de la chaîne de compilation et communication UART sur USB

Testez ce printf avec un programme de type echo.

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)

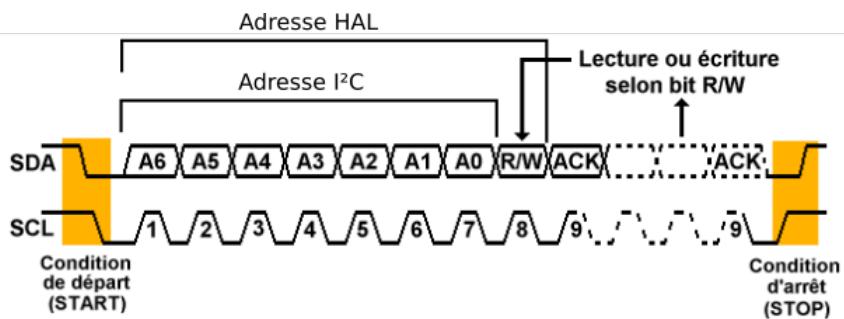
où:

- 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

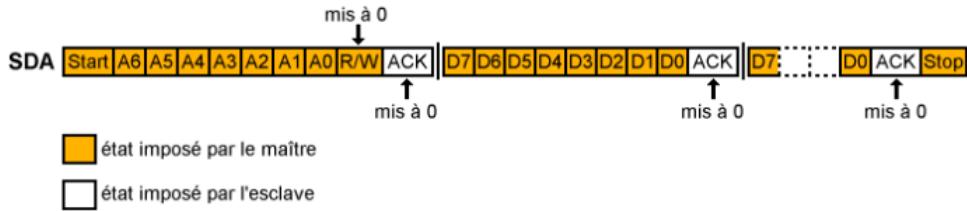
Adresse:

HAL_I2C_Master_Transmit permet d'écrire sur le bus, alors que HAL_I2C_Master_Receive permet de lire le bus. Ces 2 fonctions gère le bit R/W, mais il faut quand même lui laisser la place dans l'adresse I²C.

L'adresse I²C est officiellement sur 7 bits. L'API HAL du STM32 demande des adresses I²C sur 8bits, le LSB étant réservé au bit de R/W. Il faudra donc penser à décaler les adresses d'1 bit sur la gauche et laisser le LSB à 0.



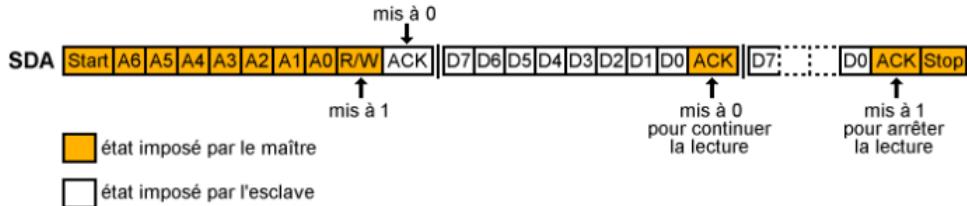
Transmit:



Le maître drive seul le bus. Le premier octet transmis sera l'adresse de l'esclave, le second octet représente l'adresse du 1er registre à écrire. Les autres octets seront les données à transmettre.

L'esclave valide chaque octet en forçant le ACK à zéro (dominant) au bon moment.

Receive:



Le maître drive le bus seulement pour l'adresse de l'esclave (1er octet). C'est ensuite l'esclave qui drive le bus pour permettre au maître de lire les données.

Le maître valide chaque octet en forçant le ACK à zéro (dominant) au bon moment, sauf le dernier (Non ACK ou NACK)

La sélection du registre à lire se fait en envoyant (transmit) auparavant l'adresse du registre à lire.

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.

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

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 les valeurs 32 bit non compensées de la pression et de la température.

Calcul des températures et des pressions 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 compensées de température et de pression sous un format lisible.

2.4. Interfaçage de l'accéléromètre

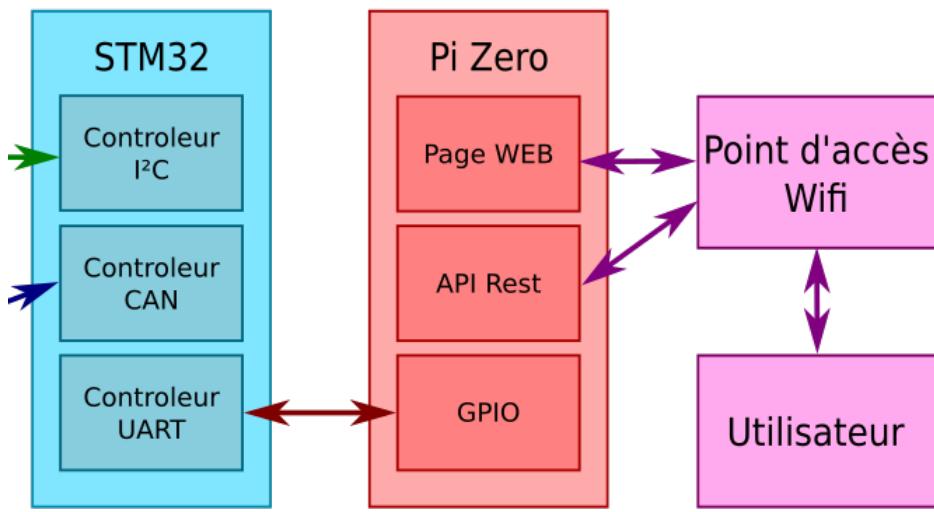
À partir du code développé pour le BMP280, commencez la mise en place de l'interfaçage de l'accéléromètre ADXL345 ou MPU9250.

Datasheets:

- [ADXL345](#)
- [MPU-9250](#)

3. TP2 - Interfaçage STM32 - Raspberry

Objectif: Permettre l'interrogation du STM32 via un Raspberry Pi Zero Wifi



L'ensemble du code développé le sera sur le Raspberry Pi Zéro.

Pensez à bien sauvegarder vos codes, ils seront effacés par le 2e groupe de TP

Vous trouverez en ressource ([ici](#)) un code simple vous permettant d'exploiter le BMP280.

3.1. Mise en route du Raspberry PI Zéro

Préparation du Raspberry

Si le connecteur GPIO n'est pas encore soudé sur votre Raspberry Pi Zero, faites-le.

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, vous pouvez utiliser:

- Rpi_Imager: <https://www.raspberrypi.com/software/>
- BalenaEtcher: <https://www.balena.io/etcher/>

Rpi_imager va vous 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
```

ou

```
SSID : D060-2Ghz  
Password : ilovelinux
```

Configuration de l'image (si vous utilisez BalenaEtcher)

Attention: avec windows, vous ne pouvez pas accéder à rootfs, et donc vous ne pourrez pas changer le hostname de votre image. Dans ce cas, utilisez de préférence Rpi_imager

Remontez la carte SD, vous devez avoir 2 partitions: boot et rootfs.

Ajoutez les fichiers suivants sur la partition boot afin de rendre votre Raspberry accessible par ssh sur le réseau wifi:

- ssh: créez un fichier qui porte ce nom sur la partition de boot (peut importe ce qu'il contient) pour lancer automatiquement le serveur SSH au démarrage.
- wpa_supplicant.conf:

```
ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev  
update_config=1  
country=FR  
  
network={  
    ssid=<Name of your wireless LAN>  
    psk=<Password for your wireless LAN>  
}
```

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

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

Pour dans le fichier cmdline.txt, pour que le noyau libère le port UART, retirez l'option suivante:

```
console=serial0,115200
```

Définition d'un nouvel utilisateur avec son mot

Sur la partition rootfs, modifiez les fichiers /etc/hostname et /etc/hosts afin de modifier le nom de votre Raspberry.

Premier démarrage

Installez la carte SD dans le Raspberry et branchez l'alimentation.

Utilisez ssh pour vous connecter à votre Raspberry. Comment le Raspberry a obtenu son adresse IP?

L'utilisateur par défaut s'appelle "pi", le mot de passe est "raspberry". Ce n'est plus possible d'utiliser ce couple de login/mdp. Utilisez ceux que vous avez définis auparavant.

3.2. Port Série

Loopback

Branchez le port série du Raspberry en boucle: RX sur TX.

Utilisez le logiciel minicom sur le raspberry pour tester le port série.

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

Une fois dans minicom configurer le port série en pressant CTRL+A suivi de O. Pensez à déactiver le contrôle de flux matériel (on utilise pas les lignes RTS/CTS).

Écrire quelques lettres au clavier. Si elles s'affichent, le loopback fonctionne (essayez en le débranchant).

CTRL+A 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 connectées aux borniers CN9 et CN10. (C'est possible en jouant avec le fer à souder: doc [Nucleo 64 page 27](#))

C'est pourquoi vous devez utiliser un **2^e** port UART sur le STM32, qui servira à la communication avec le Raspberry Pi (comme indiqué lors du [TP1](#)). Vous pouvez modifier votre fonction **printf** pour quelle affiche sur les 2 ports série en même temps.

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

Implémentez ce protocole dans le STM32.

Branchez le STM32 sur le Raspberry en prenant soin de croiser les signaux RX et TX. Les 2 fonctionnent en 3,3V donc aucune adaptation de niveau est nécessaire.

Testez ce protocole depuis le Raspberry à l'aide de minicom. Envoyez des ordres manuellement et vérifiez les valeurs renvoyées par le STM32.

3.3. Commande depuis Python

Installez pip pour python3 sur le Raspberry:

```
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: import serial

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

Réalisez un script en Python3 qui permet communiquer avec le STM32. Idéalement, réalisez une fonction par ordre du protocole.

4. TP3 - Interface REST

Objectif: Développement d'une interface REST sur le Raspberry

4.1. Installation du serveur Python

Installation

Créez votre propre utilisateur différent de pi, remplacer XXX par le nom de votre choix, avec les droits de sudo et d'accès au port série (dialout):

```
sudo adduser XXX  
sudo usermod -aG sudo XXX  
sudo usermod -aG dialout XXX
```

Déloguez vous, puis reloguez vous en tant que XXX

Si ce n'est pas déjà fait, installez pip pour python3 sur le Raspberry:

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

Créez un répertoire pour le développement de votre serveur. Dans ce répertoire, créez un fichier nommé `requirement.txt` dans lequel vous placerez le texte suivants:

```
pyserial  
flask
```

Installez ces bibliothèques par la commande: `pip3 install -r requirement.txt`

À nouveau, déloguez vous, puis reloguez vous en tant que XXX pour mettre à jour le PATH et permettre de lancer flask.

Premier fichier Web

Créez un fichier `hello.py` dans lequel vous placerez le code suivant:

```
from flask import Flask  
app = Flask(__name__)  
  
@app.route('/')  
def hello_world():  
    return 'Hello, World!\n'
```

Voilà votre nouveau serveur web! vous pouvez le lancer avec:

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

Vous pouvez tester votre nouveau serveur avec la commande `curl` (dans un 2e terminal):

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

Vous pouvez aussi ajouter les options `-s -D` à cette commande pour visualiser les headers de la réponse HTTP (regardez en particulier le champ `Server`)

Le problème est que votre serveur ne fonctionne pour le moment que sur la loopback. Cela est résolue 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, vous pouvez tester votre serveur web avec un navigateur.

4.2. Première page REST

Première route

Ajoutez 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`?

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

Pour pouvoir prétendre être RESTful, votre 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. Vous pouvez par exemple remplacer la dernière ligne de la fonction `api>Welcome_index` par:

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

(oubliez pas le `import json` en début de fichier!)

Testez le résultat. Est-ce suffisant pour dire que la réponse est bien du JSON? Observez en particulier les entêtes de la réponse: sous Firefox ou Chrome ouvrez les outils de développement (F12), sélectionnez l'onglet "réseau" et rechargez la page. Vous pouvez normalement trouver l'entête de réponse `Content-Type`: ce n'est pas du JSON!

1re solution

Il faut modifier la réponse renvoyée par flask, en ajoutant au contenu du return des entêtes personnalisés 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 du JSON, et Firefox vous 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.

Modifiez votre code pour utiliser `jsonify` et testez le.

Erreur 404

Il arrive souvent que les URL demandées soient fausses, il faut donc que votre serveur renvoie une erreur 404.

Téléchargez le fichiers `page_not_found.html` (en ressource) et placez le dans un nouveau répertoire `templates` (nom de chemin imposé par flask). Le plus simple pour créer ce fichier est de créer un fichier vide, puis de copier-coller son contenu (<shift>+<insert> sous windows). Une autre solution est d'utiliser un utilitaire de copie sur ssh: scp (pscp sous windows, à télécharger sur le site: <https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html>).

Ajoutez les lignes suivantes à votre `hello.py`:

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

Ainsi vous contrôlez la page d'erreur 404.

Modifiez la fonction `api_welcome_index` de manière à retourner cette page 404 si jamais l'index n'est pas correct. Flask fournit une fonction pour cela : `abort(404)`.

Une autre méthode aurait pu être utilisée: `redirect` avec `url_for`. plus d'info: <https://flask.palletsprojects.com/en/1.1.x/quickstart/#redirects-and-errors>

4.3. Nouvelles méthodes HTTP

Méthodes POST, PUT, DELETE...

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

Méthode POST

Pour essayer une autre méthode, utilisez l'utilitaire `curl` sur linux (par exemple directement depuis le raspberry) de cette manière:

```
curl -X POST http://ip.du.pi.0/api/welcome/14
```

ou bien utiliser l'extension **RESTED** sur firefox.

Normalement, votre raspberry doit vous insulter à coup d'erreur 405...

Et oui, il faut ajouter la liste des méthodes acceptées à votre route, par exemple

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

Mais ajouter la méthode ne suffit pas, il faut aussi que votre 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 votre serveur.

Testez à l'aide de **RESTED** ou de `curl` 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)
```

Faites en sorte notamment d'obtenir une réponse qui peuple correctement les champs `args` et `data`

API CRUD

En reprenant la fonction `api_welcome_index`, ajoutez-y les fonctions CRUD 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 doit se faire en JSON, et si une action ne renvoie rien, alors le code status de la réponse doit 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>

4.4. Et encore plus fort...

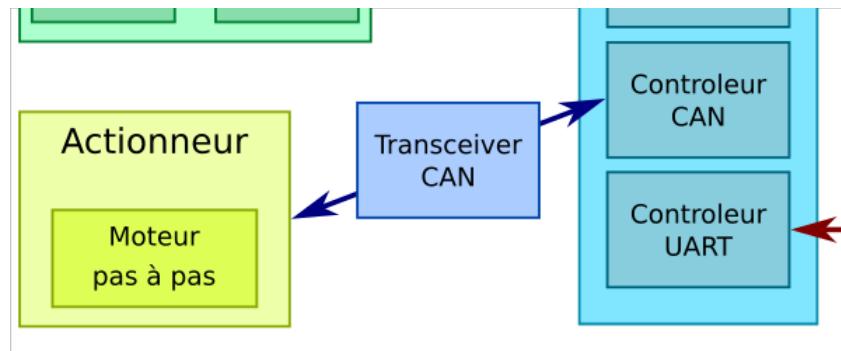
Le code écrit avec Flask pour créer une API REST est rapide, mais finalement il y a encore beaucoup de redondance...

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... Essayez de réécrire votre code avec FASTAPI, et allez voir la page [/docs](#).

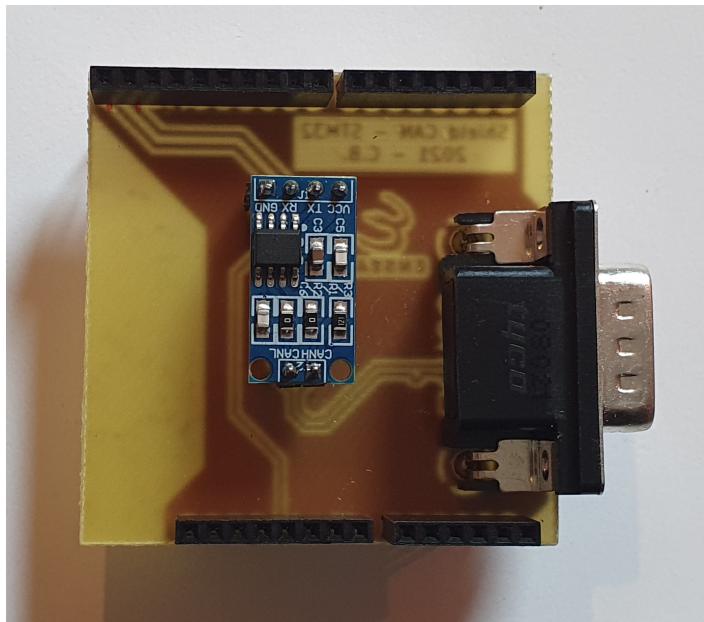
5. TP4 - Bus CAN

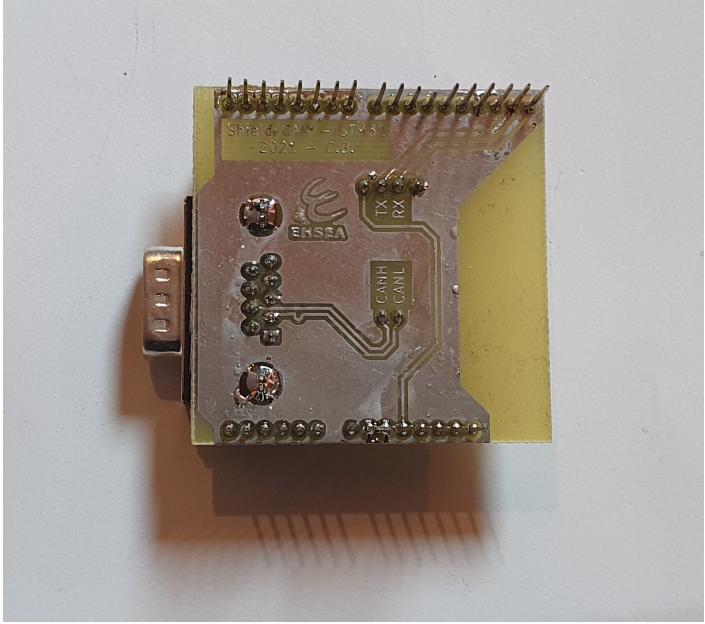
Objectif: Développement d'une API Rest et mise en place d'un périphérique sur bus CAN



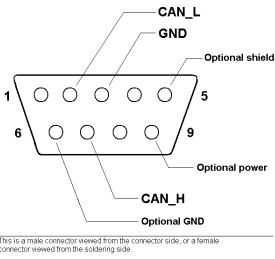
Les cartes STM32L476 sont équipées d'un contrôleur CAN intégré. Pour pouvoir les utiliser, il faut leur adjoindre un Transceiver CAN. Ce rôle est dévolu à un TJA1050 (<https://www.nxp.com/docs/en/data-sheet/TJA1050.pdf>). Ce composant est alimenté en 5V, mais possède des E/S compatibles 3,3V.

Afin de faciliter sa mise en œuvre, ce composant a été installé sur une carte fille (shield) au format Arduino, qui peut donc s'insérer sur les cartes nucléo64:





Ce shield possède un connecteur subd9, qui permet de connecter un câble au format CAN. Pour rappel, le brochage de ce connecteur est le suivant:



Seules les broches 2, 3 et 7 sont utilisées sur les câbles à votre dispositions.

Remarque: Vous pourrez noter 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.

Vous allez utiliser le bus CAN pour piloter un module moteur pas-à-pas. Ce module 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 est un peu capricieuse et ne semble tolérer qu'une vitesse CAN de **500kbit/s**. Pensez à régler CubeMx en conséquence.

Edit 2022: Il semble que ce soit surtout le ratio $\text{seg2}/(\text{seg1}+\text{seg2})$, qui détermine l'instant de décision, qui doit être aux alentours de 87%. Vous pouvez utiliser le calculateur suivant: <http://www.bittiming.can-wiki.info/>

5.1. Pilotage du moteur

Commencez par mettre en place un code simple, qui fait bouger le moteur de 90° dans un sens, puis de 90° dans l'autre, avec une période de 1 seconde.

Vous utiliserez pour cela 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` pointeur vers la structure stockant les informations du contrôleur CAN
- `CAN_TxHeaderTypeDef * pHeader` pointeur vers une structure stockant les informations du header de la trame CAN à envoyer
- `uint8_t aData[]` buffer contenant les données à envoyer
- `uint32_t * pTxMailbox` pointeur vers la boîte au lettre de transmission

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

La variable `pHeader` est une structure contenant les champs suivants, que vous devez remplir 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` 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`

La gestion de priorité du bus CAN fait que l'on ne peut pas être sûr que notre message puisse être envoyé. C'est pourquoi la fonction `HAL_StatusTypeDef` utilise une boîte au lettre: le message est stocké dans la boîte au lettre, et sera envoyé dès que le bus le permettra. Cette fonction renvoie le numéro de boîte au lettre via l'argument `pTxMailbox`, qui permet d'interroger l'état de l'envoi du message à n'importe quel moment dans le code (`HAL_CAN_IsTxMessagePending`) et même d'annuler un message en attente (`HAL_CAN_AbortTxRequest`)

5.2. Interfaçage avec le capteur

En reprenant le code des TP précédents, faites en sorte que le mouvement du moteur soit proportionnel à la valeur du capteur.

Le coefficient de proportionnalité sera stocké dans une variable. Le calcul peut parfaitement se faire à partir de la donnée issue du capteur sous forme binaire (le calcul avec la calibration n'est pas nécessaire à ce stade).

Attention: le shield utilisant les pin PB8 et PB9 pour le CAN, vous allez devoir changer les pins du bus I2C, voir même changer de contrôleur I2C.
Dans ce dernier cas, il faudra changer les noms de vos variables hi2c1 vers hic2cX.

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.