Département de Génie Electrotechnique et Automatique
Université de 8 Mai 1945 Guelma

Licence 3ème année Automatique
Module : Bus de communication et réseaux locaux industriels

TP3: Utilisation du protocole I2C

1. Introduction

Dans ce TP, nous abordons un nouveau bus et protocole de communication sériel (I2C) qui permet de relier plusieurs instruments avec la même unité de contrôle, ou contrôler le même instrument par plusieurs maîtres. Contrairement au bus SPI, le bus I2C utilise seulement deux lignes pour communiquer, ce qui réduit le coût d'installation.

En réalisant ce TP, l'étudiant pourra implémenter le protocole par la méthode de bit-banging dans l'absence du bus en hardware.

2. Principe de fonctionnement du protocole I2C

Le bus I2C est un autre mécanisme de communication et d'interfaçage différent du bus SPI, où il utilise seulement deux lignes (SDA et SCL) pour communiquer bi-directionnellement (half-duplex). Contrairement à SPI, les périphériques utilisant le protocole I2C ont une adresse unique et on communique avec eux grâce à cette adresse. En outre, I2C permet un contrôle multi-maître.

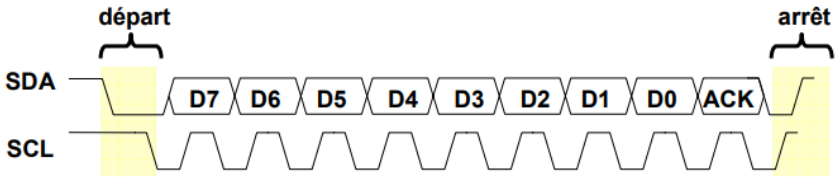

Figure 3.1. Exemple de transmission via le protocole I2C [1]

Avant de commencer la transmission, le bus doit être en repos avec SDA et SCL mis à 1 (ou HIGH). Ensuite, SDA passe à 0 et SCL garde l'état 1, et après la transmission SDA revient à 1 et SCL garde l'état 1 [1]. Le protocole I2C est basé sur MSBFirst, où il transmet le premier bit puis le valide en passant SCL à 0 et revient à 1 pour transmettre le bit suivant. A la fin, le maître transmet un bit d'acquittement 1 et l'esclave répond avec 0 pour signaler la bonne transmission.

La transmission d'une adresse se fait comme la transmission d'une donnée, sauf elle est codée sur 7 bits et le 8ème bit définit s'il s'agit d'une lecture ou d'une écriture. En outre, après

la transmission de la commande, le maître garde l'état 1 dans l'acquittement jusqu'à la réception de la donnée de l'esclave en cas d'une lecture de donnée. Puis le remet à 0 pour continuer la lecture ou à 1 pour mettre fin à la transmission [1].

3. Matériel utilisé dans ce TP

Afin de réaliser ce TP et pratiquer le protocole I2C, nous utilisons le matériel suivant :

- Une carte Arduino UNO ou Arduino Mega ;
- Un module InvenSense MPU-6050 accéléromètre et gyroscope ;
- Des fils de prototypage male-femelle (*male-male en cas d'utiliser une plaque d'essai*) ;
- Une plaque d'essai supplémentaire en cas le branchement du module n'est pas direct.

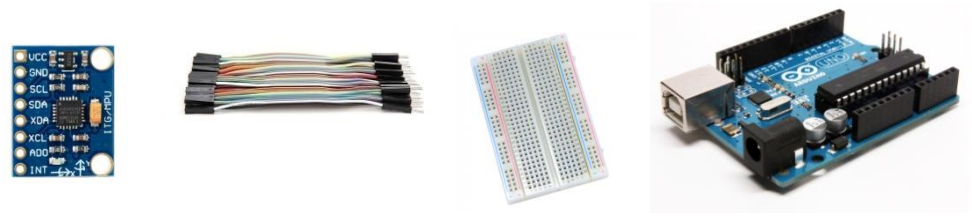

Figure 3.2. Matériel utilisé pour interfacer un afficheur LCD via le protocole SPI

2. Interfaçage d'un accéléromètre/gyroscope

Pour interfacer un équipement que ce soit un capteur ou un actionneur en Arduino, ce dernier est doté d'un bus I2C implémenté en hardware ce qui facilite l'opération. Dans le cas de l'Arduino Uno, il utilise le port analogique A4 pour le SDA (donnée) et le port A5 pour le SCL (horloge). Cependant, il faut noter que la distance maximale est un mètre pour garantir de bonnes performances, mais elle dépend également de la vitesse de transmission et la résistance électrique (câble épais ou fin).

Comme le bus SPI, on a deux possibilités d'utiliser le bus I2C, soit en bénéficiant de la bibliothèque **<Wire.h>** ou en implémentant le code manuellement pour émuler le protocole softwarement (*bitbanging*).

Dans ce TP, nous allons utiliser comme exemple le module MPU6050 qui est doté de trois axes d'accéléromètre, trois axes de gyroscope et un axe de température.

L'accéléromètre et le gyroscope sont utilisés dans de nombreuses applications telles que l'aviation pour régler la stabilité, les smartphones pour tourner l'écran, les véhicules pour détecter les pontes et les côtes, etc.

La première chose à faire est le branchement du module sur l'Arduino. Donc, pour réaliser ce TP, on a besoin d'un Arduino Uno, un module MPU6050 et des fils de prototypage. Le SCL du module est branché dans le port analogique A5, le SDA du module est branché dans

le port A4, le GND dans le GND de l'Arduino et le VCC dans la sortie 3.3v de l'Arduino, car le module MPU6050 fonctionne en 3.3v maximum.

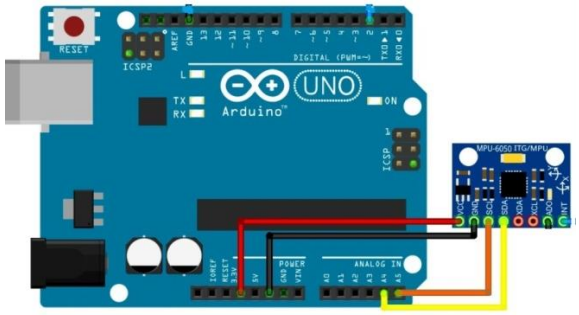

Figure 3.3. Schéma général de branchement du module MPU6050 sur l'Arduino

D'après le référentiel du constructeur [2], dès l'alimentation du module, il se met en repos par défaut et systématiquement il ne fonctionne qu'après un démarrage par l'utilisateur. Donc, on doit le réveiller en accédant au registre dédié, i.e. registre **0x6B**. En bénéficiant de la bibliothèque <Wire.h>, on initialise le bus par la fonction ***begin*** de l'objet ***Wire***, et on précise l'adresse du module (**0x68** en hexadécimal) pour pouvoir communiquer avec seulement. Ensuite, on écrase la valeur du registre **0x6B** par la valeur 0 pour démarrer le module (lignes 28-32 Figure 3.4).

L'étape suivante consiste à accéder au registre **0x3B** pour lire les données de l'accéléromètre, où on précise le nombre d'octets à lire par la fonction ***requestForm*** (ligne 14 Figure 3.4). On demande six octets, car chaque axe (les sept axes) du module est représenté par deux octets, et comme nous avons trois axes pour l'accéléromètre (x, y et z) on lit six octets. Afin de concaténer les deux octets pour trouver la valeur d'un axe, on déplace le premier octet vers la droite avec 8 bit et on ajoute (en utilisant le 'ou' binaire) le deuxième octet (e.g. ligne 18 Figure 3.4).

Une fois les valeurs des trois axes sont lises, on doit les convertir en degré angulaire pour une éventuelle utilisation d'angles, car les valeurs obtenues sont en termes de m/s² ou g. Alors, les angles en degré sont calculés par les formules suivantes [3-4], où les valeurs sont converties du radian vers le degré :

$$\mathrm{Deg_y} = \tan^{-1}\left(\frac{-Accel_x}{\sqrt{Accel_y{}^2 + Accel_z{}^2}}\right) \times \frac{180}{\pi} \qquad (5.1)$$

$$\mathrm{Deg_x} = \tan^{-1}\left(\frac{Accel_y}{\sqrt{Accel_x{}^2 + Accel_z{}^2}}\right) \times \frac{180}{\pi} \qquad (5.2)$$

où $Accel_x$ est l'accélération dans l'axe x, $Accel_y$ l'accélération dans l'axe y et $Accel_z$ est l'accélération dans l'axe z.

Afin de lire les données du gyroscope, on effectue le même principe précédent, sauf on accède au registre **0x43** (ligne 14 Figure 3.5) et on lit six octets correspondants aux trois axes du gyroscope. Cependant, le gyroscope requiert une calibration avant de l'utiliser, car il

mesure le nombre de degrés par seconde il a fait depuis sa position précédente, et il ne peut pas garder la première position (position initiale). Donc, il est sensible aux vibrations qui causent une instabilité de mise en place d'un repaire lors du démarrage. A cet égard, on effectue plusieurs lectures successives de données, puis on fait la moyenne et on la garde comme référence (lignes 32-37 Figure 3.5).

```
1.  #include <Wire.h>
2.
3.  #define RAD2DEG 180/3.14159
4.
5.  typedef struct accel_gyro_t  {
6.     int Ax , Ay , Az , Tmp , Gx , Gy , Gz;
7.  };
8.
9.  accel_gyro_t    MPU6050_read_values()  {
10.       accel_gyro_t ret;
11.       Wire.beginTransmission(0x68);
12.       Wire.write(0x3B);
13.       Wire.endTransmission(false);
14.       Wire.requestFrom(0x68 , 6 , true);
15.
16.       ret.Ax  = Wire.read()<<8 | Wire.read();
17.       ret.Ay  = Wire.read()<<8 | Wire.read();
18.       ret.Az  = Wire.read()<<8 | Wire.read();
19.
20.       return ret;
21.   }
22.
23.   void  setup() {
24.      Serial.begin(9600);
25.
26.      Wire.begin();
27.      Wire.beginTransmission(0x68);
28.      Wire.write(0x6B);
29.      Wire.write(0);
30.      Wire.endTransmission(true);
31.   }
32.
33.   void   loop() {
34.      accel_gyro_t raw = MPU6050_read_values();
35.
36.      float accel_angle_y = atan(- raw.Ax / sqrt(pow(raw.Ay , 2) + pow(raw.Az , 2)))*RAD2DEG;
37.      float accel_angle_x = atan(raw.Ay / sqrt(pow(raw.Ax , 2) + pow(raw.Az , 2)))* RAD2DEG;
38.      Serial.print(''Accel X : '');     Serial.println(accel_angle_x);
39.      Serial.print(''Accel Y : '');     Serial.println(accel_angle_y);
40.   }
```

Figure 3.4. Code source de lecture des données de l'accéléromètre à partir du module MP6050

A chaque nouvelle lecture, on soustrait la valeur de la référence de la nouvelle donnée (lignes 43-54 Figure 3.5). Puis, on divise la différence sur 131 pour trouver le nombre de deg/s, car par défaut 131 vaut 1 deg/s. A la fin, on multiplie le résultat par le nombre de secondes (lignes 47-51 Figure 3.5), i.e. différence de temps entre la nouvelle lecture et la précédente.

```
1.  #include <Wire.h>
2.
3.  #define RAD2DEG 180/3.14159
4.
5.  unsigned long last_time = 0 ;
6.  float base_x_accel   , base_y_accel   , base_z_accel ;
7.
8.  typedef struct accel_gyro_t {
9.     int Ax , Ay , Az , Tmp , Gx , Gy , Gz;
10. };
11.
12. accel_gyro_t    MPU6050_read_values() {
13.      accel_gyro_t    ret;
14.      Wire.beginTransmission(0x68);
15.      Wire.write(0x43);
16.      Wire.endTransmission(false);
17.      Wire.requestFrom(0x68 , 6 , true);
18.      ret.Gx  = Wire.read()<<8 | Wire.read();
19.      ret.Gy  = Wire.read()<<8 | Wire.read();
20.      ret.Gz  = Wire.read()<<8 | Wire.read();
21.      return ret;
22. }
23.
24. void  setup() {
25.     Serial.begin(9600);
26.     Wire.begin();
27.     Wire.beginTransmission(0x68);
28.     Wire.write(0x6B);
29.     Wire.write(0);
30.     Wire.endTransmission(true);
31.
32.     for (int i = 0; i < 10; i++) {
33.          accel_gyro_t   data = MPU6050_read_values();
34.          base_x_accel  +=  data.Gx;    base_y_accel  +=  data.Gy;      base_z_accel  +=  data.Gz;
35.          delay(100);
36.     }
37.      base_x_accel   /=  10;     base_x_accel   /=  10;     base_x_accel   /=  10;
38. }
39.
40. void   loop() {
41.     accel_gyro_t raw = MPU6050_read_values();
42.     unsigned long curr_time = millis();
43.     float gyro_x = (raw.Gx - base_x_gyro)  / 131;
44.     float gyro_y = (raw.Gy - base_y_gyro)  / 131;
45.     float gyro_z = (raw.Gz - base_z_gyro)  / 131;
46.
47.     float dt =(curr_time - last_time)/1000.0;
48.     float gyro_angle_x = gyro_x*dt;
49.     float gyro_angle_y = gyro_y*dt;
50.     float gyro_angle_z = gyro_z*dt
51.     last_time = curr_time ;
52.
53.     Serial.print(''Gyro X : '');     Serial.println(gyro_angle_x);
54.     Serial.print(''Gyro Y : '');     Serial.println(gyro_angle_y);
55.     Serial.print(''Gyro Z : '');     Serial.println(gyro_angle_z);
56. }
```

Figure 3.5. Code source de lecture des données du gyroscope à partir du module MP6050

3. Implémentation du protocole I2C en software

En suivant le principe de fonctionnement du protocole, on peut le simuler par un code source. A cet égard, on peut organiser le code source sous forme d'une classe en adoptant la programmation orientée objet (POO), ou utiliser la programmation procédurale. Dans ce TP, nous adoptons la deuxième méthode, car l'étudiant n'a pas des connaissances en POO qui sera étudiée ultérieurement (en Master).

La première étape à faire consiste à définir les variables globales utilisées par le protocole telles que le numéro du port SDA, le numéro du port SCL, le mode d'entrée utilisé (INPUT_PULLUP ou INPUT) et le temps de l'horloge (Figure 3.6).

```
1.  uint8_t I2C_sda = 2 ; // port digital N°2
2.  uint8_t I2C_scl = 3 ; // port digital N°3
3.  uint8_t I2C_input_mode = INPUT ; // ou INPUT_PULLUP
4.  uint8_t I2C_delay_us = 10 ; // délai en micro secondes
```

Figure 3.6. Déclaration des variables globales utilisées par le protocole I2C

Ensuite, nous définissons la fonction qui marque le démarrage d'une communication I2C, où on met les deux lignes (SDA et SCL) en état élevé (High) en les configurant comme entrées. Après un petit délai de temps (optionnel), on remet et on force les lignes en état bas (Low) en les configurant comme sorties (Figure 3.7). A la fin, on envoie l'adresse de l'équipement pour initialiser la communication (typiquement 0x68). ***Note :*** *les définitions des fonctions* ***sclHigh***, ***sdaHigh***, ***sdaLow*** *et* ***sclLow*** *se trouvent dans l'annexe de ce TP.*

```
1.  uint8_t  beginTransmission ( uint8_t  addr) {
2.          sclHigh();
3.          sdaHigh();
4.          delayMicroseconds (I2C_delay_us);
5.          sdaLow();
6.          delayMicroseconds (I2C_delay_us);
7.          sclLow();
8.          delayMicroseconds(I2C_delay_us);
9.          return write(addr);
10. }
```

Figure 3.7. Définition de la fonction d'initialisation de la communication I2C

Consécutivement, la fonction précédente la succède une autre fonction symétrique qui marque la fin de la communication I2C, et qui consiste à exécuter l'opération inverse de l'opération précédente (Figure 3.8).

Comme mentionné plus haut, la lecture ou l'écriture d'un octet est basé sur le mécanisme MSBF (Most Significant Bit First), ce qui exige de commencer à partir du dernier bit (le plus fort) en allant au premier bit (le plus faible). Pour écrire un bit, on monte l'horloge et on la descend après un délai (lignes 6 et 8 Figure 3.9). Après l'écriture des 8 bits, on lit le bit d'acquittement à partir de l'équipement receveur (Figure 3.10).

```
1.  void  endTransmission () {
2.          sclLow();
3.          sdaLow();
4.          delayMicroseconds (I2C_delay_us);
5.          sdaHigh();
6.          delayMicroseconds (I2C_delay_us);
7.          sclHigh();
8.          delayMicroseconds(I2C_delay_us);
9.  }
```

Figure 3.8. Définition de la fonction qui marque fin à la communication I2C

```
1.  uint8_t  write(uint8_t data) {
2.          for (uint8_t   i = 8 ; i>0 ;  --i , data<<=1) {
3.                  if(data & 0x80) sdaHigh();
4.                  else sdaLow();
5.                  delayMicroseconds (_delay_us);
6.                  sclHigh();
7.                  delayMicroseconds (_delay_us);
8.                  sclLow();
9.          }
10.         return getAck();
11. }
```

Figure 3.9. Définition de la fonction qui écrit un octet

```
1.  uint8_t  getAck() {
2.          sclLow();
3.          sdaHigh();
4.          delayMicroseconds (_delay_us);
5.          sclHigh();
6.          delayMicroseconds (_delay_us);
7.
8.          uint8_t ack = (readSda() == LOW ? ACK : NACK);
9.          delayMicroseconds (_delay_us);
10.         sclLow();
11.         return ack;
12. }
```

Figure 3.10. Définition de la fonction qui lit le bit d'acquittement

Enfin, la dernière fonction à définir est la fonction de lecture d'un octet, et qui constitue l'opération inverse de l'écriture mais en gardant toujours le mécanisme de **MSBF**. Donc, pour lire un bit on descend l'horloge et la monte en activant le SDA comme entrée (ligne 5-8 Figure 3.11). A la fin, on envoie le bit d'acquittement s'il est choisi par l'utilisateur (*acquittement* ou *sans- acquittement*).

Il est important de noter que l'adresse de l'équipement est décalée avec un bit vers la gauche, et on la rajoute un bit à droite qui désigne le mode d'accès soit écriture ou lecture (0 ou 1). Cette procédure est encapsulée dans la bibliothèque **Wire.h**, et on doit la faire manuellement lors de l'utilisation de la fonction **beginTransmission** (ligne 8 Figure 3.12).

```
uint8_t  read(bool send_ack=true) {
        uint8_t data = 0;
        for (uint8_t  i = 8 ;  i>0 ; --i) {
                data <<= 1;
                sclLow();
                sdaHigh();
                delayMicroseconds (_delay_us);
                sclHigh();
                delayMicroseconds (_delay_us);
                if(readSda()) data |= 1;
        }
        sclLow();
        if (send_ack)   sdaLow();
        else sdaHigh();
        delayMicroseconds (_delay_us);
        sclHigh();
        delayMicroseconds (_delay_us);
        sclLow();
        return data;
}
```

Figure 3.11. Définition de la fonction de lecture d'un octet

```
#include<Arduino.h>
#include "myI2C.h"

void setup()
{
   Serial.begin(115200);

   beginTransmission((0x68<< 1)) ; // écriture
   write(0x6B) ;
   write(0) ;
   endTransmission();
}

void loop()
{
    beginTransmission(0x68 << 1); // écriture
    write(0x3B);
    endTransmission();
    beginTransmission((0x68 << 1) + 1); // lecture

    int Ax  = read()<<8 | read();
    int Ay  = read()<<8 | read();
    int Az  = read()<<8 | read();

    Serial.print(Ax); Serial.print(","); Serial.print(Ay); Serial.print(","); Serial.println(Az);
}
```

Figure 3.12. Programme général d'interfaçage du module MPU6050 en utilisant la méthode de bit-banging

Références

[1] Camille Diou. Introduction au bus I2C. Université de Metz.

[2] Référentiel du module MPU6050 du constructeur InvenSense. https://www.invensense.com/wp-content/uploads/2015/02/MPU-6000-Datasheet1.pdf

[3] Mesure d'inclinaison avec l'accéléromètre. https://wiki.dfrobot.com/How_to_Use_a_Three-Axis_Accelerometer_for_Tilt_Sensing [Consulté en 10/2019]

[4] Utilisation d'un accéléromètre pour détecter l'inclinaison. https://www.digikey.com/en/articles/techzone/2011/may/using-an-accelerometer-for-inclination-sensing [Consulté en 10/2019]

Annexe

```
1.  void  sdaHigh() {
2.            pinMode(_sda , _input_mode);
3.  }
4.
5.  void  sdaLow() {
6.            digitalWrite(_sda , LOW);
7.            pinMode(_sda , OUTPUT);
8.  }
9.
10. void  sclHigh() {
11.           pinMode(_scl  , _input_mode);
12. }
13.
14. void  sclLow() {
15.           digitalWrite(_scl , LOW);
16.           pinMode(_scl , OUTPUT);
17. }
18.
19. uint8_t  readScl() {
20.           return digitalRead(_scl);
21. }
22.
23. uint8_t  readSda() {
24.           return digitalRead(_sda);
25. }
26.
```

Figure 3.13. Implémentation des méthodes de lecture et changement d'états des lignes SDA et SCL