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

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

TP2: Utilisation du protocole SPI

1. Introduction

Dans ce TP, nous utilisons le protocole de communication SPI basé sur la transmission en série pour contrôler un ou plusieurs équipements au même temps. C'est l'un des protocoles fréquemment utilisés pour interfacer les capteurs et les actionneurs, et qui permet une communication bidirectionnelle.

En réalisant ce TP, l'étudiant comprendra le fonctionnement du protocole SPI, et aura la capacité de l'implémenter par lui-même dans le cas où il n'est pas implémenté en hardware.

2. Principe de fonctionnement du protocole SPI

Le bus ou le protocole SPI est l'abréviation de *Serial Protocol Interface*, et il est basé sur la communication sérielle en full-duplex. En outre, la communication se fait en mode maître-esclave, où le maître contrôle la communication. L'avantage de ce protocole est la communication avec plusieurs esclaves en même temps, ainsi le partage du bus de communication entre tous les esclaves. Ces derniers sont toutefois sélectionnés et distingués par une ligne dédiée appelée SS (ou Slave Select). La communication SPI est basée sur quatre lignes telles que [1]:

- SCLK (Serial Clock) : utilisée pour l'horloge ;
- MOSI (Master Out Slave In) : nommée aussi SDO, SDA, DO et SO et utilisée pour envoyer des données du maître vers l'esclave ;
- MISO (Master In Slave Out) : nommée aussi SDI, DI et SI et utilisée pour recevoir des données de l'esclave par le maître ;
- SS (Slave Select) : utilisée pour sélectionner un esclave et le mettre en état actif pour communiquer avec le maître.

Les trois premières lignes sont communes entre tous les esclaves, mais la ligne SS est spécifique à chaque esclave. Lorsqu'on veut communiquer avec tous les esclaves en même temps, on met HIGH (état actif) dans le SS de tous. Contrairement, on active seulement l'esclave communiquant et les états des autres SS sont mis à LOW.

Comme mentionné que c'est une communication full-duplex, chaque bit envoyé dans la ligne MOSI lui correspond un bit reçu dans la ligne MISO. Pour envoyer un bit, l'horloge

passe de l'état 0 vers l'état 1 ou l'inverse selon le mode de transmission choisi [1]. Cependant, lors de l'envoie d'une commande via le SPI on ne reçoit pas la réponse immédiatement (e.g. lire une donnée d'un capteur), et en revanche on envoie un octet vide (ou plusieurs) pour recevoir la réponse de la commande.

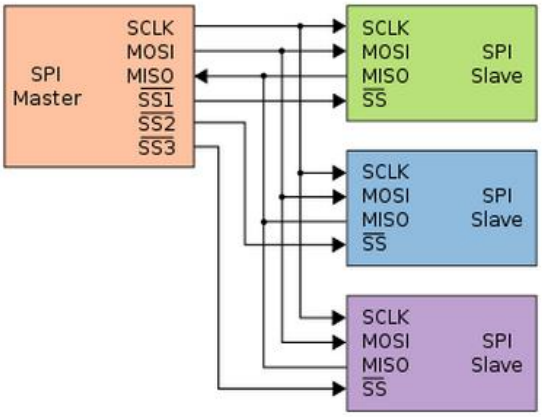

Figure 2.1. Exemple de communication avec plusieurs esclaves via le bus SPI [1]

Le protocole SPI est implémenté en hardware sur presque tous les Arduino. Par exemple, dans l'Arduino Uno, il est implémenté sur les ports 13 (SCLK), 12 (MISO), 11 (MOSI) et 10 (SS). Il faut noter que le port par défaut de SS (10) doit être configuré en sortie même s'il n'est pas utilisé, car il pourra causer un disfonctionnement du protocole au pire des cas [1].

3. Matériel utilisé dans ce TP

Afin de réaliser ce TP et pratiquer le protocole SPI, on utilisera le matériel suivant :

- Une carte Arduino UNO ou Arduino Mega ;
- Un afficheur LCD du modèle Nokia 5110 ;
- 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 ou on interface plusieurs actionneurs au même temps.

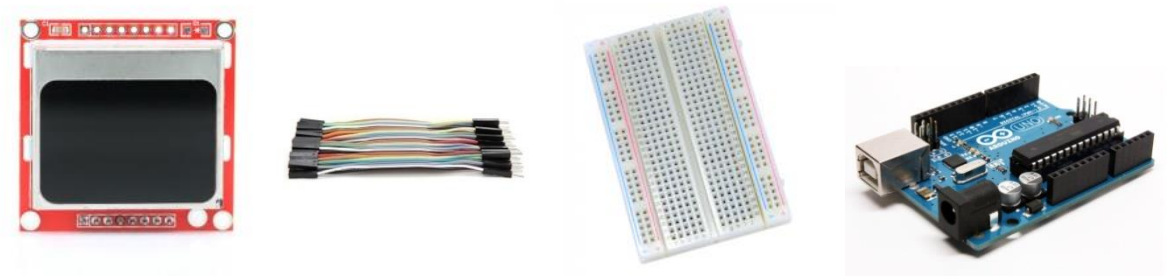

Figure 2.2. Matériel utilisé pour interfacer un afficheur LCD via le protocole SPI

4. Interfaçage d'un afficheur LCD avec SPI

Pour utiliser le protocole SPI en Arduino, on utilise la bibliothèque **<SPI.h>** pour bénéficier de l'implémentation en hardware. Sinon, on peut l'implémenter en software par la méthode de *bitbanging* [1], et on va adopter cette méthode pour mieux comprendre le fonctionnement du bus.

Dans ce TP, nous interfaçons un écran LCD Nokia 5110 avec l'Arduino en utilisant le protocole SPI. Donc, il nous faudra un écran Nokia 5110, un Arduino Uno et des fils de prototypage. Ce type d'écran fonctionne en 5v. Donc, on doit brancher le port VCC de l'écran sur la sortie 5v de l'Arduino, et pour le reste du branchement, on suit le schéma de la **Figure 4.3**.

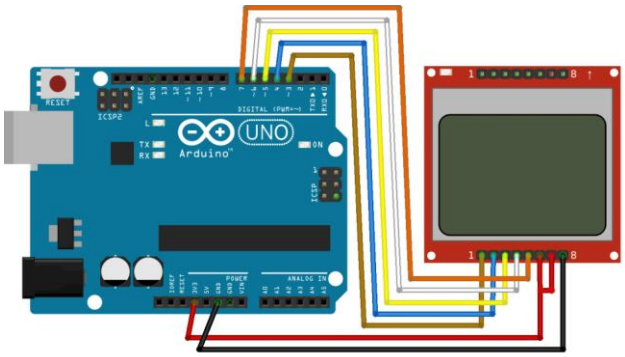

Figure 2.3. Schéma général de branchement de l'afficheur LCD Nokia 5110 sur l'Arduino [2]

L'écran Nokia 5110 est doté d'une résolution de 84x48 pixels, ce qui correspond à 84 colonnes et 48 lignes. Malheureusement, l'écriture sur un tel écran est difficile et nécessite un dessin de tous les caractères/images pixel par pixel. Par exemple, l'écriture de la lettre 'A' majuscule est faite sur une matrice de 8 lignes et 4 colonnes. Le module est doté de deux ports supplémentaires tels que RST (pour restarter le module) et DC pour désigner soit commande soit donnée (0 signifie une commande et 1 signifie une donnée).

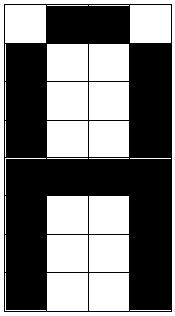

Figure 2.4. Exemple de lettre 'A' majuscule dessinée sur une matrice de 8 lignes et 4 colonnes

On a une bibliothèque qui simplifie l'utilisation de l'écran Nokia 5110, mais on ne va pas l'utiliser dans ce TP afin d'apprendre à comment implémenter le protocole manuellement. Donc, on doit créer une fonction émulant le protocole SPI pour envoyer les données.

[1] Bitbanging consiste à émuler un protocole avec un programme

Après la configuration des ports de l'Arduino (lignes 34-38 dans la Figure 2.5), on configure le module en envoyant des commandes d'initialisation du module (lignes 46-50 dans la Figure 2.5). Pour connaitre toutes les commandes, on doit consulter le référentiel du constructeur [3].

Dans le code de la Figure 2.5, la fonction **sendData** est conçue pour envoyer un octet sur la ligne MOSI, d'où elle prend en charge deux paramètres, i.e. **byte _val** (l'octet à envoyer) et **bool DC** (1 signifie une donnée et 0 signifie une commande). A l'intérieur de la fonction, on précise soit l'octet est une commande ou une donnée (ligne 8), et on désactive (désélectionne) le module (ligne 10) temporairement afin de transmettre l'octet (lignes 12-17) selon le mode MSBFirst (Most Significant Bit First). A la fin on active le module (esclave) pour recevoir l'octet (ligne 18). La fonction **setCursor** est conçue pour positionner le curseur dans l'écran correspondant à un numéro de lignes et de colonnes spécifiques. La fonction **clear** est pour écrire des blancs dans tout l'écran (effacer tout ce qui est affiché).

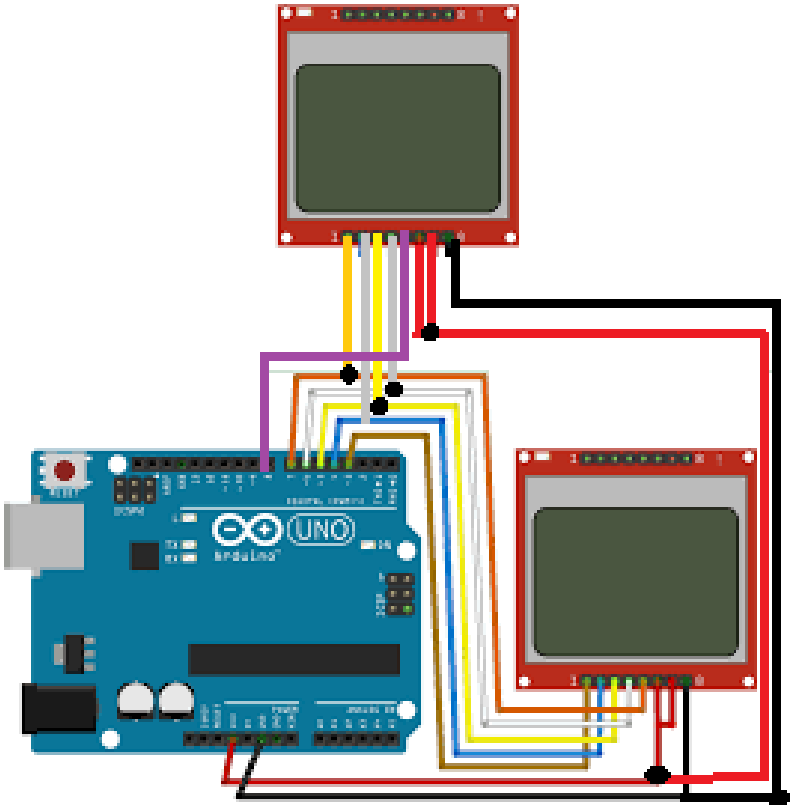

Figure 2.5. Schéma général de branchement de deux afficheurs LCD Nokia 5110 sur l'Arduino

Dans la fonction **loop**, on positionne le curseur dans la position (0, 0), et on affiche la lettre 'A' en majuscule en envoyant quatre octets correspondants à quatre colonnes (ligne 57).

Enfin, dans la fonction créée **sendData**, l'envoi de chaque bit de l'octet (à envoyer) lui correspond la réception d'un bit de l'octet d'entrée (ligne 10 dans la Figure 2.7). En général, lors de la lecture d'une valeur d'un paramètre, on envoie une commande de lecture d'un registre, et après on envoie un octet vide pour recevoir en conséquence l'octet de la valeur.

```
1.  #define LCD_CLK  7
2.  #define LCD_SDA  6
3.  #define LCD_DC   5
4.  #define LCD_RST  4
5.  #define LCD_SCE  3
6.  #define LCD_DIN  2
7.
8.  byte sendData(byte _val , bool DC) {
9.      byte ret = 0;
10.     digitalWrite(LCD_DC, DC);
11.     digitalWrite(LCD_SCE, LOW);
12.     for(int i=7; i>=0; i--) {
13.         digitalWrite(LCD_CLK, LOW);
14.         digitalWrite(LCD_SDA, bitRead(_val, i));
15.         digitalWrite(LCD_CLK, HIGH);
16.         bitWrite(ret, i, digitalRead(LCD_DIN));
17.     }
18.     digitalWrite(LCD_SCE, HIGH);
19.     return ret ;
20. }
21.
22. void setCursor(unsigned char column , unsigned char line) {
23.     sendData (0x80 | column, 0);
24.     sendData (0x40 | line, 0);
25. }
26.
27. void clear() {
28.     setCursor(0, 0);
29.     for (unsigned short i = 0 ; i < 500 ;  i++) sendData(0x00, 1);
30.     setCursor(0, 0);
31. }
32.
33. void setup() {
34.     pinMode(LCD_CLK , OUTPUT);
35.     pinMode(LCD_SDA , OUTPUT);
36.     pinMode(LCD_DC , OUTPUT);
37.     pinMode(LCD_RST , OUTPUT);
38.     pinMode(LCD_SCE , OUTPUT);
39.
40.     digitalWrite(LCD_RST , HIGH);
41.     digitalWrite(LCD_SCE , HIGH);
42.     digitalWrite(LCD_RST , LOW);
43.     delay(100);
44.     digitalWrite(LCD_RST , HIGH);
45.
46.     sendData (0x21, 0); // extended instruction set control (H=1)
47.     sendData (0x13, 0); // bias system (1:48)
48.     sendData (0xc2, 0);  // default Vop (3.06 + 66 * 0.06 = 7V)
49.     sendData (0x20, 0); // extended instruction set control (H=0)
50.     sendData (0x09, 0); // all display segments on
51.
52.     clear() ;
53. }
54.
55. void loop() {
56.     setCursor(0, 0);
57.     sendData(0x7F, 1) ;   sendData(0x88, 1) ;  sendData(0x88, 1) ;   sendData(0x7F, 1) ;
58. }
```

Figure 2.6. Code source d'interfaçage du module Nokia 5110 via le protocole SPI

5. Interfaçage de plusieurs afficheurs LCD

Comme décrit plus haut, l'un des avantages du protocole SPI est le multi-contrôle de plusieurs équipements en utilisant les mêmes lignes partagées. Donc, on refait le TP précédent, mais en reliant deux afficheurs LCD Nokia 5110 sur le même Arduino, où les lignes LCD_CLK, LCD_SDA, LCD_RST, LCD_DC sont communes. Or, chaque afficheur a une ligne LCD_SCE indépendante qui l'active ou le désactive. Pour réaliser la deuxième partie du TP, on utilise une plaque d'essai afin de brancher les lignes communes en parallèle.

```
1.  byte sendData2(byte _val , bool DC) {
2.      byte ret = 0;
3.      digitalWrite(LCD_DC, DC);
4.      digitalWrite(LCD_SCE, LOW);
5.      digitalWrite(LCD_SCE_2, LOW);
6.      for(int i=7; i>=0; i--) {
7.          digitalWrite(LCD_CLK, LOW);
8.          digitalWrite(LCD_SDA, bitRead(_val, i));
9.          digitalWrite(LCD_CLK, HIGH);
10.         bitWrite(ret, i, digitalRead(LCD_DIN));
11.     }
12.     digitalWrite(LCD_SCE, HIGH);
13.     digitalWrite(LCD_SCE_2, HIGH);
14.
15.     return ret ;
16. }
```

Figure 2.7. Code source d'interfaçage du module Nokia 5110 via le protocole SPI

Alors, on garde le même programme précédant, mais on ajoute une ligne secondaire (LCD_SCE_2) dont un numéro de port différent (e.g. port 8). Lorsqu'on veut afficher le même message sur les deux afficheurs, on met HIGH dans les deux ports LCD_SCE (port 3 et 8). Par conséquent, on modifie la fonction **sendData** précédente, et on rajoute l'activation et la désactivation du deuxième afficheur (ligne 5 et ligne 13 dans la Figure 2.7).

Il est toutefois important de savoir qu'on ne peut pas recevoir les données de plusieurs équipements au même temps sur la même ligne MISO. Donc, soit on choisit des ports différents, ou bien recevoir les données séquentiellement (l'une après l'autre).

Références

[1] La librairie SPI. http://www.mon-club-elec.fr/pmwiki_reference_arduino/pmwiki.php?n=Main.LibrairieSPI [Consulté en 09/2019]

[2] Exemple d'interfaçage de l'écran Nokia 5110 avec l'Arduino. https://www.hackster.io/vandenbrande/arduino-nokia-5110-lcd-temperature-meter-with-the-ds18b20-ad45a2 [Consulté en 09/2019]

[3] Datasheet de la puce PCD8544 de l'écran Nokia 5110. https://www.digchip.com/datasheets/parts/datasheet/364/PCD8544.php [Consulté en 09/2019]