

TP de Bus_Réseaux

Vincent LAKHMECHE

Karim JERJOUR

Ceci est le compte-rendu du tp de bus et réseau de Vincent LAKHMECHE et Karim JERJOUR

TP de Bus_Réseaux

2. TP1- Bus I2C

2.1 Capteur BMP280

2.2. Setup du STM32

Redirection du print

Test de la chaîne de compilation et communication UART sur USB

2.3. Communication I²C

Identification du BMP280 : CheckID

Configuration du BMP280

Contrôle du mode d'alimentation

Contrôle de la mesure de pression

Contrôle de la mesure de température

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

Étalonnage du composant

Récupération de la température et de la pression

Calcul des températures et des pression compensées

3. TP2 - Interfaçage STM32 - Raspberry

3.1. Mise en route du Raspberry PI Zéro

Premier démarrage

3.2 Port série

Loopback

Communication avec la STM32

Activation de l'usart dans la PI ZERO

Test d'envoi des ordres à la STM32

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

5.TP4 : CAN

5.1 Pilotage du moteur

5.2. Interfaçage avec le capteur

5.TP5 : Mise en série de l'ensemble

2. TP1- Bus I2C

2.1 Capteur BMP280

Mise en œuvre du BMP280

Toutes les communications avec l'équipement (le capteur I2C) sont réalisées en lisant et en écrivant dans des registres.

--> Ces registres sont sur une largeur de 8 bits et sont décrits dans les tableaux suivants.

Quelles sont les adresses I²C possibles pour ce composant.	On peut lire en page 28 de la datasheet (link) que l'équipement possède une adresse sur 7 bits qui est '111011x'. Les 6 premiers bits, ceux de poids fort, sont différents de 'x'. Le caractère 'x' sert à indiquer que le bit de poids faible n'est pas fixé. L'utilisateur doit le fixer et le configurer en l'adressant avec une adresse personnalisée. Pour
il y en a deux possibles en fonction de si SDO est connecté à la masse (0x76) ou à Vcc (0x77).	Dans notre cas, l'adresse I2C du BMP280 est de 0x77

Identifier le registre et la valeur permettant d'identifier ce composant	Le registre d'identification (ID) est à l'adresse 0xD0, et sa valeur est 0x58 pour le BMP280.
Le registre d'identification (ID) est à l'adresse 0xD0,	sa valeur est 0x58 pour le BMP280

3.Identifier le registre et la valeur permettant de placer le composant en mode normal	
Le registre de contrôle ctrl_meas (adresse 0xF4) permet de définir le mode de fonctionnement.	Les bits bits mode[1:0] doivent être configurés à 11 pour le mode normal.

4.Identifier les registres contenant l'étalonnage du composant
Les données d'étalonnage sont stockées dans les registres de 0x88 à 0xA1

5.Identifier les registres contenant la température (ainsi que le format)	
Les données de température sont réparties sur trois registres : 0xFA (MSB), 0xFB (LSB), et 0xFC (XLSB).	Le format est une valeur non signée de 20 bits : le MSB contient les bits [19:12], le LSB les bits [11:4], et le XLSB les bits [3:0].

6.Identifier les registres contenant la pression (ainsi que le format)	Les données de pression sont également réparties sur trois registres : 0xF7 (MSB), 0xF8 (LSB), et 0xF9 (XLSB).
Les données de pression sont également réparties sur trois registres : 0xF7 (MSB), 0xF8 (LSB), et 0xF9 (XLSB).	Le format est une valeur non signée de 20 bits, similaire à celui des données de température.

7.Identifier les fonctions permettant le calcul de la température et de la pression compensées, en format entier 32 bits.	La fonction pour la compensation de la température est bmp280_compensate_T_int32(BMP280_S32_t adc_P) et celle pour la pression est bmp280_compensate_P_int64(BMP280_S32_t adc_T). Ces fonctions utilisent des entiers 32 bits pour les calculs de compensation.
--	--

2.2. Setup du STM32

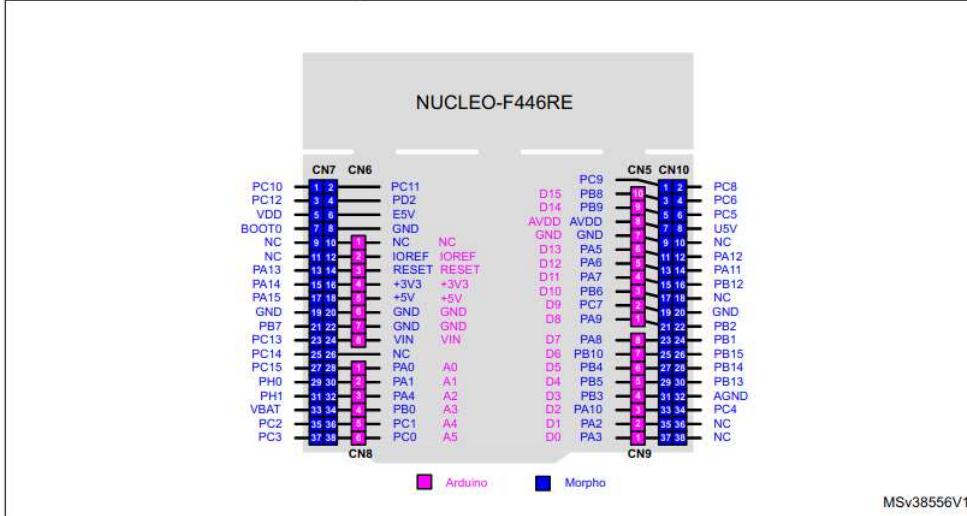
La carte utilisée est une carte de développement STM32 NUCLEO-F446RE

Il nous faut

La figure suivante montrent les signaux connectés par défaut aux connecteurs ARDUINO® Uno V3 (CN5, CN6, CN8, CN9) et au connecteur ST morpho (CN7 et CN10), on retrouve une figure pour chaque carte STM32 Nucleo page 35 de la datasheet :

[user manual : dm00105823-stm32-nucleo64-boards-mb1136-stmicroelectronics.pdf](#)

Figure 25. NUCLEO-F446RE



On doit câbler la carte fille qui contient les composants(capteurs) dont le BMP280 . Il faut faire **attention au câblage** qui a été une source d'erreur et une grande perte de temps.

Le points à contrôler sont :

- les pins GPIO sur lesquels l'i2c est configurée dans l'ioc
- les pins PHYSIQUE sur lesquels on branche SCL cable jaune et SDA cable blanc : cf voir la photo
- les pins physiques sur lesquels on branche VCC (sur un pin 3.3V de la nucleo) c-a-d le câble rouge, et GND sur un pin GND de la nucleo

PHOTO du câblage sur la STM32 pour la connexion logicielle (pins) :

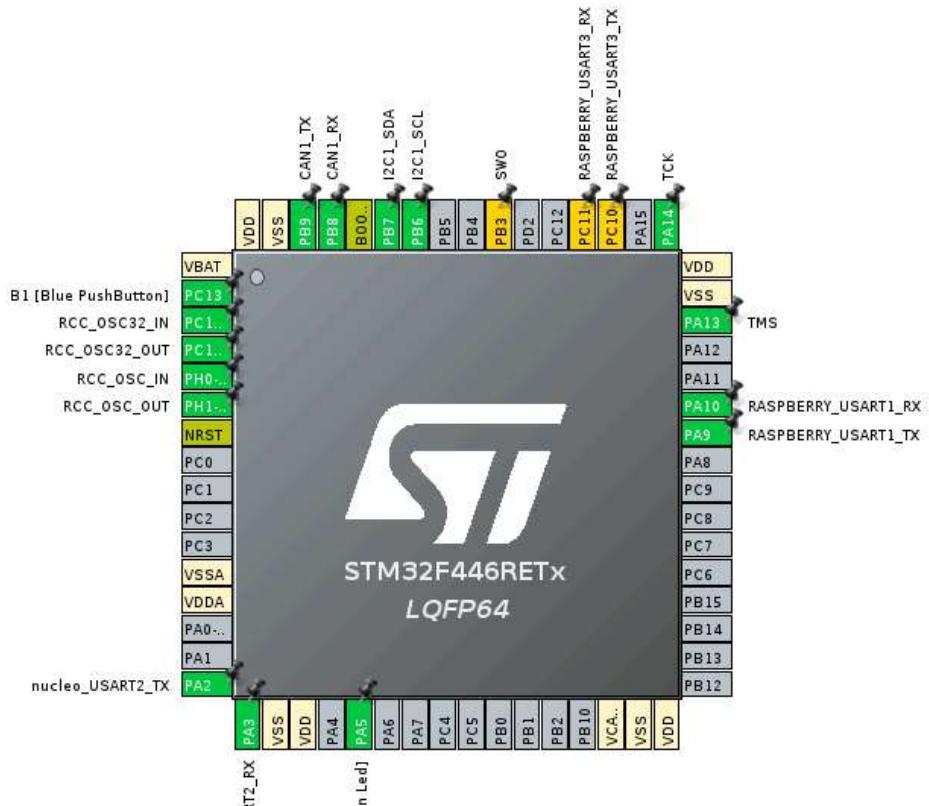
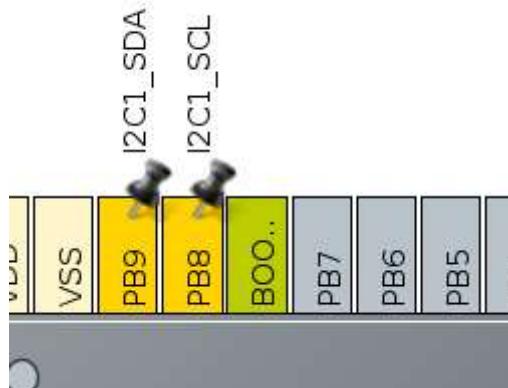


image-20241011090144010-I2C :



I2C1_SDA sur PB9

I2C1_SCL sur PB8

MAJ du tp4 : il faut déplacer l'i2c de ces pins afin de les libérer pour le bus CAN toujours avec l'I2C1

PB7 pour SDA

PB6 pour SCL

UART 2

TX sur PA2 et RX PA3

D'un UART pour la PIO. (TP2)

UART ? = usart 1 PA10 et PA9

PIN

Les pins PB9 et PB8 se trouvent à droite de la carte en haut comme on peut le voir sur la figure de la carte ci-dessus.

Attention le pin PB8 commence au deuxième pins sur les connecteurs morpho.

On rappel que le signal SDA sur PB9 et SCL sur PB8.

Tandis quel 3.3 V pour VCC et le GND sont à gauche de la carte(on peut utiliser d'autres GND situé sur la carte qu'à gauche mais visuellement c'est plus pratique pour vérifier le câblage s'ils sont isolé d'un côté et les pins I2C d'un autre).

Pour la partie CAN, on doit utiliser les pins PB8 ET PB9 d'après le sujet de TP, il faudra donc penser à déplacer les connexions pour le CAN sur d'autres pins si on souhaite utiliser PB8 et PB9

D'un CAN

*****Problèmes cube ide*****

```
sudo apt-get install libncurses5
Lecture des listes de paquets... Fait
Construction de l'arbre des dépendances... Fait
Lecture des informations d'état... Fait
Les paquets supplémentaires suivants seront installés :
  libtinfo5
Les NOUVEAUX paquets suivants seront installés :
  libncurses5 libtinfo5
0 mis à jour, 2 nouvellement installés, 0 à enlever et 6 non mis à jour.
Il est nécessaire de prendre 207 ko dans les archives.
Après cette opération, 883 ko d'espace disque supplémentaires seront utilisés.
Souhaitez-vous continuer ? [O/n] o
Réception de :1 http://fr.archive.ubuntu.com/ubuntu jammy-updates/universe amd64
libtinfo5 amd64 6.3-2ubuntu0.1 [100 kB]
Réception de :2 http://fr.archive.ubuntu.com/ubuntu jammy-updates/universe amd64
libncurses5 amd64 6.3-2ubuntu0.1 [107 kB]
207 ko réceptionnés en 0s (974 ko/s)
Sélection du paquet libtinfo5:amd64 précédemment désélectionné.
(Lecture de la base de données... 283299 fichiers et répertoires déjà installés.)
Préparation du dépaquetage de .../libtinfo5_6.3-2ubuntu0.1_amd64.deb ...
Dépaquetage de libtinfo5:amd64 (6.3-2ubuntu0.1) ...
Sélection du paquet libncurses5:amd64 précédemment désélectionné.
Préparation du dépaquetage de .../libncurses5_6.3-2ubuntu0.1_amd64.deb ...
Dépaquetage de libncurses5:amd64 (6.3-2ubuntu0.1) ...
```

```
Paramétrage de libtinfo5:amd64 (6.3-2ubuntu0.1) ...
Paramétrage de libncurses5:amd64 (6.3-2ubuntu0.1) ...
Traitement des actions différées (« triggers ») pour libc-bin (2.35-
Ubuntu3.8) ...
```

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 */
```

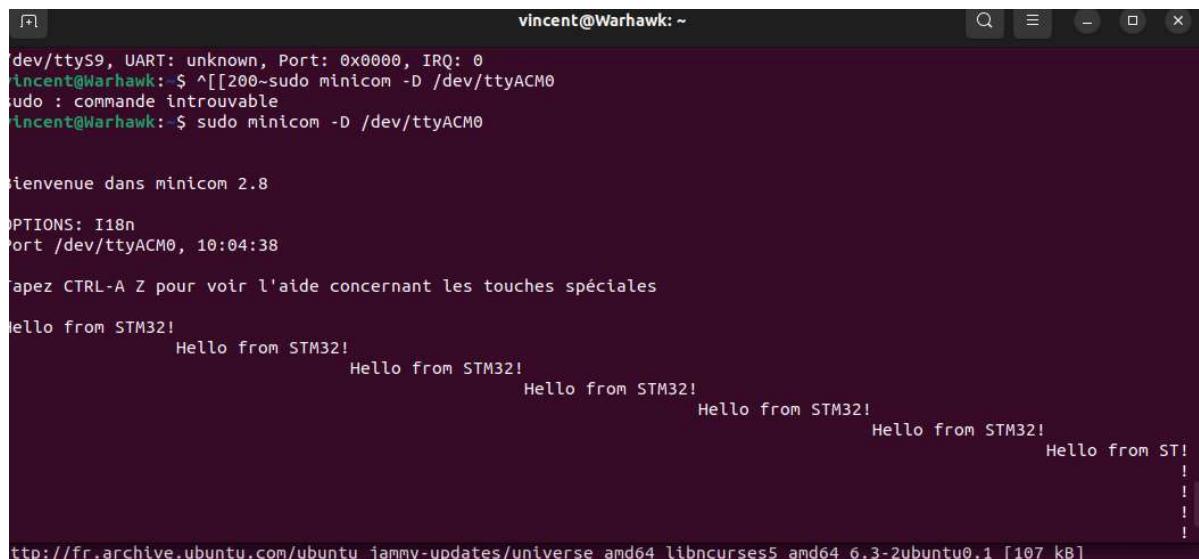
Test de la chaîne de compilation et communication UART sur USB

Si `ttyAMA0` est utilisé pour la console système et que vous souhaitez l'utiliser pour votre propre application, vous devrez le libérer

Utiliser la commande suivante pour lancer `minicom` sur le bon port :

```
sudo minicom -D /dev/ttyACM0 # pour la nucleo  
minicom -b 115200 -o -D /dev/ttyAMA0 #pour la pi zero
```

Une fois la commande lancé, le code écrit dans le main (la ligne du printf) doit s'afficher dans le terminal minicom

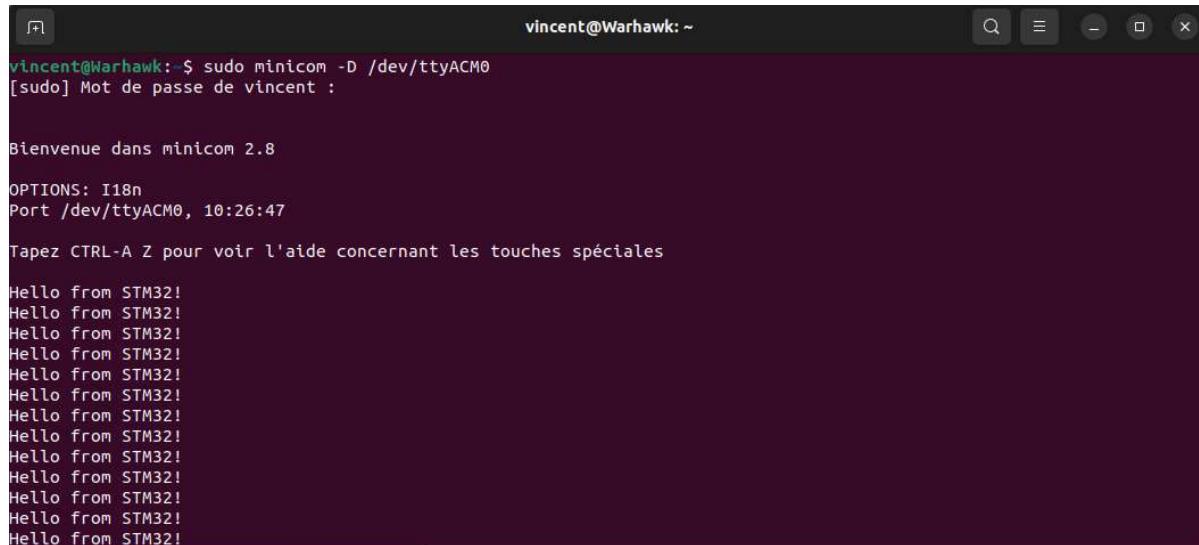


```
vincent@Warhawk: ~  
[dev/ttyS9, UART: unknown, Port: 0x0000, IRQ: 0  
vincent@Warhawk: $ ^[[200~sudo minicom -D /dev/ttyACM0  
sudo : commande introuvable  
vincent@Warhawk: $ sudo minicom -D /dev/ttyACM0  
  
Bienvenue dans minicom 2.8  
  
OPTIONS: I18n  
Port /dev/ttyACM0, 10:04:38  
Tapez CTRL-A Z pour voir l'aide concernant les touches spéciales  
Hello from STM32!  
Hello from STM32!
```

L'affichage est décalé car il manquait le \r pour le retour chariot

```
printf("Hello from STM32!\r\n");
```

Maintenant l'affichage est centré à gauche comme on peut le voir :



```
vincent@Warhawk: ~  
[sudo] Mot de passe de vincent :  
  
Bienvenue dans minicom 2.8  
  
OPTIONS: I18n  
Port /dev/ttyACM0, 10:26:47  
Tapez CTRL-A Z pour voir l'aide concernant les touches spéciales  
  
Hello from STM32!  
Hello from STM32!
```

2.3. Communication I²C

On va ajouter tout le code nécessaire pour manipuler le composant dans des fonctions dont la syntaxe pour rédiger leurs signatures sera BMP280_fonction_a_coder(). Elle seront déclarer dans le fichier header BMP280_vincent.h et implémenter dans le fichier qui sera appelé BMP280_vincent.c.

Identification du BMP280 : CheckID

On doit vérifier que le contenu du registre correspond bien à la datasheet.

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

La première fonction qu'on code est donc la vérification de l'id de l'équipement I2C .

On obtient sa réponse avec la fonction HAL_Receive (). Le buffer passé en argument de cette fonction sera écrasé par les data transmises par le composant I2C. Ce buffer doit être de type uint8_t sinon les octets transmis ne seront pas bien stockés.

Nous avons donc implémenté la fonction checkID().

code de la fonction checkID() :

```
int BMP280_checkID(void) {

    //Fonctionnel au 23 oct 18h59 <<<<< aussi 9 nov 19h34 organisé

    uint8_t buf[10];
    buf[0]=BMP280_ID_REG;

    retour = HAL_I2C_Master_Transmit(&hi2c1, BMP280_ADDR, buf, 1, -1);

    printf("transmission in progress\r\n");

    if(retour != HAL_OK){

        printf("\n problem during transmission \r\n");

        return 1;

    }
    else{
        printf("no problem during transmission | transmission complete ->
reception in progress\r\n");
    }

    retour = HAL_I2C_Master_Receive(&hi2c1, BMP280_ADDR, buf, 1, -1);
    if(retour != HAL_OK){

        printf("\n problem during reception I2C\r\n");

        return 1;

    }
    else{

        printf("\n reception complete ... : Register Id content : 0x%x\r\n",
buf[0]);
    }
}
```

```
}
```

```
Welcome to minicom 2.8
OPTIONS: I18n
Port /dev/serial0, 19:17:09
Press CTRL-A Z for help on special keys
main

reception complete ... : Register Id content : 0x58
main
□
```

On peut lire comme contenu du registre : 0x58. C'est bien la valeur attendue pour le registre 0xD0

Configuration du BMP280

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

La configuration à mettre en place dans le composant est la suivante: mode normal, Pressure oversampling x16, Temperature oversampling x2

Pour cela il faut écrire les bonnes valeurs dans les bons registres conformément à la datasheet.

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 (avec Receive()), la valeur reçue sera la nouvelle valeur du registre

La lecture de la valeur reçue en 2 nous permet de vérifier que la configuration a bien été écrite dans le registre.

Registres concernés pour établir la configuration voulu dans la documentation : mode normal - Pressure oversampling x16 - Temperature oversampling x2

#####

4.2 Memory map

The memory map is given in Table 18 below. Reserved registers are not shown.

Table 18: Memory map

Register Name	Address	bit7	bit6	bit5	bit4	bit3	bit2	bit1	bit0	Reset state
temp_xlsb	0xFC			temp_xlsb<7:4>	0	0	0	0	0	0x00
temp_lsb	0xFB				temp_lsb<7:0>					0x00
temp_msb	0xFA				temp_msb<7:0>					0x80
press_xlsb	0xF9			press_xlsb<7:4>	0	0	0	0	0	0x00
press_lsb	0xF8				press_lsb<7:0>					0x00
press_msb	0xF7				press_msb<7:0>					0x80
config	0xF5	t_sb[2:0]			filter[2:0]			spi3w_en[0]		0x00
ctrl_meas	0xF4	osrs_t[2:0]			osrs_p[2:0]		mode[1:0]			0x00
status	0xF3				measuring[0]		im_update[0]			0x00
reset	0xE0				reset[7:0]					0x00
id	0xD0				chip_id[7:0]					0x58
calib25...calib00	0xA1...0xB8				calibration data					individual

Registers:	Reserved registers	Calibration data	Control registers	Data registers	Status registers	Revision	Reset
Type:	do not write	read only	read / write	read only	read only	read only	write only

4.2.1 Register 0xF3 “status”

Tous les registres à configurer sont des sous parties du registre **ctrl_meas** : les bits 7 à 5 pour l'oversampling de température , 4 à 2 pour l'oversampling de la pression.

il faudra donc modifier les bits de bon poids à l'adresse du registre **ctrl_meas** pour modifier le paramètre voulu (mode power normal, oversampling) mais tous situés à l'adresse 0xF4

Les paramètres binaires à mettre sont aussi décrits page 25 datasheet pour chaque paramètres d'oversampling et de mode voulu il y a un mot binaire associé

4.3.3 Register 0xF3 “status”

The “status” register contains two bits which indicate the status of the device.

Table 19: Register 0xF3 “status”

Register 0xF3 “status”	Name	Description
Bit 3	measuring[0]	Automatically set to ‘1’ whenever a conversion is running and back to ‘0’ when the results have been transferred to the data registers.
Bit 0	im_update[0]	Automatically set to ‘1’ when the NVM data are being copied to image registers and back to ‘0’ when the copying is done. The data are copied at power-on-reset and before every conversion.

4.3.4 Register 0xF4 “ctrl_meas”

The “ctrl_meas” register sets the data acquisition options of the device.

Table 20: Register 0xF4 “ctrl_meas”

Register 0xF4 “ctrl_meas”	Name	Description
Bit 7, 6, 5	osrs_t[2:0]	Controls oversampling of temperature data. See chapter 3.3.2 for details.
Bit 4, 3, 2	osrs_p[2:0]	Controls oversampling of pressure data. See chapter 3.3.1 for details.
Bit 1, 0	mode[1:0]	Controls the power mode of the device. See chapter 3.6 for details.

Contrôle du mode d'alimentation

Il faut placer le composant en mode normal, les configurations possibles et leurs valeurs binaires associées du registre mode[1:0] sont décrites dans le tableau suivant page 15 :

3.6 Power modes

The BMP280 offers three power modes: sleep mode, forced mode and normal mode. These can be selected using the mode[1:0] bits in control register 0xF4.

Table 10: mode settings

mode[1:0]	Mode
00	Sleep mode
01 and 10	Forced mode
11	Normal mode

Adresse registre ctrl_meas	Valeur à écrire
0xF4	Les bits mode[1:0] doivent être configurés à 11 pour le mode normal.

Contrôle de la mesure de pression

On souhaite paramétriser Pressure oversampling à x16

page 12 : Pour activer l'oversampling de la pression (et la mesure) il faut sélectionner les bits [2:0] du registre de control osrs_p à l'adresse 0xF4. Les configurations possibles et valeurs associés du registre osrs_p ne pas sont décrites p.13 dans le tableau ci-dessous.

3.3.1 Pressure measurement

Pressure measurement can be enabled or skipped. Skipping the measurement could be useful if BMP280 is used as temperature sensor. When enabled, several oversampling options exist. Each oversampling step reduces noise and increases the output resolution by one bit, which is stored in the XLSB data register 0xF9. Enabling/disabling the measurement and oversampling settings are selected through the `osrs_p[2:0]` bits in control register 0xF4.

Table 4: `osrs_p` settings

Oversampling setting	Pressure oversampling	Typical pressure resolution	Recommended temperature oversampling
Pressure measurement skipped	Skipped (output set to 0x80000)	–	As needed
Ultra low power	×1	16 bit / 2.62 Pa	×1
Low power	×2	17 bit / 1.31 Pa	×1
Standard resolution	×4	18 bit / 0.66 Pa	×1
High resolution	×8	19 bit / 0.33 Pa	×1
Ultra high resolution	×16	20 bit / 0.16 Pa	×2

In order to find a suitable setting for `osrs_p`, please consult chapter 3.4.

Mais dans le tableau 21 ci-dessous les exemples de configuration avec l'oversampling sont détaillés page25 :

4.3.4 Register 0xF4 “ctrl_meas”

The “`ctrl_meas`” register sets the data acquisition options of the device.

Table 20: Register 0xF4 “`ctrl_meas`”

Register 0xF4 “ <code>ctrl_meas</code> ”	Name	Description
Bit 7, 6, 5	<code>osrs_t[2:0]</code>	Controls oversampling of temperature data. See chapter 3.3.2 for details.
Bit 4, 3, 2	<code>osrs_p[2:0]</code>	Controls oversampling of pressure data. See chapter 3.3.1 for details.
Bit 1, 0	<code>mode[1:0]</code>	Controls the power mode of the device. See chapter 3.6 for details.

Table 21: register settings `osrs_p`

<code>osrs_p[2:0]</code>	Pressure oversampling
000	Skipped (output set to 0x80000)
001	oversampling ×1
010	oversampling ×2
011	oversampling ×4
100	oversampling ×8
101, Others	oversampling ×16

On va choisir le mot binaire ‘101’ qui correspond à oversampling ×16 ce que l'on souhaite comme configuration.

La valeur ‘101’ à écrire dans le code est notée 0b101.

Contrôle de la mesure de température

On souhaite paramétrer Temperature oversampling à x2

Pour activer l'oversampling de la température (et la mesure), il faut sélectionner les bits [2:0] du registre de control osrs_t toujours à l'adresse 0xF4

Configurations possibles et valeurs possible du registre osrs_t sont tout de suite décrites dans le tableau 5 suivant page 13 :

3.3.2 Temperature measurement

Temperature measurement can be enabled or skipped. Skipping the measurement could be useful to measure pressure extremely rapidly. When enabled, several oversampling options exist. Each oversampling step reduces noise and increases the output resolution by one bit, which is stored in the XLSB data register 0xFC. Enabling/disabling the temperature measurement and oversampling setting are selected through the osrs_t[2:0] bits in control register 0xF4.

Table 5: osrs_t settings

osrs_t[2:0]	Temperature oversampling	Typical temperature resolution
000	Skipped (output set to 0x80000)	-
001	×1	16 bit / 0.0050 °C
010	×2	17 bit / 0.0025 °C
011	×4	18 bit / 0.0012 °C
100	×8	19 bit / 0.0006 °C
101, 110, 111	×16	20 bit / 0.0003 °C

It is recommended to base the value of osrs_t on the selected value of osrs_p as per Table 4. Temperature oversampling above ×2 is possible, but will not significantly improve the accuracy of the pressure output any further. The reason for this is that the noise of the compensated pressure value depends more on the raw pressure than on the raw temperature noise. Following the recommended setting will result in an optimal noise-to-power ratio.

Ou dans le tabelau 22 page 25 :



Table 22: register settings osrs_t

osrs_t[2:0]	Temperature oversampling
000	Skipped (output set to 0x80000)
001	oversampling ×1
010	oversampling ×2
011	oversampling ×4
100	oversampling ×8
101, 110, 111	oversampling ×16

4.3.5 Register 0xF5 “config”

The “config” register sets the rate, filter and interface options of the device. Writes to the “config” register in normal mode may be ignored. In sleep mode writes are not ignored.

Table 23: Register 0xF5 “config”

On va choisir le mot binaire ‘010’ qui correspond à oversampling x2, ce que l’on souhaite comme configuration.

Il faut écrire dans les bits 7 à 5 du registre le mot binaire ‘010’ qu’on notera dans le code C “0b010”.

Fin explications de la datasheet

On va ajouter tout le code nécessaire pour configurer le composant dans une fonction BMP280_config() qui sera dans le fichier driver.c qui sera accompagné de son fichier driver.h

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, lorsque l’on essaie de lire le bus, sera la nouvelle valeur du registre
-

Code bpm

```
uint8_t config = (0b010<<5) | (0b101<<2) | (0b11);
```

(0b010<<5) pour remplir les bits [7:5] soit les bit 7 à 5 soit **osrs_t [2:0]**

0b010 << 5 :

- **0b010** est 2 en décimal.
- Décalage de 5 bits vers la gauche : **0b01000000**, soit **64** en décimal

(0b101<<2) pour remplir les bits [4:2] (bits 4 à 2) soit **osrs_p [2:0]**

0b101 << 2 :

- **0b101** est 5 en décimal.
- Décalage de 2 bits vers la gauche : **0b00010100**, soit **20** en décimal.

(0b11) pour remplir les bits [1:0] soit **mode [1:0]**

0b11 :

- **0b11** est 3 en décimal et n'est pas décalé.

```
Port /dev/serial0, 19:17:09
vkuser@vkpi: ~
Press CTRL-A Z for help on special keys

main

reception complete ... : Register Id content : 0x58
main

reception complete ... : Register Id content : 0x58
main

config : no problem during transmission ... transmission complete -> reception in progress
config : well configured - configuration value = hexadecimal : 0x57 | decimal : 0d87
```

La valeur de **config** est donc **0x87** en hexadécimal, soit **0d57** en décimal, et **0b01010111** en binaire.

On lit bien sur l'USART la valeur renvoyé par le capteur suite à la configuration, affiché en hexadécimal grâce à %x :

Printf - paramètres :

utiliser le spécificateur de format %d pour afficher un entier

%u est utilisé pour afficher le contenu de la variable en décimal (entiers non signé)

%x sert à afficher le contenu de la variable en hexadécimal

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

Étalonnage du composant

Pour récupérer les valeurs d'étalonnage, on envoie l'adresse la plus basse du registre des valeurs d'étalonnage et on réceptionne les 24 valeurs d'étalonnage.

3.11.2 Trimming parameter readout

The trimming parameters are programmed into the devices' non-volatile memory (NVM) during production and cannot be altered by the customer. Each compensation word is a 16-bit signed or unsigned integer value stored in two's complement. As the memory is organized into 8-bit words, two words must always be combined in order to represent the compensation word. The 8-bit registers are named calib00...calib25 and are stored at memory addresses 0x88...0xA1. The corresponding compensation words are named dig_T# for temperature compensation related values and dig_P# for pressure compensation related values. The mapping is shown in Table 17.

Table 17: Compensation parameter storage, naming and data type

Register Address LSB / MSB	Register content	Data type
0x88 / 0x89	dig_T1	unsigned short
0x8A / 0x8B	dig_T2	signed short
0x8C / 0x8D	dig_T3	signed short
0x8E / 0x8F	dig_P1	unsigned short
0x90 / 0x91	dig_P2	signed short
0x92 / 0x93	dig_P3	signed short
0x94 / 0x95	dig_P4	signed short
0x96 / 0x97	dig_P5	signed short
0x98 / 0x99	dig_P6	signed short
0x9A / 0x9B	dig_P7	signed short
0x9C / 0x9D	dig_P8	signed short
0x9E / 0x9F	dig_P9	signed short
0xA0 / 0xA1	reserved	reserved

3.11.3 Compensation formula

page 21

Adresse registres étalonnage	Valeur à écrire
0x88[136 en décimal] à 0xA1 [160 en décimal]	Les bits mode[7:0] doivent être configurés à 11 pour le mode normal.

Code pour l'étalonnage

On va faire une demande de lecture qui commence avec transmit() puis read().

La lecture se déroule avec l'envoie de l'adresse du registre 0x88 (qui correspond au registre calib00 de calibration (étalonnage) la plus basse valeur des adresses des registres d'étallonnage) et on reçoit 1 octet correspondant au contenu du registre.

On ajoute la constante dans le fichier BMP280_vincent.h suivante :

```
typedef uint32_t BMP280_U32_t;
typedef int32_t BMP280_S32_t;
typedef int64_t BMP280_S64_t;

static const uint8_t BMP280_ADDR = 0x77 << 1; // Use 8-bit address
// static const uint8_t BMP280_ADDR = 0x76 << 1; // Use 8-bit address

static const uint8_t BMP280_ID_REG = 0xD0;//id du registre selon la doc

static const uint8_t BMP280_TRIM_REG_MSB = 0x88;
```

La fonction BMP280_calib() contient le code suivant pour avoir les registres d'étallonnage :

```
void BMP280_calib(void) {
    //*****à supprimer

    // uint8_t etalo = 0x88; //Adresse du premier registre contenant les valeurs
    //d'étalonnage
    //buf_adr[0]=etalo; //buffer contenant l'adresse du registre d'étalonnage

    //*****
    uint8_t buf_data[24]; //buffer pour les données reçues des registre

    //on demande la lecture avec le transmit

    //BMP280_TRIM_REG_MSB est déclarée dans BMP280_vincent.h

    // L'adresse de la variable BMP280_TRIM_REG_MSB est passée à la fonction qui
    //attends une adresse en paramètre (celle sur laquelle pointerait le pointeur
    //pData)
    retour = HAL_I2C_Master_Transmit(&hi2c1, BMP280_ADDR, &BMP280_TRIM_REG_MSB, 1,
    HAL_MAX_DELAY);

    if (retour != HAL_OK) {
        printf("Probleme calib I2C\r\n");
        return;
    }
    //on lit la réponse qu'on stocke dans le buffer buf_data
    retour = HAL_I2C_Master_Receive(&hi2c1, BMP280_ADDR, buf_data,
    24,HAL_MAX_DELAY);

    if (retour != HAL_OK) {
        printf("Problem calib I2C\r\n");
    }

    else { //on affiche les données d'étalonnage reçues
```

```

    }

    dig_T1 = buf_data[0] | (buf_data[1] << 8);
    dig_T2 = buf_data[2] | (buf_data[3] << 8);
    dig_T3 = buf_data[4] | (buf_data[5] << 8);
    dig_P1 = buf_data[6] | (buf_data[7] << 8);
    dig_P2 = buf_data[8] | (buf_data[9] << 8);
    dig_P3 = buf_data[10] | (buf_data[11] << 8);
    dig_P4 = buf_data[12] | (buf_data[13] << 8);
    dig_P5 = buf_data[14] | (buf_data[15] << 8);
    dig_P6 = buf_data[16] | (buf_data[17] << 8);
    dig_P7 = buf_data[18] | (buf_data[19] << 8);
    dig_P8 = buf_data[20] | (buf_data[21] << 8);
    dig_P9 = buf_data[22] | (buf_data[23] << 8);

    for (int i = 0; i < 24; i++) {
        printf("\ni :%d\r\n", i);
        printf("dans boucle for //registre d'etallage calib= %d \r\n ", buf_data[i]);
    }
}

```

```

#include "stdio.h"
#include "stdlib.h"

#include "main.h"
#include "BMP280_vincent.h"

extern I2C_HandleTypeDef hi2c1;

uint16_t dig_T1;
int16_t dig_T2;
int16_t dig_T3;
uint16_t dig_P1;
int16_t dig_P2;
int16_t dig_P3;
int16_t dig_P4;
int16_t dig_P5;
int16_t dig_P6;
int16_t dig_P7;
int16_t dig_P8;
int16_t dig_P9;

//envoyer une trame avec l'adresse du registre à l'aide de la fonction
HAL_I2C_Master_Transmit().
//buf[0]=BMP_ID_REG;
//HAL_I2C_Master_Transmit(&hi2c1, BMP280_ADDR, buf, 1, -1);

//HAL_I2C_Master_Receive(&hi2c1, BMP280_ADDR, buf, 1, -1);

```

```

//printf("Idreg: 0x%x\r\n", buf[0]);
```

```

int BMP280_checkID(void){

    uint8_t buffer[1];
    buf[0] = BMP280_ID_REG;
    retour = HAL_I2C_Master_Transmit(&hi2c1, BMP280_ADDR, &ID_address, 1,
HAL_MAX_DELAY);

    if(retour != HAL_OK){

        printf("Problème avec l'I2C\r\n");

        return 1;

    }

    retour = HAL_I2C_Master_Receive(&hi2c1, BMP280_ADDR, buffer, 1,
HAL_MAX_DELAY);

    if(retour != HAL_OK){

        printf("Problème avec l'I2C\r\n");

        return 1;

    }

    printf("L'ID du capteur est 0x%x\r\n",buffer[0]);

    return 0;
}
```

```

int BMP280_config(void){

    //uint8_t buffer[1];

    uint8_t buf[2];

    buf[0]= ctrl_meas;

    buf[1]= config;//mot binaire à écrire pour conf correctement

    retour = HAL_I2C_Master_Transmit(&hi2c1, BMP280_ADDR, buf, 2, HAL_MAX_DELAY);

    if(retour != HAL_OK){

        printf("Problème avec l'envoie I2C\r\n");

        return 1;

    } <

    retour = HAL_I2C_Master_Receive(&hi2c1, BMP280_ADDR, buf, 1, HAL_MAX_DELAY);
}

```

```

if(retour != HAL_OK){

    printf("Problème avec la lecture I2C\r\n");

    return 1;

}

if(buffer[0] == config){

    printf("La config reçue est bien reçue et elle vaut %d\r\n", buffer[0]);

    return 0;

}

return 0;

}

void BMP280_calib(void){
//*****à supprimer

// uint8_t etalo = 0x88; //Adresse du premier registre contenant les valeurs
d'étalonnage
//buf_adr[0]=etalo; //buffer contenant l'adresse du registre d'étalement

//*****
buf_data[24];//buffer pour les données reçues des registre

//on demande la lecture avec le transmit

//BMP280_TRIM_REG_MSB est déclarée dans BMP280_vincent.h

// L'adresse de la variable BMP280_TRIM_REG_MSB est passée à la fonction qui
atteint une adresse en paramètre (celle sur laquelle pointerait le pointeur
pData)
    retour=HAL_I2C_Master_Transmit(&hi2c1,BMP280_address,&BMP280_TRIM_REG_MSB,1,
HAL_MAX_DELAY);

    if(retour != HAL_OK){
        printf("Problème avec le transmit() de l'I2C\r\n");
    }
//on lit la réponse qu'on stocke dans le buffer buf_data
    retour = HAL_I2C_Master_Receive(&hi2c1,BMP280_address, buf_data, 24,
HAL_MAX_DELAY);

    if(retour != HAL_OK){

        printf("Problème avec la lecture des données d'étalement I2C \r\n");
    }
}

```

```

    •     return 1;
}
else{//on affiche les données d'étallonnage reçues

    for(int i=0; i<25; i++){
        printf("registre d'étallonnage calib [%d] \r\n ",buf_data[i]);

    }
dig_T1 = buf_data[0] | (buf_data[1]<<8);
dig_T2 = buf_data[2] | (buf_data[3]<<8);
dig_T3 = buf_data[4] | (buf_data[5]<<8);
dig_P1 = buf_data[6] | (buf_data[7]<<8);
dig_P2 = buf_data[8] | (buf_data[9]<<8);
dig_P3 = buf_data[10] | (buf_data[11]<<8);
dig_P4 = buf_data[12] | (buf_data[13]<<8);
dig_P5 = buf_data[14] | (buf_data[15]<<8);
dig_P6 = buf_data[16] | (buf_data[17]<<8);
dig_P7 = buf_data[18] | (buf_data[19]<<8);
dig_P8 = buf_data[20] | (buf_data[21]<<8);
dig_P9 = buf_data[22] | (buf_data[23]<<8);
}

BMP280_get_temperature(){

}

```

il faut décaler l'octet écrit qui est de poids fort (c-a-d le deuxième octet reçu dans chaque couple d'octet par exemple pour buf_data[1] et buf_data[0] c'est buf_data[1] qui est l'octet de poids fort) dans chaque registre de calibration pour les stockés dans les variables t1,t2,t3,p1,...

```
dig_T1 = buf_data[0] | (buf_data[1]<<8<<8);
```

On met sur 16 bit, soit 2 octets, l'octet contenu dans buf_data[1] alors qu'il était sur 1 octet auparavant.

```
(buf_data[1]<<8);
```

Cela nous permettra de faire un 'OU' logique avec l'autre octet non décalé (buf_data[0]). L'octet de buf_data[0] sera convertit sur 16 bits en plaçant un octet de '0' en octet de poids fort (MSB) devant l'octet de donnée contenu dans buf_data[0]

exemple :

0x0034 est égal à 0x34 sauf que l'on précise dans la première forme qu'il y a un octet de 0 à écrire avant d'écrire les bits en base 2 de 0x34.

Valeur de buf_data[0] (LSB) :

- 00000000 00110100 (0x0034)

Et vu que tout l'octet de buf_data[1] a été décalé vers le MSB. Il restera l'octet de poids fort une fois le 'OU' effectué .

Ainsi tout le contenu écrit dans buf_data [1] soit un octet entier sera simplement recopié lors de l'opération logique .

On fait pareil pour chaque registre, on stocke dans un variable une fois le décalage fait.

```
dig_T1 = buf_data[0] | (buf_data[1]<<8);
dig_T2 = buf_data[2] | (buf_data[3]<<8);
dig_T3 = buf_data[4] | (buf_data[5]<<8);
dig_P1 = buf_data[6] | (buf_data[7]<<8);
dig_P2 = buf_data[8] | (buf_data[9]<<8);
dig_P3 = buf_data[10] | (buf_data[11]<<8);
dig_P4 = buf_data[12] | (buf_data[13]<<8);
dig_P5 = buf_data[14] | (buf_data[15]<<8);
dig_P6 = buf_data[16] | (buf_data[17]<<8);
dig_P7 = buf_data[18] | (buf_data[19]<<8);
dig_P8 = buf_data[20] | (buf_data[21]<<8);
dig_P9 = buf_data[22] | (buf_data[23]<<8);
```

Récupération de la température et de la pression

Récupération de la température

On va coder une fonction qui va renvoyer le type BMP280_S32_t. Pour cela, on va définir ce type dans le .h

```
typedef int32_t BMP280_S32_t;
```

Ainsi que l'adresse du registre contenant la température et pression

```
static const uint8_t BMP280_TEMP_REG_MSB = 0xFA; //Adresse du registre contenant
la température
static const uint8_t BMP280_PRES_REG_MSB = 0xF7; //Adresse du registre contenant
la pression
```

Il faut recevoir la température dans un buffer de type `uint8_t` et `int` sinon on obtiens des valeurs températures incorrectes

```
BMP280_S32_t BMP280_get_temperature() {
```

```

BMP280_S32_t adc_T;

//buffer pour les données reçues de la part des registres , ici 3 pour la
température

    //uint8_t ! pas int!. Si int buf_data[3] : valeur mal stockée = valeur
température incorrecte
    uint8_t buf_data[3];

//on demande la lecture avec le transmit

//BMP280_TEMP_REG_MSB est déclarée dans BMP280_vincent.h

// l'adresse de la variable BMP280_TEMP_REG_MSB est passée à la fonction qui
attend une adresse en paramètre (celle sur laquelle pointerait le pointeur pData)
retour = HAL_I2C_Master_Transmit(&hi2c1, BMP280_ADDR, &BMP280_TEMP_REG_MSB,
1,
HAL_MAX_DELAY);

if (retour != HAL_OK) {
    //^printf("Probleme I2C\r\n");
}

//on lit la réponse qu'on stocke dans le buffer buf_data
retour = HAL_I2C_Master_Receive(&hi2c1, BMP280_ADDR, buf_data,
3,HAL_MAX_DELAY);

if (retour != HAL_OK) {

    //^printf("problem i2c");

    return 1;
}

adc_T = ((BMP280_S32_t) (buf_data[0]) << 12) | ( (BMP280_S32_t) (buf_data[1])
<< 4) | ( (BMP280_S32_t) (buf_data[2]) >> 4);

//printf("%05lX", adc_T);

//^printf("%d \r\n", adc_T);

return adc_T;
}

```

```
BMP280_S32_t BMP280_get_pressure() {
```

```

BMP280_S32_t adc_P;

//uint8_t ! pas int!. Si int buf_data[3] : valeur mal stockée = valeur
pression incorrecte
uint32_t buf_data[3];//buffer pour les données reçues de la part des
registres , ici 3 pour la pression

//on demande la lecture avec le transmit

//BMP280_PRES_REG_MSB est déclarée dans BMP280_vincent.h

// l'adresse de la variable BMP280_PRES_REG_MSB est passée à la fonction qui
attend une adresse en paramètre (celle sur laquelle pointerait le pointeur pData)
retour = HAL_I2C_Master_Transmit(&hi2c1, BMP280_ADDR, &BMP280_PRES_REG_MSB,
1,HAL_MAX_DELAY);

if (retour != HAL_OK) {
    //printf("problem i2c\r\n");
}

//on lit la réponse qu'on stocke dans le buffer buf_data
retour = HAL_I2C_Master_Receive(&hi2c1, BMP280_ADDR, buf_data,
3,HAL_MAX_DELAY);

if (retour != HAL_OK) {

    //printf("problem i2c \r\n");

    return 1;
}

adc_P = ((BMP280_S32_t) (buf_data[0]) << 12) | ((BMP280_S32_t) (buf_data[1])
<< 4) | ((BMP280_S32_t) (buf_data[2]) >> 4);

//printf("Pressure adc_P:      0d ");

//printf("%05lx", adc_P);

//printf("%d\r\n", adc_P);

return adc_P;
}

```

Calcul des températures et des pression compensées

On utilise le code mis à disposition par bosch dans la datasheet que l'on ajoute dans le fichier BMP280_vincent.c

Juste avant dans le fichier .h on déclare les types nécessaire au retour de ces fonctions et on déclare les fonctions :

```
BMP280_S32_t bmp280_compensate_T_int32(BMP280_S32_t adc_T)
```

Et on déclare ces fonctions :

```
//déclarations fonctions de compensation de bosch
BMP280_S32_t bmp280_compensate_T_int32(BMP280_S32_t );
BMP280_U32_t bmp280_compensate_P_int64(BMP280_S32_t );
```

Il faut modifier dans le corps des fonctions les signes tiret longs "-" (ou dash) par des tirets court classique pour que le compilateur reconnaisse l'opération de soustraction "-".

Lors de l'appel de ces fonctions il faudra passer les variables adc_T et adc_P en argument pour recevoir les valeurs compensées retournées par la fonction.

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.

On appelle les fonctions dans la boucle while dans le fichier main.c

```
/* Initialize all configured peripherals */
MX_GPIO_Init();
MX_USART2_UART_Init();
MX_I2C1_Init();
MX_USART1_UART_Init();
MX_CAN1_Init();
/* USER CODE BEGIN 2 */

HAL_UART_Receive_IT(&huart1, RxBuff,RX_BUFF_SIZE);

BMP280_S32_t temp_uncompen;
BMP280_S32_t pres_uncompen;

//déclaration des variables contenant la température non compensée
BMP280_U32_t temp_comp;
BMP280_U32_t pres_comp;

BMP280_checkID();
BMP280_config();
BMP280_calib(); //récupère données étalonnage
```

```

printf("main\r\n");

//***** Connexion au moteur Avec fonctions du driver
*****



//enable_can();

//*****FOnctionnel avec
Mr.B*****



//TP4 partie CAN d
//début *****



/*
//Activer le module can
HAL_StatusTypeDef start_can= HAL_CAN_Start(&hcan1);
if(start_can!= HAL_OK) {

    // Gérer l'erreur d'envoi
    printf("erreur démarrage CAN\r\n");

}

*/



/*
//déclaration du pHeader
CAN_TxHeaderTypeDef pHeader;
uint8_t aData[3]; // Tableau pour les données à transmettre

// Configuration du champ pHeader
pHeader.StdId = 0x61;           // Identifiant standard pour la commande
"Angle"
pHeader.IDE = CAN_ID_STD;      // (0x00000000U) !< Standard Id
pHeader.RTR = CAN_RTR_DATA;    // (0x00000000U) !< Data frame
pHeader.DLC = 2;
pHeader.TransmitGlobalTime = DISABLE;

aData[0] = 90; // D0 : 90° en hexadécimal (0x5A)
aData[1] = 0x00; // D1 : Angle positif

uint32_t pTxMailbox; // variable pour stocker l'indice de la boîte aux
lettres CAN

// Envoi du message CAN avec l'angle de 90°
HAL_StatusTypeDef status = HAL_CAN_AddTxMessage(&hcan1, &pHeader, aData,
&pTxMailbox);

```

```

// vérification du statut d'envoi
if (status != HAL_OK) {
    // Gérer l'erreur d'envoi
    printf("erreur CAN\r\n");
}
else{
    printf("premiere conf reussi\r\n");
}
//fin partie CAN ****
*****



//tx_can(pHeader, aData, pTxMailbox);

/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    //printf("\n on rentre dans le while\r\n");

    //capter la température
    temp_uncompen= BMP280_get_temperature(); //récupérer la température
    temp_comp=bmp280_compensate_T_int32(temp_uncompen); //récupérer la
température

    //printf("T=%ld%ld.%ld%ld_C\r\n", (temp_comp/1000)%10, (temp_comp/100)%10,
(temp_comp/10)%10, temp_comp%10);

    //Les Affichages

    //Température

    printf("valeur non compensée de la température %u \r\n",temp_uncompen);

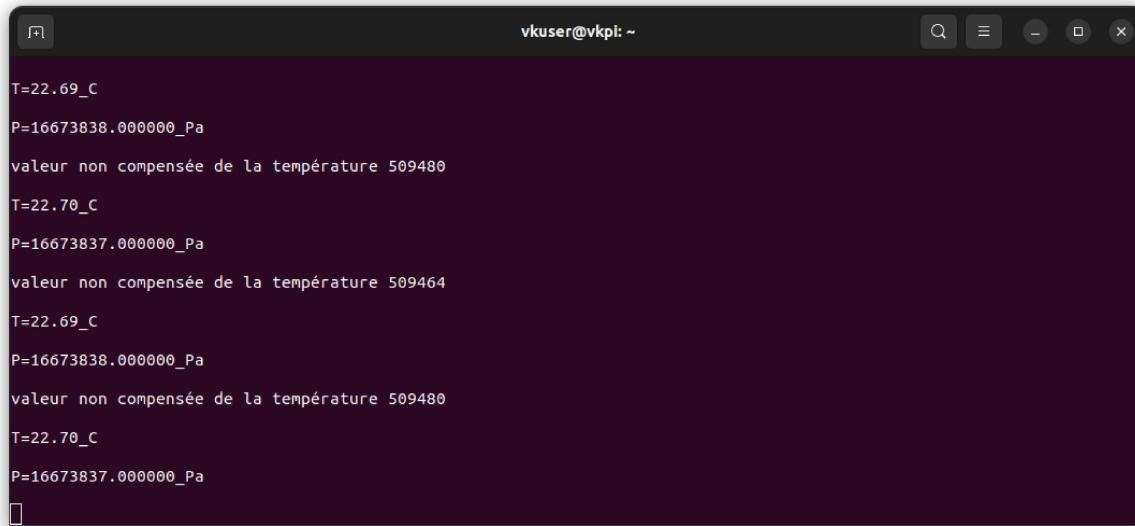
    printf("T=%ld%ld.%ld%ld_C\r\n", (temp_comp/1000)%10, (temp_comp/100)%10,
(temp_comp/10)%10, temp_comp%10);

    //Pression
    pres_uncompen=BMP280_get_pressure(); //récupérer la pression
    pres_comp=bmp280_compensate_P_int64(pres_uncompen); //compenser la
pression
    //HAL_Delay(1000);
}

```

```
printf("P=%f_Pa \r\n", ((float)(pres_comp))/256);
```

Ce qui affiche :



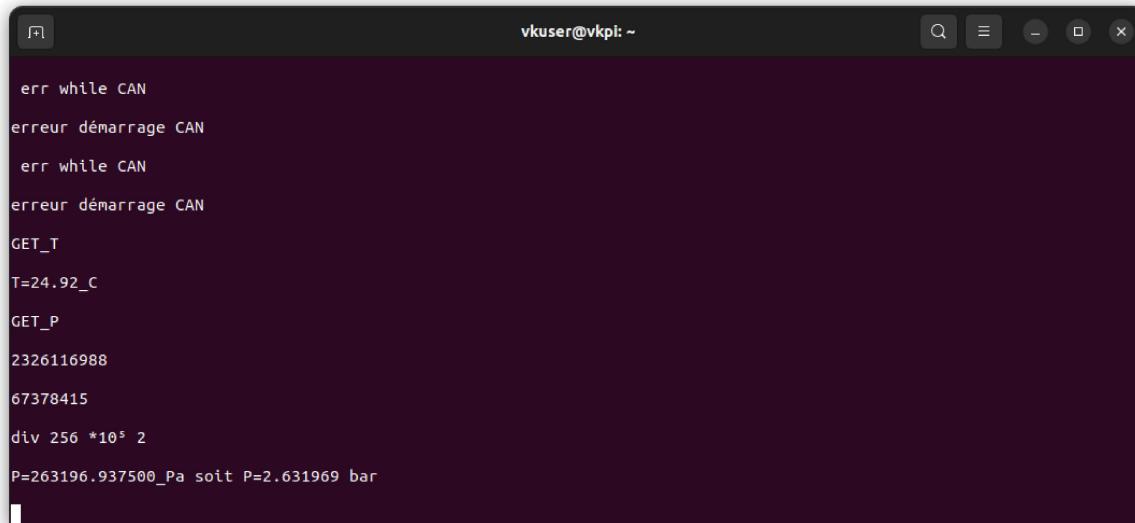
```
vkuser@vkpi: ~
T=22.69_C
P=16673838.000000_Pa
valeur non compensée de la température 509480
T=22.70_C
P=16673837.000000_Pa
valeur non compensée de la température 509464
T=22.69_C
P=16673838.000000_Pa
valeur non compensée de la température 509480
T=22.70_C
P=16673837.000000_Pa
```

La température est affiché au format T=22.72_C .

La pression est renvoyée en **format Q24.8**, où les 24 bits représentent la partie entière de la pression et les 8 bits représentent la partie fractionnaire.

Ainsi, la pression compensée est donnée sous la forme **P / 256**, où **P** est l'entier brut en **Pa**.

la doc a mentionné qu'une valeur de **24674867** correspond à une pression de **963.86 hPa**.



```
vkuser@vkpi: ~
err while CAN
erreur démarrage CAN
err while CAN
erreur démarrage CAN
GET_T
T=24.92_C
GET_P
2326116988
67378415
div 256 *10^5 2
P=263196.937500_Pa soit P=2.631969 bar
```

3. TP2 - Interfaçage STM32 - Raspberry

3.1. Mise en route du Raspberry PI Zéro

Premier démarrage

On installe sur la carte sd raspbian os lite 32 bits avec l'outil pi imager en précisant comme user (pi0) et comme mdp (password).

Pour le réseau, on configure ESE_Bus_Network pour le ssid et ilovelinux pour le mdp.
L'adresse IP est attribuée par le routeur : 192.168.88.237.

231 disparue

carte sd de vincent

On a refait les étapes d'installation avec d'autres paramètres de compte et de réseau.

nom d'hôte: vkpi

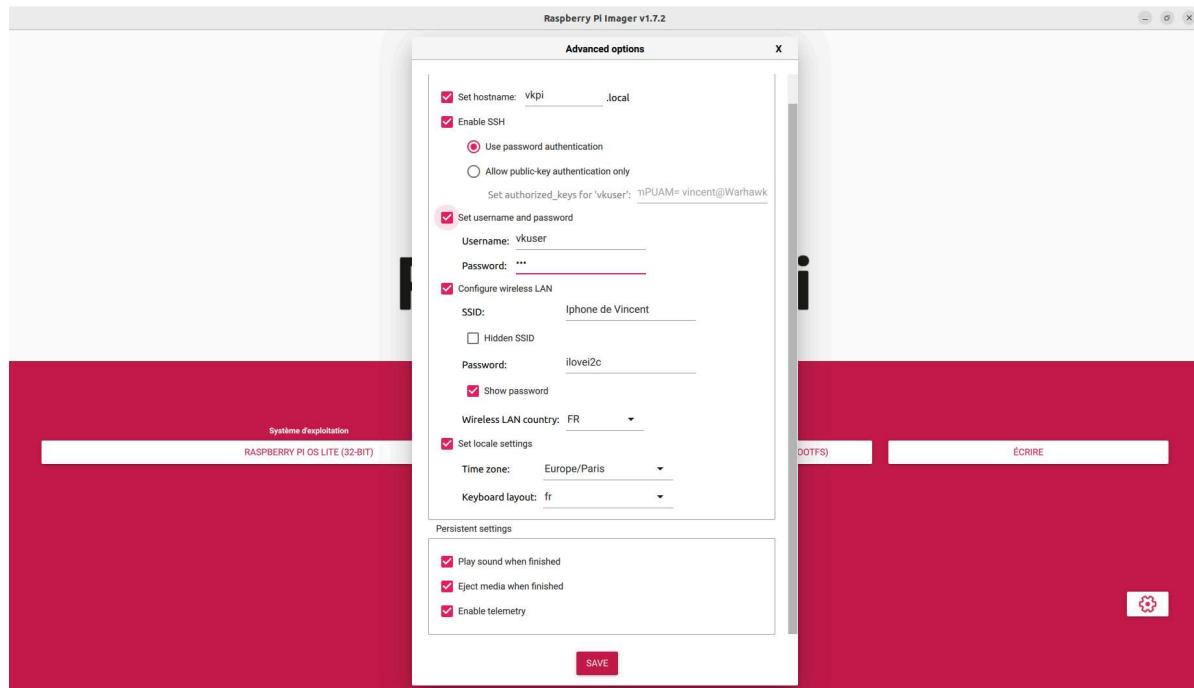
user : vkuser

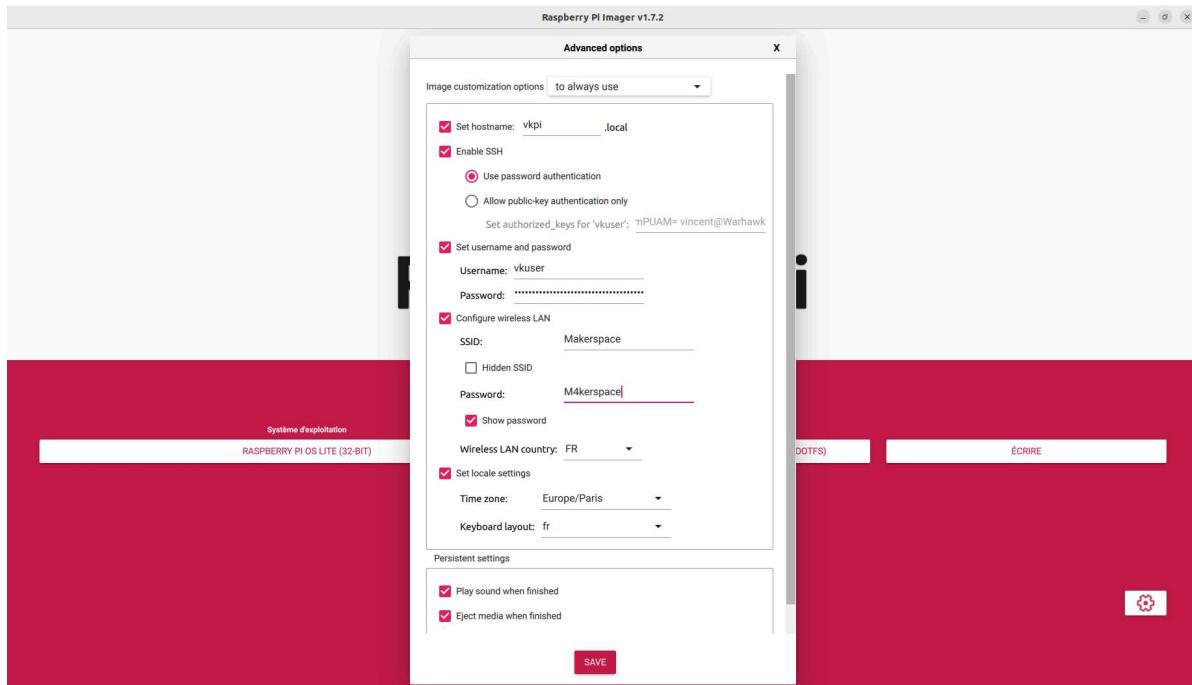
mot de passe : mdp

Réseau

ssid: iPhone de vincent

mot de passe : mdp





avec le réseau wifi du makerspace :

user : vkuser

mdp: mdp

3.2 Port série

Loopback

On modifie les fichiers suivant pour pouvoir accéder à ce port série sur la pi zero :

-/boot/firmware/config.txt

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
```

-/boot/firmware/cmdline.txt

```
console=serial0,115200
```

Dans minicom il faut configurer les paramètre suivants en appuyant sur ctrl+A puis O

```
come to minicom 2.8

I+-----+
t| A - Serial Device      : /dev/ttyAMA0
| B - Lockfile Location   : /var/lock
s| C - Callin Program     :
| D - Callout Program    :
| E - Bps/Par/Bits       : 115200 8N1
| F - Hardware Flow Control: No
| G - Software Flow Control: No
| H - RS485 Enable        : No
| I - RS485 Rts On Send   : No
| J - RS485 Rts After Send: No
| K - RS485 Rx During Tx  : No
| L - RS485 Terminate Bus : No
| M - RS485 Delay Rts Before: 0
| N - RS485 Delay Rts After : 0
|
| Change which setting? □
+-----+
```

Serial port setup>F - Hardware Flow Control : no> il faut appuyer sur la touche F

Puis appuyer sur ctrl+A puis Z

```
pi0@pi0: ~

Welcome to minicom 2.8 +-----+ Minicom Command Summary +-----+
OPTIONS: I18n | Port /dev/ttyAMA0, 09:0 | Commands can be called by CTRL-A <key>
Press CTRL-A Z for help | Main Functions | Other Functions |
| Dialing directory...D | run script (Go)....G | Clear Screen.....C |
| Send files.....S | Receive files.....R | cOnfigure Minicom..O |
| comm Parameters....P | Add linefeed.....A | Suspend minicom....J |
| Capture on/off.....L | Hangup.....H | exit and reset.....X |
| send break.....F | initialize Modem...M | quit with no reset.Q |
| Terminal settings..T | run Kermit.....K | Cursor key mode....I |
| lineWrap on/off....W | local Echo on/off..E | Help screen.....Z |
| Paste file.....Y | Timestamp toggle...N | scroll Back.....B |
| Add Carriage Ret...U |
|
| Select function or press Enter for none.□
+-----+
```

Il faut appuyer sur E pour activer le local Echo on/off, ainsi lorsque l'on appuie sur une touche elle est automatiquement renvoyé dans le terminal à la suite du caractère qu'on vient de taper comme montré ci-dessous dans la figure 'loopback test'

```

pi0@pi0: ~
Welcome to minicom 2.8
OPTIONS: I18n
Port /dev/ttyAMA0, 09:02:35
Press CTRL-A Z for help on special keys
zzzzzzzzzz
tt

```

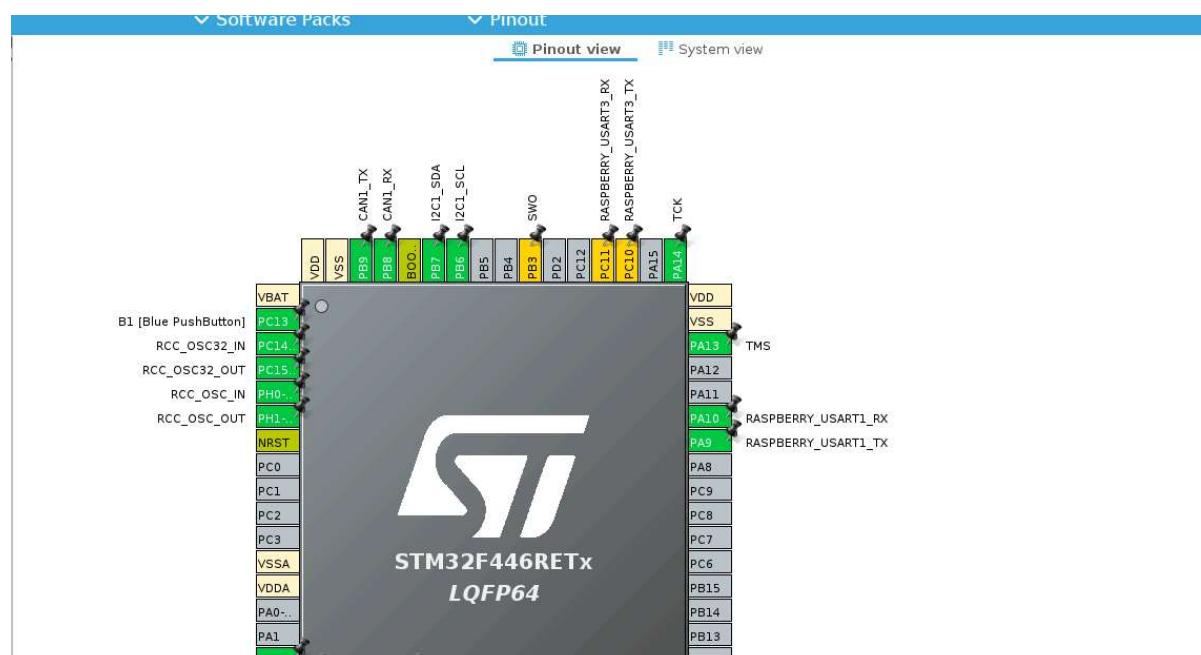
figure loopback test

Par exemple, en tapant t il se réaffiche grâce au echo qui affiche les caractères du rx sur le terminal.

Communication avec la STM32

On teste la connexion en UART, on connecte Rx et Tx et on utilise minicom pour s'auto-envoyer des messages. Une fois que ça fonctionne, on peut connecter Rx sur Tx de la STM32 et Tx sur Rx de la STM32, on utilise l'UART1 et les pins PA9 et PA10. Il ne faut pas oublier la masse.

Attention au cablage , il faut câbler sur pa10 et pa9 et



On a besoin d'un autre port UART qui va communiquer à la pi zero, nous avons activé l'UART1 sur les pins

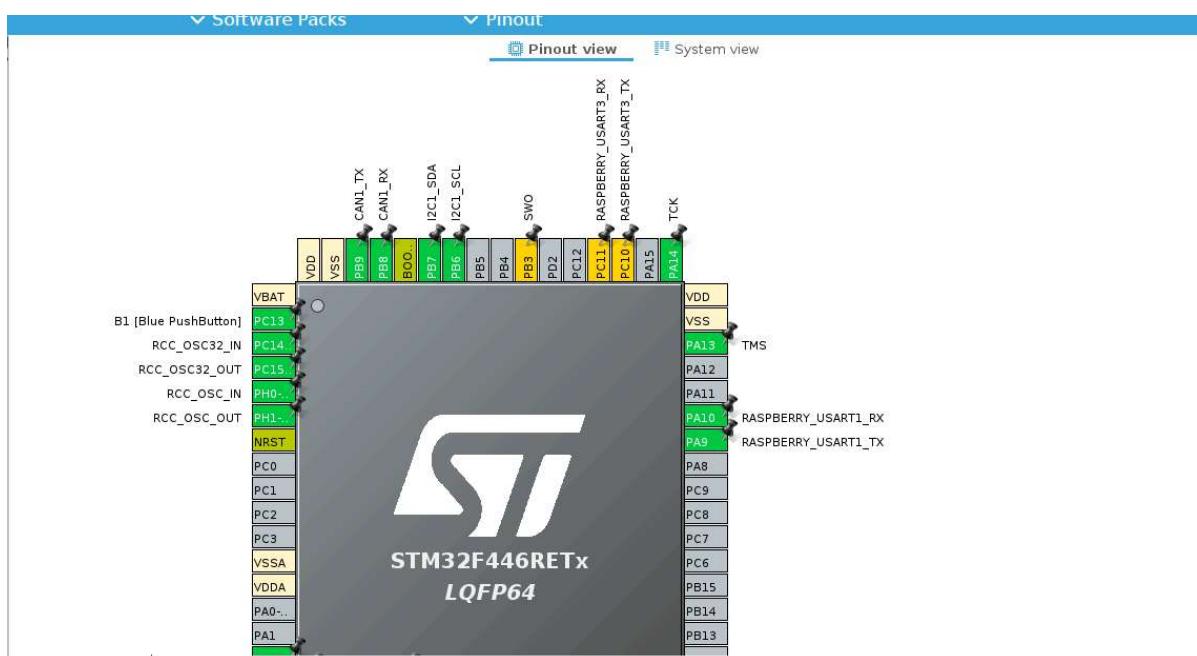
pin PC10 USART1_RX

pin PC9 USART1_TX

La pi zero est branché sur les pins PA10 et PA9 de la pi zero (avec le GND).

pin PC5 USART3_RX

pin PB10 USART3_TX



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 (USART 2) sur USB, en ajoutant le code suivant au fichier `stm32f4xx_hal_msp.c`. On va modifier la fonction `printf` pour quelle affiche sur les 2 ports série en même temps. On part de code :

```
/* 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); //transmission à la nucleo
    HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, 0xFFFF); //transmission à la pi zero
    return ch;
}
/* USER CODE END 1 */

```

On ajoute également la transmission vers l'usart1 pour assurer la redirection du printf dans la console de l'USART1.

On rajoute l'écriture dans l'USART1 avec la fonction Transmit dans stm32f4xx_hal_msp.c:

```
HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, 0xFFFF);
```

Ce qui donne le code suivant pour la fonction printf

```

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);
    HAL_UART_Transmit(&huart1, (uint8_t *)&ch, 1, 0xFFFF);
    return ch;
}

```

Activation de l'usart dans la PI ZERO

On active l'usart dans la PI ZERO : `sudo raspi-config > advanced options>seiral port activate`

Communication entre la PI zero et le STM32

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

Requête du RPI ZERO	Réponse du STM	Commentaire
GET_T	T=+12.50_C	Température compensée sur 10 caractères

Requête du RPI ZERO	Réponse du STM	Commentaire
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

Pour écrire ce protocole on va écrire une fonction **void dial_pi()** dans BMP280_vincent.c. La taille du tableau qui reçoit les données de l'UART est définie dans le fichier BMP280_vincent.h

```
#define RX_BUFF_SIZE 7
```

```
void dial_pi(){

    if(strncmp(RxBuff, "GET_T", 5)==0) {

        //déclaration des variables contenant la température non compensée
        temp_uncompen= BMP280_get_temperature(); //récupérer la température

        //déclaration des variables contenant la température compensée
        BMP280_U32_t temp_comp;

        temp_comp=bmp280_compensate_T_int32(temp_uncompen); //récupérer la
        température compensé

        //ancien printf
        //printf("%u \r\n",temp_comp);// AFFICHÉ TEMPÉRATURE compensée sur
        l'usart

        printf("T=%1d%1d.%1d%1d_C\r\n", (temp_comp/1000)%10, (temp_comp/100)%10,
        (temp_comp/10)%10,temp_comp%10);
        //T=+12.50_C      Température compensée sur 10 caractères
        //GET_P      P=102300Pa

        //HAL_Delay(1000);
        //interaction sur le moteur de la température
        can_change_temp(temp_comp);

    }

    if (strncmp(RxBuff,"GET_P",5)==0){

        //déclaration des variables contenant la pression non compensée
        //pres_uncompen= BMP280_get_pressure(); //récupérer la température
```

```

//déclaration des variables contenant la pression compensée
BMP280_U32_t pres_comp;

pres_uncompen=BMP280_get_pressure(); //récupérer la pression non
compensée

pres_comp=bmp280_compensate_P_int64(pres_uncompen); //compenser la
pression

//printf("%u \r\n",pres_uncompen);
//printf("%u \r\n",pres_comp);
//printf("div 256 *105 %u \r\n", ((pres_comp) / (25600000) ));

printf("P=%f_Pa \r\n",((float)(pres_comp))/256);

}

if (strncmp(RxBuff,"SET_K=",10)==0){
    //K_pid=RxBuff[7];
    K_pid=0;

    printf("K %d",K_pid);

    //
    //    BMP280_U32_t pres_comp;
    //    pres_uncompen=BMP280_get_pressure(); //récupérer la pression non
compensée
    //
    //    pres_comp=bmp280_compensate_P_int64(pres_uncompen); //compenser
la pression
    //
    //
    //    printf("%u \r\n",pres_uncompen);
    //
    //    printf("%u \r\n",pres_comp);

}

if(strncmp(RxBuff,"GET_A",5)==0){
    a_pid=145;
    printf("A=%d\r\n",a_pid);
}

// Réinitialiser RxBuff après chaque commande
//memset(RxBuff, 0, RX_BUFF_SIZE);

}

void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)

```

```

{
    dial_pi();
    HAL_UART_Receive_IT(&huart1, RxBuff, RX_BUFF_SIZE);

}

```

On compare le contenu des messages stockés dans le bufer de l'USART pour lancer la bonne section de code dans dial_pi().

Test d'envoi des ordres à la STM32

On se connecte à la pi zero en ssh puis on ouvre minicom avec la commande suivante :

```
minicom -D /dev/serial0
```

On reçoit bien la température et la pression.

3.3 Commande depuis Python

Création de l'environnement à l'aide des commandes au tableau: voir photo pipenv 25/10

Suite à un problème de version de blinker, on procéde à l'installation sans le requirements.txt, package par package.

On établi également un environnement pour notre futur serveur.

4. TP3- Interface REST

4.1. Installation du serveur Python

Configuration et Initialisation

- Configuration d'un serveur web avec Flask pour gérer les routes API.
- Connexion au port série pour communiquer avec la STM32, permettant de recevoir et envoyer des données (température, pression, échelle).

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

Le décorateur permet d'accéder et de se positionner dans le lien /api/welcome/.

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

Le fragment permet l'affichage de chaque lettre de Welcome, de 0 à 6 donc, none ensuite.

4.2 Première page REST

La fonction `json.dumps()` convertit bien un objet Python en une chaîne JSON, mais elle ne configure pas automatiquement les **en-têtes HTTP** pour indiquer que la réponse est du type JSON. C'est pourquoi, ici, même si le contenu retourné est formaté comme du JSON, l'en-tête HTTP **Content-Type** n'indique pas correctement `application/json`.

Les clients se basent sur l'en-tête `Content-Type` pour interpréter correctement la réponse.

Si le `Content-Type` n'est pas défini comme `application/json`, certains clients pourraient traiter la réponse comme du texte brut, même si son contenu est du JSON valide.

C'est pour cela que l'on a recourt à Flask.

4.3. Nouvelles méthodes HTTP

Gestion des Routes API

- `/api/welcome/`
 - : Route pour gérer un message d'accueil avec les méthodes :
 - **GET** : Récupère le message d'accueil actuel.
 - **POST** : Met à jour le message d'accueil avec une nouvelle valeur.
 - **DELETE** : Réinitialise le message d'accueil à une chaîne vide.

- `/api/welcome/<int:index>`
 - : Route pour manipuler une lettre spécifique du message d'accueil selon l'index, avec les méthodes :
 - **GET** : Récupère la lettre à l'index donné.
 - **POST** : Remplace le message d'accueil par un nouveau message.
 - **PUT** : Insère un mot au niveau de l'index spécifié dans le message d'accueil.
 - **PATCH** : Remplace une lettre spécifique à l'index donné.

- o **DELETE** : Supprime la lettre à l'index spécifié.

Gestion des Capteurs

- Température (`/api/temp/`)
 - o **GET** : Renvoie toutes les valeurs de température enregistrées.
 - o **POST** : Récupère une nouvelle mesure de température du STM32 et l'ajoute au tableau.
- Température avec Index (`/api/temp/<int:index>`)
 - o **GET** : Renvoie la valeur de température à un index donné.
 - o **DELETE** : Supprime la valeur de température à l'index donné.
- Pression (`/api/pres/`)
 - o **GET** : Renvoie toutes les valeurs de pression enregistrées.
 - o **POST** : Récupère une nouvelle mesure de pression du STM32 et l'ajoute au tableau.
- Pression avec Index (`/api/pres/<int:index>`)
 - o **GET** : Renvoie la valeur de pression à un index donné.
 - o **DELETE** : Supprime la valeur de pression à l'index donné.

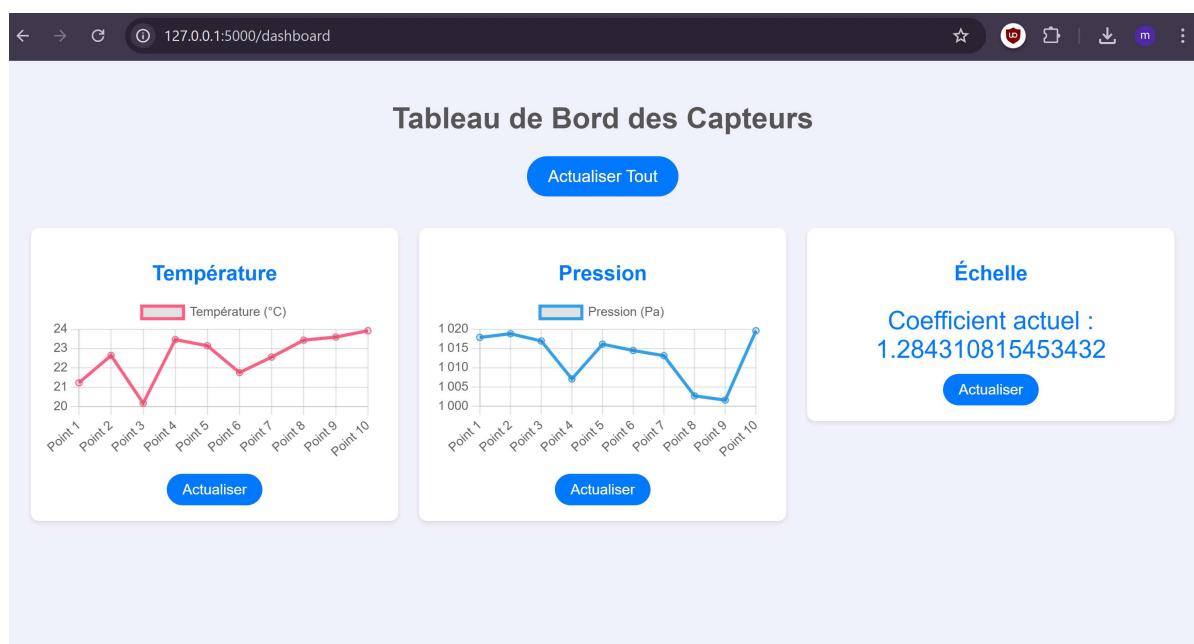
Gestion de l'Échelle

- `/api/scale/` : Récupère le coefficient d'échelle du STM32.
- `/api/scale/<int:index>` : Modifie le coefficient d'échelle dans le STM32 en envoyant une nouvelle valeur.

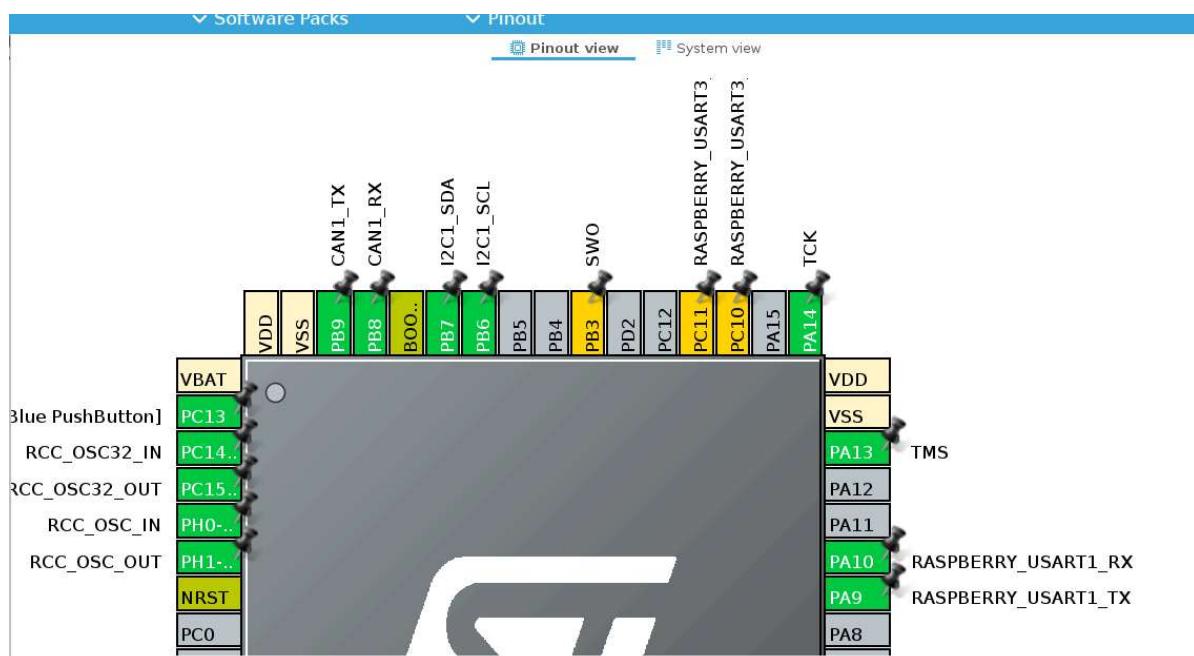
Nous créons un dashboard.html pour recueillir l'ensemble des données de température et de pression recueillies et les compiler sous forme de graphes.

Nous pouvons envoyer nos requête avec RESTED qui déclenche le code python et l'obtention de la température et de la pression au format JSON.

Ce dashboard est construit à l'aide d'un tableau composé des valeurs renvoyées sur l'uart préalablement par passées par des fonctions de parcing pour isoler la valeur numérique de la pression et de la température.(T= _C)



5.TP4 : CAN



On peut voir sur l'image ci-dessus que l'on a déplacé les pins I2C pour pouvoir mettre le CAN TX sur PB9 et le CAN RX sur PB8.

Il faut activer le CAN sur les pin PB8 et PB9 dans l'ioc.

Il faut configurer la clock à 80 MHz dans clock configuration avec l'ide qui calcule les coefficients pour nous afin de fixer la fréquence de communication avec le bus can à 500 kbits/s.

prescaler	segment 1	segment 2
10 que l'on divise par 2 donc 5	13	2

5.1 Pilotage du moteur

Pour piloter le moteur, nous avons codé plusieurs fonctions. Premièrement la fonction `enable_can()` est la première appelée dans le main.

On active le module can dans ce bout de code :

```

//*****Activer le module
can
HAL_StatusTypeDef start_can= HAL_CAN_Start(&hcan1);
if(start_can!= HAL_OK) {

    // Gérer l'erreur de démarrage
    printf("erreur start can_config CAN\r\n");

}

else{
    printf("[enable_can] start CAN OK\r\n");
}

```

Ensuite on initialise les variables utiles pour le CAN (pHeader et aData).

```

//*****déclaration du
pHeader*****


//CAN_TxHeaderTypeDef pHeader;
//uint8_t aData[3]; // Tableau pour les données à transmettre

// Configuration du champ pHeader
pHeader.StdId = 0x61;           // Identifiant standard pour la commande
"Angle"
pHeader.IDE = CAN_ID_STD;      // (0x00000000U) !< Standard Id
pHeader.RTR = CAN_RTR_DATA;    // (0x00000000U) !< Data frame
pHeader.DLC = 2;
pHeader.TransmitGlobalTime = DISABLE;

aData[0] = 90; // D0 : 90° en hexadécimal (0x5A)
aData[1] = 0x00; // D1 : Angle positif

```

Enfin, on envoie cette configuration au moteur via le bus CAN grâce à la fonction `tx_can`

```

HAL_StatusTypeDef tx_can(CAN_TxHeaderTypeDef pHeader2,uint8_t* aData2,uint32_t
pTxMailbox2){

    HAL_StatusTypeDef error_out;
    //uint32_t pTxMailbox; // variable pour stocker l'indice de la boîte aux
lettres CAN

    // Envoi du message CAN avec l'angle de 90°
    HAL_StatusTypeDef status = HAL_CAN_AddTxMessage(&hcan1, &pHeader2, aData2,
&pTxMailbox2);
    error_out=status;
    // vérification du statut d'envoi
    if (status != HAL_OK) {
        // Gérer l'erreur d'envoi
        printf("erreur HAL_CAN_AddTxMessageCAN\r\n");
    }
}

```

```

    else{
        printf("[tx_can]  envoi reussi\r\n");
    }

    return error_out;
}

}

```

La fonction `tx_can()` sert à envoyer avec `HAL_CAN_AddTxMessage` au moteur les nouvelles données pour changer l'angle, par exemple, contenu dans la variable `aData[]`.

```

//*****transmission au can de la configuration*****
HAL_StatusTypeDef conf_errout=tx_can(pHeader, aData, pTxMailbox);
// vérification du statut d'envoi
if (conf_errout != HAL_OK) {
    // Gérer l'erreur d'envoi
    printf("erreur config_base_ CAN\r\n");
}
else{
    printf("config base  envoi reussi\r\n");
}

```

La fonction `can_setter()` sert à faire tourner le moteur dans le sens opposé. Elle est utilisé dans la boucle while pour faire bouger le moteur de +90° à -90 ° en continu.

```

void can_setter(CAN_TxHeaderTypeDef pHeader, uint8_t* aData,uint32_t pTxMailbox){

    // CAN part start  while loop
    aData[0] = 90; // D0 : 90° en hexadécimal (0x5A)
    aData[1] = 1-aData[1]; //


    HAL_StatusTypeDef rota_out=tx_can(pHeader, aData, pTxMailbox);
    if (rota_out != HAL_OK) {
        // Gérer l'erreur d'envoi
        printf(" err rota CAN\r\n");
    }
    else {
        printf("change rotation complete \r\n");
    }
}

```

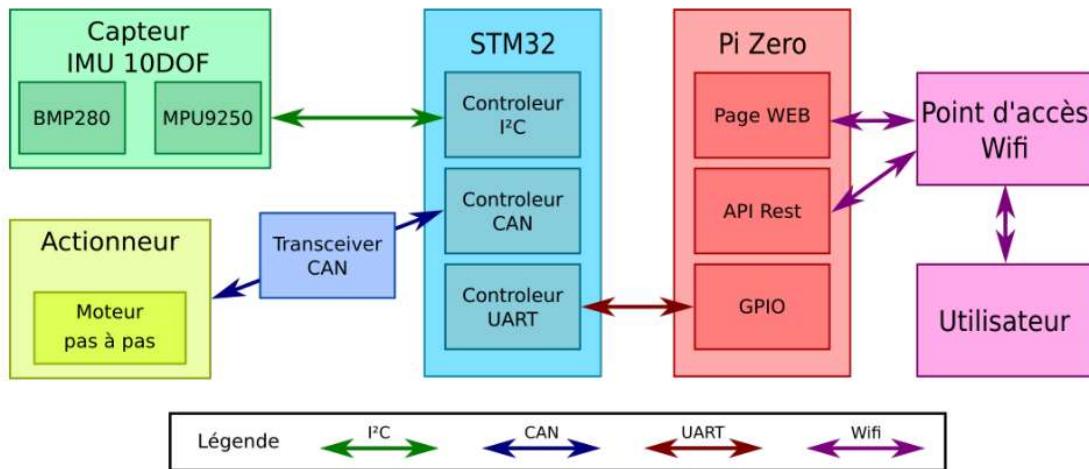
5.2. Interfaçage avec le capteur

On fait maintenant en sorte que lorsqu'une nouvelle valeur de température est renvoyée par le capteur, celle-ci soit utilisé pour établir un coefficient pour obtenir un angle de rotation du moteur. Cet angle est ensuite envoyé via le bus CAN au moteur pour que celui-ci tourne en fonction de la température. Le tout dans la fonction `can_change_temp()` qui sert à bouger le moteur en fonction de la variation de température, elle est appelée lors de la requête `GET_T` dans le code de `dial_pi()`

```
void can_change_temp(BMP280_U32_t temp_comp){  
  
    enable_can();  
  
    //TP4 partie CAN d  
    //début *****  
  
    // récupérer temps*****  
    // BMP280_S32_t temp_uncompen;  
    //  
    // temp_comp=bmp280_compensate_T_int32(temp_uncompen); //récupérer la  
    //température compensé  
    // //déclaration des variables contenant la température non compensée  
    // temp_uncompen= BMP280_get_temperature(); //récupérer la température  
  
    //déclaration des variables contenant la température compensée  
    BMP280_U32_t old_temp;  
  
    //*****temp_comp = temperature *****  
  
    if(fabs(temp_comp-old_temp)>2){  
  
        if(temp_comp>old_temp){//On bouge dans le sens 0x00 +  
  
            aData[1]=0x00;  
            aData[0]=(int)(10*(temp_comp-old_temp));  
            HAL_CAN_AddTxMessage(&hcan1, &pHeader, aData, &pTxMailbox);  
            //printf("adata0= %d\r\n",aData[0]);  
  
            old_temp=temp_comp;  
        }  
        else{//On bouge dans le sens 0x01 -  
            aData[1]=0x01;  
            aData[0]=(int)((old_temp-temp_comp));  
            HAL_CAN_AddTxMessage(&hcan1, &pHeader, aData, &pTxMailbox);  
            //printf("adata0= %d\r\n",aData[0]);  
            old_temp=temp_comp;  
        }  
    }  
}
```

5.TP5 : Mise en série de l'ensemble

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

Comme visible sur la vidéo fournie, la température fait varier la position du moteur proportionnellement à son augmentation relative.

```
vkuser@vkpi: ~
config : well configured - configuration value = hexadecimal : 0x57 | decimal : 0d87
main
GET_T
T=22.96_C
[enable_can] start CAN OK
[tx_can] envoi réussi
config base envoi réussi
GET_P
P=16643235.000000_Pa
GET_A
A=145
```

On récupère également la pression qui est stockée dans un tableau python.

La mesure de l'angle n'a pas pu être récupérée sur l'interface REST, même si elle est disponible depuis minicom.

Les données sont ensuite compilés dans le dashboard qui permet de voir les évolutions au fur et à mesure des requêtes envoyées et d'avoir une vue d'ensemble.

Nous avons donc l'ensemble du système qui fonctionne.

Vidéo finale disponible en annexe