MAKERS FOR LIFE

MakAir

# Mesure avec un Débitmètre

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 débitmètre intégré dans le MakAir. La première partie du document est une analyse des caractéristiques du débitmètre. La deuxième partie portera sur la récupération des données de la mesure du débit d'air.

# Table des matières

Table des figures ..... 2
C'est quoi un débitmètre ? ..... 2
La communication I2C ..... 3
Récupération de la mesure du débit d'air ..... 4
Câblage du capteur ..... 5
Initialisation du capteur ..... 6
Récupération des données ..... 7
Mise en place dans le loop() ..... 8
Mise en place d'un Timer ..... 9
Annexe ..... 11

# Table des figures

Figure 1 : Caractéristiques du Honeywell Zephyr ..... 3
Figure 2 : Start/Stop bus I2C ..... 4
Figure 3 : Bus I2C du Honeywell Zephyr ..... 4
Figure 4 : Câblage du Honeywell Zephyr ..... 5
Figure 5 : Prototypage du débitmètre ..... 5
Figure 6 : Code d'initialisation du capteur ..... 6
Figure 7 : Code de récupération des données en I2C ..... 8
Figure 8 : Code d'intégration d'un timer ..... 9

# C'est quoi un débitmètre ?

Un débitmètre est un instrument utilisé pour mesurer le débit linéaire, non linéaire, massique ou volumétrique d'un liquide ou d'un gaz. Dans notre cas, nous voulons mesurer le débit d'air, donc nous allons nous porter vers des débitmètres plus adaptés dans ce domaine. Pour sélectionner un débitmètre, nous devons nous concentrer sur ces différents paramètres :

- **Plage de mesure du capteur** : c'est un paramètre important puisqu'il faut un capteur qui puisse mesurer toute la plage de débit d'air que le système va produire. Il y a différentes unités dans les fiches technique des débitmètres (SLPM, l/min, $Cm^3$/min).
- **La pression nominale** : c'est un paramètre important particulièrement dans notre cas puisque le système va produire de l'air avec une pression non négligeable, donc il faut que le débitmètre puisse supporter cette pression
- **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
- **La section d'installation** : le débitmètre comporte un trou pour insérer un tuyau qui va transporter l'air envoyée. Il est important de prendre en compte quel tuyau nous allons utiliser et si le débitmètre est adapté à la section du tuyau pour pouvoir l'installer. Une différence trop importante de section entre le tuyau et le capteur pourrait créer une perte de charge, voire pire : des fortes turbulences sur l'écoulement d'air qui perturberaient la mesure
- **La tension nominale d'alimentation** : 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 débitmètre pour qu'il fonctionne correctement.

- **Interface de communication** : le dernier paramètre important, c'est la communication entre le microcontrôleur et le débitmètre 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.
- **La technologie du capteur** : fil chaud VS capteur différentiel

Pour la réalisation du respirateur PPC, nous restons sur le débitmètre d'inspiration du MakAir qui est le Honeywell Zephyr. C'est un débitmètre adapté au domaine médical et très performant pour la mesure d'un débit d'air. On peut retrouver ces différentes caractéristiques ci-dessous :

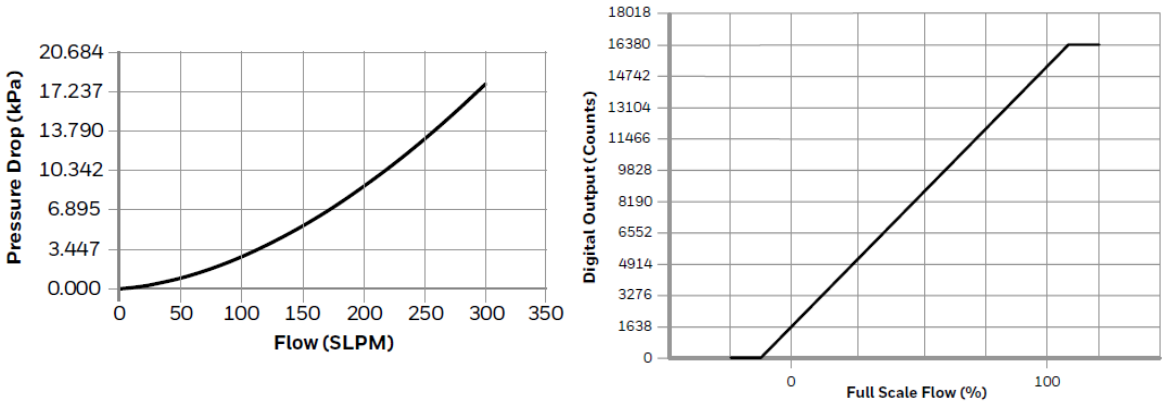



*Figure 1 : Caractéristiques du Honeywell Zephyr*

| | |
|---|---|
| **Supply voltage** | 3,3 Vdc |
| **Plage de débit** | 0 - 200 SLPM |
| **Résolution** | 0,029 SLPM |
| **Pression max** | 60 Psi |
| *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 2 signaux :

- Signal de donnée : SDA
- Signal d'horloge : SCL

Chaque protocole de communication communique grâce à un 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

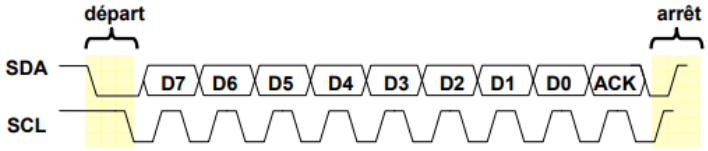



*Figure 2 : Start/Stop bus I2C*

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 exemple du débitmètre du MakAir :

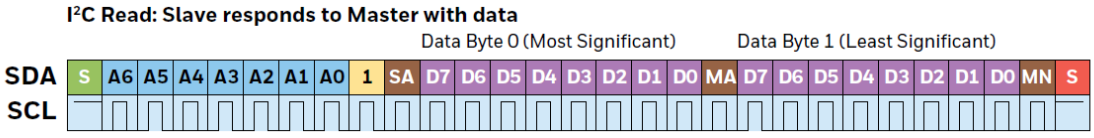



*Figure 3 : Bus I2C du Honeywell Zephyr*

On peut voir que le bus commence par l'adresse du capteur pour le maître (microcontrôleur), ensuite un premier octet de donnée est envoyé et enfin un deuxième octet est envoyé. Le maître reçoit une donnée sur 16 bits. Ce bus comporte quelques particularités :

- SA : Mise à 0 de SDA par l'esclave
- MA : Mise à 0 de SDA par le maître
- MN : Mise à 1 de SDA par le maître pour l'arrêt de la trame

## Récupération de la mesure du débit d'air

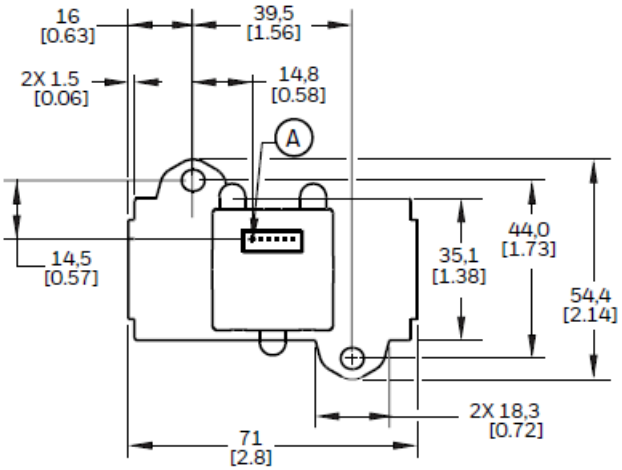
Nous allons passer à la partie réalisation, nous allons voir comment récupérer la mesure du débit d'air grâce à un programme.

Pour récupérer les données du débitmètre 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 logiciellement 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). Nous allons mettre en place ces 2 solutions.

## Câblage du capteur

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




Ⓐ Pin 1.

| Pin 1 | Pin 2 | Pin 3 | Pin 4 | Pin 5 | Pin 6 |
|---|---|---|---|---|---|
| NC | SCL | VVDD | ground | SDA | NC |

*Figure 4 : Câblage du Honeywell Zephyr*

Il suffit donc de 4 fils pour câbler le débitmètre. On peut l'ajouter au schéma de prototypage :

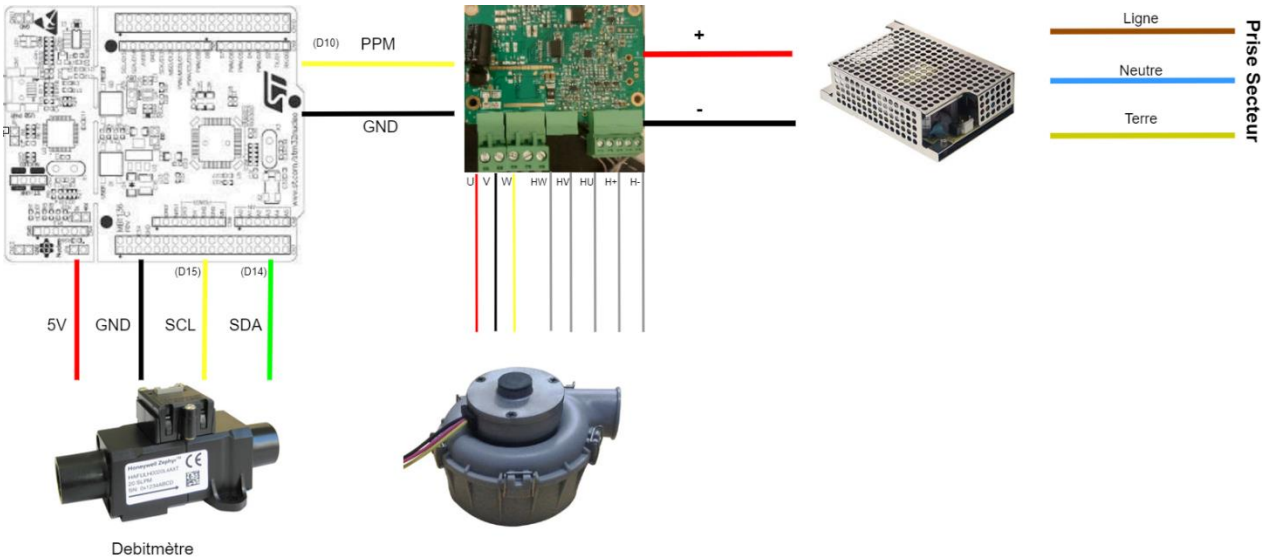



*Figure 5 : Prototypage du débitmètre*

Maintenant le câblage réalisé, on peut passer à l'initialisation du débitmètre

## 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 bien précis à mettre en place pour que la récupération des données se fasse 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 :

```
////Constantes du programme////
#define MFM_HONEYWELL_HAF_I2C_ADDRESS 0x49
#define PIN_I2C_SDA PB9
#define PIN_I2C_SCL PB8

#include <Wire.h>

////Variables Débitmètre////
uint8_t slpm = 0;
int debit[2];
int32_t mfmInspiratoryLastValue = 0;
int32_t mfmInspiratoryAirFlow = 0;
union {
    uint16_t i;
    int16_t si;
    unsigned char c[2];
} mfmLastData;

void setup() {
    Serial.begin(115200);
    Wire.setSDA(PIN_I2C_SDA);
    Wire.setSCL(PIN_I2C_SCL);
    Wire.begin();
    sensor_Init();
    PWM_Blower();
}

void sensor_Init(){ //lire le numéro de série

    Wire.begin();
    Wire.beginTransmission(MFM_HONEYWELL_HAF_I2C_ADDRESS);
    Wire.write(0x02);  // Force reset
    uint8_t txOk = Wire.endTransmission();
    Wire.end();
    delay(30); // délai necessaire pour séparer le cycle d'écriture et de lecture

    uint32_t sn = 0;
    Wire.begin();
    Wire.beginTransmission(MFM_HONEYWELL_HAF_I2C_ADDRESS);
    uint8_t rxcount = Wire.requestFrom(MFM_HONEYWELL_HAF_I2C_ADDRESS, 2);
    sn = Wire.read();
    sn <<= 8;
    sn |= Wire.read();  // first transmission is serial number register 0
    sn <<= 8;
    delay(2);  // if you do not wait, sensor will send again register 0
    rxcount += Wire.requestFrom(MFM_HONEYWELL_HAF_I2C_ADDRESS, 2);
    sn |= Wire.read();
    sn <<= 8;
    sn |= Wire.read();  // second transmission is serial number register 1
    Serial.print("Numéro de série : ");
    Serial.println(sn);
    Wire.end();
}
```

*Figure 6 : Code d'initialisation du capteur*

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

- `Wire.setSDA(PIN_I2C_SDA); / Wire.setSCL(PIN_I2C_SCL);` : Ces fonctions permettent de configurer les signaux SDA et SLC sur les pins PB9 et PB8
- `Wire.begin();` : Cette fonction permet de débuter une communication I2C
- `Wire.beginTransmission(MFM_HONEYWELL_HAF_I2C_ADDRESS);` : Cette fonction permet de faire une requête de transmission à l'esclave
- `Wire.write(0x02);` : Cette fonction permet au maître d'écrire à l'esclave, ici l'écriture d'un 0x02 permet de forcer le reset du capteur
- `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.end();` : Cette fonction permet de mettre fin à la communication I2C entre le maître et l'esclave.
- `Wire.requestFrom(MFM_HONEYWELL_HAF_I2C_ADDRESS, 2);` : Cette fonction est une requête du maître pour que l'esclave lui envoie 2 octets de données (les 2 octets que l'on a pu précédemment concernant le Honeywell Zephyr). Cette fonction permet de définir la taille du bus I2C qui va être transmis
- `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 un décalage vers la gauche (`sn <<= 8`) afin d'éviter d'écraser l'octet reçu par le prochain octet qui va être reçu. Nous recommençons l'opération de requête et de lecture une deuxième fois pour récupérer le deuxième registre du capteur afin d'avoir toutes les données du débitmètre.

On récupère à la fin de cette initialisation du capteur : son numéro de série qui est lui est propre lors de la communication I2C. Maintenant que les étapes d'initialisation du débitmètre 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 débitmètre et va les traiter pour les rendre exploitables. Elle est sous 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 le loop()

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

```
void loop() {
  airFlow();

}

void airFlow(){ //lire le débit
     Wire.begin();
     uint8_t readCount = Wire.requestFrom(MFM_HONEYWELL_HAF_I2C_ADDRESS, 2);
     mfmLastData.c[0] = Wire.read();
     mfmLastData.c[1] = Wire.read();
            // Wire.endTransmission() send a new write order followed by a stop. Useless and
the
            // sensor often nack it.
     Wire.end();

     mfmInspiratoryLastValue = (uint32_t)(mfmLastData.c[1] & 0xFFu); // masquage
0b00000000000000000000000011111111 & données du registre 1 (falcutatif)
     mfmInspiratoryLastValue |= ((uint32_t)mfmLastData.c[0]) << 8) & 0x0000FF00u; // masquage
0b00000000000000001111111100000000 & données du registre 0 (falcutatif)

            // Theorical formula: Flow(slpm) = 200*((rawvalue/16384)-0.1)/0.8
            // float implementation, 1 liter per minute unit
     mfmInspiratoryAirFlow = 200 * (((uint32_t)mfmInspiratoryLastValue / 16384.0) - 0.1) /
0.8;
            // (Output value in SLPM)

            // fixed float implementation, 1 milliliter per minute unit
     //mfmInspiratoryAirFlow = (((10 * mfmInspiratoryLastValue) - 16384) * 1526);
     Serial.print("debit d'air : ");
     Serial.print(mfmInspiratoryAirFlow);
     Serial.println(" SLPM");
}
```

*Figure 7 : Code de récupération des données en I2C*

Nous retrouvons les fonctions « Wire » expliquaient précédemment, mais 2 formules sont mises en place et ont chacun leur utilité :

- `mfmInspiratoryLastValue = (uint32_t)(mfmLastData.c[1] & 0xFFu);` : Cette formule est complexe à première vue mais est au final assez simple à comprendre. Le but ici est de faire un masquage (un ET avec 32 bits à 1 sur la données récupérée) sur une des cellules du tableau de données « mfmLastData.C », cela permet d'éviter de récupérer des données incohérentes

- `mfmInspiratoryAirFlow = 200 * (((uint32_t)mfmInspiratoryLastValue / 16384.0) - 0.1) / 0.8;` : Cette formule est donnée dans la datasheet du capteur Honeywell Zephyr, elle permet de convertir la donnée reçue en une valeur de débit d'air (en SLPM).

Toutes ces étapes permettent de récupérer le débit d'air mesurée par le capteur.

## Mise en place d'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 « airflow » 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 ESC_PPM_PERIOD 20000 // 50Hz
#define AIRFLOW_PERIOD 200000 // 5Hz
#define MIN_BLOWER_SPEED 300u
#define MAX_BLOWER_SPEED 1800u
#define DEFAULT_BLOWER_SPEED 900u
#define PIN_ESC_BLOWER  PB6  // D10 / TIM4_CH1
#define TIM_CHANNEL_ESC_BLOWER 1 //Channel 1 de la pin D10
#define TIM_CHANNEL_AIRFLOW 4 //Channel 4 de la pin D15
#define BlowerSpeed2MicroSeconds(value) map(value, 0, 1800, 1000, 1950) //Intervalle [1ms à
1,95 ms] sur la variation de vitesse du blower
#define MFM_HONEYWELL_HAF_I2C_ADDRESS 0x49
#define PIN_I2C_SDA PB9
#define PIN_I2C_SCL PB8

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

HardwareTimer* hardwareTimer1;  // Timer Blower command
HardwareTimer* hardwareTimer2;  // Timer Debitmetre command

void setup() {
  Serial.begin(115200);
  Wire.setSDA(PIN_I2C_SDA);
  Wire.setSCL(PIN_I2C_SCL);
  Wire.begin();
  sensor_Init();
  PWM_Blower();
  Airflow_Timer();
}

void loop() {

}

void Airflow_Timer() {
  hardwareTimer2 = new HardwareTimer(TIM1); //Sélectionne le timer 4 pour la pin D10
  hardwareTimer2->setOverflow(AIRFLOW_PERIOD, MICROSEC_FORMAT); //Définit la période/fréquence
de la PWM
  hardwareTimer2->refresh(); //réninitialise le timer
  hardwareTimer2->setMode(TIM_CHANNEL_AIRFLOW, 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
  hardwareTimer2->setCaptureCompare(TIM_CHANNEL_AIRFLOW, 0 , PERCENT_COMPARE_FORMAT);
//Définit le rapport cyclique de la PWM
  hardwareTimer2->attachInterrupt(TIM_CHANNEL_AIRFLOW, airFlow); //Active la fonction à chaque
interruption
  hardwareTimer2->resume(); //Lance le timer
}
```

*Figure 8 : 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* hardwareTimer2 : 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 »

- hardwareTimer2 = new HardwareTimer(TIM1) : Cette commande permet d'instancier l'objet HardwareTimer en sélectionnant TIM1

- hardwareTimer2->setOverflow(AIRFLOW_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.

- hardwareTimer2->refresh() : Cette commande met à jour tous les registres du timer

- hardwareTimer2->setMode(TIM_CHANNEL_AIRFLOW,TIMER_OUTPUT_COMPARE) : Cette commande permet de sélectionner le channel du timer et le mode du timer (OC dans notre cas)

- hardwareTimer2->setCaptureCompare(TIM_CHANNEL_AIRFLOW,0, PERCENT_COMPARE_FORMAT) : Cette commande permet de sélectionner le channel et le début du compteur.

- hardwareTimer2->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 débitmètre s'activera tous les 0,2s. Notre programme du débitmètre est donc fini.

# Annexe

```
#define ESC_PPM_PERIOD 20000 // 50Hz
#define AIRFLOW_PERIOD 200000 // 5Hz
#define MIN_BLOWER_SPEED 300u
#define MAX_BLOWER_SPEED 1800u
#define DEFAULT_BLOWER_SPEED 900u
#define PIN_ESC_BLOWER  PB6  // D10 / TIM4_CH1
#define TIM_CHANNEL_ESC_BLOWER 1 //Channel 1 de la pin D10
#define TIM_CHANNEL_AIRFLOW 4 //Channel 4 de la pin D15
#define BlowerSpeed2MicroSeconds(value) map(value, 0, 1800, 1000, 1950)
//Intervalle [1ms à 1,95 ms] sur la variation de vitesse du blower
#define MFM_HONEYWELL_HAF_I2C_ADDRESS 0x49
#define PIN_I2C_SDA PB9
#define PIN_I2C_SCL PB8

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

HardwareTimer* hardwareTimer1;  // Timer Blower command
HardwareTimer* hardwareTimer2;  // Timer Debitmetre command

////Variables Blower////
static int speedBlower = MAX_BLOWER_SPEED;

////Variables Débitmètre////
uint8_t slpm = 0;
int debit[2];
int32_t mfmInspiratoryLastValue = 0;
int32_t mfmInspiratoryAirFlow = 0;
union {
  uint16_t i;
  int16_t si;
  unsigned char c[2];
} mfmLastData;

void setup() {
  Serial.begin(115200);
  Wire.setSDA(PIN_I2C_SDA);
  Wire.setSCL(PIN_I2C_SCL);
  Wire.begin();
  sensor_Init();
  PWM_Blower();
  Airflow_Timer();
}

void loop() {

}

void airFlow() { //lire le débit
  Wire.begin();
  uint8_t readCount = Wire.requestFrom(MFM_HONEYWELL_HAF_I2C_ADDRESS, 2);
  mfmLastData.c[0] = Wire.read();
  mfmLastData.c[1] = Wire.read();
  // Wire.endTransmission() send a new write order followed by a stop.
Useless and the
  // sensor often nack it.
  Wire.end();
```

```
  mfmInspiratoryLastValue = (uint32_t)(mfmLastData.c[1] & 0xFFu); //
masquage 0b00000000000000000000000011111111 & données du registre 1
(falcutatif)
  mfmInspiratoryLastValue |= (((uint32_t)mfmLastData.c[0]) << 8) &
0x0000FF00u; // masquage 0b00000000000000001111111100000000 & données du
registre 0 (falcutatif)

  // Theorical formula: Flow(slpm) = 200*((rawvalue/16384)-0.1)/0.8
  // float implementation, 1 liter per minute unit
  mfmInspiratoryAirFlow = 200 * (((uint32_t)mfmInspiratoryLastValue /
16384.0) - 0.1) / 0.8;
  // (Output value in SLPM)

  // fixed float implementation, 1 milliliter per minute unit
  //mfmInspiratoryAirFlow = (((10 * mfmInspiratoryLastValue) - 16384) *
1526);
  Serial.print("debit d'air : ");
  Serial.print(mfmInspiratoryAirFlow);
  Serial.println(" SLPM");
}
void sensor_Init() { //lire le numéro de série

  Wire.begin();
  Wire.beginTransmission(MFM_HONEYWELL_HAF_I2C_ADDRESS);
  Wire.write(0x02);  // Force reset
  uint8_t txOk = Wire.endTransmission();
  Wire.end();
  delay(30);

  uint32_t sn = 0;
  Wire.begin();
  Wire.beginTransmission(MFM_HONEYWELL_HAF_I2C_ADDRESS);
  uint8_t rxcount = Wire.requestFrom(MFM_HONEYWELL_HAF_I2C_ADDRESS, 2);
  sn = Wire.read();
  sn <<= 8;
  sn |= Wire.read();  // first transmission is serial number register 0
  sn <<= 8;
  delay(2);  // if you do not wait, sensor will send again register 0
  rxcount += Wire.requestFrom(MFM_HONEYWELL_HAF_I2C_ADDRESS, 2);
  sn |= Wire.read();
  sn <<= 8;
  sn |= Wire.read();  // second transmission is serial number register 1
  Serial.print("Numéro de série : ");
  Serial.println(sn);
  Wire.end();
}

void PWM_Blower() {
  pinMode(PIN_ESC_BLOWER, OUTPUT);
  hardwareTimer1 = new HardwareTimer(TIM4); //Sélectionne le timer 4 pour
la pin D10
  hardwareTimer1->setOverflow(ESC_PPM_PERIOD, MICROSEC_FORMAT); //Définit
la période/fréquence de la PWM
  //hardwareTimer1->setOverflow(50, HERTZ_FORMAT);
  hardwareTimer1->refresh(); //réninitialise le timer
  hardwareTimer1->setMode(TIM_CHANNEL_ESC_BLOWER,
TIMER_OUTPUT_COMPARE_PWM1, PIN_ESC_BLOWER); //Permet de mettre le mode PWM
sur le timer (Autres modes : Input Capture/Ouput Capture/One-pulse
  // Set PPM width to 1ms
  hardwareTimer1->setCaptureCompare(TIM_CHANNEL_ESC_BLOWER,
BlowerSpeed2MicroSeconds(DEFAULT_BLOWER_SPEED), MICROSEC_COMPARE_FORMAT);
```

```
  //hardwareTimer1->setCaptureCompare(TIM_CHANNEL_ESC_BLOWER, 10 ,
PERCENT_COMPARE_FORMAT); //Définit le rapport cyclique de la PWM
  hardwareTimer1->resume(); //Lance le timer
}

void Airflow_Timer() {
  hardwareTimer2 = new HardwareTimer(TIM1); //Sélectionne le timer 4 pour
la pin D10
  hardwareTimer2->setOverflow(AIRFLOW_PERIOD, MICROSEC_FORMAT); //Définit
la période/fréquence de la PWM
  hardwareTimer2->refresh(); //réninitialise le timer
  hardwareTimer2->setMode(TIM_CHANNEL_AIRFLOW, 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
  hardwareTimer2->setCaptureCompare(TIM_CHANNEL_AIRFLOW, 0 ,
PERCENT_COMPARE_FORMAT); //Définit le rapport cyclique de la PWM
  hardwareTimer2->attachInterrupt(TIM_CHANNEL_AIRFLOW, airFlow); //Active
la fonction à chaque interruption
  hardwareTimer2->resume(); //Lance le timer
}
```