MAKERS FOR LIFE

MakAir

Mesure avec un Capteur de pression

BOUTRY Loan

Alternant sur le projet MakAir

Ce document fait partie d'une série de livrables concernant la conception et le prototypage d'un respirateur à pression positive continue pour diminuer l'apnée du sommeil d'un patient

Ce livrable a pour but d'expliquer le fonctionnement du capteur de pression. La première partie du document est une analyse des caractéristiques du capteur. La deuxième partie portera sur la récupération des données de la mesure de la pression.

Table des matières

Table des figures

C'est quoi un Capteur de pression ?

La pression est définie comme la force appliquée par un liquide ou un gaz sur une surface et est généralement mesurée en unités de force par unité de surface. Les unités communes sont le Pascal (Pa), le Bar (barre), N / mm2 ou psi (livres par pouce carré). Par conséquent, un capteur de pression est un instrument composé à la fois d'un élément sensible à la pression pour déterminer la pression réelle appliquée au capteur et de certains composants pour convertir cette information en un signal de sortie. Pour adapter notre capteur à notre système, nous devons faire attention à certains paramètres :

- **Type de mesure** : Il y a 3 types de mesure avec un capteur de pression : absolue, relative et différentielle

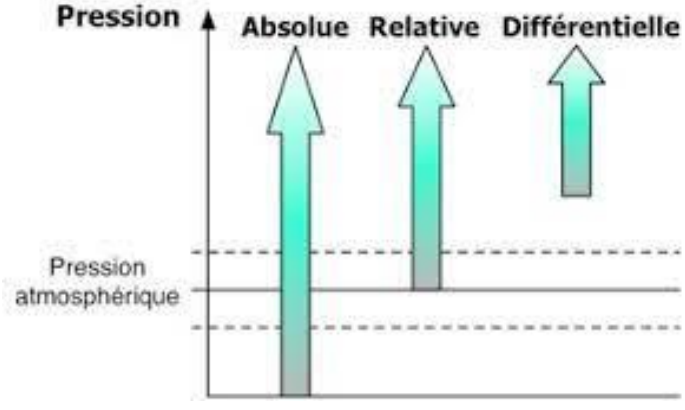


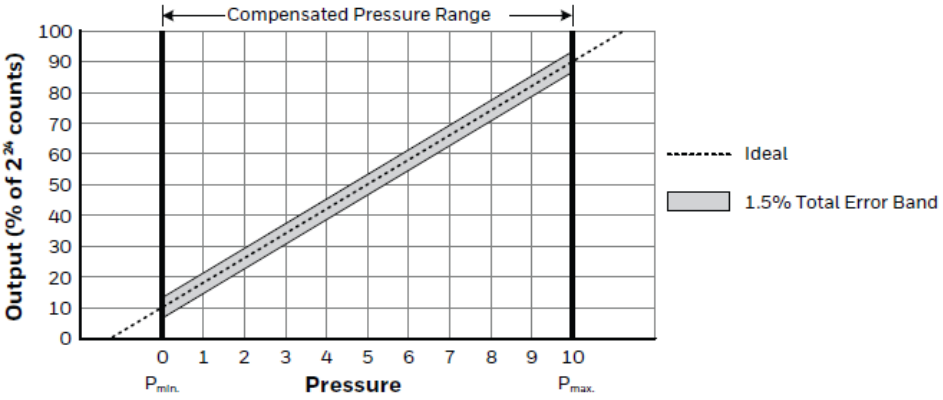
Figure 1 : Les différents types de mesure d'une pression

- Les capteurs de pression absolue mesurent la pression par rapport à une référence vide donc une pression à 0. Ils sont conçus avec une référence qui est une chambre fermée dans le capteur (presque vide)
- Les capteurs de pression relative mesurent la pression par rapport à une pression de référence qui est la pression atmosphérique. Ils sont conçus avec une référence qui est l'air extérieur, grâce à un petit trou en dessous du capteur
- Les capteurs de pression différentielle mesurent la différence entre deux pressions, c'est souvent utilisé pour mesurer les niveaux de fluide et les débits. Ils sont conçus avec une référence qui est une autre sortie du capteur

Pour notre système (et en général dans le domaine médical), nous choisissons une mesure différentielle afin de comparer la pression interne et externe du système.

- **Plage de mesure du capteur** : c'est un paramètre important puisqu'il faut un capteur qui puisse mesurer toute la plage de pression de l'air que le système va produire. Il y a différentes unités liées à la pression comme nous avons pu le voir en introduction. Pour notre système, il est obligatoire d'avoir des valeurs négatives puisque la pression va être différente entre l'inspiration et l'expiration du patient
- **La précision et répétabilité du capteur** : ce sont des paramètres communs à tous les capteurs mais qui restent indispensable à prendre en compte. Les choix se font en fonction du domaine d'utilisation
- **L'entrée du capteur** : Il est important de savoir comment on va pouvoir mesurer une pression en faisant passer de l'air dans notre capteur. Pour un capteur différentiel, nous avons 2 entrées pour pouvoir insérer l'air externe et interne du système
- **La tension nominale** : C'est un paramètre plus spécifique à l'électronique, mais il est important de la savoir afin de connaître la tension que le système doit apporter au capteur de pression pour qu'il fonctionne correctement.
- **La tension de sortie du capteur** : Ce paramètre concerne la sécurité du microcontrôleur. Il faut faire attention à la tension maximale que peut sortir le capteur afin d'éviter d'endommager les pins (ex : pin STM32 = 3,3V max en entrée).
- **Interface de communication** : le dernier paramètre important, c'est la communication entre le microcontrôleur et le capteur de pression puisque c'est par cette voie que les données de la mesure vont être envoyées et reçues. Cela peut se faire avec une sortie analogique, une communication I2C ou encore une communication SPI. Dans notre cas, ça sera en communication I2C.

Pour la réalisation du respirateur PPC, nous prenons un capteur de pression différent du MakAir puisque celui n'est plus disponible chez les fournisseurs. Nous prenons le capteur de pression : ABP2MRRT060MD2A3XX. C'est un capteur de pression adapté au domaine médical et assez performant pour la mesure de la pression d'un respirateur PPC. Vous pouvez retrouver le choix technologique du capteur de pression en annexe afin d'avoir un exemple du procédé de recherche de capteurs. Pour notre capteur de pression, on peut retrouver sa caractéristique ci-dessous :



Pressure example 1: Transfer Function A (10% to 90%)

$$\text{Output (\% of } 2^{24} \text{ counts)} = \frac{80\%}{P_{max.} - P_{min.}} \times (\text{Pressure}_{applied} - P_{min.}) + 10\%$$

Pressure example 2: Transfer Function B (30% to 70%)

$$\text{Output (\% of } 2^{24} \text{ counts)} = \frac{40\%}{P_{max.} - P_{min.}} \times (\text{Pressure}_{applied} - P_{min.}) + 30\%$$

Figure 2 : Caractéristique du ABP2MRRT060MD2A3XX

Supply voltage	3,3 Vdc
Précision	± 0.25 %
Plage de mesure	-60 à 60 mbar
Communication	**I2C**

La communication I2C

C'est un protocole de communication en série très commun en électronique qui a été développé au début des années 80 par Philips pour permettre de relier facilement à un microprocesseur les différents circuits d'un téléviseur moderne.

La communication I2C permet de faire communiquer entre eux des composants électronique très divers grâce à 3 signaux :

- Signal de donnée : SDA
- Signal d'horloge : SCL
- Signal de référence électrique : masse

Chaque protocole de communication communique grâce à bus et chaque bus comporte leurs caractéristiques. Les caractéristiques du bus I2C sont :

- 1 adresse unique pour chaque périphérique
- Bus multi-maître, détection des collisions et arbitrage
- Bus série, 8 bits, bidirectionnel à 100 kbps (standard mode), 400 kbps (fast mode), 3,2 Mbps (high-speed mode)

On peut décomposer le bus I2C comme cela :

- Le bus doit être au repos avant la prise de contrôle SDA et SCL à 1
- Pour transmettre des données, il faut surveiller :
    - La condition de départ : SDA passe à 0, SCL reste à 1
    - La condition d'arrêt : SDA passe à 1, SCL reste à 1

- Après avoir vérifié que le bus est libre, puis pris le contrôle de celui-ci, le circuit en devient le maître : c'est lui qui génère le signal d'horloge

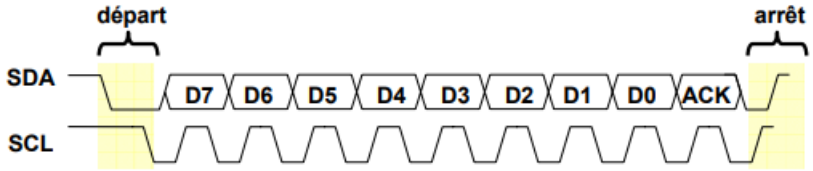


Figure 3 : Start/Stop bus I2C

Sur un core Arduino, la communication I2C est gérée par la librairie **Wire.h**

Si l'on rentre dans la partie plus concrète de l'application de l'I2C dans un système. Chaque capteur a un bus I2C différent, c'est-à-dire que chaque octet du bus correspond à une donnée différente du capteur. On peut prendre en exemple le capteur de pression :

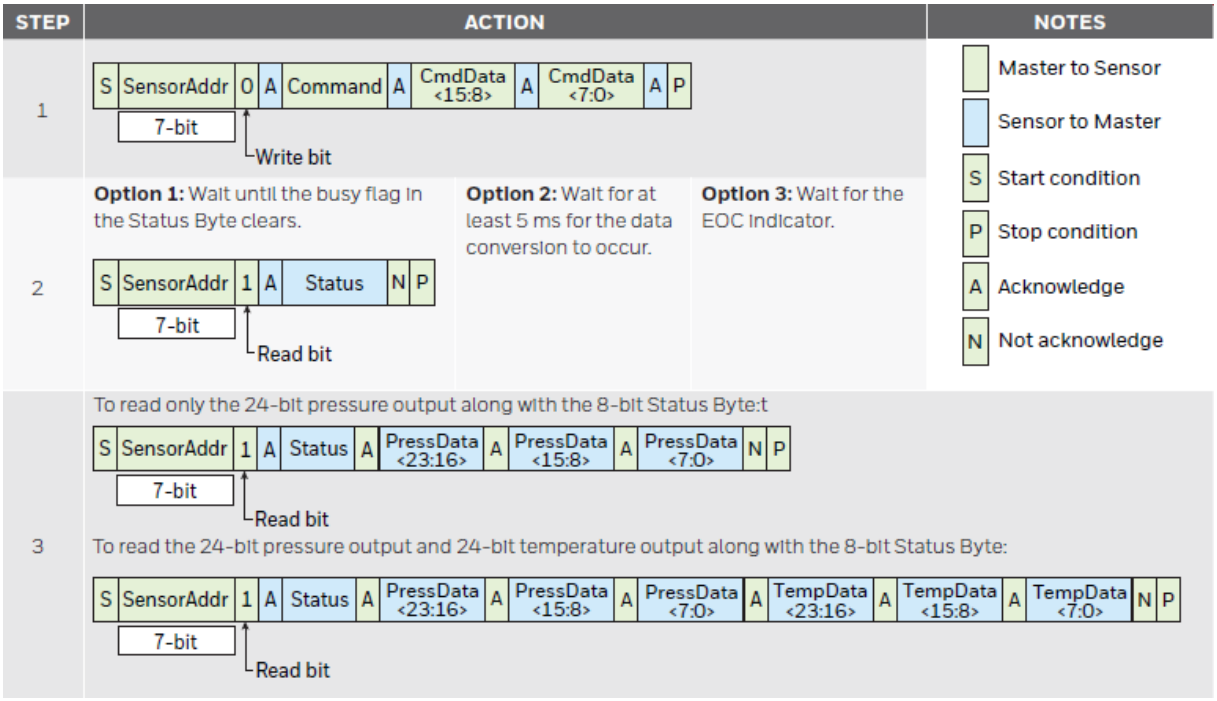


Figure 4 : Bus I2C du Capteur de pression ABP2 Series

Ce tableau explique les étapes à réaliser afin de récupérer les données du capteur. La première étape est la requête du Master au Slave. Cette étape permet d'initialiser le capteur par la communication I2C. La troisième étape est l'instant où le capteur envoie ses données au microcontrôleur, on peut voir qu'il y a 2 bus différents : un pour récupérer la pression seule et l'autre récupérer la pression et la température environnante. Petite remarque : on peut observer que le bus commence toujours par l'adressage du capteur par le Master, cela est obligatoire afin de reconnaître le capteur sur le bus I2C.

Récupération de la mesure de la pression

Nous allons passer à la partie réalisation, nous allons voir comment récupérer la mesure de la pression de l'air envoyé.

Pour récupérer les données du capteur de pression en I2C, il y a plusieurs étapes à faire :

- Câblage du capteur : Cette étape se passe plutôt en Hardware puisqu'il consiste à câbler les pins du capteur sur des ports adaptés sur le microcontrôleur. Il suffit par la suite d'activer ces pins dans l'initialisation.

- Initialisation du capteur : Cette étape est importante pour débuter la communication entre le maître et l'esclave. On va donc configurer l'adresse du capteur et le bus I2C que doit recevoir le maître.

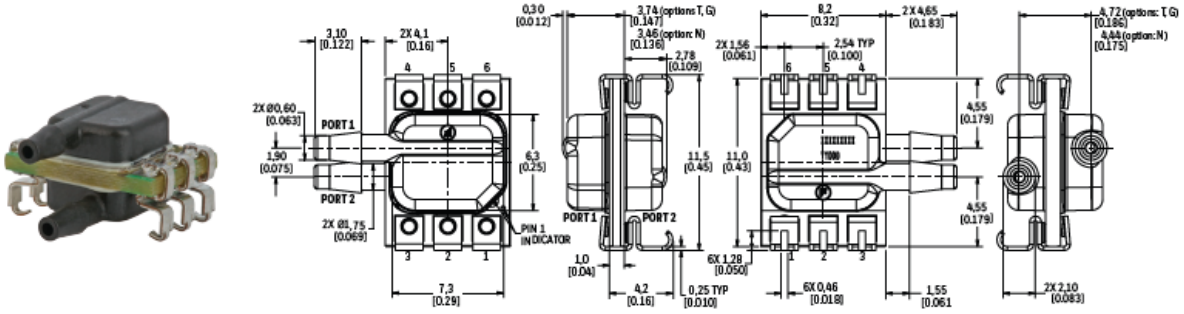
Récupération des données : Cette étape permet de récupérer le bus I2C et prendre les données communiquées par l'esclave. On peut aussi choisir si l'on intègre la fonction de mesure dans un loop() ou dans un timer configuré (cela dépend de l'application). La différence est non négligeable en fonction du système à concevoir :

- **Loop()** est un bloc du code qui va se répéter infiniment lorsque le système est en marche. C'est-à-dire que le code va être toujours être répété à chaque fin d'exécution du code. Cela peut être avantageux puisque le code sera toujours exécuté mais va être moins avantageux sur le temps d'exécution du système qui va augmenter.

- **Un Timer** est un élément configuré sur le microprocesseur afin qu'il réalise une ou des fonctions à chaque fois que le compteur (qui est configuré en fonction de la période du timer) a fini son décompte. Ce compteur s'incrémente en tâche du fond du code exécuté dans le loop(). Cela permet de réaliser des fonctions asynchrone au loop() à des périodes définies par le timer et donc d'optimiser le temps d'exécution du système.

Câblage du capteur

La première étape est de se renseigner sur l'intégration Hardware du capteur de pression. Normalement toutes les informations sont présentes sur les datasheets des composants. Dans le cas du ABP2MRRT060MD2A3XX, on retrouve les différents pins (communication I2C) qui vont être reliés au microcontrôleur :



OUTPUT TYPE	PIN 1	PIN 2	PIN 3	PIN 4	PIN 5	PIN 6
I^2C	GND	V_{DD}	EOC	NC	SDA	SCL

Figure 5 : Câblage du ABP2MRRT060MD2A3XX

GND : Ground/Masse - VDD : Voltage Drain (Positive Supply Voltage)
EOC : End-of-conversion indicator - NC : Not Connected
SDA : Serial Data Line - SCL : Serial Clock Line

EOC : Cette broche est mise à 1 lorsqu'une mesure et un calcul sont terminés

Il suffit donc de 5 fils pour câbler le capteur de pression. On peut l'ajouter au schéma de prototypage :

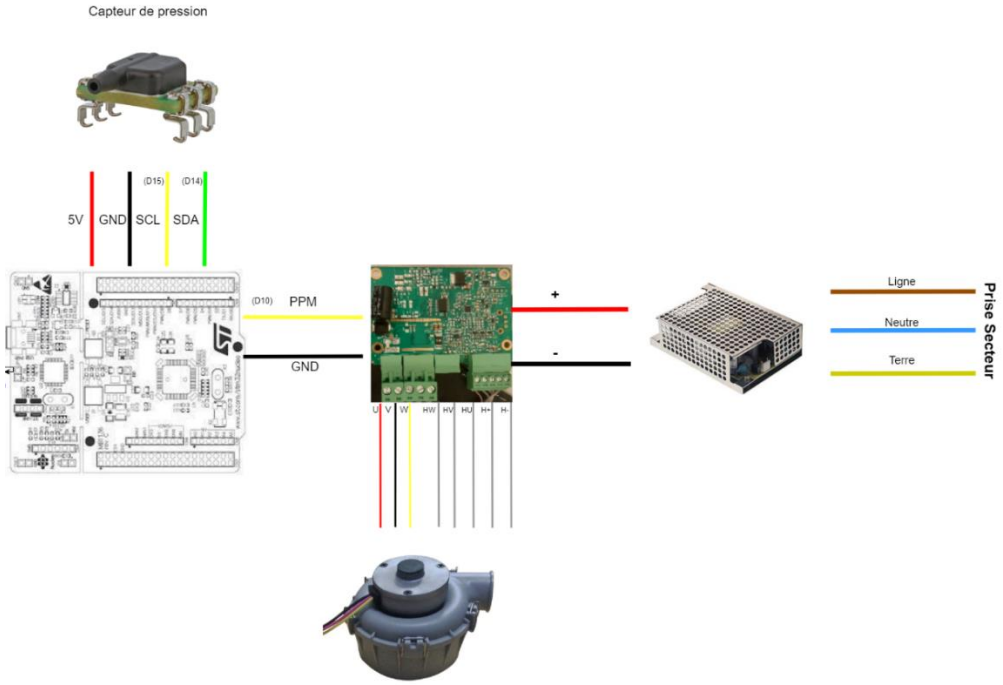


Figure 6 : Prototypage du capteur de pression

Maintenant le câblage réalisé, on peut passer à l'initialisation du capteur de pression

Initialisation du capteur

La deuxième étape va permettre d'initialiser le capteur afin de préparer la communication I2C entre le maître et l'esclave. Cette étape est un protocole précis à mettre en place pour que la récupération des données se fassent correctement.

Avant de commencer, il est bien de préciser que l'on utilise la libraire « Wire.h » qui est la librairie la plus commune pour réaliser une communication I2C sur Arduino IDE. Chaque fonction sera expliquée au fur à mesure de la découverte du code. Voici le programme d'initialisation du capteur :

```
Wire.begin();

Wire.beginTransmission(id);
int stat = Wire.write (cmd, 3); // write command to the sensor
stat |= Wire.endTransmission();
delay(10);
Wire.requestFrom(id, 7); // read back Sensor data 7 bytes
```

Figure 7 : Code d'initialisation du capteur

Nous allons analyser ce code en expliquant les différentes fonctions importantes pour l'initialisation :

- `Wire.begin();` : Cette fonction permet de débuter une communication I2C, en général cette fonction est mise dans le « setup() »
- `Wire.beginTransmission(id);` : Cette fonction permet de faire une requête de transmission à l'esclave , id correspond à l'adresse I2C du capteur = 0x28

- `Wire.write(cmd,3);` : Cette fonction réalise la première étape de la récupération des données du capteur comme nous avons pu le voir précédemment :

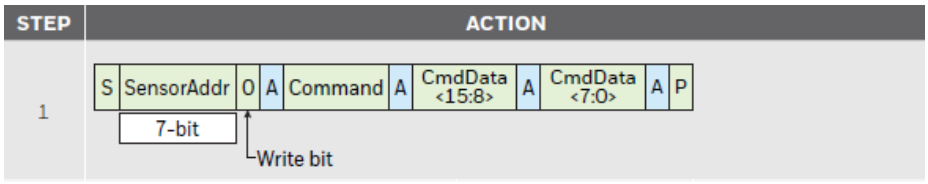


Figure 8 : Etape 1 de la communication I2C

- `Wire.endTransmission();` : Cette fonction permet de mettre fin à l'écriture et renvoie un message qui indique si la transmission est OK ou si il y a eu une erreur
- `Wire.requestFrom(id, 7);` : Cette fonction est une requête du maître pour que l'esclave lui envoie 7 octets de données (les 7 octets que l'on a pu analyser précédemment concernant la pression et la température). Cette fonction permet de définir la taille du bus I2C qui va être transmis

Maintenant que l'étape d'initialisation du capteur de pression est faite, on peut passer à la récupération de la mesure du capteur.

Récupération des données

La troisième étape va permettre de récupérer les données du capteur de pression et va les traiter pour les rendre exploitables. On fait le choix de récupérer la pression et la température :

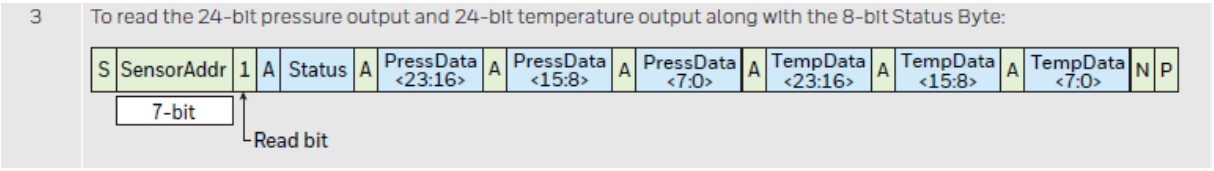


Figure 9 : Etape 3 de la communication I2C

Elle sous la forme d'une fonction que l'on peut mettre dans le loop() ou dans un timer. La différence est non négligeable en fonction du système à concevoir :

- **Loop()** est un bloc du code qui va se répéter infiniment lorsque le système est en marche. C'est-à-dire que le code va être toujours être répété à chaque fin d'exécution du code. Cela peut être avantageux puisque le code sera toujours exécuté mais va être moins avantageux sur le temps d'exécution du système qui va augmenter.

- **Un Timer** est un élément configuré sur le microprocesseur afin qu'il réalise une ou des fonctions à chaque fois que le compteur (qui est configuré en fonction de la période du timer) a fini son décompte. Ce compteur s'incrémente en tâche du fond du code exécuté dans le loop(). Cela permet de réaliser des fonctions asynchrone au loop() à des périodes définies par le timer et donc d'optimiser le temps d'exécution du système.

Mise en place dans un loop()

Dans ce cas, on met cette fonction dans le loop().

```
void Pressure() {
  Wire.beginTransmission(id);
  int stat = Wire.write (cmd, 3); // write command to the sensor
  stat |= Wire.endTransmission();
  delay(10);
  Wire.requestFrom(id, 7); // read back Sensor data 7 bytes
  int i = 0;
  for (i = 0; i < 7; i++) {
    data [i] = Wire.read();
  }
  press_counts = data[3] + data[2] * 256 + data[1] * 65536; // calculate digital pressure
counts
  temp_counts = data[6] + data[5] * 256 + data[4] * 65536; // calculate digital temperature
counts
  temperature = (temp_counts * 200 / 16777215) - 50; // calculate temperature in deg c
  percentage = (press_counts / 16777215) * 100; // calculate pressure as percentage of full
scale
  //calculation of pressure value according to equation 2 of datasheet
  pressure = ((press_counts - outputmin) * (pmax - pmin)) / (outputmax - outputmin) + pmin;
  dtostrf(press_counts, 4, 1, cBuff); // dtostrf() may be the function you need if you have a
floating point value that you need to convert to a string.
  dtostrf(percentage, 4, 3, percBuff);
  dtostrf(pressure, 4, 3, pBuff);
  dtostrf(temperature, 4, 3, tBuff);
  /*
    The below code prints the raw data as well as the processed data
    Data format : Status Register, 24-bit Sensor Data, Digital Counts, percentage of full
scale
    pressure,
    pressure output, temperature
  */
  sprintf(printBuffer, " % x\t % 2x % 2x % 2x\t % s\t % s\t % s\t % s \n", data[0], data[1],
data[2], data[3], cBuff, percBuff, pBuff, tBuff);
  Serial.print(pBuff);
  Serial.print(" KPa; ");
  Serial.print(percBuff);
  Serial.println(" % ");
  delay(10);
}
```

Figure 10 : Code de récupération des données en I2C

On peut voir que les principales instructions réalisées sont le traitement des données lues, nous avons quand même la récupération des 7 octets de données par :

- `Wire.read();` : Cette fonction est la lecture des données demandées à l'esclave. Cette fonction est limitée à 1 octet. C'est pour cela que l'on fait une boucle de 7 afin de récupérer tous les octets.

Comme dit précédemment, la suite concerne le traitement des données lues pour les rendre exploitable. Nous allons nous intéresser au traitement des données de la pression.

- `press_counts = data[3] + data[2] * 256 + data[1] * 65536;` : Cette formule permet de mettre à l'échelle les données de pression lues.
- `percentage = (press_counts / 16777215) * 100;` : Cette formule permet de calculer la pression en pourcentage de la pleine échelle d'échantillonnage
- `pressure = ((press_counts - outputmin) * (pmax - pmin)) / (outputmax - outputmin) + pmin;` : Cette formule permet de correspondre les données lues sur une unité de pression, nous pouvons donc avoir une pression avec une unité choisie, dans notre cas on met la pression en mbar. Pour modifier l'unité, il faut modifier les variables pmin et pmax. Cette formule est retrouvable sur la datasheet.

Toutes ces étapes permettent de récupérer la pression de l'air mesurée par le capteur. Le programme complet est en Annexe du livrable.

Mise en place dans un Timer

Dans ce cas, on met la fonction dans un Timer pour la mettre en tâche de fond et de pouvoir gérer le temps d'activation de celle-ci. Les Timer/Counter ont généralement 4 modes d'utilisation : IC (Input Compare), OC (Output Compare), PWM et Pulse counter. C'est le mode OC qui nous intéresse dans notre cas puisque celui-ci va activer la fonction « Pressure » vue précédemment périodiquement (la période est définie par nous).

Pour utiliser le timer/counter sur le STM32F411RE, nous allons prendre la librairie « hardwareTimer » pour avoir des fonctions configurant le timer/counter. Voici comment se présente le code :

```
#define PRESSURE_PERIOD 200000 // 5Hz
#define TIM_CHANNEL_PRESSURE 2 //Channel 2 de la pin D3
#define PRESSURE_SDA PB3
#define PRESSURE_SCL PB10

#include<Wire.h>
#include "HardwareTimer.h"

HardwareTimer* hardwareTimer3;  // Timer Pressure command


void setup() {

  Serial.begin(115200);


  while (!Serial) {
    delay(10);
  }

  Wire.setSDA(PRESSURE_SDA);
  Wire.setSCL(PRESSURE_SCL);
  Wire.begin();
  Pressure_Timer();
  sprintf(printBuffer, "\nStatus Register, 24 - bit Sensor data, Digital Pressure Counts,\
Percentage of full scale pressure, Pressure Output, Temperature\n");
  Serial.println(printBuffer);

}

void loop() {

}


void Pressure_Timer() {
  hardwareTimer3 = new HardwareTimer(TIM2); //Sélectionne le timer 4 pour la pin D10
  hardwareTimer3->setOverflow(PRESSURE_PERIOD, MICROSEC_FORMAT); //Définit la
période/fréquence de la PWM
  hardwareTimer3->refresh(); //réninitialise le timer
  hardwareTimer3->setMode(TIM_CHANNEL_PRESSURE, TIMER_OUTPUT_COMPARE); //Permet de mettre le
mode PWM sur le timer (Autres modes : Input Capture/Ouput Capture/One-pulse
  // Set PPM width to 1ms
  hardwareTimer3->setCaptureCompare(TIM_CHANNEL_PRESSURE, 0 , PERCENT_COMPARE_FORMAT);
//Définit le rapport cyclique de la PWM
  hardwareTimer3->attachInterrupt(TIM_CHANNEL_PRESSURE, Pressure); //Active la fonction à
chaque interruption
  hardwareTimer3->resume(); //Lance le timer
}
```

Figure 11 : Code d'intégration d'un timer

Avant de commencer, il est conseillé d'aller regarder la librairie « hardwareTimer.h » afin de voir toutes les méthodes de la librairie et les différents attributs (Voir Annexe 1). Nous allons décrire les différentes fonctions pour connaître leurs paramètres et rôles :

- HardwareTimer* hardwareTimer3 : Cette commande permet de faire appel à la librairie hardwareTimer.h à travers un objet. C'est grâce à cet objet que l'on va pouvoir utiliser les fonctions de la librairie « hardwareTimer »

- hardwareTimer3 = new HardwareTimer(TIM2) : Cette commande permet d'instancier l'objet HardwareTimer en sélectionnant TIM2

- hardwareTimer3->setOverflow(PRESSURE_PERIOD, MICROSEC_FORMAT) : Cette commande permet de configurer la période du timer. Dans notre cas, ça va définir la période d'activation de la fonction à 5 Hz.

- hardwareTimer3->refresh() : Cette commande met à jour tous les registres du timer

- hardwareTimer3->setMode(TIM_CHANNEL_PRESSURE,TIMER_OUTPUT_COMPARE) : Cette commande permet de sélectionner le channel du timer et le mode du timer (OC dans notre cas)

- hardwareTimer3->setCaptureCompare(TIM_CHANNEL_PRESSURE,0, PERCENT_COMPARE_FORMAT) : Cette commande permet de sélectionner le channel et le début du compteur.

- hardwareTimer3->resume() : Cette commande permet de lancer le timer et d'activer le channel du timer sélectionné.

En intégrant ce programme, notre fonction récupérant et traitant les données du capteur de pression s'activera tous les 0,2s. Notre programme du capteur de pression est donc fini.

Annexe

Choix technologique du capteur de pression

Solution trouvée	Technologie	Avantage	Inconvénient	Caractéristiques	Prix	Choix
Capteur de pression différentiel	Sensirion SDP8 ou SEN0343 (prototypage avec Arduino)	Très fiable Très stable Précis Plage de mesure suffissante	Assez encombrant Communication I2C	Supply voltage : 3,3V Range : -500 to + 500 Pa Communication I2C Accuracy : 0,1 Pa Répétitivité : 0,05 Pa	36€ ou 44€	
	Honeywell APB2 series ABP2DDAN005NDAA5XX	Fiable Précis Plage de mesure suffissante boîtier SIP		Range : ± 1 kPa Supply Voltage ; 3,3V Sortie Analogique	42 €	
	MPXV7002DP	Fiable Stable Précis Plage de mesure suffissante	boîtier SOIC	Supply Voltage : 5V Accuracy : ± 2,5% Range : ± 2kPa Sortie Analogique	32 €	
	MPX5010DP	Fiable Stable Précis boîtier SIP	Plage de mesure positive	Supply Voltage 5V Accuracy : 5% Range : 0 à 10 kPa Sortie Analogique	33 €	
	Sensirion SDP1000	Fiable Précis boîtier PPS Sortie analogique	prix élevé	Supply Voltage : 5V Accuracy : ± 1,5% Range : -20 à 500Pa Sortie Analogique (0 ,25 à 4V)	89 €	
	ABP2MRRT060MD2A3XX	Fiable Stable Précis boîtier SMD-6 Sortie 3,3V Plage de mesure de suffisante	Communication I2C	Supply Voltage : 3,3V Accuracy : ± 0.25 % Range : -60 à 60 mbar 204 Samples/s Communication I2C	28 €	X

Figure 12 : Choix technologique

Ce tableau a été créé afin de mettre en valeur la technologie de chaque capteur trouvé et de voir leurs avantages et inconvénients pour notre système. On compare aussi les caractéristiques importantes des capteurs pour voir lequel est le plus adapté à notre cas d'utilisation. On peut ajouter le prix et la disponibilité (non présent sur le tableau) pour savoir si le capteur répond au budget et s'il est disponible chez des fournisseurs de composants électronique

Programme du capteur de pression

```
#define PRESSURE_PERIOD 200000 // 5Hz
#define TIM_CHANNEL_PRESSURE 2 //Channel 2 de la pin D3
#define PRESSURE_SDA PB3
#define PRESSURE_SCL PB10

#include<Arduino.h>
#include<Wire.h>
#include "HardwareTimer.h"

HardwareTimer* hardwareTimer3;  // Timer Pressure command

uint8_t id = 0x28; // i2c address
uint8_t data[7]; // holds output data
uint8_t cmd[3] = {0xAA, 0x00, 0x00}; // command to be sent
double press_counts = 0; // digital pressure reading [counts]
double temp_counts = 0; // digital temperature reading [counts]
double pressure = 0; // pressure reading [bar, psi, kPa, etc.]
double temperature = 0; // temperature reading in deg C
double outputmax = 15099494; // output at maximum pressure [counts]
double outputmin = 1677722; // output at minimum pressure [counts]
double pmax = 60; // maximum value of pressure range [bar, psi, kPa, etc.]
double pmin = -60; // minimum value of pressure range [bar, psi, kPa, etc.]
double percentage = 0; // holds percentage of full scale data
char printBuffer[200], cBuff[20], percBuff[20], pBuff[20], tBuff[20];

void setup() {

  Serial.begin(115200);


  while (!Serial) {
    delay(10);
  }

  Wire.setSDA(PRESSURE_SDA);
  Wire.setSCL(PRESSURE_SCL);
  Wire.begin();
  Pressure_Timer();
  sprintf(printBuffer, "\nStatus Register, 24 - bit Sensor data, Digital
Pressure Counts,\
Percentage of full scale pressure, Pressure Output, Temperature\n");
  Serial.println(printBuffer);

}

void loop() {

}

void Pressure() {
  Wire.beginTransmission(id);
  int stat = Wire.write (cmd, 3); // write command to the sensor
  stat |= Wire.endTransmission();
  delay(10);
  Wire.requestFrom(id, 7); // read back Sensor data 7 bytes
  int i = 0;
  for (i = 0; i < 7; i++) {
    data [i] = Wire.read();
  }
```

```
  press_counts = data[3] + data[2] * 256 + data[1] * 65536; // calculate
digital pressure counts
  temp_counts = data[6] + data[5] * 256 + data[4] * 65536; // calculate
digital temperature counts
  temperature = (temp_counts * 200 / 16777215) - 50; // calculate
temperature in deg c
  percentage = (press_counts / 16777215) * 100; // calculate pressure as
percentage of full scale
  //calculation of pressure value according to equation 2 of datasheet
  pressure = ((press_counts - outputmin) * (pmax - pmin)) / (outputmax -
outputmin) + pmin;
  dtostrf(press_counts, 4, 1, cBuff); // dtostrf() may be the function you
need if you have a floating point value that you need to convert to a
string.
  dtostrf(percentage, 4, 3, percBuff);
  dtostrf(pressure, 4, 3, pBuff);
  dtostrf(temperature, 4, 3, tBuff);
  /*
    The below code prints the raw data as well as the processed data
    Data format : Status Register, 24-bit Sensor Data, Digital Counts,
percentage of full scale
    pressure,
    pressure output, temperature
  */
  sprintf(printBuffer, " % x\t % 2x % 2x % 2x\t % s\t % s\t % s\t % s \n",
data[0], data[1], data[2], data[3], cBuff, percBuff, pBuff, tBuff);
  Serial.print(pBuff);
  Serial.print(" KPa; ");
  Serial.print(percBuff);
  Serial.println(" % ");
  delay(10);
}

void Pressure_Timer() {
  hardwareTimer3 = new HardwareTimer(TIM2); //Sélectionne le timer 4 pour
la pin D10
  hardwareTimer3->setOverflow(PRESSURE_PERIOD, MICROSEC_FORMAT); //Définit
la période/fréquence de la PWM
  hardwareTimer3->refresh(); //réninitialise le timer
  hardwareTimer3->setMode(TIM_CHANNEL_PRESSURE, TIMER_OUTPUT_COMPARE);
//Permet de mettre le mode PWM sur le timer (Autres modes : Input
Capture/Ouput Capture/One-pulse
  // Set PPM width to 1ms
  hardwareTimer3->setCaptureCompare(TIM_CHANNEL_PRESSURE, 0 ,
PERCENT_COMPARE_FORMAT); //Définit le rapport cyclique de la PWM
  hardwareTimer3->attachInterrupt(TIM_CHANNEL_PRESSURE, Pressure); //Active
la fonction à chaque interruption
  hardwareTimer3->resume(); //Lance le timer
}
```