

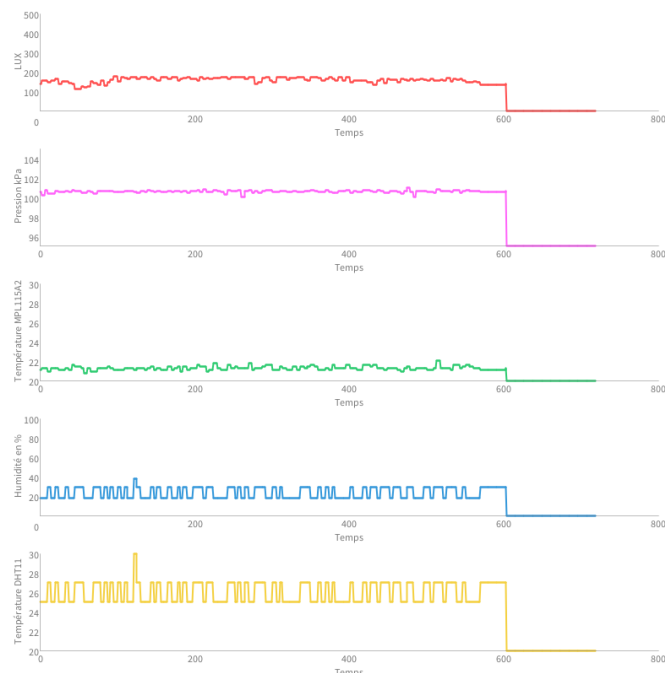
Bordeaux INP – ENSEIRB-MATMECA

Filière Systèmes Électroniques Embarqués

## ES9EN326 Capteurs pour l'embarqué

Station météo sans fil

Adrien CLAIN - Clément SAVARY



# Table des matières

I.	Présentation global du projet.....	3
1)	Objectifs du projet.....	3
2)	Capteurs mis à disposition .....	3
3)	Arduino, xBee et processing.....	4
II.	Interaction avec les capteurs .....	5
1)	Capteur de Luminosité TEMT6000 .....	5
2)	Capteur de pression et de température MPL115A2 .....	8
a.	Mise en place du capteur MPL115A2.....	8
b.	Communication i2C .....	9
3)	Capteur d'humidité et de température DHT11.....	18
a.	Mise en place du capteur DHT11 .....	18
b.	Communication série bidirectionnelle .....	19
III.	Emission des données .....	24
IV.	Traitement des données .....	25
1)	Récupération des données avec Processing .....	25
2)	Affichage des données sur L'IHM .....	26
V.	Annexes .....	28

## I. Présentation global du projet

### 1) Objectifs du projet

L'objectif de ce projet SEE est de réaliser une station météo sans fil. Elle sera composée d'une partie émission distante munie de différents capteurs, et d'une partie réception locale reliée à un ordinateur. Nous devons être en mesure de présenter un système capable d'afficher sur l'écran d'un ordinateur les valeurs instantanées des capteurs embarqués.

L'architecture retenue pour cette station est présentée en Figure 1.

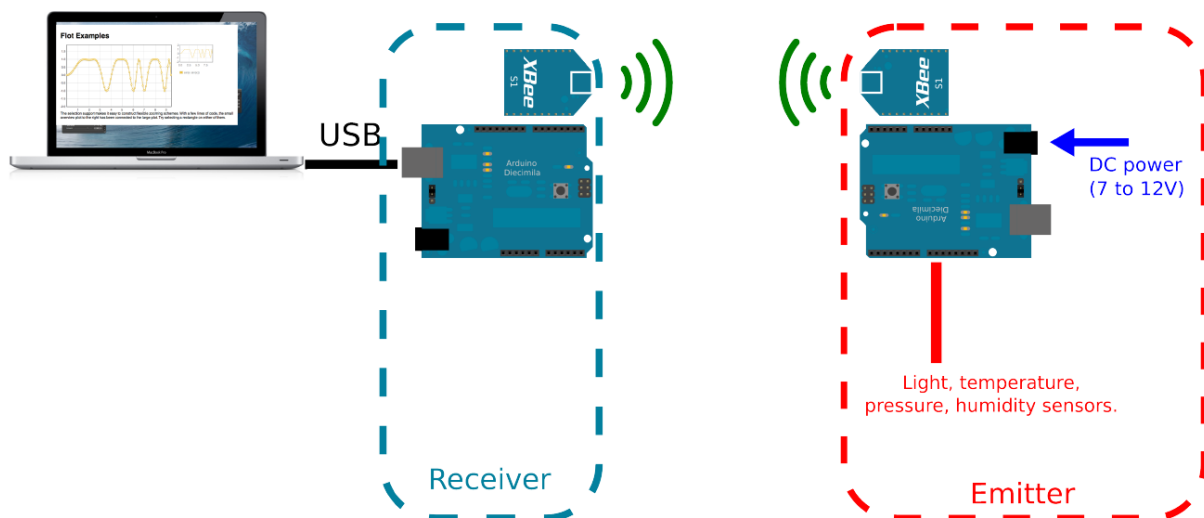


Figure 1 : Architecture de la station météo

Chaque module se compose :

- D'une carte Arduino UNO avec microcontrôleur ATMEGA328P.
- D'un shield Wireless Proto ou SD.
- D'un module xBee.

Le module récepteur doit être relié au PC par USB et tirera son alimentation directement par le port. Le module émetteur sera alimenté par une alim

### 2) Capteurs mis à disposition

Pour prélever les différentes mesures instantanées de pression, température et d'humidité, nous avons à disposition 3 capteurs :

- Un capteur de luminosité Vishay TEMT6000.
- Un capteur de pression Freescale MPL115A2, donnant également une mesure de température.
- Un capteur d'humidité/température Sensirion DHT11.

L'étude des datasheets va permettre de connaître le protocole de communication des composants, et l'exploitation des données reçues.

### 3) Arduino, xBee et processing

Nous allons programmer des cartes Arduino UNO afin d'interagir avec les différents capteurs. La programmation se fera par le biais de l'environnement Arduino (langage proche du C). Les entrées et sorties de cette carte Arduino UNO sont définies ci-dessous.

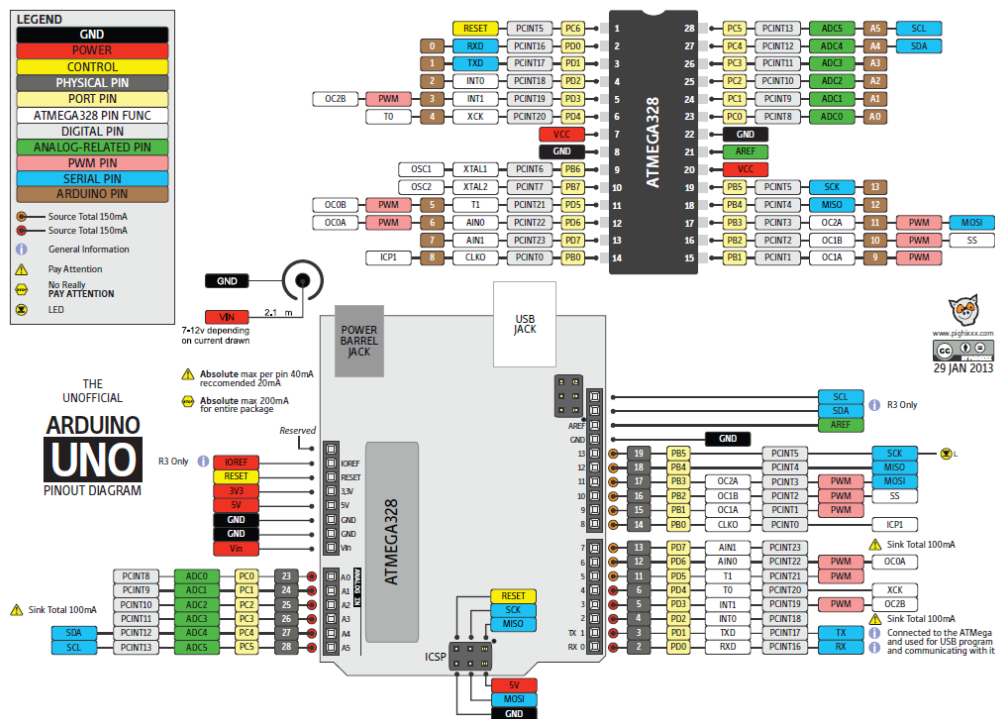


Figure 2 : Brochage de l'Arduino UNO

La communication sans fil entre les deux cartes Arduino UNO mises à votre disposition va s'effectuer grâce à des modules XBee de chez Digi International. Ces modules radio sont basés sur le standard IEEE 802.15.4, et sont adaptés pour les applications impliquant une faible communication et une faible portée.

La bande utilisée pour les modules fournis est la bande 2.4GHz, et la puissance de sortie de 1mW. La portée dans des conditions optimales est d'environ 100m

Nous allons également développer l'Interface Homme Machine, sous l'environnement Processing. Ce logiciel va nous permettre de créer un affichage graphique des données issues des capteurs.

## II. Interaction avec les capteurs

Dans cette partie, nous allons décrire l'interaction entre la carte Arduino et les différents capteurs. Les capteurs détiennent leur propre protocole de communication, une lecture préliminaire des datasheet a été faite pour comprendre le fonctionnement des capteurs.

### 1) Capteur de Luminosité TEMT6000

Ce capteur détient un phototransistor NPN où sa base va être polarisée grâce à un apport de lumière visible (440nm à 800nm).

Nous pouvons schématiser le capteur TEMT6000 comme suit, une résistance de Pull-Down de 10kohms est présente afin de convertir le courant du courant collecteur en tension avec la loi d'ohm :

$$U_{SIG} = R_1 \times I$$

$$U_{SIG} = I \times 10\,000$$

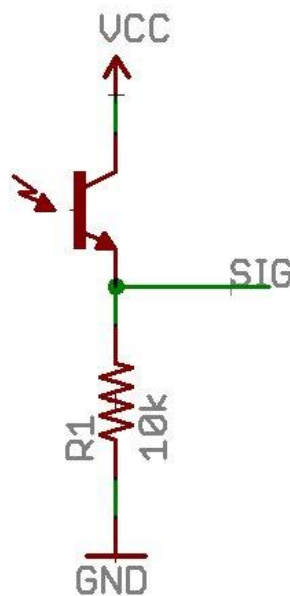


Figure 3 : Schéma équivalent du TEMT6000

Nous connectons la sortie SIG du capteur à la broche A0 de l'Arduino (broche 23 de l'ATMEGA). Cette entrée est connectée à un CAN (Convertisseur Analogique Numérique) qui va renvoyer une donnée sur 10 bits, soit une donnée comprise entre 0 et 1023.

L'alimentation du capteur est de 5V, donc nous connectons cette broche aux +5V généré par l'Arduino.

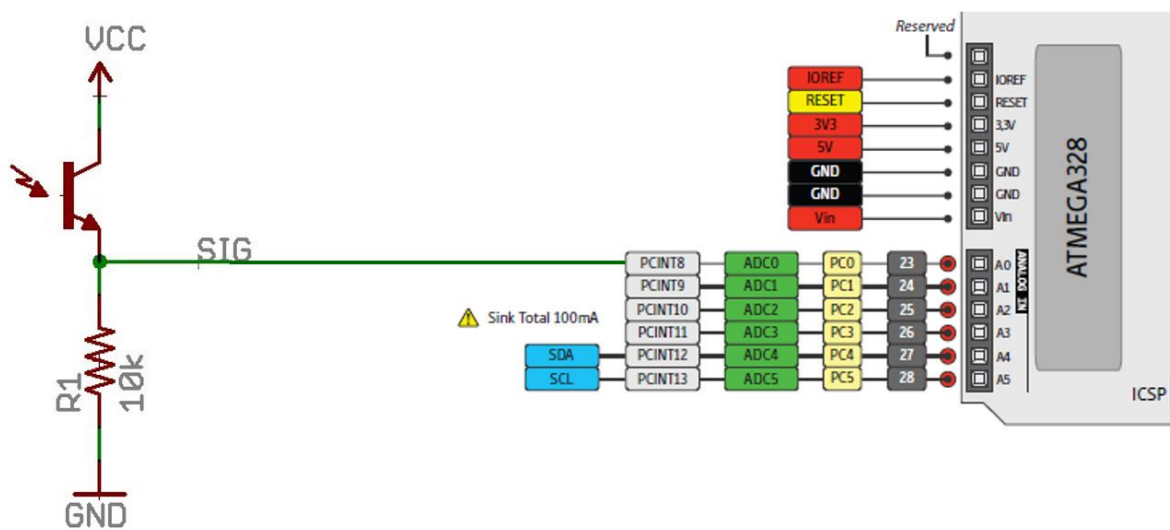


Figure 4 : Schématique du brochage du capteur avec l'Arduino

Le signal de sortie du capteur va donc varier entre 0V et +5V en fonction de la luminosité ambiante. Sachant que notre capteur détient une loi linéaire nous obtenons les valeurs extrêmes suivantes :

Valeur tension signal SIG (V)	Valeur en sortie du CAN
0V	0
5V	1023

Les valeurs en sortie du CAN peuvent donc être déterminées avec l'équation suivante :

$$CAN_{value} = U_{SIG} \times \frac{1024}{5V}$$

Le courant du collecteur est déduit avec la loi d'ohm :

$$I = \frac{U_{SIG}}{R} = \frac{CAN_{value} \times \frac{5V}{1024}}{10k}$$

Sachant que la loi du capteur est linéaire avec un coefficient direct de 2 (cf. datasheet), nous pouvons déterminer la luminosité en fonction du courant :

$$Lum = I \times 2 = \frac{CAN_{value} \times \frac{5V}{1024}}{10k} \times 2$$

La fonction qui permet de calculer la luminosité grâce aux équations précédentes est la suivante :

```
float ReadLux(int pin)
{
    float valueLum_bit = 0;
    float valueLum_Volt = 0;
    float valueLum_Amp = 0;
```

```

float valueLum_Lux = 0;

valueLum_bit = analogRead(pin);
valueLum_Volt = valueLum_bit * 5 / 1024;           // U = bit*5V/1024
valueLum_Amp = (valueLum_Volt / 10000) * 1000000; // I=U/R=u/10k en microAmp
valueLum_Lux = valueLum_Amp * 2;                  // loi linéaire

return valueLum_Lux;
}

```

Figure 5 : Fonction de calcul de la luminosité capteur TEMT6000

Pour afficher les valeurs renvoyées par la carte Arduino sur le port série, nous utilisons les fonctions de communication « Serial » de la librairie Arduino.

```

#define PIN_TEMT6000 A0

void setup() {
    Serial.begin(9600); // Initialisation port Série
}

void loop() {
    Serial.print("Valeur en LUX : ");
    Serial.println(valueLum_Lux);
    delay(100);
}

```

Figure 6 : Code pour afficher valeur de la luminosité sur le port série de l'ordinateur

Nous obtenons la luminosité en Lux sur le terminal de l'IDE :

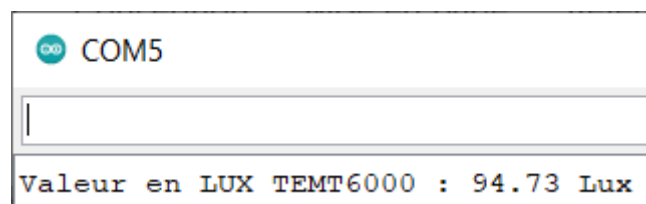


Figure 7 : Valeur luminosité affichée

Nous verrons dans la partie 0 comment envoyer ses données via un module WIFI xBee.

## 2) Capteur de pression et de température MPL115A2

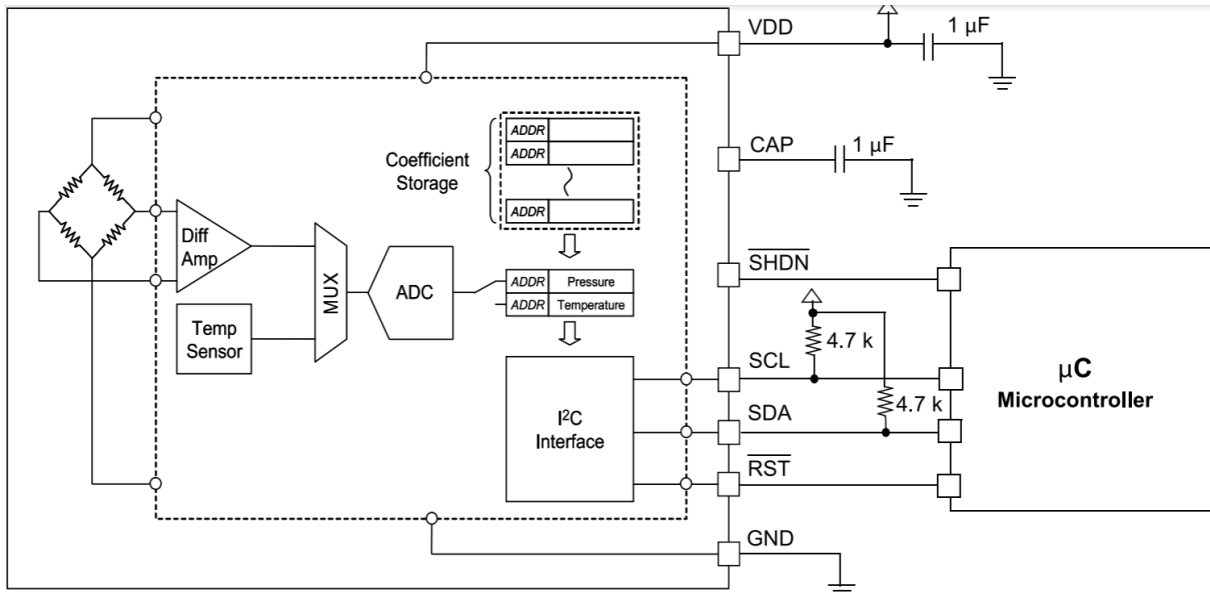
### a. Mise en place du capteur MPL115A2

Ce capteur détient une liaison série i2C pour la communication avec d'autre équipement. L'Arduino détient deux broches (A4 et A5) permettant de faire un liaison i2C :

- SCL : ligne pour horloge de synchronisation
- SDA : ligne pour transmission des données

Nous câblons les broches suivantes au +5V de l'Arduino :

- RST : pour désactiver la remise à zéro du capteur (désactivation interface i2C)
- SHDD : pour activer le capteur
- VDD : pour alimenter le capteur



Pin	Name	Function
1	VDD	Power Supply Connection: VDD range is 2.375 V to 5.5 V.
2	CAP	1 µF connected to ground.
3	GND	Ground
4	SHDN	Shutdown: Connect to GND to disable the device. When in shutdown, the part draws no more than 1 µA supply current and all communications pins (RST, SCL, SDA) are high impedance. Connect to VDD for normal operation.
5	RST	Reset: Connect to ground to disable I <sup>2</sup> C communications.
6	NC	No connection
7	SDA <sup>[1]</sup>	Serial data I/O line
8	SCL <sup>[1]</sup>	Serial clock input.

Figure 8 : Brochage du capteur MPL115A2



Nous obtenons donc le montage suivant :

## ARDUINO UNO

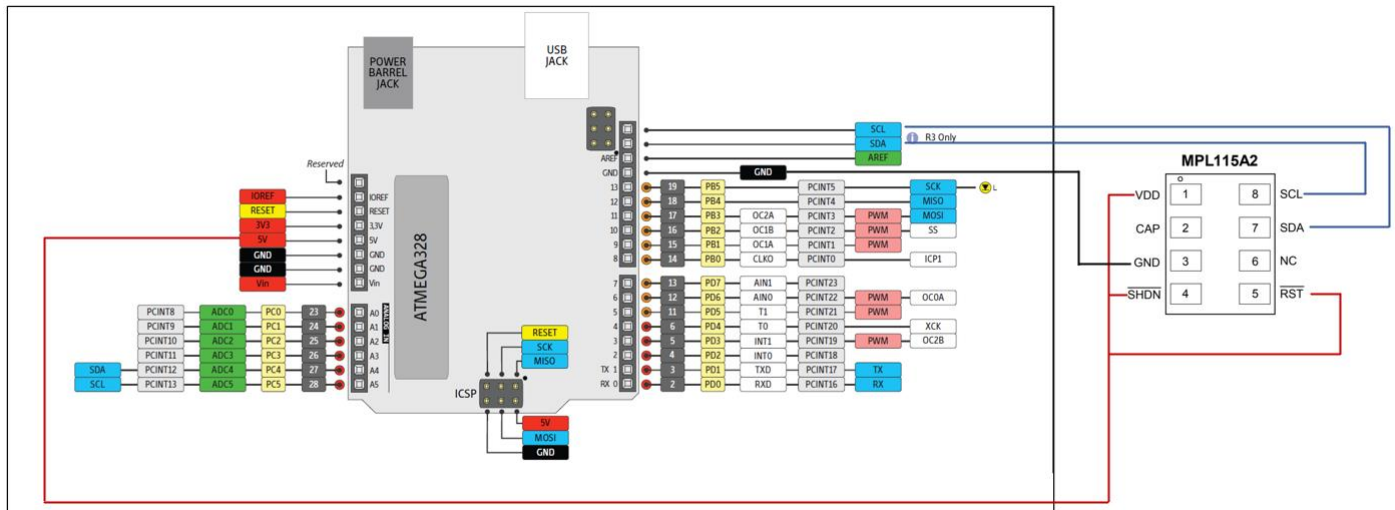


Figure 9 : Schématique du branchement du capteur avec la carte Arduino UNO

### b. Communication i2C

Nous devons maintenant mettre en place la communication entre la carte Arduino et le capteur avec la liaison i2C. Pour cela, nous devons lire attentivement la datasheet du capteur.

La datasheet nous indique les tâches à réaliser pour l'utilisation du capteur :

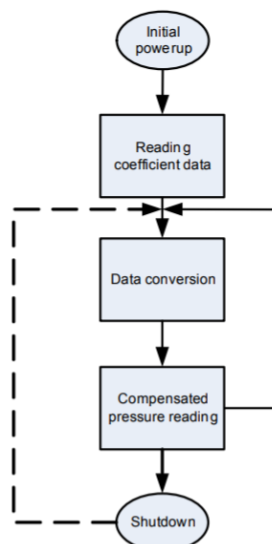


Figure 10 : Flow chart

Nous avons ainsi 5 étapes :

- **Alimentation de la carte** : tous les éléments du circuit sont actifs, les broches du port 2C sont à haute impédance et les registres associés sont effacés. L'appareil passe alors en mode veille.
- **Lecture des coefficients** : des coefficients propres et uniques au capteurs sont stockés dans des registres spécifiques. Cette lecture est réalisée une seule fois vu que les coefficients ne sont pas modifiés.
- **Conversion des données** : C'est la première étape qui est effectuée chaque fois qu'une nouvelle lecture de pression est requise, ce qui est initiée par l'hôte envoyant la commande CONVERT (0x12). Les principaux circuits du système sont activés (réveil) en réponse à la commande et une fois la conversion terminée, le résultat est placé dans les registres de sortie du CAN de pression et de température. La conversion se termine dans le temps de conversion maximum. Le capteur passe ensuite en mode veille en attente de commande.
- **Lecture de la pression compensée** : Une fois que la conversion a eu suffisamment de temps pour se terminer, le microcontrôleur (maître) lit le résultat des registres de sortie CAN et calcule la pression compensée. Ceci doit être fait en utilisant les données de coefficient du MPL115A et des valeurs brutes de sortie CAN de pression et de température. À partir de cette étape, le microcontrôleur peut soit attendre ou revenir à la conversion de données pour obtenir la prochaine lecture de pression. On peut également arrêter le capteur pour économiser de l'énergie.
- **Arrêt système** : le maître peut désactiver le capteur, cela videra tous les registres.

#### *i. Lecture des coefficients*

Comme vu précédemment, nous allons réaliser cette action une seule fois car ce sont des données de compensation intégrées lors de la calibration du capteur en usine et elle est propre au capteur. Ces coefficients vont permettre de calculer la compensation à apporter à la pression mesurée :

$$P_{COMP} = a_0 + (b_1 + c_{12} \times T_{ADC}) \times P_{ADC} + b_2 \times T_{ADC}$$

La datasheet nous indique les adresses des registres contenant ces coefficients :

Address	Name	Description
00h	Padc_MSB	10-bit Pressure ADC output value MSB
01h	Padc_LSB	10-bit Pressure ADC output value LSB
02h	Tadc_MSB	10-bit Temperature ADC output value MSB
03h	Tadc_LSB	10-bit Temperature ADC output value LSB
04h	a0_MSB	a0 coefficient MSB
05h	a0_LSB	a0 coefficient LSB
06h	b1_MSB	b1 coefficient MSB
07h	b1_LSB	b1 coefficient LSB
08h	b2_MSB	b2 coefficient MSB
09h	b2_LSB	b2 coefficient LSB
0Ah	c12_MSB	c12 coefficient MSB
0Bh	c12_LSB	c12 coefficient LSB

Figure 11 : Adresses des registres du capteur

Les coefficients sont donc stockés de l'adresse 0x04 à 0x0B. Il y a deux registres pour décrire un coefficient (MSB et LSB), il faudra ainsi concaténer les deux registres pour traiter un seul coefficient.

Pour la lecture des coefficients, nous avons une fonction (mise dans le SetUp) qui reçoit un tableau de 4 cases et va les remplir des 4 coefficients (a0, b1, b2 et c12) :

```
void ReadCoef (float coef[]);
```

Nous utilisons la librairie i2C Arduino pour la communication avec le capteur. Nous réalisons une boucle for pour lire les 8 registres contenant les valeurs des coefficients en suivant les étapes suivantes :

- 1) Commencer la transmission
- 2) Envoyer une requête au capteur pour la lecture d'un registre (spécifié par l'adresse du registre)
- 3) Stopper la transmission
- 4) Demande au capteur d'envoyer la valeur du registre spécifié
- 5) Attente et stockage d'un octet dans le microcontrôleur s

Ces étapes sont décrites dans la fonction ReadCoef :

```

for (int i = 0 ; i < 8 ; i++){           //lecture registre 0x04 à 0x0B (coefficient)
Wire.beginTransmission(I2c_ADDR_MPL); //demande de transmission au capteur en adresse 0x60
Wire.write(ADDR_REG_A0_MSB + i);        //lecture des différents registres
Wire.endTransmission();                 //fin de la transmission
Wire.requestFrom(I2c_ADDR_MPL, 1);      //envoi d'un octet sur le composant à l'adresse 0x60

if (Wire.available() == 1){             //si un octet dispo alors on lance la lecture
    reg[i] = Wire.read();                //stockage des coefficient
}
}

```

Figure 12 : Lecture des registres contenant les valeurs des coefficients

Comme vu à la Figure 11, les coefficients sont écrits dans deux registre de 8 bits, l'un représentant le MSB et l'autre le LSB du coefficient. Il faut donc réaliser une concaténation des valeurs prélevée, pour cela il faut décaler le MSB (bits de poids forts) de 8 bits afin de mettre le résultat sur 16 bits, par exemple :

Valeur sur 8 bits du MSB	1111 1111
Valeur sur 16 bits du MSB avec décalage	1111 1111 0000 0000

Figure 13 : Décalage binaire

Maintenant pour ajouter le LSB (bits de poids faibles), nous réalisons un masquage entre le MSB sur 16 bits et le LSB sur 8 bits afin d'ajouter le LSB. Par exemple :

Valeur sur 8 bits du LSB	1010 0101
Valeur sur 16 bits du MSB avec décalage	1111 1111 0000 0000
Valeur du coefficient sur 16 bits	1111 1111 1010 0101

Figure 14 : Concaténation MSB et LSB

On réalise ces opérations sur tous les coefficients prélevés. Néanmoins sur la datasheet il est précisé que le coef c12 est codé sur 14 bits et non 16 bits, les deux LSB valent 0.

	a0	b1	b2	c12
Total Bits	16	16	16	14

$c12 \text{ MS byte} = c12[13:6] = [c12_{b13}, c12_{b12}, c12_{b11}, c12_{b10}, c12_{b9}, c12_{b8}, c12_{b7}, c12_{b6}]$   
 $c12 \text{ LS byte} = c12[5:0] \& "00" = [c12_{b5}, c12_{b4}, c12_{b3}, c12_{b2}, c12_{b1}, c12_{b0}, 0, 0]$

Figure 15 : Taille total des coefficients

Nous allons donc supprimer les deux LSB sur le résultat obtenu sur 16 bits, par exemple :

Valeur sur 8 bits du LSB	1010 0100
Valeur sur 16 bits du MSB avec décalage	1111 1111 <b>0000 0000</b>
Valeur du coefficient sur 16 bits	1111 1111 <b>1010 0100</b>
Valeur du coefficient sur 14 bits (seulement c12)	1111 1111 <b>1010 01</b>

Figure 16 : Suppression des deux LSB pour le coefficient C12

Ces opération pour concaténer les MSB et LSB sont décrites ci-dessous :

```
//concaténation du MSB et LSB
coef[0] = (reg[0] <<8) | reg[1];
coef[1] = (reg[2] <<8) | reg[3];
coef[2] = (reg[4] <<8) | reg[5];
coef[3] = ((reg[6] <<8) | reg[7]) >> 2 ; //suppression des 2 derniers bits non utilisés
```

Figure 17 : Code C pour déterminer les coefficient à partir du MSB et du LSB

La datasheet nous renseigne que les données sont codées en binaire signé, ceci implique la présence de bit de signe, bits entier et de bits fractionnaires :

	a0	b1	b2	c12	Padc	Tadc
<b>Total Bits</b>	16	16	16	14	10	10
<b>Sign Bits</b>	1	1	1	1	0	0
<b>Integer Bits</b>	12	2	1	0	10	10
<b>Fractional Bits</b>	3	13	14	13	0	0

a0 Signed, Integer Bits = 12, Fractional Bits = 3 :

Coeff a0 =  $S \cdot I_{11} I_{10} I_9 I_8 I_7 I_6 I_5 I_4 I_3 I_2 I_1 I_0 \cdot F_2 F_1 F_0$

b1 Signed, Integer Bits = 2, Fractional Bits = 13 :

Coeff b1 =  $S \cdot I_1 I_0 \cdot F_{12} F_{11} F_{10} F_9 F_8 F_7 F_6 F_5 F_4 F_3 F_2 F_1 F_0$

b2 Signed, Integer Bits = 1, Fractional Bits = 14 :

Coeff b2 =  $S \cdot I_0 \cdot F_{13} F_{12} F_{11} F_{10} F_9 F_8 F_7 F_6 F_5 F_4 F_3 F_2 F_1 F_0$

c12 Signed, Integer Bits = 0, Fractional Bits = 13, dec pt zero pad = 9 :

Coeff c12 =  $S \cdot 0 \cdot 000\ 000\ 000\ F_{12} F_{11} F_{10} F_9 F_8 F_7 F_6 F_5 F_4 F_3 F_2 F_1 F_0$

Figure 18 : Description du codage binaire des coefficients

Nous avons créé la fonction `checkSigned` permettant de vérifier le signe de la valeur reçue. Celle-ci va réaliser un complément à 2 si jamais un bit de signe est présent (MSB à 1) :

```
float checkSigned(float number){
    if (number > 32767) { //si bit de signe à 1 (> 0b10 0000 0000 00000 = 0d32767)
        number = number - 65536; //+1 et inversion de tous les bits(compléments à 2)
    }
    return number;
}
```

Figure 19 : Fonction de vérification de signe

Nous appliquons ainsi cette fonction dans la fonction ReadCoef :

```
//prise en compte du bit de signe (MSB)
coef[0] = checkSigned(coef[0]);
coef[1] = checkSigned(coef[1]);
coef[2] = checkSigned(coef[2]);
coef[3] = checkSigned(coef[3]);
```

Une fois le bit de signe vérifié, nous devons retrouver un chiffre à virgule à partir de la mantisse. Pour cela, on connaît d'après la Figure 18 le nombre de bit alloué pour la partie fractionnaire :

- A0 : 3 bits
- B1 : 13 bits
- B2 : 14 bits
- C12 : 22 bits

Pour retrouver notre chiffre en valeur flottante, il suffit de diviser par  $2^n$ , avec n le nombre de bit fractionnaire.

```
//Conversion bit mantisse vers variable flottante
coef[0] = coef[0]/8; //3 bits fractionnaires :  $2^3 = 8$ 
coef[1] = coef[1]/8192; //13 bits fractionnaires :  $2^{13} = 8192$ 
coef[2] = coef[2]/16384; //14 bits fractionnaires :  $2^{14} = 16384$ 
coef[3] = coef[3]/4194304; //22 bits fractionnaires :  $2^{22} = 4194304$ 
```

Figure 20 : Code permettant de retrouver la valeur flottante à partir d'une mantisse

## ii. Lecture de la température et de la pression

Nous allons maintenant prélever la valeur de la température  $T_{ADC}$  avec le registre 0x02 et 0x03 (Figure 11) et de la pression  $P_{ADC}$  avec le registre 0x00 et 0x01 avec la fonction Read\_Pressure\_Temp.

```
void Read_Pressure_Temp (float Tab_press_temp[]);
```

Cette fonction va remplir le tableau `Tab_press_temp` (2 cases de 16 bits) avec la valeur de la température et de la pression. Cette fonction va d'abord envoyer la commande « Start Conversions » soit la valeur 0x12 au capteur afin qu'il puisse prélever les données et de le stocker dans les registres 0x00 à 0x03.

On rajoute un délai de 100 ms avant d'envoyer une deuxième commande pour permettre l'envoi des valeurs des registres 0x00 à 0x03 vers le microcontrôleur.

On peut ensuite stocker les 4 octets dans la mémoire avant de les traiter.

```
#define START_CONV_TEMP_PRESS 0x12
#define Read_Pressure_ADC_MSB 0x00
#define I2c_ADDR_MPL          0x60

Wire.beginTransmission(I2c_ADDR_MPL); //demande de transmission au capteur en adresse 0x60
Wire.write(START_CONV_TEMP_PRESS);    //Commencer la conversion
Wire.endTransmission();               //fin de la transmission
delay(100);                           //ajout d'un délai entre deux transmissions

Wire.beginTransmission(I2c_ADDR_MPL); //nouvelle transmission
Wire.write(ADDR_REG_Padc_MSB);         //demande envoi donnée dès le reg 0x00
Wire.endTransmission();               //fin de transmission
Wire.requestFrom(I2c_ADDR_MPL, 4);    //envoi des 4 octets du capteur (@ 0x60) vers µc

if (Wire.available() == 4){ //si 4 octets sont disponibles alors on lance la lecture
    reg[0] = Wire.read();      //stockage Padc MSB
    reg[1] = Wire.read();      //stockage Padc LSB
    reg[2] = Wire.read();      //stockage Tadc MSB
    reg[3] = Wire.read();      //stockage Tadc LSB
}
```

Comme précédemment, nous devons concaténer le MSB et le LSB de  $T_{ADC}$  et  $P_{ADC}$  avec la même méthode :

```
Tab_press_temp[0] = (reg[0] << 8 | reg[1]) >> 6;
Tab_press_temp[1] = (reg[2] << 8 | reg[3]) >> 6;
```

### iii. Calcul température et pression compensée

Nous avons à présent toutes les données nécessaire aux calculs de la pression et de la température compensée. Pour rappel, dans la datasheet nous avons les relations suivantes :

$$P_{COMP} = a_0 + (b_1 + c_{12} \times T_{ADC}) \times P_{ADC} + b_2 \times T_{ADC}$$

$$Pression(kPa) = P_{comp} \times \left( \frac{115 - 50}{1023} \right) + 50$$

$$Temperature = \frac{Temp - 498}{-5.35} + 25$$

Pour chaque équation, nous avons réalisé une fonction spécifique (calculPcomp, conversionPression, calculTemperature) :

```
float calculPcomp(float a1, float b1, float b2, float c12, unsigned int Padc,
                 unsigned int Tadc ){
    return a1 + (b1 + c12 * Tadc)*Padc + b2*Tadc;
}

float conversionPression(float Pcomp){
    return Pcomp*(0.06353861)+50;
}

float calculTemperature (float Temp){
    return (Temp - 498) / (-5.35) + 25;
}
```

Figure 21 : Fonction renvoyant la pression et la température compensée

Nous utilisons ces fonctions dans la fonction Loop d'Arduino afin de de calculer la pression et la température en boucle toutes les 100 ms et d'afficher les résultat sur le terminal de l'IDE d'Arduino:

```
void loop() {
    Read_Pressure_Temp(Tab_press_temp);
    Pcomp = calculPcomp(coef[0],coef[1],coef[2],coef[3], Tab_press_temp[0], Tab_press_temp[1]);
    Pressure_MPL11A2 = conversionPression(Pcomp);
    Temperature_MPL11A2 = calculTemperature(Tab_press_temp[1]);

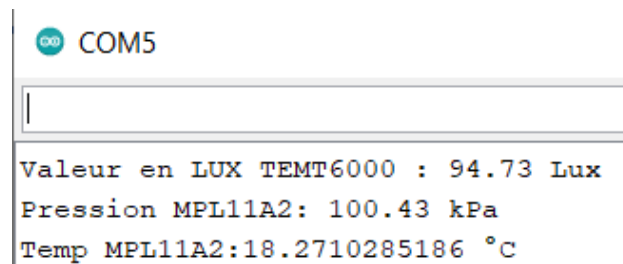
    Serial.println(Pressure_MPL11A2);
    Serial.println(Temperature_MPL11A2, DEC);
    Serial.print("Valeur en LUX TEMT6000 : ");
    Serial.print(valueLum_Lux_TEMT6000);
    Serial.println(" Lux");
    Serial.print("Pression MPL11A2: ");
```



```
Serial.print(Pressure_MPL11A2);  
Serial.println(" kPa");  
Serial.print("Temp MPL11A2:");  
Serial.print(Temperature_MPL11A2, DEC);  
Serial.println(" °C")  
delay(100);  
}
```

Figure 22 : Fonction Void Loop pour afficher les données du capteur MPL115A2

Nous obtenons l’affichage suivant :



The screenshot shows the Arduino IDE terminal window with the title bar 'COM5'. The terminal output displays three lines of sensor data: 'Valeur en LUX TEMT6000 : 94.73 Lux', 'Pression MPL11A2: 100.43 kPa', and 'Temp MPL11A2:18.2710285186 °C'.

```
COM5  
Valeur en LUX TEMT6000 : 94.73 Lux  
Pression MPL11A2: 100.43 kPa  
Temp MPL11A2:18.2710285186 °C
```

Figure 23 : Affichage des données du capteur MPL115A2 sur le terminal de l'IDE

### 3) Capteur d'humidité et de température DHT11

#### a. Mise en place du capteur DHT11

Nous passons maintenant à l'implémentation du code pour le contrôle du capteur DHT11. Celui-ci permet de récupérer l'humidité (%) et la température (°C) ambiante.

Ce capteur présente 3 broches qui nous pouvons connecter avec un microcontrôleur comme suit :

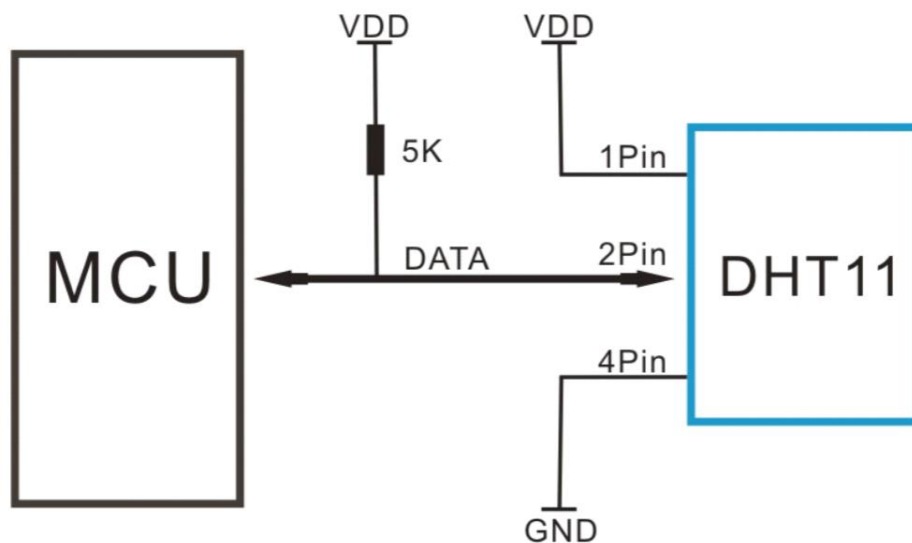


Figure 24 : Brochage du capteur DHT11

Nous connectons donc les différentes broches comme suit :

- VDD au +5V de carte Arduino UNO
- GND à la masse de la carte Arduino UNO
- Data à la broche 4 de la carte Arduino UNO

La résistance de Pull-Up de 5kohms est déjà présente sur le capteur donc pas besoin d'en connecter une dans notre montage.

Nous obtenons ainsi le montage suivant :

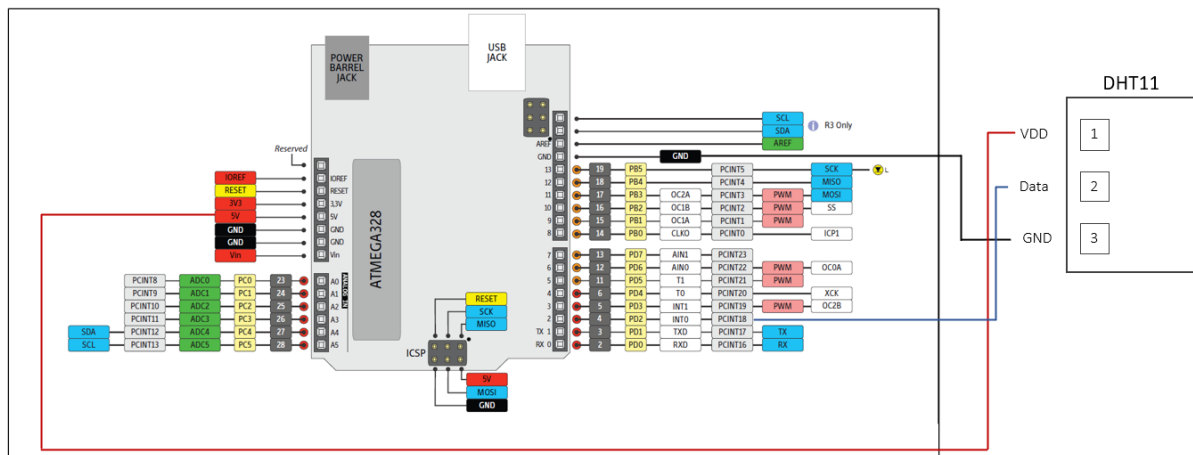


Figure 25 : Schéma du montage du capteur DHT11 sur la carte Arduino UNO

#### b. Communication série bidirectionnelle

Nous utilisons la broche numérique n°4 de la carte car le DHT11 utilise une communication série du type *Single-Wire Two-Way* ce qui signifie que nous avons affaire à une liaison bidirectionnelle sur une seule voie.

La datasheet nous indique qu'une communication dure 40 ms et que les données sont réparties comme suit :

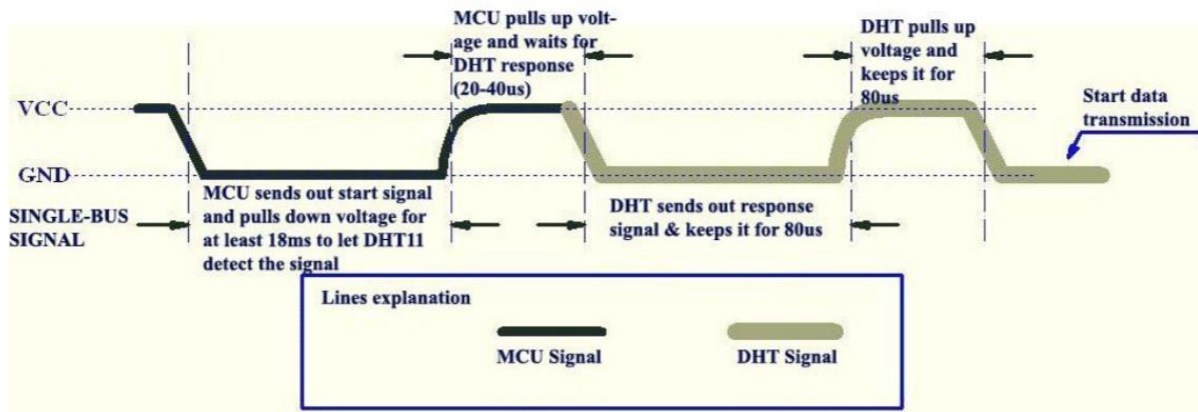
Humidité		Température		
8 bits	8 bits	8 bits	8 bits	8 bits
Partie entière humidité	Partie décimale humidité	Partie entière température	Partie décimale température	Checksum

La partie décimale est toujours égale à 0 car ceci n'a pas encore été développé. Nous devons donc traiter les parties entières des données et le Checksum pour vérifier la validité de la trame reçue.

#### iv. Débuter la communication avec un Start Signal

Pour commencer la communication avec le capteur nous devons réaliser les étapes qui suivent sachant que la résistance de Pull Up force la référence à +5V :

- 1) Forcer la ligne à 0V pendant 18 ms
- 2) Forcer la ligne à 5V entre 20 et 40  $\mu$ s
- 3) Attente de 40 ms pour attendre la réponse totale du capteur



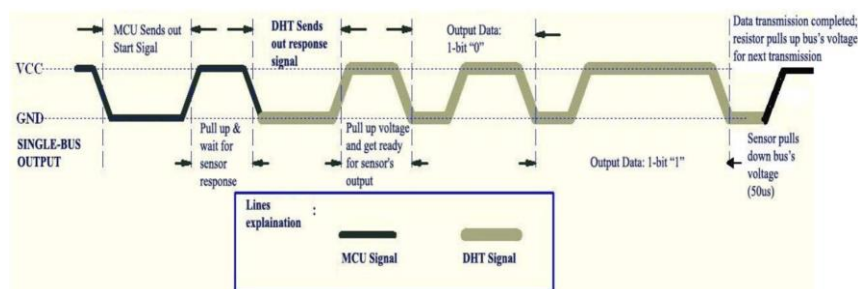
La fonction `startCom(int pin)` va permettre de réaliser les étapes décrites précédemment.

```
void startCom(int pin)
{
    pinMode(pin, OUTPUT);    //pin 4 configurée en OUTPUT (en mode écriture)
    digitalWrite(pin, LOW);  //pin 4 niveau bas
    delay(18);               //niveau bas durant 18 ms (cf. datasheet)
    digitalWrite(pin, HIGH); //pin 4 niveau haut
    delayMicroseconds(40);   //niveau haut durant 40 ms (cf. datasheet)
    pinMode(pin, INPUT);     //pin 4 en INPUT (en mode réception)
}
```

#### v. Réponse du capteur vers le microcontrôleur

Une fois que le Start Signal est détecté par le capteur, celui-ci va réaliser les étapes suivantes :

- 1) Force la ligne Data à 5V pendant 80  $\mu$ s
- 2) Force la ligne Data à 0V pendant 50  $\mu$ s (Strat Transmission)
- 3) Envoi des données (40 bits) :
  - a. Signal à 5V entre 26 et 28  $\mu$ s pour une donnée binaire à 0
  - b. Signal à 5V pendant 70  $\mu$ s pour une donnée binaire à 1
- 4) Force la ligne Data à 0V pour arrêter la transmission



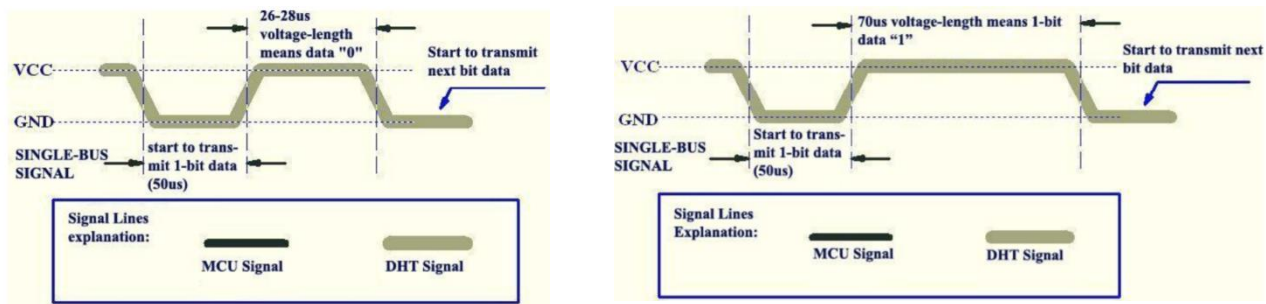


Figure 29 : Envoi d'une donnée 1 ou 0 en fonction de la durée du signal à l'état haut

Nous avons codé la fonction `void Wait_pin_etat(int pin, bool etat)` permettant de vérifier l'état de ligne afin d'en déduire les données :

```
void Wait_pin_etat(int pin, bool etat)
{
    unsigned int timeOUT = 10000;

    while(digitalRead(pin) == etat) //si la pin reste à l'état rentrée dans la fct on décrémente le timeOUT
    {
        if (timeOUT-- == 0) Serial.println("ERROR_TIMEOUT");
    }
}
```

Figure 30 : Fonction permettant de vérifier l'état de la ligne Data

Nous avons maintenant la fonction `readCom(int pin, float *humidity, float *temperature)` qui va permettre de lire les données envoyées par le capteur et de déterminer les valeurs de température et d'humidité. Nous allons décrire succinctement le contenu de cette fonction.

Dans un premier temps, nous démarrons la communication avec le capteur et nous attendons qu'il nous envoie un signal pour commencer la transmission.

```
//on démarre la com avec le capteur
startCom(pin);

//on vérifie que le capteur nous envoie un état bas (cf. datasheet)
Wait_pin_etat(4,0);

//après un état bas, on vérifie que le capteur nous envoie un état haut (cf. datasheet)
Wait_pin_etat(4,1);
```

Figure 31 : Code permettant de réaliser la communication avec le capteur et attente de réponse

Nous stockons ensuite les 40 bits envoyés par le capteur avec décision sur la valeur (0 ou 1) de la donnée. Sachant que nous travaillons sur des octets, nous concaténons 8 bits par 8 bits dans les 5 cases du tableau « octet ».

```
for (int i=0; i<40; i++) //i = 40 car 40 bits
{
    Wait_pin_etat(4,0); //attente du niveau bas pour début de com (cf. datasheet)

    unsigned long t = micros(); //temps en µs de l'instant t0

    Wait_pin_etat(4,1); //attente du niveau haut pour connaître la valeur du bit

    if ((micros() - t) > 40) { //detection de 1 ou 0 (si >40µs => 1 sinon 0)
        octet[index] = octet[index] | (1 << cpt); //stockage du bit reçu dans l'octet (Attention
        //MSB reçu en 1er, cpt initialisé à 7)
    }
    if (cpt == 0)
    {
        cpt = 7;
        index++;
    }
    else cpt--;
}
```

Figure 32 : Code permettant de stocker les 40 bits dans le buffer "octet"

Après avoir stocké 5 octets reçus dans un tableau, nous pouvons déterminer la valeur de la température et de l'humidité. Nous rappelons que la partie décimale n'est pas prise en compte car elle n'a pas encore été développée par le constructeur.

Nous prenons aussi en compte le checksum.

```
*humidity    = (float)octet[0]; //valeur entière de l'humidité
*temperature = (float)octet[2]; //valeur entière de la température

uint8_t sum = octet[0] + octet[2];

if (octet[4] != sum) return ERROR_CHECKSUM_DHT11;
return 0;
```

Figure 33 : Relever de la température et de l'humidité avec gestion du checksum

Avec le fichier Emetteur.ino nous pouvons afficher les différentes données du capteur DHT11.

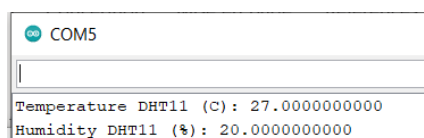


Figure 34 : Affichage de la température et de l'humidité sur le port série de l'IDE Arduino

```
void setup() {  
  }  
  
void loop() {  
  
  readCom(PIN_DHT, &humidity_DHT11, &temperature_DHT11);  
  
  Serial.print("Humidity DHT11 (%): ");  
  Serial.println(humidity_DHT11, DEC);  
  Serial.print("Temperature DHT11 (C): ");  
  Serial.println(temperature_DHT11, DEC);  
  
  delay(500);  
}
```

*Figure 35 : Code pour afficher valeurs du capteur DHT11*

### III. Emission des données

Pour la réception des données, nous avons utilisé deux modules Xbee, un pour l'émission et un pour la réception. La mise en place de la communication Xbee est assez simple, car le module émetteur va directement envoyer les données qui ont été écrites sur le port série. Pour le récepteur, il suffit de lire les données reçues sur le port série.

Le récepteur écrit sur le port série les données reçues par le module Xbee, cela nous permettra de les récupérer par la suite via le port série pour l'affichage sur notre IHM.

```
void setup()
{
    Serial.begin(9600);
}
void loop()
{
    while (Serial.available()){

        Serial.write(Serial.read());
    }
}
```

Nous avons défini le protocole suivant pour l'envoi des données :

122.07	:100.14	:18.8317756652	:146.0000000000	:12.0000000000
Lux	Pression kPa	Température MPL115A2	Humidité	Température DHT11

Nous envoyons les données en série en utilisant « : » comme séparateur entre chaque donnée.



## IV. Traitement des données

### 1) Récupération des données avec Processing

Pour pouvoir créer notre affichage de donnée nous utilisons le logiciel Processing. La première étape est de récupérer les données écrites sur le port série. Pour cela nous utilisons la librairie « processing.serial » afin d'utiliser le port série dans Processing .

```
import processing.serial.*;

Serial myPort; // Create object from Serial class
String val;    // Data received from the serial port
float list[];

void setup() {

    String portName = Serial.list()[3]; //change the 0 to a 1 or 2 etc. to match your port
    myPort = new Serial(this, portName, 9600);

}

void draw() {

    if ( myPort.available() > 0){

        val = myPort.readStringUntil('\n');

        if(val !=null) list = float(split(val, ':'));

    }

}
```

Figure 36 : Code de réception Processing

La variable « val » va contenir les différentes trames reçues :

```
119.14:100.10:18.6448593139:34.0000000000:24.0000000000
120.12:100.28:18.8317756652:146.0000000000:12.0000000000
121.09:100.33:19.0186920166:146.0000000000:12.0000000000
122.07:100.14:18.8317756652:146.0000000000:12.0000000000
121.09:100.14:18.8317756652:146.0000000000:12.0000000000
122.07:100.42:18.8317756652:146.0000000000:12.0000000000
121.09:100.14:18.8317756652:146.0000000000:12.0000000000
120.12:100.14:18.8317756652:146.0000000000:12.0000000000
122.07:100.14:18.8317756652:146.0000000000:12.0000000000
```

Figure 37 : Exemple de trames reçues par Processing

Pour pouvoir retrouver les données utiles nous utilisons la fonction « split » qui comporte deux paramètres : le message à fragmenter (ici « val ») et le caractère de séparation (ici « : »).

Nous enregistrons le résultat dans un tableau de string du nom de « list ».

```
List [0] = 119.14 => Lux  
List [1] = 100.10 => Pression  
List [2] = 18.6448593139 => Température MPL115A2  
List [3] = 34.0000000000 => Humidité  
List [4] = 24.0000000000 => Température DHT11
```

## 2) Affichage des données sur L'IHM

Pour la partie graphique nous utilisons la librairie « processing.serial », cette librairie nous permet notamment d'afficher les axes et des figures

Les deux fonctions principales sont les suivantes :

`setData(X, Y)` : Permet de définir les axes X et Y, et de lire la valeur Y.

`draw(CordX_start, CordY_start, Largeur_femmettre, hauteur_femmettre)` : Permet d'afficher une figure de taille définie aux coordonnées X,Y avec comme point d'origine le haut à gauche de l'écran d'affichage.

Afin d'afficher une courbe dynamique, nous allons créer une fonction qui va venir décaler un tableau, c'est à dire que chaque case du tableau va venir prendre la valeur de la case qui lui précède.

```
void decalagebuffer(float buff[])  
{  
    for (int i = buff.length-1; i > 0; i--) {  
        buff[i] = buff[i-1];  
    }  
}
```

A chaque cycle on vient remplir la première case du tableau avec la valeur d'un capteur, on affiche le tableau à l'aide de la fonction « Draw », ensuite on décale le tableau avec notre fonction « decalagebuffer » et on recommence le cycle pour créer un affichage dynamique.

Afin d'afficher les 5 graphes correspondant aux relevés de nos 5 capteurs, nous allons créer 5 tableaux qui vont contenir chacun une donnée d'un capteur, et on affiche ensuite ces données une à une pour ensuite les décaler. Toute cette démarche nous permet donc de réaliser notre affichage de données météorologiques.

```

void draw() {

    background(255);

    myXYchart.setData(xData, buffer);
    myXYchart.draw(10,0,width-20, (height/5));

    myXYchart1.setData(xData, buffer1);
    myXYchart1.draw(10,(height/5)*1,width-20, (height/5));

    myXYchart2.setData(xData, buffer2);
    myXYchart2.draw(10,(height/5)*2,width-20, (height/5));

    myXYchart3.setData(xData, buffer3);
    myXYchart3.draw(10,(height/5)*3,width-20, (height/5));

    myXYchart4.setData(xData, buffer4);
    myXYchart4.draw(10,(height/5)*4,width-20, (height/5));

    if ( myPort.available() > 0){

        val = myPort.readStringUntil('\n');

        if(val !=null) list = float(split(val, ':'));

    if(list.length==5){
        buffer[0]=list[0];
        buffer1[0]=list[1];
        buffer2[0]=list[2];
        buffer3[0]=list[3];
        buffer4[0]=list[4];}
        decalagebuffer(buffer);
        decalagebuffer(buffer1);
        decalagebuffer(buffer2);
        decalagebuffer(buffer3);
        decalagebuffer(buffer4);

    }

}

```

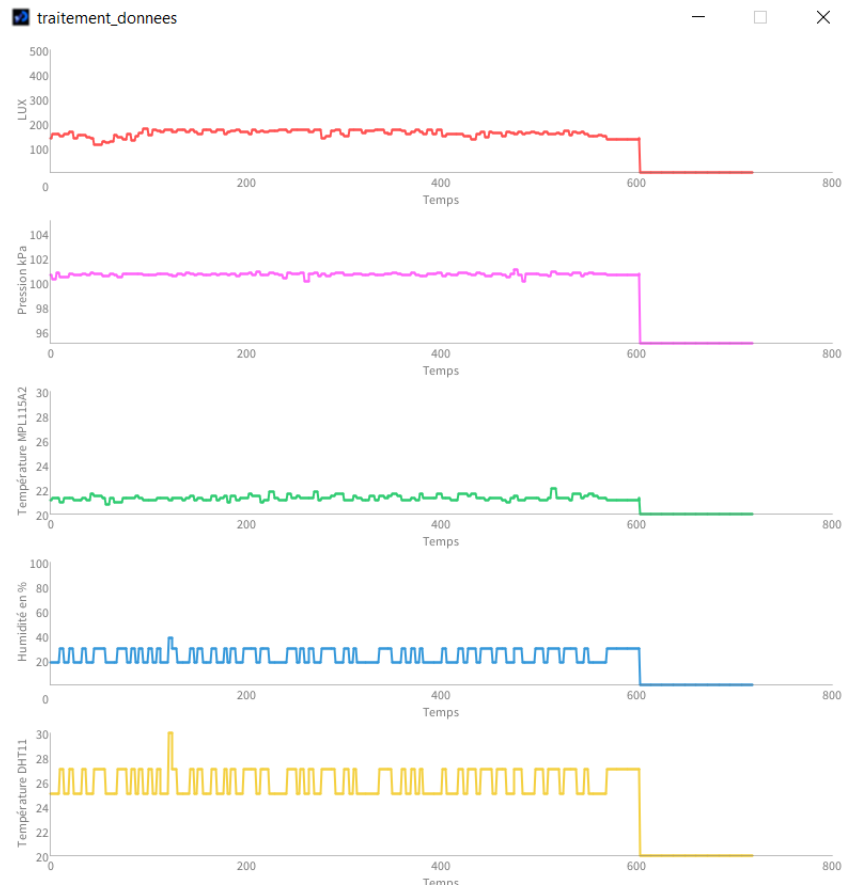


Figure 38 : Affichage de nos données météorologiques en temps réel

## V. Annexes

Montage carte Arduino avec les différents capteurs.

