République Algérienne Démocratique et Populaire

Ministère de l'Enseignement Supérieur et de la Recherche Scientifique

Université 8 Mai 1945 Guelma

Faculté des Sciences et de la Technologie

Département de

Génie Electrotechnique et Automatique

Polycopié de Travaux Pratiques

Module : Bus de communication et réseaux locaux industriels

Niveau

Cycle Licence 3ème Année Automatique

Année universitaire 2019/2020

Réalisé par

Mr. ABAINIA Kheireddine

Préambule

Ce fascicule s'adresse particulièrement aux étudiants de troisième année cycle Licence en Automatique. C'est une compilation de travaux pratiques visant à initier l'étudiant au domaine des systèmes embarqués qui a une relation étroite avec l'automatisme, ainsi l'ensemble des TP permet d'assimiler et de pratiquer les protocoles et les bus de communication en interfaçant quelques dispositifs (capteurs et actionneurs).

Tous les TP se font sur l'environnement libre (open source) Arduino à cause de sa simplicité et de la disponibilité de la plateforme (gratuite), ce qui permettra aux étudiants de pratiquer et de refaire les TP à la maison. A un budget modéré, l'étudiant peut acquérir le matériel utilisé pendant les TP afin de consolider ses connaissances et s'approfondir dans le domaine.

En premier lieu, le premier TP vise à exploiter la plateforme de développement Arduino et se familiariser avec son logiciel, ainsi on fait un premier pas dans le domaine des systèmes embarqués. Comme annexe du même TP, on fait un petit rappel sur le langage de programmation C++ en faisant la correspondance avec Matlab, vu que ce dernier à été étudié en deuxième année Tronc Commun (i.e. Sciences et Techniques). Ensuite, dans les autres TP, l'étudiant apprendra à communiquer avec des instruments et les contrôler en utilisant un protocole de communication (ou interfaçage).

Ce document présente une série de six (06) TP au total dont chacun adresse un bus et un protocole de communication, or un des TP est supplémentaire et adresse la commande des différents moteurs. Afin de réussir l'ensemble des TP, il est impératif d'acquérir au moins une carte de développement Arduino Uno Rev 3 (ou autre carte Arduino) dans le laboratoire, ainsi il est recommandé de télécharger la dernière version de l'éditeur Arduino du site web officiel *https://www.arduino.cc/en/Main/Software*.

Pour une éventuelle remarque ou suggestion, le lecteur doit contacter l'auteur sur sa boite email *k.abainia@gmail.com* ou *k.abainia@univ-guelma.dz*.

Sommaire

Préambule

TP 1: Introduction à l'Arduino

1. Introduction…….....1
2. Découverte de la plateforme de développement Arduino……....1
3. Vue générale sur le logiciel Arduino……....3
4. Fonctions de base……..……....5
5. Contrôle d'une LED avec un Arduino……..……....8
6. Contrôle avancé d'une LED avec un bouton poussoir……..……....9
7. Implémentation d'un système de switch……....……....11

Références……11

Annexe ……12

TP 2: Utilisation du protocole SPI

1. Introduction…….....1
2. Principe de fonctionnement du protocole SPI……..…….1
3. Matériel utilisé dans ce TP……...….2
4. Interfaçage d'un afficheur LCD avec SPI…….3
5. Interfaçage de plusieurs afficheurs LCD…….…..6

Références…….…….6

TP 3: Utilisation du protocole I2C

1. Introduction…….....1
2. Principe de fonctionnement du protocole I2C……...……1
3. Matériel utilisé dans ce TP……..….2
4. Interfaçage d'un accéléromètre/gyroscope……...……...….2
5. Implémentation du protocole I2C en software…….……6

Références……..9

Annexe ……..……9

TP 4: Utilisation du bus CAN

1. Introduction…….....1
2. Principe de fonctionnement du bus CAN……...1
3. Matériel utilisé dans ce TP……..….2
4. Communication entre deux Arduino via le bus CAN……..3

Références…….…5

TP 5: Utilisation du protocole ModBus

1. Introduction…….....1
2. Principe de fonctionnement du protocole ModBus……..….1
3. Matériel utilisé dans ce TP……..…..3
4. Interfaçage d'un compteur d'énergie monophasé via le protocole ModBus……..…3
Références……..7

TP Supplémentaire: Commande des moteurs par Arduino

1. Introduction…….....1
2. Commande d'un moteur à courant continu…….…1
3. Commande d'un servo-moteur…….…3
4. Commande d'un moteur Brushless……6
5. Commande d'un moteur pas-à-pas…….……...7
Références…….……9

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

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

TP1: Introduction à l'Arduino

1. Introduction

Dans ce TP, nous allons faire une initiation à la plateforme de développement Arduino, où nous commençons par voir l'interface graphique de l'éditeur sur lequel on écrit le code source. Par la suite, nous réalisons un simple montage à base de LED (petite ampoule) et la commander automatiquement.

La programmation Arduino se base principalement sur deux fonctions : fonction **setup()** et fonction **loop()**. La fonction ***setup*** s'exécute seulement une fois au démarrage de la carte (particulièrement lors du branchement de l'alimentation). Donc, à l'intérieur de cette fonction on configure la carte et les ports, ainsi on initialise les dispositifs et les variables globales utilisées dans le programme général. Par contre, la fonction ***loop***, comme son nom l'indique, elle s'exécute continuellement sans arrêt (boucle infinie) jusqu'au débranchement de l'alimentation. Donc, dans cette fonction, on met le programme à exécuter en permanence.

2. Découverte de la plateforme de développement Arduino

Le projet Arduino a été inventé par un groupe d'enseignants et d'étudiants d'une école de design italienne, et ceci dans le but de faciliter le prototypage en électronique à toutes les tranches d'utilisateurs (i.e. électroniciens, informaticiens, musiciens, etc.).

La plateforme de développement est constituée d'une partie hardware (une carte électronique) and d'une partie software, où le tout est open source ce qui veut dire que n'importe qui peut utiliser, modifier et contribuer à l'amélioration de la plateforme gratuitement. Le rôle de la carte Arduino est de recevoir un programme, le stocker dans une mémoire permanente et l'exécuter pour interagir avec le monde physique (i.e. capteurs et actionneurs).

Toutes les cartes Arduino sont équipées d'un microcontrôleur/microprocesseur d'architecture RISC, soit de la famille Atmel ou de la famille ARM. Donc, selon l'usage des utilisateurs, il y a une multitude de cartes, chacune correspond à une application. Les cartes Arduino se diffèrent par leur taille, mémoire volatile, vitesse de traitement et mémoire programme. Dans le tableau ci-dessous (Tableau 1.1), on résume la différence entre quelques cartes Arduino en termes de hardware.

Tableau 1.1. Comparaison entre quelques cartes Arduino [1]

Nom	Processeur	Voltage (volt)	Vitesse (Mhz)	Analogique In/Out	Numérique IO/PWM	EEPROM (Kb)	SRAM (Kb)	Flash (Kb)
101	Intel® Curie	7-12	32	6/0	14/4	-	24	196
Mega 2560	ATmega2560	7-12	16	16/0	54/15	4	8	256
Micro	ATmega32U4	7-12	16	12/0	20/7	1	2.5	32
Pro Mini		5-12	16	6/0	14/6	1	2	32
Uno	ATmega328P	5-12	16	6/0	14/6	1	2	32
Due	ATSAM3X8E	7-12	84	12/2	54/12	-	96	512
Zero	ATSAMD21G18	7-12	48	6/1	14/10	-	32	256
MKRZero	SAMD21 Cortex-M0+	3.3	48	7/1	22/12	-	32	256

Vu que les microcontrôleurs ne peuvent pas interagir directement avec les ordinateurs, les cartes Arduino sont dotées d'une puce (ATmega16U2) d'interfaçage et de communication avec l'ordinateur.

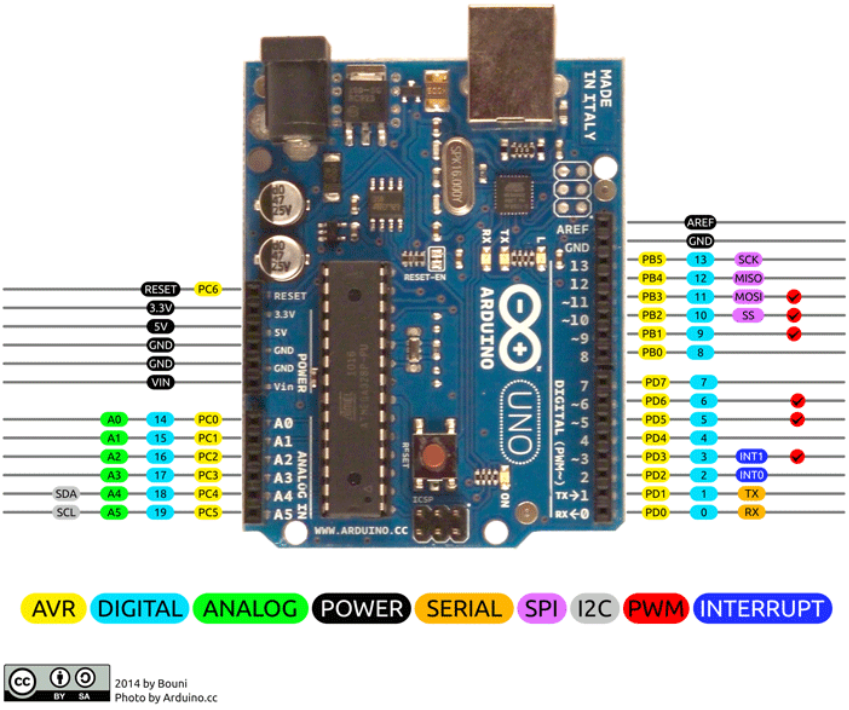

Figure 1.1. Carte Arduino UNO avec étiquetage des ports [2]

A partir de ce TP, nous utilisons la carte Arduino UNO qui est dotée de 14 ports numériques et 6 ports analogiques (Figure 1.1). Outre le nombre de ports de la carte, elle est dotée de protocole SPI implémenté en hardware, protocole I2C implémenté en hardware, signal PWM dans six ports (11, 10, 9, 6, 5 et 3). Elle est dotée également de deux interruptions (port 2 et 3) et un protocole UART/Serial implémenté en hardware (port 0 et 1).

Une fois le câble USB est branché dans l'ordinateur et dans l'Arduino, ce dernier est

alimenté par défaut avec cinq volts (5v) sortant du port USB de l'ordinateur. Sinon, la carte peut être alimentée par une alimentation externe via une fiche jack ayan un voltage d'entrée compris entre 5-12 volts, et ce dernier est passé par un régulateur de voltage (typiquement un circuit L7805CV) pour le réguler à cinq volts. La carte Arduino UNO peut délivrer une tension de sortie 5v et une autre 3.3v pour alimenter des capteurs et actionneurs fonctionnant à ces voltages.

Les ports numériques peuvent lire ou écrire une valeur binaire telle que HIGH qui correspond à TRUE ou1 (ou 5 volt) et LOW qui correspond à FALSE ou 0 (0 volt), tandis que les ports analogiques lisent une valeur numérique comprise entre 0-1024 ce qui correspond à un intervalle entre 0-5 volt.

Enfin, pour programmer la carte Arduino on utilise généralement un environnement libre et multiplateforme (i.e. Arduino IDE), ainsi le langage de programmation C/C++. On trouve dans cet environnement toutes les bibliothèques nécessaires qui facilitent l'interaction avec les dispositifs, ainsi on peut télécharger et rajouter d'autres bibliothèques du site web officiel de l'Arduino ou d'autres serveurs tels que Github.

3. Vue générale sur le logiciel Arduino

Il y a pratiquement deux logiciels Arduino pour écrire un code source et le compiler, où le premier est une version téléchargeable sur un ordinateur avec n'importe quel système d'exploitation (i.e. MacOS, Windows ou Linux), tandis que le deuxième est une version web utilisée en ligne. L'avantage de cette dernière est d'avoir toujours la dernière mise à jour des bibliothèques, mais peut être non appréciée par les utilisateurs qui veulent garder leur code secret.

Le logiciel comprend principalement un menu, une barre d'outils, une barre de progression et trois fenêtres dont la première (couleur blanche) est l'éditeur de texte (ou code source), la deuxième fenêtre (couleur noire avec texte orange) est le log du processus de compilation et de télé-versement du programme, et la troisième fenêtre affiche le numéro de la ligne actuelle (à gauche) et le type de carte sélectionné avec le port COM dédié (à droite). En accédant à la barre d'outils, on peut bénéficier de quelques raccourcis pour effectuer les opérations fréquentes telles que la compilation du code source, le télé-versement du programme créé, l'ouverture d'un fichier, la sauvegarde d'un fichier et la création d'un nouveau projet.

En cliquant sur le menu '*fichier*', on accède aux opérations de base qui sont communes entre la plupart des éditeurs (préférences, sauvegarde, ouverture, nouveau, etc.), ainsi on accède un ensemble d'exemple de codes sources prêts à utiliser (sous-menu '*exemple*' du menu '*fichier*'). En cliquant sur le menu 'édition', on accède aux opérations d'édition de texte (copier, coller, remplacer, commenter, etc.).

Dans le menu '*croquis*' ou '*sketch*', on trouve l'outil de compilation et l'outil de télé-versement du programme créé, ainsi on trouve un autre outil qui permet d'exporter le fichier binaire généré par la compilation du code source pour une utilisation externe (fichier portant

l'extension '.hex'). Comme l'interfaçage des dispositifs (capteurs et actionneurs) est basé sur l'utilisation des bibliothèques contenant des fonctions facilitant la programmation, i.e. en écrivant quelques lignes au lieu des centaines et des milliers de lignes, il y a un sous-menu '*inclure une bibliothèque*' ou '*include library*' pour inclure la définition [1] des fonctions utilisées par une bibliothèque quelconque. En outre, on peut également rajouter une nouvelle bibliothèque qui n'existe pas parmi l'ensemble des bibliothèques prédéfinies, soit en la téléchargeant automatiquement du serveur (en cliquant sur '*gérer les bibliothèques…*' ou '*manage libraries…*'), soit en l'ajoutant manuellement à partir d'un fichier zip (en cliquant sur '*ajouter la bibliothèque .ZIP…*' ou '*add .ZIP library…*').

Figure 1.2. Capture d'écran sur le logiciel Arduino version 1.8.5 pour Windows

Parmi les menus les plus importants, il y a le menu '*outils*' ou '*tools*' qui est conçu pour interagir avec la carte Arduino (Figure 1.3). Plus spécifiquement, à partir de ce menu on peut choisir le type de carte sur laquelle on veut télé-verser le programme en cliquant sur le sous-menu '*Carte*' ou '*Board*'. On a également la possibilité de choisir le numéro de port COM sur lequel la carte est connectée en cliquant sur le sous-menu '*port*', car il est évidant qu'on peut connecter plusieurs cartes Arduino sur le même ordinateur, mais chacune est branchée dans un port USB différent.

Outre le choix de la carte, on peut choisir un programmateur [2] pour télé-verser le programme dans le microcontrôleur de la carte (par défaut le programmateur d'Arduino est

[1]La définition des fonctions se trouve dans les fichiers headers portant l'extension .h, et leur implémentation se trouve dans les fichiers .cpp

[2]Un programmateur est un outil pour programmer les microcontrôleurs et les mémoires EEPROM

'*AVRISP mkII*'). Dans le cas où le microcontrôleur de l'Arduino (i.e. Atmega328) est remplacé par l'utilisateur, il doit contenir le bootloader d'Arduino pour pouvoir comprendre les programmes Arduino. Donc, pour faire ceci, on clique sur '*Graver la séquence d'initialisation*' ou '*Burn Bootloader*' pour installer le bootloader dans le nouveau microcontrôleur qui vient pratiquement vierge (ou vide). On peut lancer le moniteur série (ou Serial Monitor) et le traceur série (Serial Plotter) à partir du menu '*outils*' ou '*tools*', où les deux outils sont conçus pour interagir en temps réel avec la carte Arduino et lire/écrire des données via le bus UART ou Serial.

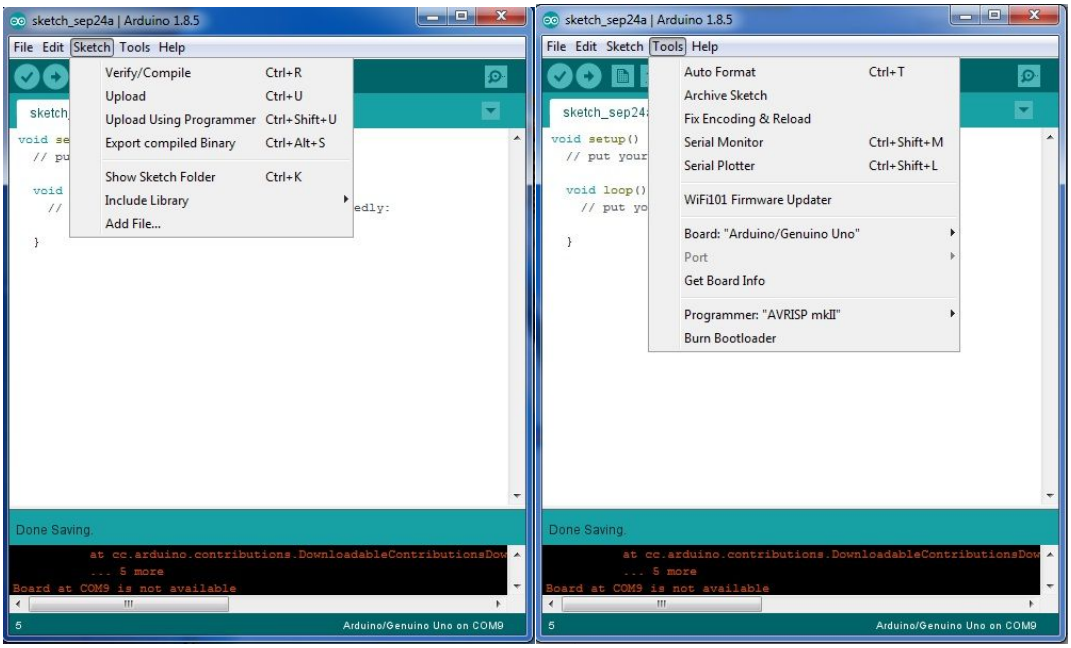

Figure 1.3. Capture d'écran sur le logiciel Arduino IDE. A gauche une illustration du menu « Sketch » et à droite une illustration du menu « Tools »

Enfin, le dernier menu est le menu '*Help*' ou '*Aide*' qui contient un référentiel d'apprentissage et un guide en cas de soucis de compilation et télé-versement.

4. Fonctions de base

Pour manipuler l'Arduino, il y a un ensemble de fonctions fréquemment utilisées presque dans chaque programme. Dans cette section, on essaie de mentionner quelques unes et les expliquer selon le référentiel officiel de l'Arduino [4].

Nous commençons par la première fonction qui permet de configurer les ports de l'Arduino, soit en entrée (ou input) soit en sortie (ou output). C'est la fonction **pinMode(pin, mode)** qui prend en charge de deux paramètres, i.e. le numéro du port (pin) et le mode (INPUT, OUTPUT ou INPUT_PULLUP). Le mode OUTPUT configure le port en étant sortie, ce qui veut dire un signal sort du port de la carte vers le dispositif. Les deux modes INPUT et INPUT_PULLUP configurent le port en étant entrée, ce qui veut dire un signal sort du dispositif vers le port de la carte. Cependant, la différence entre les deux réside dans l'état du

bouton quand il est appuyé, où dans le mode INPUT_PULLUP on lit l'état HIGH quand le bouton est en repos et l'état LOW quand il est appuyé.

Après la configuration du port, on effectue soit une lecture soit une écriture, et pour faire cela on utilise des fonctions dépendant du type du port (numérique ou analogique). Pour envoyer un signal via un port numérique, on utilise la fonction **digitalWrite(pin, value)** qui prend en charge un numéro de port et la valeur à écrire (soit HIGH/1 ou soit LOW/0). Bien évidemment le contraire de la fonction ***digitalWrite*** est **digitalRead(pin)** qui prend en charge d'un seul paramètre (le numéro du port) et renvoie une valeur booléenne (soit HIGH/1 ou soit LOW/0).

Les fonctions similaires utilisées avec les ports analogiques sont **analogWrite(pin, value)**et **analogRead(pin)**, où le deuxième paramètre de la première fonction est une valeur numérique entre 0 et 255. La fonction analogRead renvoie une valeur numérique comprise entre 0 et 1023 ce qui correspond à 0-5v, donc on applique la règle des trois pour trouver le voltage d'entrée. Par exemple, lorsqu'on lit la valeur 980, elle correspond à 4.7v (962 * 5 / 1023).

Pour faire un délai de retard entre les instructions du programme, on utilise la fonction **delay(milliSeconds)** dans laquelle on passe le temps en milli secondes. Cette instruction met la carte ou le microcontrôleur en repos pendant un moment tout en gardant l'état des ports tels qu'ils sont avant cette instruction. Donc, elle est très utile dans les protocoles de communication pour régler la fréquence du signal inter-échangé, ainsi pour faire des animations (e.g. clignotement d'une LED).

Il y a une autre alternative de la fonction ***delay*** et qui travaille sur l'ordre de micro secondes, c'est la fonction **delayMicroseconds(microSeconds)** qui reçoit la valeur en micro secondes comme paramètre. Dans la même famille des fonctions, on trouve la fonction **millis()**qui renvoie le nombre de milli secondes passés depuis le démarrage de la carte Arduino (branchement de l'alimentation).

Enfin, si on veut échanger des données entre la carte Arduino et l'ordinateur en temps réel via le câble USB en utilisant le bus UART, on a trois fonctions dédiées. La première fonction est la fonction ***begin*** (de l'objet ***Serial***) qui initie la communication avec l'ordinateur, elle reçoit en paramètre la vitesse de communication en baud (ou bits par seconde). Donc, on fait appel à ***begin*** dans la fonction principale **setup()**, où on précise la vitesse telle que 4800, 9600, 19200, etc.

Pour écrire une valeur (numérique ou autre) de l'Arduino vers l'ordinateur, ou autrement on affiche une valeur acquise depuis l'Arduino, on fait appel à la fonction **print(value)** ou **println(value)** appartenant au même objet ***Serial***. Donc, on écrit **Serial.print(value)** ou **Serial.println(value)**. On peut même spécifier le format d'affichage tel que BIN (binaire), OCT (octal), DEC (décimal) et HEX (hexadécimal) en mettant un de ces formats comme deuxième paramètre de la fonction, par exemple **Serial.println(value, HEX)**. On note que la différence entre les deux fonctions d'affichage ***print*** et ***println*** réside dans le retour à la ligne (***line feed*** en anglais) ajouté par la deuxième fonction (**println**).

Tableau 1.2. Résumé de quelques fonctions fréquemment utilisées en Arduino

Fonction	Valeur à renvoyer	Paramètres	Description
pinMode(pin, mode)	Rien	pin : port mode : OUTPUT, INPUT ou INPUT_PULLUP	Configurer un port en sortie ou entrée
digitalWrite(pin, value)	Rien	pin : port value : HIGH ou LOW	Ecrire une valeur numérique
digitalRead(pin)	Valeur booléenne	pin : port	Lire une valeur numérique
analogWrite(pin, value)	Rien	pin : port value : 0 - 255	Ecrire une valeur analogique
analogRead(pin)	Valeur entière 0-1023	pin : port	Lire une valeur analogique
delay(milliSeconds)	Rien	Milli secondes	Faire un retard en milli secondes
delayMicroseconds(microSeconds)	Rien	Micro secondes	Faire un retard en micro secondes
millis()	Valeur entière longue	Rien	Savoir le temps écoulé en milli secondes depuis le démarrage
Serial.begin(speed)	Rien	speed : 4800, 9600, etc.	Initialiser la communication sérielle
Serial.print(value)	Nombre d'octets affichés	value : numérique ou autre	Ecrire une donnée dans une communication sérielle
Serial.println(value)	Nombre d'octets affichés	value : numérique ou autre	Ecrire une donnée et retourne à la ligne dans une communication sérielle
Serial.read()	Un byte	Rien	Lire un byte via la communication sérielle
Serial.readBytes(buffer, size)	Nombre d'octets à enregistrer dans le buffer	buffer : tableau size : taille en octet	Lire une suite de bytes via la communication sérielle
Serial.parseInt()	Valeur entière	Rien	Lire une valeur reçue via la communication sérielle comme entier
Serial.parseFloat()	Valeur réelle	Rien	Lire une valeur reçue via la communication sérielle comme flottant

Pour faire l'inverse et écrire une valeur depuis l'ordinateur vers l'Arduino, ou acquérir une valeur par l'Arduino depuis l'ordinateur, il y a une multitude de fonctions selon l'usage et l'application. Par exemple, la fonction **Serial.read()** lit les données octet par octet et la fonction **Serial.readBytes(buffer, size)** lit les données en vrac et les sauvegarde dans un

buffer avec une taille passée comme paramètre. Il y a également deux fonctions qui convertissent les données d'entrée en valeur numérique entière ou réelle, ces fonctions sont **Serial.parseInt()** et **Serial.parseFloat()**. Donc, on résume les fonctions citées précédemment dans le Tableau 1.2.

5. Contrôle d'une LED avec un Arduino

Dans cette section, on fait notre première expérimentation en contrôlant une LED avec l'Arduino. Donc, dans ce projet nous utilisons un Arduino, une LED 3mm ou 5mm de diamètre (de n'importe quelle couleur), des fils de prototypage, une plaque d'essai et une résistance 220 Ohm. En fait, la résistance sert pour protéger la LED contre le survoltage, car la LED fonctionne sur 3v max et le voltage sortant du port de l'Arduino Uno (5 volts) l'endommagera au fils du temps.

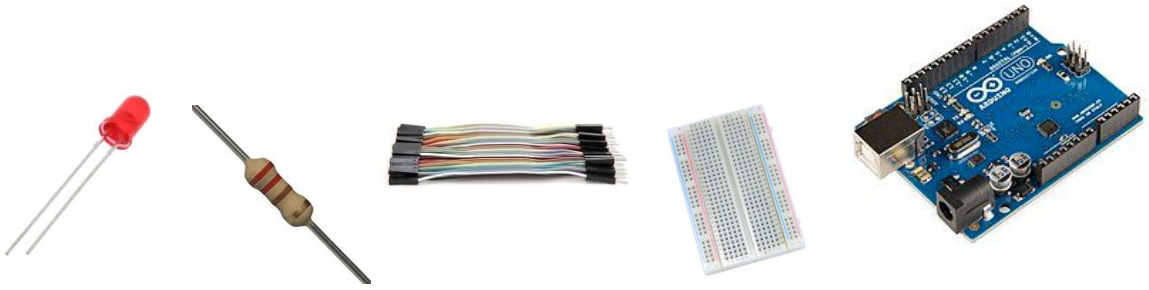

Figure 1.4. Matériel utilisé pour contrôler une LED 5mm avec l'Arduino

Donc, on relit le pole positif (anode) de la LED à n'importe quel port numérique de l'Arduino (par exemple le port 9), et le pole négatif (cathode) à une extrémité de la résistance et l'autre extrémité de celle-ci au GND de l'Arduino (Figure 1.5).

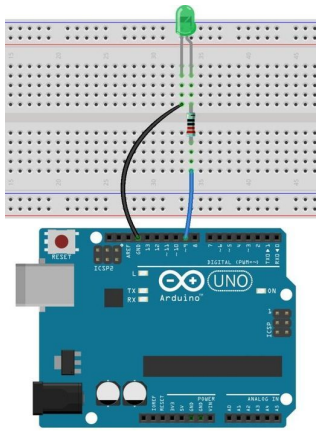

Figure 1.5. Schéma de branchement de la LED sur l'Arduino

Après avoir fait le branchement, on passe à la programmation de la carte Arduino en utilisant le logiciel Arduino IDE. Donc, une fois le logiciel est démarré, on crée un nouveau projet et on le nomme « controle_LED ». Bien évidemment le port sur lequel la LED est branchée doit être configuré en étant sortie, car on a besoin d'un signal qui sort de l'Arduino

vers la LED. Plus précisément, soit il y a un courant sortant qui allume la LED, soit il n'y a pas de courant et la LED s'éteint.

Donc, la première chose à faire c'est de configurer le port numéro 9 en tant que sortie avec la fonction ***pinMode*** en choisissant le mode OUTPUT. Ensuite, dans la fonction ***loop***, on allume et éteint la LED avec la fonction ***digitalWrite*** en jouant avec HIGH et LOW pour passer et couper le courant. Il est souhaitable d'ajouter un petit délai entre les deux états de la LED pour voir le résultat clairement, car si on ne fait pas un délai on risque de ne pas voir la transition entre les deux états. Une fois le code fait sans erreurs, on le compile et on le téléverse vers la carte Arduino. Comme résultat, on obtient une LED clignotante sans arrêt.

Une astuce très pratique pour contourner les erreurs dues aux numéros de ports, notamment lorsqu'on a plusieurs ports à interagir avec, consiste à définir les numéros de ports dans des constantes (ligne 1 Figure 1.6) et utiliser des noms au lieu des numéros (ligne 4, 8 et 10 Figure 1.6).

```
1.  #define PIN_LED 9
2.
3.  void setup() {
4.  pinMode(PIN_LED, OUTPUT) ;
5.  }
6.
7.  void loop() {
8.  digitalWrite(PIN_LED, HIGH) ;
9.  delay(1000) ;
10. digitalWrite(PIN_LED, LOW) ;
11. delay(1000) ;
12. }
```

Figure 1.6. Code source pour clignoter une LED avec l'Arduino

6. Contrôle avancé d'une LED avec un bouton poussoir

Afin de rendre le TP précédent plus dynamique, on rajoute un autre composant pour contrôler l'état de la LED par l'utilisateur. Pour faire cela, on ajoute un bouton poussoir, où l'état de la LED dépend de l'état du bouton.

Figure 1.7. Bouton poussoir

Le but est de lire l'état du bouton, et dans le cas où il est appuyé on allume la LED, sinon celle-ci s'éteint. Donc, on relit l'une des extrémités du bouton à un port numérique différent du premier, ainsi l'autre extrémité avec le GND de l'Arduino.

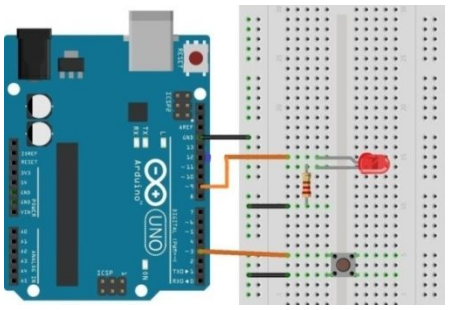

Figure 1.8. Schéma du branchement de la LED et le bouton poussoir sur l'Arduino

La prochaine étape consiste à configurer le nouveau port en étant entrée pour pouvoir lire des signaux externes, et ceci est effectué en utilisant la fonction ***pinMode*** avec le mode INPUT_PULLUP. Ensuite, à l'intérieur de la fonction ***loop***, on ajoute un test de l'état du bouton pour savoir s'il est appuyé ou non en utilisant la fonction ***digitalRead***.

```
1.  #define PIN_LED 9
2.  #define PIN_BOUTON 3
3.
4.  void setup() {
5.  pinMode(PIN_LED, OUTPUT) ;
6.  pinMode(PIN_BOUTON, INPUT_PULLUP) ;
7.  }
8.
9.  void loop() {
10.     if(digitalRead(PIN_BOUTON) == HIGH)
11.     {
12. digitalWrite(PIN_LED, HIGH) ;
13.     }
14.     else
15.     {
16. digitalWrite(PIN_LED, LOW) ;
17.     }
18. }
```

Figure 1.9. Code source pour contrôler la LED avec un bouton poussoir

Le résultat du code de la Figure 1.9 est malheureusement défavorable, car on doit continuer à enfoncer le bouton pour que la LED reste allumée. Alors pour y remédier, on ajoute une variable booléenne qui garde l'état de la LED, et au lieu de changer l'état de la LED directement on inverse l'état de la variable booléenne une fois le bouton appuyé.

Comme expliquez dans le premier TP, le point d'exclamation signifie la négation d'une valeur booléenne. Donc, la ligne 14 signifie (Figure 1.10) l'inversion de la valeur de la variable ***etat_led*** (1 devient 0 et 0 devient 1). Aussi, dans la ligne 16, on a remplacé la valeur HIGH et LOW par la variable ***etat_led***, car celle-ci contient soit 1 soit 0 (systématiquement HIGH ou LOW).

7. Implémentation d'un système de switch

Ceci est un exercice à faire par les étudiants eux-mêmes, et consiste à implémenter un système de switch qui contrôle deux LEDs par deux boutons poussoirs. Lorsque le premier

bouton est appuyé la première LED s'allume et la deuxième s'éteint, et le contraire lorsqu'on appuie sur le deuxième bouton. Dans le cas où on appuie sur un bouton correspondant à une LED allumée, alors elle s'éteint (éteindre toutes les LEDs).

```
1.  #define PIN_LED 9
2.  #define PIN_BOUTON 3
3.
4.  bool etat_led = false ;
5.
6.  void setup() {
7.  pinMode(PIN_LED, OUTPUT) ;
8.  pinMode(PIN_BOUTON, INPUT_PULLUP) ;
9.  }
10.
11. void loop() {
12.     if(digitalRead(PIN_BOUTON) == HIGH)
13.     {
14.         etat_led = ! etat_led ;
15.     }
16. digitalWrite(PIN_LED, etat_led) ;
17. }
```

Figure 1.10. Code source amélioré pour contrôler la LED avec un bouton poussoir

Références

[1] Comparaison entre plusieurs Arduino. https://www.arduino.cc/en/products/compare [consulté en 09/2019]

[2] Description de la carte Arduino Uno. https://components101.com/microcontrollers/arduino-uno [consulté en 09/2019]

[3] Référence du langage C++. https://cppreference.com [consulté en 09/2019]

[4] Référence des fonctions Arduino. https://www.arduino.cc/reference/en/[consultée en 09/2019]

Annexe

Dans cette section, on fait un petit rappel sur le langage C++ en correspondant les mots clés réservés et la syntaxe avec ceux de Matlab. Il est toutefois recommandé de maitriser les bases de Matlab, vu que l'étudiant a déjà étudié le langage Matlab pendant la deuxième année tronc commun en Sciences et Techniques, où il a appris les bases de Matlab pendant le premier semestre et a pratiqué le module '*analyse numérique*' sous Matlab pendant le deuxième semestre.

Le langage C++ permet à la fois une programmation procédurale et une programmation orientée objet, où la première est basée sur la déclaration/utilisation de fonctions et la deuxième se base sur la structuration du programme sous forme d'objets ou de classes (sera étudié durant le Master). Principalement, le corps général d'un programme Arduino (typiquement en C++) se décompose en trois parties : la première est dédiée à l'inclusion des bibliothèques, la deuxième est dédiée à l'initialisation de l'environnement et la configuration du microcontrôleur (s'exécute une seule fois au démarrage) et la troisième consiste à exécuter le programme en permanence (boucle infinie).

1. Inclusion des bibliothèques

Comme tous les langages de programmation, l'utilisation de bibliothèques facilite la programmation, où on écrit quelques lignes de code au lieu de dizaine (voire des centaines de lignes) pour faire une petite manipulation. Donc, plusieurs lignes de code sont regroupées dans une fonction portant un nom bien déterminé, et on n'a qu'à faire appeler cette dernière autant de fois que nous voulons pour effectuer la tâche.

Dans l'environnement Arduino, il y a une multitude de bibliothèques prédéfinies et qui viennent par défaut avec l'éditeur, et pour bénéficier de ces bibliothèques, on les inclut avec la directive **#include**. En Arduino (comme en C++), il est recommandé (selon les normes de codage) d'inclure seulement les bibliothèques nécessaires au début (plus haut) du programme (Figure 1.4). Pour inclure une bibliothèque, on met le préprocesseur **#include** succédé par <>et à l'intérieur on met le nom de la bibliothèque (portant l'extension **.h** ou non dépendant de la bibliothèque).

```
1.  #include<Wire.h>
2.
3.  void setup() {
4.   }
5.
6.  void loop() {
7.  }
```

Figure 1.11. Exemple de code Arduino vide avec l'inclusion d'une bibliothèque

2. Déclaration de variables

Contrairement à Matlab dans lequel les variables sont utilisées directement sans

déclaration au préalable et sans type, en C++ on doit déclarer chaque variable qu'on utilise avec un type bien déterminé selon l'usage et l'application. A cet égard, on trouve plusieurs types couramment utilisés en Arduino comme résumé dans le Tableau 1.2.

Tableau 1.3. Description de quelques variables utilisées en Arduino

Type	Taille (octet)	Description
byte	1	Valeur entière
char	1	Caractère
int/long	4	Valeur entière
short	2	Valeur entière
long long	8	Valeur entière
float	4	Valeur réelle
double	8	Valeur réelle
bool	1 bit	Valeur booléenne (0 ou 1)

Le choix du type de la variable doit être judicieux, car la mémoire des microcontrôleurs est très restreinte (2 Kb dans l'Arduino Uno). Plus précisément, lorsqu'on veut déclarer une variable entière et on sait bien que notre valeur ne dépassera pas 255, il est fortement conseillé d'utiliser le type **byte** au lieu le type **int**, car le premier est de taille 1 octet et le deuxième est de taille 4 octets. Par exemple, un tableau de type **int** et de taille 500 occupera tout l'espace de la mémoire SRAM[3], tandis qu'un autre tableau de type **byte** peut contenir jusqu'à 2000 cases dans le cas où aucune autre variable est déclarée.

Pour déclarer une variable, on met le type de la variable succédé par un espace, puis le nom de la variable succédé par un point virgule pour marquer fin à l'instruction. Il est préférable de donner des noms significatifs aux variables, et dans le cas où le nom est composé (plusieurs mots) on sépare les mots par des tirets bas '_'. Par exemple, pour déclarer une variable de type **byte** qui représente l'état d'une LED [4] indicatrice correspondant à un moteur, on la déclare comme suite :

byte led_moteur **;**

La déclaration des tableaux suit le même principe, mais en précisant la taille du tableau (e.g. nombre de lignes, nombre de colonnes, etc.). Donc, après le nom du tableau et avant le point virgule, on met des crochets [ ] en précisant la taille à l'intérieur, et on fait cette manipulation pour chaque dimension. Par exemple, pour déclarer une matrice (tableau de deux dimensions) de type réel, de 5 lignes et de 7 colonnes, on écrit comme suit :

float tableau[5][7] **;**

[3]SRAM est la mémoire volatile du microcontrôleur
[4]LED est l'abréviation de *light emitting diode* en anglais et une petite ampoule lumineuse

3. Instructions de contrôle

Les instructions de contrôle se divisent en deux telles que les tests et les boucles. En C++, il y a trois types de tests (if, else et switch) pour effectuer des comparaisons comme Matlab, mais sans l'instruction **end** à la fin. Le test ***if*** est succédé par des parenthèses ( ) et à l'intérieur on met la clause (ou condition), puis on écrit l'instruction à exécuter terminée par un point virgule après la parenthèse fermante. Dans le cas où il y a un ensemble d'instructions à exécuter si le test est satisfait, on met l'ensemble des instructions entre deux accolades { } comme illustré par la Figure 1.12 (lignes 3-8).

```
1.  if(valeur_1 == valeur_2) allumerLed() ;
2.
3.  if(valeur_1 == valeur_2)
4.  {
5.      allumerLed() ;
6.  demarrerMoteur() ;
7.   controllerEtat() ;
8.   }
```

Figure 1.12. Exemple de différents tests ***if*** avec une seule instruction et plusieurs instructions à exécuter

Le test ***else*** est la négation du test ***if*** et suit ce dernier, donc on ne peut pas l'utiliser sans le test ***if***. Le test ***else*** ne contient pas une condition, mais en revanche il contient des instructions à exécuter comme le cas de ***if*** (une ou plusieurs instructions).

Le '*ou*' logique est exprimé avec deux barres || en C++, le '*et*' avec && et la négation avec un point d'exclamation '!' (en Matlab '~'). Pour la comparaison numérique, on utilise les simples opérateurs (== , != , < , > , >= ,<=).

Enfin, le dernier test est le test ***switch*** qui remplace plusieurs tests ***if*** d'égalité. Donc, on ne peut pas l'utiliser lorsqu'on a des tests d'inégalité, de négation, d'infériorité ou de supériorité. L'avantage du test ***switch*** est la rapidité d'exécution par rapport à plusieurs tests ***if***.

Pour bénéficier de l'avantage de ***switch***, on met le mot clé réservé ***switch*** succédé par des parenthèses contenant la variable à tester, puis on met des accolades et à l'intérieur les valeurs à tester avec (Figure 1.13). On note qu'il est impératif d'utiliser une variable dans ***switch*** et non une valeur, car cela engendra une erreur de compilation.

Les valeurs à tester sont exprimées par des blocs de ***case*** (mot clé réservé), où on met le mot ***case*** succédé par la valeur (numérique ou caractère seulement) et deux points ' :'. Ensuite, on met les instructions à exécuter et à la fin on termine avec l'instruction ***break*** pour marquer fin aux tests (Figure 1.13). Il faut noter que si on ne met pas l'instruction ***break***, le programme exécute les instructions du prochain bloc ***case*** et continue à faire ça jusqu'à trouver l'instruction ***break*** ou arriver à la fin (l'accolade fermante). Ce dernier cas est plus pratique lorsqu'on veut traduire le '*ou*' logique dans ***switch*** vu que ce n'est pas autorisé qu'avec le test ***if***. Par exemple, si on a un test *if(variable==3 || variable ==11)*, on traduit cela

par deux blocs ***case*** sans mettre ***break*** à la fin du premier (Figure 1.7).

```
1.  if(variable == 3) action1() ;
2.  if(variable == 5) action2() ;
3.  if(variable == 10) action3() ;
4.
5.  switch(variable)
6.  {
7.  case3 :
8.  action1() ;
9.  break ;
10. case5 :
11. action2() ;
12. break ;
13. case10 :
14. action3() ;
15. break ;
16. }
```

Figure 1.13. Exemple de plusieurs tests ***if*** et leur conversion en test ***switch***

```
1.  switch(variable)
2.  {
3.  case3 :
4.  case11 :
5.          action() ;
6.  break ;
7.  }
```

Figure 1.14. Exemple d'utiliser un '*ou*' logique dans le test ***switch***

En C++ les deux boucles (***for*** et ***while***) existent avec une syntaxe différente à celle de Matlab. Par exemple, la boucle ***for*** en C++ contient trois parties séparées par des points virgules à l'intérieur d'une parenthèse. Dans la première partie, on déclare et on initialise les variables à utiliser dans la boucle, la deuxième partie représente le critère d'arrêt (généralement une condition) et la troisième partie c'est la partie incrémentation et dans laquelle on met les variables qui s'incrémentent à chaque itération (Figure 1.15).

```
1.  for(int i=0 ; i < 10 ; i++)
2.  {
3.  }
4.
5.  while (condition)
6.   {
7.   }
8.
9.  do
10.  {
11.     ….
12.  } while (condition) ;
```

Figure 1.15. Exemple d'utilisation de la boucle ***for***, la boucle ***while*** et la boucle ***do while***

La boucle ***while*** est quasi-similaire à celle de Matlab, sauf la condition est mise entre parenthèse, ainsi le bloc d'instructions est mis entre deux accolades (le cas de plusieurs

instructions). Il existe une autre variante de la boucle ***while*** qui s'appelle ***dowhile***, et son rôle consiste à exécuter le bloc d'instructions au moins une fois avant de tester la condition (contrairement à la boucle ***while***).

4. Fonctions et procédures

Une fonction ou une procédure est un sous-programme du programme principal, où on peut le réutiliser autant de fois. En particulier, lorsqu'on a un ensemble d'instructions qui se répètent fréquemment dans le programme, on peut les regrouper dans un petit programme auquel on peut attribuer un nom (nom de la fonction). En Arduino, on utilise couramment les fonctions pour interfacer les dispositifs et de configurer la carte. La différence entre une fonction et une procédure réside dans le type de renvoie, où la première renvoie une valeur (numérique, caractères ou autre) et la deuxième ne renvoie rien.

```
1.  void maProcedure()  {
2.  }
3.
4.  void maProcedure_2(int param_1, bool param_2)  {
5.  }
6.
7.  int maFonction(int param_1, bool param_2)  {
8.       ….
9.  return valeur ;
10. }
```

Figure 1.16. Exemple de déclaration de procédure sans paramètres, avec paramètres et déclaration de fonction

Pour déclarer une procédure, on met le nom de la procédure précédé par le type ***void***(signifie vide en français) et succédé par une parenthèse contenant les paramètres (optionnels selon l'application). La déclaration d'une fonction est similaire à la déclaration de procédure, mais en utilisant un type différent de ***void***, et à la fin du programme (de la fonction) on renvoie une valeur en utilisant le mot clé ***return*** suivi par la valeur/variable (Figure 1.16).

Pour appeler une fonction/procédure prédéfinie ou définie par l'utilisateur, on l'appelle juste avec son nom comme le cas du Matlab et on passe les paramètres (dans le cas échéant).

Enfin, on rappelle que chaque instruction en C++ se termine par un point virgule ' **;**', sauf les tests, la boucle ***for***, la boucle ***while*** et la déclaration des fonctions.

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.

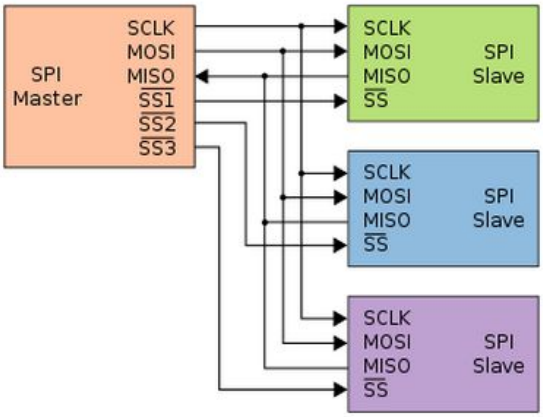

Figure 2.1. Exemple de communication avec plusieurs esclaves via le bus SPI [1]

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.

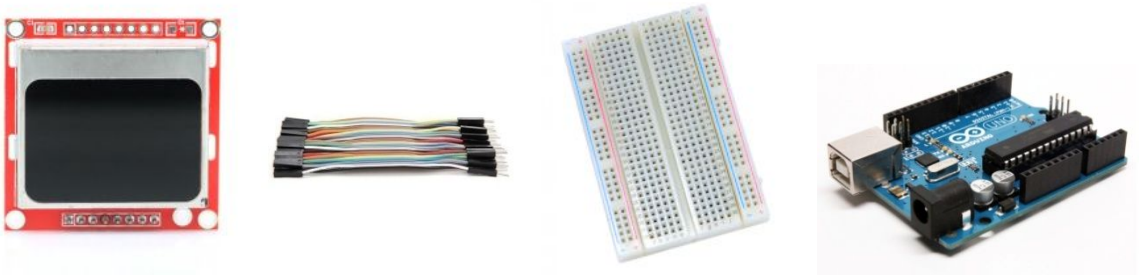

Figure 2.2. Matériel utilisé pour interfacer un afficheur LCD via le protocole SPI

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**.

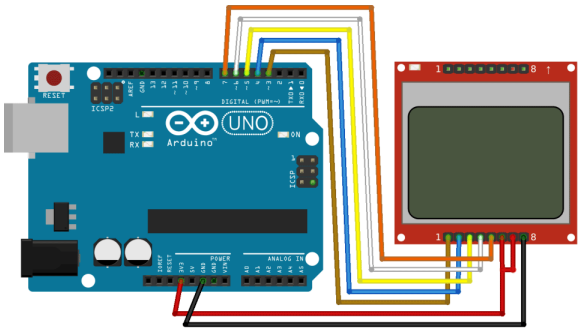

Figure 2.3. Schéma général de branchement de l'afficheur LCD Nokia 5110 sur l'Arduino [2]

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).

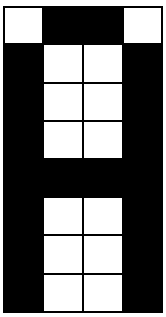

Figure 2.4. Exemple de lettre 'A' majuscule dessinée sur une matrice de 8 lignes et 4 colonnes

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.

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

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

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).

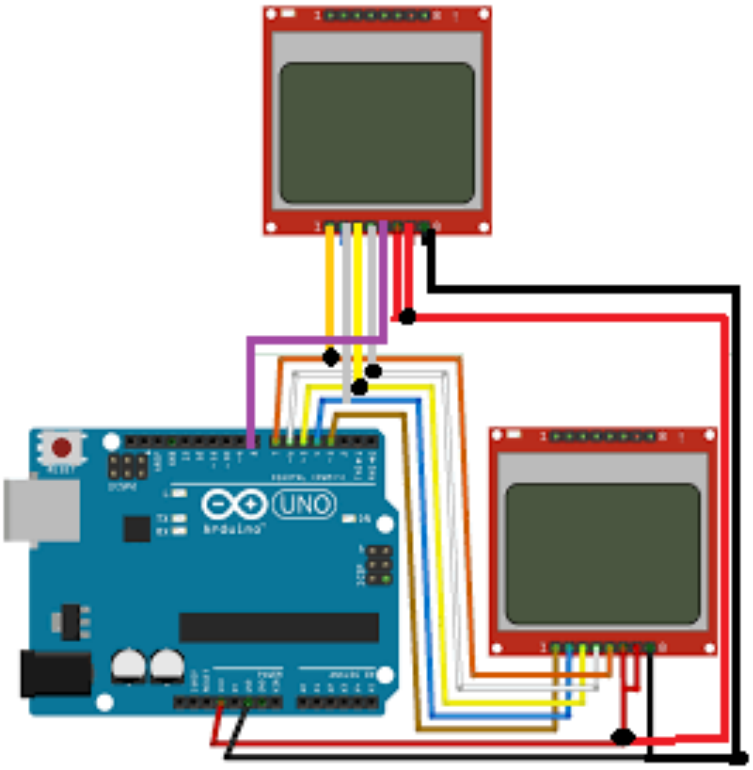

Figure 2.5. Schéma général de branchement de deux afficheurs LCD Nokia 5110 sur l'Arduino

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.

```
#define LCD_CLK  7
#define LCD_SDA  6
#define LCD_DC   5
#define LCD_RST  4
#define LCD_SCE  3
#define LCD_DIN  2

byte sendData(byte _val , bool DC) {
byte ret = 0;
digitalWrite(LCD_DC, DC);
digitalWrite(LCD_SCE, LOW);
for(int i=7; i>=0; i--) {
digitalWrite(LCD_CLK, LOW);
digitalWrite(LCD_SDA, bitRead(_val, i));
digitalWrite(LCD_CLK, HIGH);
bitWrite(ret, i, digitalRead(LCD_DIN));
    }
digitalWrite(LCD_SCE, HIGH);
    return ret ;
}

void setCursor(unsignedchar column , unsignedchar line) {
    sendData (0x80 | column, 0);
    sendData (0x40 | line, 0);
}

void clear() {
    setCursor(0, 0);
for (unsignedshort i = 0 ; i <500 ;  i++) sendData(0x00, 1);
    setCursor(0, 0);
}

void setup() {
    pinMode(LCD_CLK , OUTPUT);
    pinMode(LCD_SDA , OUTPUT);
    pinMode(LCD_DC , OUTPUT);
    pinMode(LCD_RST , OUTPUT);
    pinMode(LCD_SCE , OUTPUT);

    digitalWrite(LCD_RST , HIGH);
    digitalWrite(LCD_SCE , HIGH);
    digitalWrite(LCD_RST , LOW);
    delay(100);
    digitalWrite(LCD_RST , HIGH);

    sendData (0x21, 0); // extended instruction set control (H=1)
    sendData (0x13, 0); // bias system (1:48)
    sendData (0xc2, 0);  // default Vop (3.06 + 66 * 0.06 = 7V)
    sendData (0x20, 0); // extended instruction set control (H=0)
    sendData (0x09, 0); // all display segments on

    clear() ;
}

void loop() {
    setCursor(0, 0);
    sendData(0x7F, 1) ;   sendData(0x88, 1) ;  sendData(0x88, 1) ;   sendData(0x7F, 1) ;
}
```

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]

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.

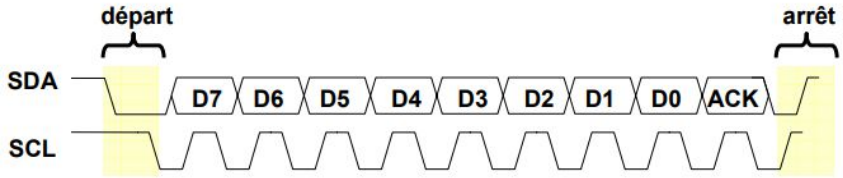

Figure 3.1. Exemple de transmission via le protocole I2C [1]

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.

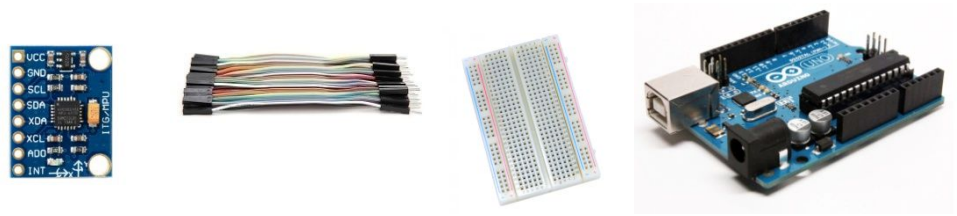

Figure 3.2. Matériel utilisé pour interfacer un afficheur LCD via le protocole SPI

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.

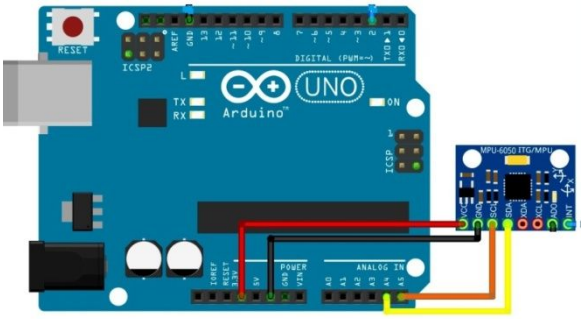

Figure 3.3. Schéma général de branchement du module MPU6050 sur l'Arduino

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 :

$$\text{Deg}_y = \tan^{-1}\left(\frac{-Accel_x}{\sqrt{Accel_y{}^2 + Accel_z{}^2}}\right) \times \frac{180}{\pi} \quad (5.1)$$

$$\text{Deg}_x = \tan^{-1}\left(\frac{Accel_y}{\sqrt{Accel_x + Accel_z{}^2}}\right) \times \frac{180}{\pi} \quad (5.2)$$

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.

```
#include <Wire.h>

#define RAD2DEG 180/3.14159

unsigned long last_time = 0 ;
float base_x_accel  , base_y_accel  , base_z_accel ;

typedef struct accel_gyro_t {
int Ax , Ay , Az , Tmp , Gx , Gy , Gz;
};

accel_gyro_t  MPU6050_read_values() {
    accel_gyro_t  ret;
Wire.beginTransmission(0x68);
Wire.write(0x43);
Wire.endTransmission(false);
Wire.requestFrom(0x68 , 6 , true);
    ret.Gx  = Wire.read()<<8 | Wire.read();
    ret.Gy  = Wire.read()<<8 | Wire.read();
    ret.Gz  = Wire.read()<<8 | Wire.read();
return ret;
}

void  setup() {
Serial.begin(9600);
Wire.begin();
Wire.beginTransmission(0x68);
Wire.write(0x6B);
Wire.write(0);
Wire.endTransmission(true);

for (int i = 0; i <10; i++) {
        accel_gyro_t  data = MPU6050_read_values();
        base_x_accel  +=  data.Gx;   base_y_accel  +=  data.Gy;    base_z_accel  +=  data.Gz;
        delay(100);
   }
    base_x_accel  /=  10;    base_x_accel  /=  10;    base_x_accel  /=  10;
}

void  loop() {
   accel_gyro_t raw = MPU6050_read_values();
unsigned long curr_time = millis();
float gyro_x = (raw.Gx - base_x_gyro)  / 131;
    float gyro_y = (raw.Gy - base_y_gyro)  / 131;
    float gyro_z = (raw.Gz - base_z_gyro)  / 131;

float dt =(curr_time - last_time)/1000.0;
float gyro_angle_x = gyro_x*dt;
float gyro_angle_y = gyro_y*dt;
float gyro_angle_z = gyro_z*dt
    last_time = curr_time ;

    Serial.print(''Gyro X : '');    Serial.println(gyro_angle_x);
    Serial.print(''Gyro Y : '');    Serial.println(gyro_angle_y);
    Serial.print(''Gyro Z : '');    Serial.println(gyro_angle_z);
}
```

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.           returndigitalRead(_scl);
21. }
22.
23. uint8_t  readSda() {
24.           returndigitalRead(_sda);
25. }
26.
```

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

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

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

TP4: Utilisation du bus CAN

1. Introduction

Dans ce TP, nous abordons un des bus utilisés dans le milieu industriel et qui est tolérant aux perturbations dues aux bruits. Plus particulièrement, nous utilisons le bus Can qui est à l'origine un bus destiné pour communiquer les équipements de l'automobile. Dans ce TP, nous utilisons la carte Arduino avec un module CAN pour faire communiquer deux Arduino via le bus CAN, dans l'absence d'équipements industriels pour les interfacer avec l'Arduino.

En réalisant le TP, l'étudiant aura la capacité de communiquer (lire et écrire des données) avec un tableau de bord d'un véhicule, automate ou autre dispositif utilisant le bus CAN.

2. Principe de fonctionnement du bus CAN

Le bus CAN est un bus système qui permet une transmission sérielle, et il est fréquemment utilisé dans l'industrie, notamment le domaine de l'automobile. Il permet de raccorder plusieurs équipements et unités de contrôle sur un seul bus ayant seulement deux lignes, où la communication se fait par tour de rôle [1]. L'autre avantage de ce bus est de permettre une communication de longue distance qui se diffère selon le débit de transmission.

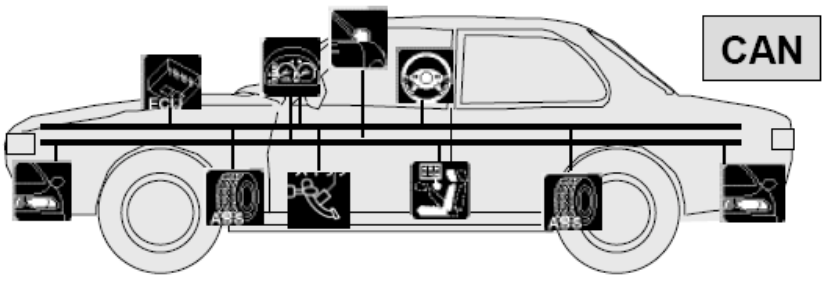

Figure 4.1. Interconnexion de plusieurs capteurs et actionneurs via le bus CAN [2]

Comme mentionné précédemment, la communication via le bus CAN est basée sur deux fils (CAN High/CanH et CAN Low/CanL), et ceux-ci engendre un signal différentiel. En d'autres termes, la lecture des bits transmis se fait par la mesure de la différence de tensions entre les deux fils. Vu la perturbation électromagnétique altère les deux fils, la différence de tension reste identique [1], et c'est pour cette raison les véhicules (beaucoup de bruitages)

utilisent le bus CAN.

Par défaut, en repos la tension des deux fils (CanH et CanL) est mise à 2.5v, et elle se change dans le cas de transmission. Plus spécifiquement, lors de transmission du bit 0 (un bit dominant), la tension de CanH augmente de 1v et la tension de CanL diminue avec 1v, et donc, pour transmettre 0, CanH devient 3.5v et CanL devient 1.5v. Tandis que pour transmettre un bit 1 (bit récessif), les tensions gardent la valeur initiale (2.5v).

Il se peut qu'il y ait une perturbation dans les fils, ce qui engendra une différence de tensions au moment du repos. A cet égard, le bus CAN contient un dispositif **transceiver** qui vérifie les signaux de CanH et CanL s'ils sont déclenchés ou non. En outre, il y a une résistance de 120 Ohm (deux résistances de 60 Ohm de chaque côté) entre les deux fils CanH et CanL pour éliminer les parasites aux réfléchissements des signaux sur les extrémités.

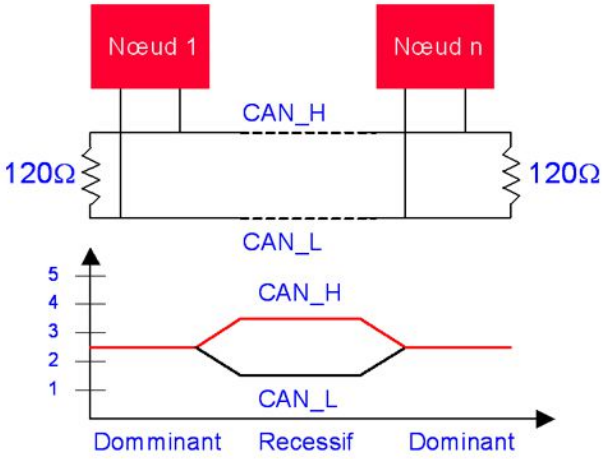

Figure 4.2. Illustration de différence de tension dans la communication CAN [3]

Le bus CAN est basé sur le mécanisme de diffusion (broadcasting) des paquets à tous les dispositifs connectés, et cela permet à tout le monde de lire les données circulantes. Il y a deux types de paquets tels que le paquet standard et le paquet étendu [4], où chaque paquet a une taille différente (CAN standard 11 bits et CAN étendu 29 bits) et une structure contenant des informations, i.e. ID, CRC, ACK, etc. Pour plus d'informations sur le principe de fonctionnement et le détail des paquets, l'étudiant doit consulter le chapitre '*Bus CAN*' du cours '*Bus de communication et réseaux locaux industriels*'.

3. Matériel utilisé dans ce TP

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

- Deux cartes Arduino UNO ou Arduino Mega ;
- Deux modules CAN Bus ;
- Des fils de prototypage male-femelle (*male-male en cas d'utiliser une plaque d'essai*) ;

Figure 4.3. Matériel utilisé pour faire communiquer deux Arduino via le bus CAN

4. Communication entre deux Arduino via le bus CAN

Dans ce TP, nous allons faire communiquer deux Arduino via le bus CAN vu que les équipements de l'automobile ne sont pas à la porter de tout le monde pour les interfacer. Et pour ce faire, on a besoin de deux Arduino Uno, deux modules CAN Bus MCP2515 et des fils de prototypages.

Le module MCP2515 met en place une interface SPI pour pouvoir interagir avec les cartes de développement qui ne disposent pas de bus CAN comme l'Arduino. Donc, la communication entre l'Arduino et le module se fait à travers le protocole SPI, et le module traduit les commandes et les données reçues en paquets CAN pour les envoyer dans le bus CAN.

Le module MCP2515 est équipé d'un circuit MCP2551 qui gère les signaux CanH et CanL (transceiver), ainsi ce circuit est alimenté par 5v et permet de connecter jusqu'à 112 nœuds sur le bus.

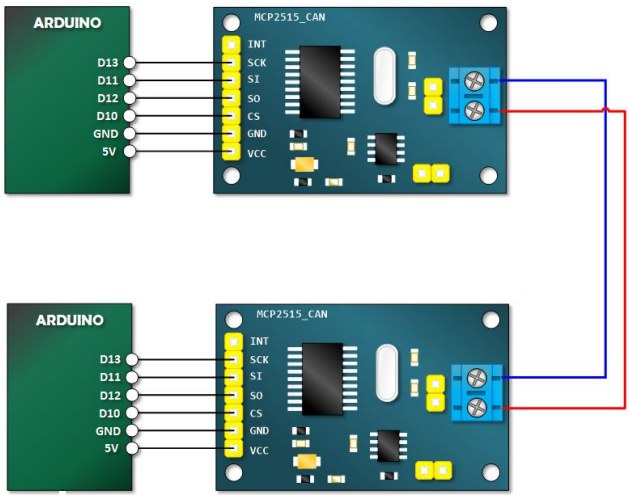

Figure 4.4. Schéma de branchement des modules MCP2515 sur les Arduino

Donc, afin de réaliser ce TP, on a besoin de deux Arduino Uno, des fils de prototypage et deux modules MCP2515. On essaie de lire un message à partir du ***Serial Monitor*** du premier Arduino, le transmettre au deuxième Arduino via le bus CAN et l'afficher dans le ***Serial***

monitor du deuxième Arduino. Par conséquent, on relit le port VCC du module à la sortie 5v de l'Arduino, le GND dans le GND, le CS dans le port 10, le SO dans le port 12, le SI dans le port 11 et le SCK dans le port 13 (Figure 4.4). Il est impératif de vérifier si les deux fils de CAN (rouge et bleu) ne sont pas inversés, sinon la communication ne se fera pas.

Etant la bibliothèque[1] d'interfaçage du module MCP2515 n'est pas disponible par défaut sur l'éditeur Arduino, on doit la télécharger et l'installer. Ensuite, le premier Arduino doit être programmé en étant émetteur et l'autre en étant récepteur.

Tableau 4.1. Liste des vitesses de transmission utilisées par MCP2515

Vitesses de transmission
CAN_5KBPS, CAN_10KBPS, CAN_20KBPS, CAN_31K25BPS, CAN_33KBPS, CAN_40KBPS, CAN_50KBPS, CAN_80KBPS, CAN_83K3BPS, CAN_95KBPS, CAN_100KBPS, CAN_125KBPS, CAN_200KBPS, CAN_250KBPS, CAN_500KBPS, CAN_1000KBPS

```
1.  #include <SPI.h>
2.  #include <mcp2515.h>
3.  struct can_frame canMsg;
4.  char buffer[8] ;
5.  MCP2515 mcp2515(10);
6.
7.  void setup()  {
8.      Serial.begin(9600);
9.      SPI.begin();
10.
11.     mcp2515.reset();
12.     mcp2515.setBitrate(CAN_500KBPS , MCP_8MHZ);
13.     mcp2515.setNormalMode();
14. }
15.
16. void loop()  {
17. while(Serial.available())   {
18. Serial.readBytes(buffer, 8) ;
19.         canMsg.can_id  = 0x036;
20.         canMsg.can_dlc = 8;
21.
22.         memcpy(canMsg.data , buffer , 8) ;
23.
24.         mcp2515.sendMessage(&canMsg);
25.         delay(1000);
26.     }
27. }
```

Figure 4.5. Code source d'envoi un message via le bus CAN en utilisant MCP2515

La première chose à faire avant d'utiliser le module MCP2515 consiste à inclure les bibliothèques nécessaires (SPI et MCP2515), et configurer le module en choisissant le numéro du port CS du module (pratiquement 10), puis en précisant la vitesse de transmission parmi une liste prédéfinie (Tableau 4.1). En outre, on précise la vitesse de l'horloge que le module doit utiliser telle que 8Mhz, 16Mhz et 20Mhz (ligne 12 Figure 4.5). Consécutivement, on définit le mode de communication (ligne 13 Figure 4.5).

[1] *https://github.com/autowp/arduino-mcp2515/archive/master.zip*

```
1.  #include <SPI.h>
2.  #include <mcp2515.h>
3.  struct can_frame canMsg;
4.  char buffer[8] ;
5.  MCP2515 mcp2515(10);
6.
7.  void setup()  {
8.      Serial.begin(9600);
9.      SPI.begin();
10.
11.     mcp2515.reset();
12.     mcp2515.setBitrate(CAN_500KBPS , MCP_8MHZ);
13.     mcp2515.setNormalMode();
14. }
15.
16. void loop() {
17.     if (mcp2515.readMessage(&canMsg) == MCP2515::ERROR_OK)  {
18.         memcpy(buffer , canMsg.data , 8) ;
19.         Serial.println(buffer) ;
20.     }
21. }
```

Figure 4.6. Code source de réception d'un message via le bus CAN en utilisant MCP2515

Après avoir configuré le module, l'émetteur prépare un paquet de données (de taille 11 octets) représenté par une structure de données (4 octets ID, 1 octet DLC et 8 octets DATA). Donc, on lit le message tapé sur le ***Serial Monitor*** et on le sauvegarde dans les 8 octets de DATA du paquet CAN pour le transmettre sur le bus en utilisant la fonction **sendMessage** de l'objet MCP2515. Contrairement dans le récepteur, on utilise la fonction **readMessage** pour lire les données reçues dans le bus et on les affiche dans le ***Serial Monitor*** du deuxième Arduino.

Références

[1] Découverte du bus CAN. *https://arduino103.blogspot.com/2018/04/canbus-la-decouverte-du-bus-can-et-du.html* [Consulté en 10/2019]

[2] Exposé sur le Bus CAN. *http://igm.univ-mlv.fr/~dr/XPOSE2009/BusCAN/intro_can.html*[Consulté en 10/2019]

[3] Cours Systèmes Embarqués: Le Bus CAN**.** *https://www.technologuepro.com/cours-systemes-embarques/cours-systemes-embarques-Bus-CAN.htm* [Consulté en 10/2019]

[4] Tutoriel sur la lecture des données d'automobiles via le bus CAN. *https://publicism.info/engineering/penetration/3.html* [Consulté en 10/2019]

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

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

TP5: Utilisation du protocole ModBus

1. Introduction

Dans ce TP, nous abordons un autre bus de communication industrielle évolué et un nouveau protocole de communication. En particulier, nous utilisons le bus RS485 avec le protocole ModBus pour interagir avec un compteur de consommation électrique supportant le même protocole. Le but de ce TP consiste à initier l'étudiant pour faire un montage de bonne gestion de consommation électrique (consommation intelligente).

Comme le compteur de consommation fonctionne en 220v, il est hautement recommander de prendre toutes les précautions afin d'éviter tout dégât (un choc électrique). En particulier, on doit porter des gants d'électricité et on veuille à respecter les directives de raccordement.

2. Principe de fonctionnement du protocole ModBus

Le protocole ModBus est un protocole libre créé par la société Modicon et est basé sur une structure hiérarchisée entre le maître et les esclaves [1]. La communication via ModBus est une communication half-duplex utilisant deux ou quatre fils qui permettent de transmettre 9600-19200 bits/s, où on distingue deux scénarios de communication. Le premier scénario consiste à transmettre un message par le maître et attendre la réponse de l'esclave, tandis que le deuxième consiste à diffuser un message à tous les esclaves sans attendre une réponse. Il est important de noter que les esclaves ne peuvent pas communiquer entre eux, et chaque esclave demande une permission auprès du maître pour envoyer la réponse.

START	ADDRESS	FUNCTION	DATA	CRC CHECK	END
T1–T2–T3–T4*	8 BITS	8 BITS	*n* x 8 BITS	16 BITS	T1–T2–T3–T4*

*For T1–T2–T3–T4, 3.5 character times at no communication.

Modbus Frame Structure-RTU Mode

Figure 5.1. Structure d'un message du protocole ModBus [2]

Le message échangé contient quatre champs tels que le numéro de l'esclave codé sur 1 octet (0-64), l'instruction à exécuter codée sur 1 octet (19 instructions possibles), les données

composées de plusieurs caractères et le contrôle d'erreur CRC [1]codé sur 2 octets. On note que l'adresse 0x00 est réservée au mode de diffusion, ainsi la taille maximale des données est 256 octets.

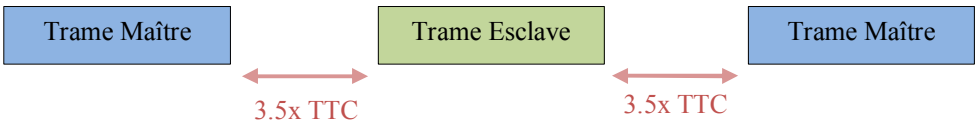

Figure 5.2. Echange de données entre le maître et l'esclave. TTC signifie le temps de transmission d'un caractère

Chaque message est délimité par un silence de minimum 3.5 fois le temps de transmission d'un caractère (TTC). Au cours de la transmission, s'il y a un délai de 1.5 TTC entre la transmission de deux caractères successifs, le système détecte une erreur de transmission. Un caractère en ModBus est une suite de 11 bits dont 1 bit de début, 8 bits de données, 1 bit de parité et 1 bit d'arrêt.

Le protocole ModBus est basé sur la transmission sérielle et fonctionne à base de RS232, RS422, RS485 et Ethernet TCP/IP, où chaque support de transmission a des avantages et des inconvénients. Comme nous nous intéressons au RS485, il offre la possibilité de communiquer avec 32 périphériques sur une distance de 1200m maximum, ainsi il est tolérant aux parasites.

Tableau 5.1. Liste des codes de fonctions utilisées par le protocole ModBus [1]

Code	Nature des fonctions MODBUS
H'01'	Lecture de n bits de sortie consécutifs
H'02'	Lecture de n bits de sortie consécutifs
H'03'	Lecture de n mots de sortie consécutifs
H'04'	Lecture de n mots consécutifs d'entrée
H'05'	Ecriture de 1 bit de sortie
H'06'	Ecriture de 1 mot de sortie
H'07'	Lecture du statut d'exception
H'08'	Accès aux compteurs de diagnostic
H'09'	Téléchargement, télé déchargement et mode de marche
H'0A'	Demande de CR de fonctionnement
H'0B'	Lecture du compteur d'événements
H'0C'	Lecture des événements de connexion
H'0D'	Téléchargement, télé déchargement et mode de marche
H'0E'	Demande de CR de fonctionnement
H'0F'	Ecriture de n bits de sortie
H'10'	Ecriture de n mots de sortie
H'11'	Lecture d'identification
H'12'	Téléchargement, télé déchargement et mode de marche
H'13'	Reset de l'esclave après erreur non recouverte

Il est possible de communiquer avec 248 périphériques, mais en utilisant des répéteurs sur le même réseau. Vu que la transmission par défaut est basée sur deux fils, on peut utiliser quatre fils pour passer en mode full-duplex qui permet d'obtenir un débit de transmission plus

[1]CRC est l'abréviation de *Cyclic Redundancy Control*

rapide.

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 compteur de consommation électrique SDM230-MODBUS de la marque EASTRON;
- Un module RS485 pour Arduino ;
- Des fils de prototypage male-femelle (*male-male en cas d'utiliser une plaque d'essai*) ;
- Un câble électrique de section 1.5mm ou 2.5mm pour raccorder le compteur à la prise ;
- Un équipement quelconque fonctionnant en 220v comme un ampoule.

Figure 5.3. Matériel utilisé pour faire communiquer deux Arduino via le bus CAN

4. Interfaçage d'un compteur d'énergie monophasé via le protocole ModBus

Dans ce TP, nous interfaçons un compteur de consommation électrique avec un Arduino via le protocole ModBus, et nous essayons de manipuler le compteur à distance. Donc, on a besoin d'une carte Arduino, un module MAX485 qui convertit de RS485 ModBus vers un signal TTL[2] et vice versa, des fils de prototypage et un compteur d'énergie dotée d'une interface ModBus comme le modèle SDM230 de la marque EASTRON. Donc, dans ce TP, l'Arduino sert comme maître et le compteur est utilisé comme esclave.

La première étape à faire consiste à relier l'Arduino avec le module MAX485, et pour ce faire on alimente le module (VCC) avec 5v de l'Arduino et on branche le GND du module

[2]TTL est l'abréviation de *Transistor-to-Transistor Logic*

avec celui de l'Arduino. Ensuite, on branche le port A (ligne non inversée) dans le TX+/RX+ du compteur et le port B (ligne inversée) dans le TX-/RX- du compteur.

On branche les quatre ports qui restent tels que R0 (receiver output), RE (receiver enable), DE (driver enable), DI (driver input) dans quatre ports numériques de l'Arduino (e.g. 0, 3, 3 et 1, respectivement). On remarque qu'on a utilisé les ports du bus UART avec lequel l'Arduino communique avec le ***Serial Monitor*** de l'ordinateur. Dans le cas où on interagit avec ce dernier, il est préférable d'utiliser la bibliothèque ***SerialSoftware*** pour émuler le bus UART dans des ports différents (e.g. ports 4 et 5).

Figure 5.4. Compteur de consommation électrique SDM230-MODBUS de la marque EASTRON

Chacun des quatre ports correspond à un usage. Par exemple, le port R0 est destiné à lire les données entrantes, DI pour envoyer des données sortantes, RE pour activer le mode de réception et DE pour activer le mode d'envoi. On note qu'on a utilisé le même port numérique (numéro 3) pour RE et DE, car lorsque RE est mis à LOW le mode de réception est activé, tandis que lorsque DE est mis à HIGH le mode d'envoi est activé.

Figure 5.5. Module MAX485 ModBus qui convertit du RS485 vers TTL et vice-versa

Pour interfacer le module ***MAX485*** afin de recevoir et envoyer des données, on a deux possibilités, soit on utilise la bibliothèque ***ModBusRtu***[3] qui doit être téléchargée, ou bien on envoie les paquets octet par octet en utilisant l'objet ***Serial***. En référant à la liste des codes des fonctions utilisées dans le protocole ModBus (Tableau 7.1), le code 0x04 est dédié pour lire une suite de caractères consécutifs d'entrée. Donc, on envoie un paquet contenant l'adresse de

[3] *https://github.com/smarmengol/Modbus-Master-Slave-for-Arduino/blob/master/ModbusRtu.h*

l'esclave (0x01 par défaut) et le code de la fonction de lecture.

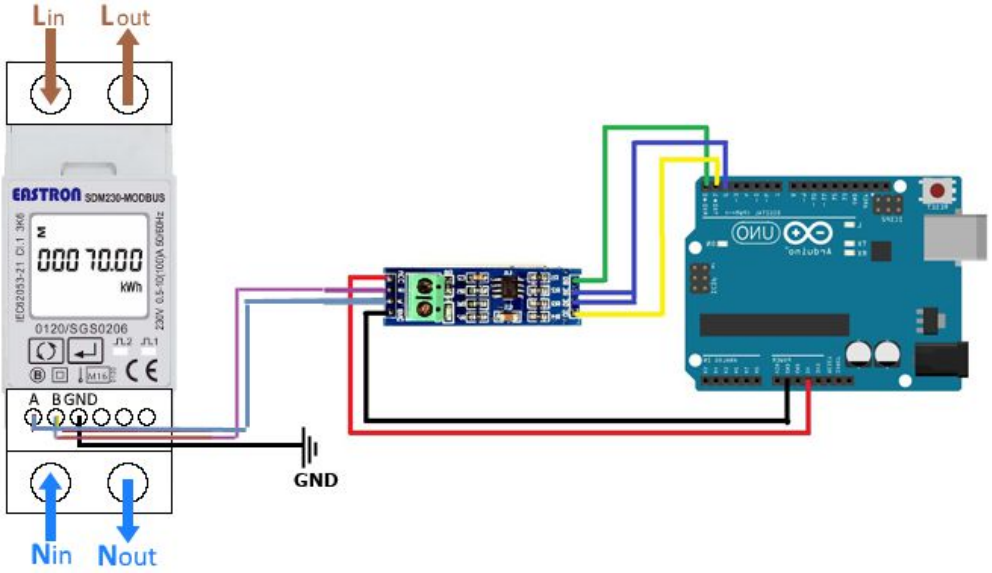

Figure 5.6. Schéma de branchement du module MAX485 et le compteur SDM230 sur l'Arduino

Adresse d'esclave	Code fonction	Adresse de début (Hi)	Adresse de début (Lo)	Nombre de points (Hi)	Nombre de points (Lo)	CRC (Hi)	CRC (Lo)

Figure 5.7. Structure d'un paquet d'une requête

Afin de transmettre le paquet de commande de lecture, on active le mode d'écriture avec l'état HIGH dans DE et systématiquement on bloque la lecture (RE = HIGH). Puis, en se référant au manuel[4] du constructeur du module, on prépare le paquet contenant 8 octets dont la structure illustrée par la Figure 5.7. Il faut savoir toutefois que le module EASTRON SDM230 accepte seulement quelques codes de fonctions tels que 0x03, 0x04, 0x08 et 0x0F. Par conséquent, le module répond avec 9 octets dont la structure illustrée par la Figure 5.8, ainsi pour recevoir le paquet on doit mettre LOW dans RE.

Adresse d'esclave	Code fonction	Nombre d'octets	Premier registre (Hi)	Premier registre (Lo)	Second registre (Hi)	Second registre (Lo)	CRC (Hi)	CRC (Lo)

Figure 5.8. Structure d'un paquet de réponse à une requête

A titre d'exemple, pour lire le voltage d'entrée (alternatif) à partir du module SDM230, on met l'adresse 0x01 comme adresse d'esclave, 0x04 comme code de fonction pour lire plusieurs caractères consécutifs d'entrée (Tableau 5.1), 0x00 0x00 comme adresse de début d'après le manuel (lire le voltage), 0x00 0x02 comme nombre de registres à lire (2 registres à lire) et enfin la valeur de CRC (e.g. 0x71, 0xCB).

[4]Manuel d'utilisation de ModBus avec SDM230. *https://www.desouttertools.com/resource-centre/file/download/126505189*

0x01	0x04	0x00	0x00	0x00	0x02	0x71	0xCB

Figure 5.9. Exemple de requête de lecture du voltage d'entrée via le protocole ModBus

```
1.  #define RE_DE 4
2.
3.  int Buffer[9];
4.  byte VoltageReadCommand[] ={0x01 , 0x04 , 0x00 , 0x00 , 0x00 , 0x02 , 0x71 , 0xCB};
5.
6.  void setup()
7.  {
8.  Serial.begin(2400);
9.
10. pinMode(RE_DE , OUTPUT);
11. digitalWrite(RE_DE , HIGH);
12. }
13.
14. void loop()
15. {
16. digitalWrite(RE_DE , HIGH);
17.
18. for (int i=0; i <8; i++) Serial.write( VoltageReadCommand[i] );
19. delay(50);
20.
21. digitalWrite(RE_DE , LOW);
22.
23. while(Serial.available() <9);
24. for (int i=0; i <9; i++) Buffer[i] = Serial.read();
25.
26. Serial.flush();
27. Serial.println();
28.
29.     Serial.print(Buffer[3] , HEX); Serial.print("");
30. Serial.print(Buffer[4] , HEX); Serial.print("");
31. Serial.print(Buffer[5] , HEX); Serial.print("");
32. Serial.print(Buffer[6] , HEX); Serial.print("");
33.     Serial.println();
34.
35. float x;
36.     ((byte*) &x)[3]= Buffer[3];
37.     ((byte*) &x)[2]= Buffer[4];
38.     ((byte*) &x)[1]= Buffer[5];
39.     ((byte* )&x)[0]= Buffer[6];
40.
41. Serial.print(x , 2);
42. Serial.println("Volts");
43. delay(500);
44. }
```

Figure 5.10. Code source de lecture du voltage d'entrée à partir du module SDM230

Après la réception du paquet de réponse, on extrait les quatre octets représentant le voltage et on les converti en valeur flottante (lignes 35-39 Figure 5.10), car le courant alternatif n'est pas stable et contient la virgule.

Références

[1] Liaison série Modbus RS485. Cours réseau de terrain, Université de Grenoble

[2] Format du message ModBus. *https://www.rfwireless-world.com/Terminology/Modbus-message-frame.html* [Consulté en 10/2019]

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

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

TP Supplémentaire: Commande des moteurs par Arduino

1. Introduction

Dans ce TP, nous allons contrôler différents types de moteurs dotés de différentes technologies tels que le moteur à courant continu, le servo-moteur, le moteur brushless et le moteur pas-à-pas. Généralement, chaque type de moteur nécessite un driver spécial pour le commander, à l'exception des servo-moteurs qui embarquent le driver à l'intérieur du boitier.

2. Commande d'un moteur à courant continu

Le moteur à courant continu (ou DC réfère à *Direct Current*) est un moteur qu'on trouve un peu partout, notamment dans les jouets des enfants (véhicules et autres) vu son coût très faible. Ce type de moteur est doté de deux fils seulement, et dés qu'on branche l'alimentation sur les deux fils il commence à tourner à une vitesse rapide. Toutefois, lorsque l'alimentation est inversée (le plus + de l'alimentation dans le moins – du moteur et le moins – de l'alimentation dans le plus + du moteur) le moteur tourne dans le sens inverse.

Chaque moteur a un voltage maximum à supporter (sinon il se brûle) et auquel il tourne à sa grande vitesse. Malheureusement, on ne peut pas varier la vitesse de rotation s'il est branché directement dans l'alimentation, ni de changer le sens de rotation en cours de marche. Le seul moyen de contrôler la vitesse est d'abaisser le voltage d'entrée, ce qui requiert une alimentation variable qui est défavorable.

Figure 1. Moteur à courant continu

Donc, pour contrôler le sens de rotation on utilise généralement un circuit électronique qui s'appelle '*pont-H*' qui est constitué de quatre éléments de commutation, i.e. généralement des transistors. La solution la plus pratique consiste à utiliser un circuit intégré basé sur un

pont-H et encastré dans un module (appelé *driver* ou *shield*) tel que L298N, L293D, MX1508, etc.

Les modules à base de *pont-H* ne permettent pas seulement le contrôle du sens, mais également la vitesse de rotation grâce au signal PWM[1]. Ce dernier est un signal pseudo-analogique, ce qui veut dire un signal analogique (0 et 1) avec une fréquence fixe. Le principe du signal PWM consiste à générer des impulsions égales ou inférieures à la période. En outre, plus la durée d'impulsion se rapproche de la période plus l'actionneur (i.e. moteur, ampoule, etc.) converge vers un fonctionnement à sa grande puissance (ou vitesse).

Figure 2. Exemple de différents drivers à base de pont-H. A gauche un driver L298N, au centre un driver L293D et à droite un driver MX1508.

En général, le cycle de service (pourcentage) de fonctionnement peut être calculé en divisant la durée d'impulsion T_0 par la période T et en multipliant le résultat par 100. Le signal PWM doit être généré à une fréquence de 20 KHz (i.e. T= 50µs) au minimum pour éviter d'être dans la gamme audible (perçue par l'être humain).

$$\text{Pourcentage} = \frac{T_0}{T} \times 100 \qquad (3.1)$$

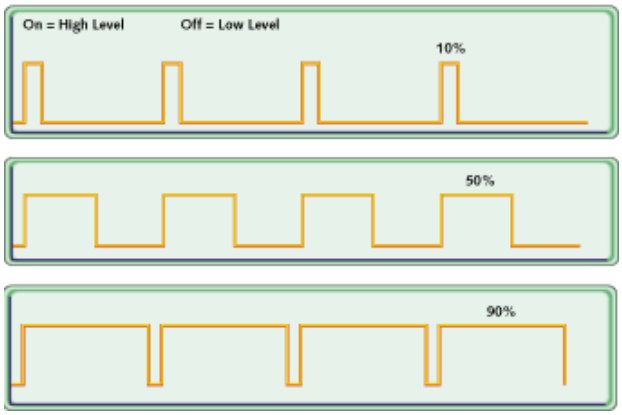

Figure 3. Exemple de différents signaux PWM [1]

[1]PWM est l'abréviation de *Pulse Modulation Width*

Par exemple, pour une fréquence de 20 KHz correspondant à une période de 50µs, si la vitesse est 90% de la vitesse maximale (90% de cycle de service), on envoie une impulsion pendant 45µs (Equation 3.2) et les 5µs restantes on envoie 0 (Figure 3.3).

$$\begin{cases} 50\mu s \rightarrow 100\% \\ t \rightarrow 90\% \end{cases} \Longrightarrow t = \frac{90\% \times 50\mu s}{100\%} = 45\mu s \qquad (3.2)$$

Alors, pour générer un signal PWM en Arduino on a deux solutions, où la première consiste à implémenter le mécanisme softwarement (émulation par programme) en envoyant une impulsion pendant une durée de temps et un repos pour le reste du temps. Par contre, la deuxième solution consiste à bénéficier du PWM implémenté hardwarement dans quelques ports de l'Arduino Uno, et utiliser une fonction qui facilite la programmation. La fonction qui génère le signal PWM est ***analogWrite***, où il est préférable de l'utiliser avec un port doté de PWM. Par exemple si on veut un cycle de service 50%, on passe 127 à la fonction (255/2 = 127) et ainsi de suite. On rappelle que cette fonction accepte une valeur comprise entre 0-255, ce qui exige un mappage du pourcentage dans cet intervalle.

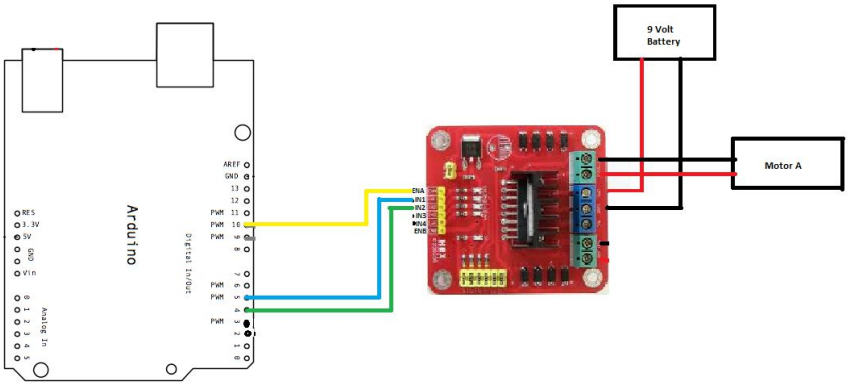

Figure 4. Schéma de branchement du moteur DC, driver L298N et l'Arduino

Dans ce TP, on va commander un moteur à courant continu avec un driver L298N, et celui-ci permet de contrôler deux moteurs dans deux directions ou quatre moteurs dans une seule direction. Donc, on utilise au total trois ports de l'Arduino configurés en tant que sorties (signal sortant de l'Arduino), dont deux entre eux sont destinés pour contrôler la direction et un pour contrôler la vitesse de rotation (PWM).

On branche les deux fils du moteur dans le compartiment A, et les deux fils de l'alimentation dans GND (-) et VMS (+). En outre, on relit les trois fils sortant de l'Arduino avec IN1, IN2 et ENABLE du moteur A (Figure 3.4). Enfin, on tape le code de la Figure 3.5 pour contrôler le moteur dans deux directions, où il est tourné à sa vitesse maximale dans les deux directions, puis à moitié de vitesse maximale.

3. Commande d'un servo-moteur

Le servo-moteur est un cas particulier des moteurs à courant continu, où il peut tourner

degré par degré. Il est constitué principalement de moteur à courant continu et d'un ensemble d'engrenages réducteurs de vitesse, ainsi une carte électronique de contrôle de l'énergie et un capteur de position (capteur rotatif). La plupart des servo-moteurs tournent entre 0° et 180°, mais dans des cas particuliers on trouve des servo-moteurs qui tournent 360°.

```
#define ENABLE_A 10
# define IN1_A 5
# define IN2_A 4

void setup() {
pinMode(ENABLE_A, OUTPUT) ;
pinMode(IN1_A, OUTPUT) ;
pinMode(IN2_A, OUTPUT) ;
}

void loop() {
digitalWrite(ENABLE_A, 255) ;
analogWrite(IN2_A, LOW) ;
    analogWrite(IN1_A, HIGH) ;
delay(1000) ;
    analogWrite(IN1_A, LOW) ;
analogWrite(IN2_A, HIGH) ;
delay(1000) ;

    digitalWrite(ENABLE_A, 127) ;
analogWrite(IN2_A, LOW) ;
analogWrite(IN1_A, HIGH) ;
    delay(1000) ;
    analogWrite(IN1_A, LOW) ;
analogWrite(IN2_A, HIGH) ;
delay(1000) ;
}
```

Figure 5. Code source de contrôle d'un moteur à courant continu avec un driver L298N

Figure 6. Servo-moteur SG90 de la marque TowerPro

Un servo-moteur est équipé de trois fils, dont deux sont dédiés pour l'alimentation et un pour recevoir un signal. En particulier, le fil marron (ou noir) est le GND, le fil rouge (ou orange) est le VCC (ou +5v) et le fil jaune est celui qui reçoit le signal à partir d'un port numérique. Il faut noter que la majorité des servo-moteurs fonctionnent avec 5 volts max (sinon ils se brûlent), sauf les servo-moteurs industriels (gros servo-moteurs).

On note également que plus le couple du servo est élevé plus celui-ci consomme un courant élevé. Néanmoins, il est préférable d'alimenter toujours le servo avec une alimentation externe dotée d'un ampérage élevé, notamment lorsqu'on a plusieurs servos ou un servo à grand couple. En outre, il faut toujours brancher le négatif de l'alimentation externe avec celui de l'Arduino.

Pour manipuler les servos avec l'Arduino, on utilise la bibliothèque <Servo.h>. D'après le référentiel officiel de l'Arduino, l'utilisation de cette bibliothèque désactive le fonctionnement de PWM sur les ports 9 et 10 des Arduino Uno [2], que ce soit des servos branchés sur ces ports ou non.

Pour configurer le port sur lequel le servo est branché, il suffit d'utiliser la fonction dédiée **attach(port)** de l'objet **Servo** au lieu de **pinMode**. Ensuite, pour tourner le servo à un angle donné, on utilise la fonction **write(angle)** de l'objet **Servo**. Malheureusement, cette fonction n'est pas fiable avec tous les servos, car certains constructeurs ne respectent pas les normes, et par conséquent le servo ne sera pas positionné correctement. Par exemple, si on le positionne à 90°, il peut être positionné à moins de cette valeur ou plus (même avec une virgule). Comme une solution alternative, la communauté a mis en disposition une autre fonction qui travaille en bas niveau du signal envoyé, c'est la fonction **writeMicroseconds(microseconds)**de l'objet **Servo**. La majorité des servos qui suivent les normes fonctionnent entre 1000µs (0°) et 2000µs (180°). Donc, on fait la règle des trois pour trouver le nombre de micro secondes à envoyer correspondant à l'angle voulu (e.g. 90° = 1500µ).

```
1.  #include<Servo.h>
2.  #define PIN_SERVO 12
3.
4.  Servo mon_servo ;
5.
6.  void setup() {
7.      mon_servo.attach(PIN_SERVO) ;
8.  mon_servo.write(0) ;
9.  }
10.
11. void loop() {
12.     mon_servo.write(90) ;
13. delay(1000) ;
14.
15.     mon_servo.writeMicroseconds(2000) ;
16. delay(1000) ;
17. }
```

Figure 7. Code source de manipulation d'un servo-moteur avec deux fonctions différentes

Après l'exécution du code de la Figure 3.7, nous voyons que le servo se positionne à l'angle 0° (s'il n'est pas dans cet angle par défaut). Ensuite, il se positionne à l'angle 90° et **180**° consécutivement, et comme la fonction *loop* est une boucle infinie, le servo répète les mêmes actions continuellement. Dans le cas où le servo ne tourne pas 180°, on change la valeur 2000 (ligne 15 Figure 3.7) jusqu'à tomber sur le nombre exact.

Nous remarquons que le servo bouge d'un angle à un autre à une vitesse assez rapide

(presque non percevable à l'œil), ce qui causera une instabilité de mouvement du corps attaché au servo (un choc peut endommager le montage ou le servo lui même). Donc, il est conseillé de bouger le servo degré par degré jusqu'à atteindre l'angle souhaité. En outre, il est préférable d'ajouter un petit retard (une dizaine de milli secondes) entre deux degrés successifs (Figure 3.8), bien évidemment dans le cas où il n'y a aucune contrainte de temps dans le système réalisé.

```
1.  #include<Servo.h>
2.  #define PIN_SERVO 12
3.
4.  Servo mon_servo ;
5.
6.  void setup() {
7.      mon_servo.attach(PIN_SERVO) ;
8.  mon_servo.write(0) ;
9.  }
10.
11. void loop() {
12. for(int i=0 ; i<=90 ; i++) {
13.         mon_servo.write(i) ;
14.         delay(10) ;
15.     }
16. }
```

Figure 8. Code source de manipulation d'un servo-moteur degré par degré

4. Commande d'un moteur Brushless

Le moteur Brushless est une évolution du moteur à courant continu dont l'architecture interne est différente. Plus particulièrement, les aimants permanents sont fixes et placés sur le rotor qui tend à suivre un champ magnétique tournant. Le moteur Brushless est équipé de trois fils (trois phases) dont chacun est relié à la tension et à la masse à la fois [3]. En outre, la commande de ce type de moteur nécessite un driver spécial pour passer le courant entre deux fils en alternance, et peut contrôler un moteur triphasé sans capteur en variant la vitesse via le signal PWM.

Figure 9. Exemple de moteur Brushless utilisé dans les drones

Les moteurs Brushless se diffèrent par le nombre de Kv (vitesse) et le voltage de

fonctionnement. Généralement, on utilise le driver ESC[2] qui contient deux fils d'alimentation d'entrée, un fils d'acquisition de signal PWM pour contrôler la vitesse et trois fils de sortie pour alimenter le moteur triphasé.

Figure 10. Driver ESC pour commander un moteur Brushless

Le contrôleur ESC étant doté d'une puce interne, il peut être configurable, notamment la plage d'impulsion (en termes de micro secondes). Alors, il faut noter que la plupart des ESC sont configurés pour fonctionner entre 1000-2000µs. Alors, on peut bénéficier de la bibliothèque <Servo.h> pour contrôler le moteur Brushless en utilisant la fonction **writeMicroseconds**.

```
1.  #include<Servo.h>
2.  #define PIN_BRUSHLESS 11
3.
4.  Servo brushless ;
5.
6.  void setup() {
7.      brushless.attach(PIN_BRUSHLESS) ;
8.  }
9.
10. void loop() {
11.     brushless.writeMicroseconds(1200) ;        // fonctionner à 20% de la vitesse max
12. delay(1000) ;
13.     brushless.writeMicroseconds(1500) ;        // fonctionner à 50% de la vitesse max
14. delay(1000) ;
15.     brushless.writeMicroseconds(2000) ;        // fonctionner à 100% de la vitesse max
16. delay(1000) ;
17. }
```

Figure 11. Code source de commande de moteur Brushless en variant la vitesse

5. Commande d'un moteur pas-à-pas

Enfin, le quatrième type des moteurs fonctionnant en courant continu est le moteur pas-à-pas qui est basé sur une technologie totalement différentes des autres moteurs, ainsi il est doté d'une très grande précision de rotation. Il y a trois types de moteur pas-à-pas tels que moteur à réluctance variable, moteur à aimants permanents et le moteur hybride combinant les deux.

[2]ESC est l'abréviation de *Electronic Speed Controller*

Les trois types sont des moteurs à quatre phases, donc équipés de quatre fils d'alimentation et de contrôle ce qui exige l'utilisation d'un driver spécial. Cependant, le but principal est de contrôler le nombre de pas effectués par le moteur et non la vitesse, car ce type de moteur généralement tourne lentement.

Figure 12. Moteur pas-à-pas Nema17

Il y a une multitude de drivers dédiés qui se diffèrent selon la puissance du moteur (voltage et ampérage), ainsi le nombre de pas par seconde. Dans ce TP, on utilise seulement le driver A4988 qui permet de contrôler des petits moteurs pas-à-pas (typiquement Nema 17), et qui supporte jusqu'à 35v et 2A.

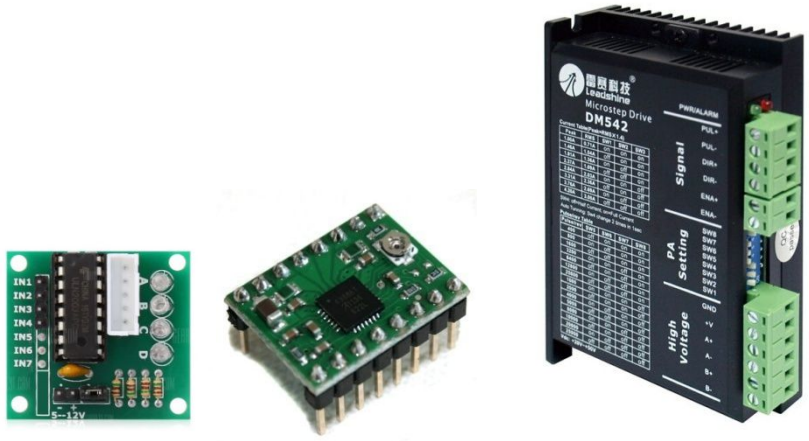

Figure 13. Différentes catégories de driver de moteur pas-à-pas. A droite c'est driver ULN2003, au centre un driver A4988 et à gauche à driver DM542

Pour réaliser ce TP, on a besoin d'une plaque d'essai, des fils de prototypage, un driver A4988, un moteur pas-à-pas, une alimentation 12v et un Arduino Uno. La première chose à faire c'est de brancher le moteur au driver dans les ports A1, A2, B1 et B2. En d'autres termes, on branche le premier pôle du moteur dans A1 et A2, puis le deuxième pôle dans B1 et B2. Ensuite, on connecte le port STP à un port numérique de l'Arduino (exemple port 3) et le port DIR (pour la direction) à un autre port numérique de l'Arduino (exemple port 2). Enfin, la dernière chose à brancher est l'alimentation externe (12v jusqu'à 35v).

Pour faire tourner le moteur à un pas, on envoie une impulsion à STP pendant une durée,

puis on remet le port à 0. Donc, si on veut bouger à plusieurs pas, on fait une boucle. En outre, pour contrôler la direction soit on envoie HIGH ou LOW au port DIR.

```
1.  #define PIN_DIR 2
2.  #define PIN_STEP 3
3.
4.  void setup() {
5.  pinMode(PIN_DIR, OUTPUT);
6.  pinMode(PIN_STEP, OUTPUT);
7.  }
8.
9.  void loop() {
10.   digitalWrite(PIN_DIR, HIGH);
11.
12. for (int i = 0; i <200; i++) {
13. digitalWrite(PIN_STEP, HIGH);
14. delayMicroseconds(2000);
15. digitalWrite(PIN_STEP, LOW);
16. delayMicroseconds(2000);
17.   }
18. }
```

Figure 11. Code source de contrôle de moteur Brushless en variant la vitesse

Références

[1] Introduction au signal PWM. https://barrgroup.com/Embedded-Systems/How-To/PWM-Pulse-Width-Modulation [consulté en 09/2019]

[2] Référence de la bibliothèque Servo. https://www.arduino.cc/en/reference/servo [consulté en 09/2019]

[3] Modélisation des moteurs BrushLess. https://eduscol.education.fr [consulté en 09/2019]