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.

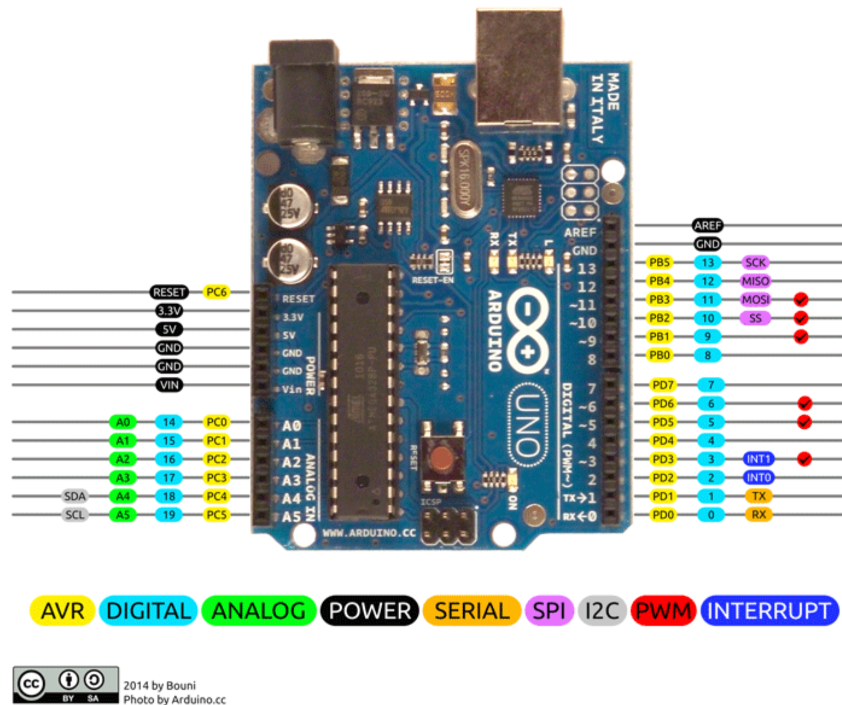

Figure 1.1. Carte Arduino UNO avec étiquetage des ports [2]

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…*').

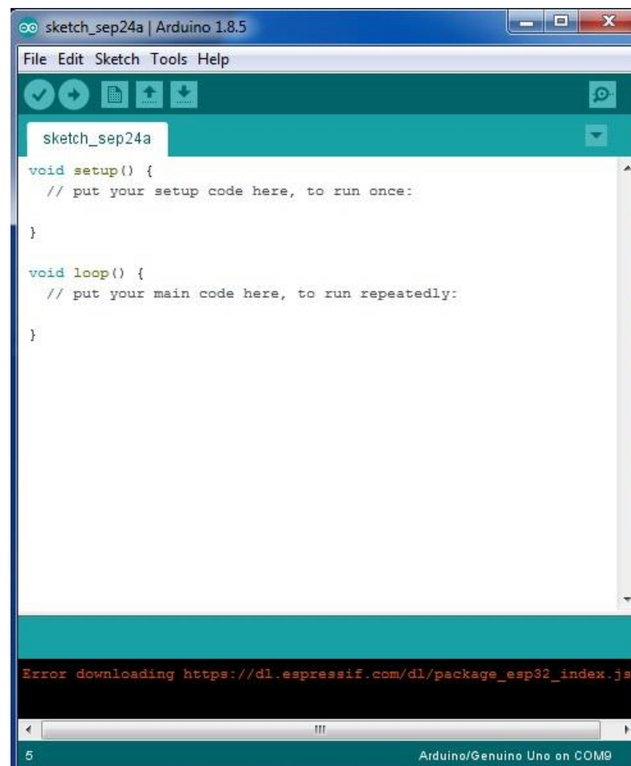

Figure 1.2. Capture d'écran sur le logiciel Arduino version 1.8.5 pour Windows

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 évident 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.

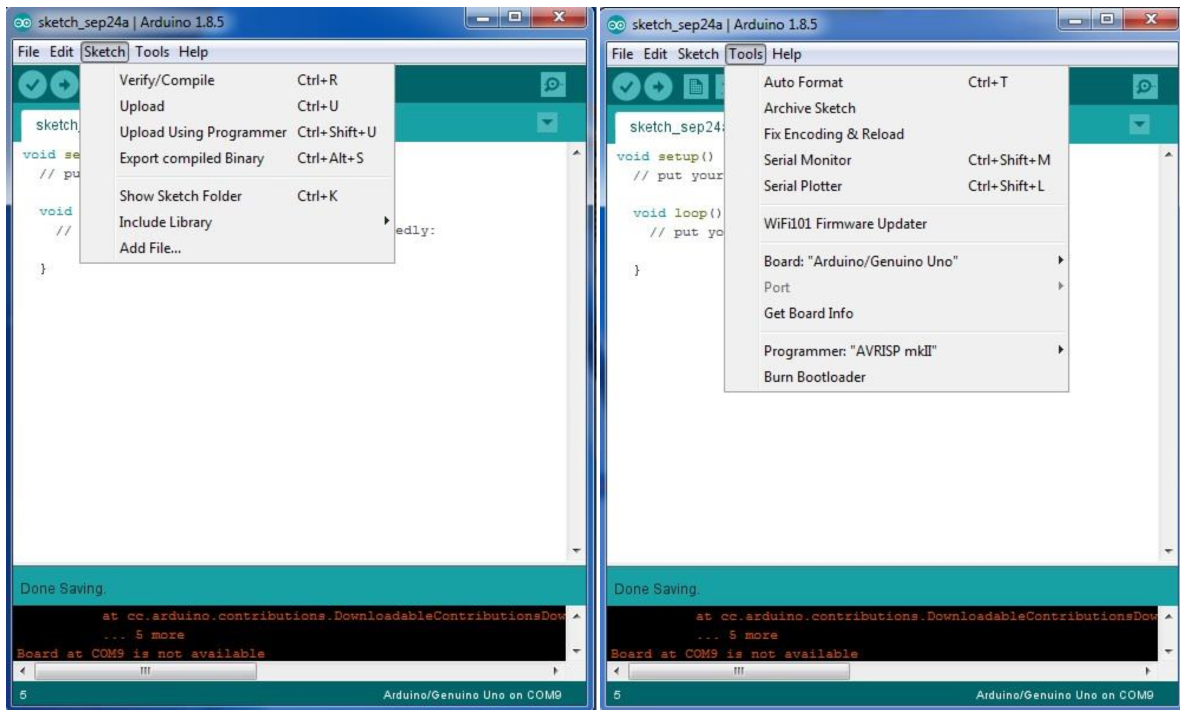

Figure 1.3. Capture d'écran sur le logiciel Arduino IDE. A gauche une illustration du menu « Sketch » et à droite une illustration du menu « Tools »

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.

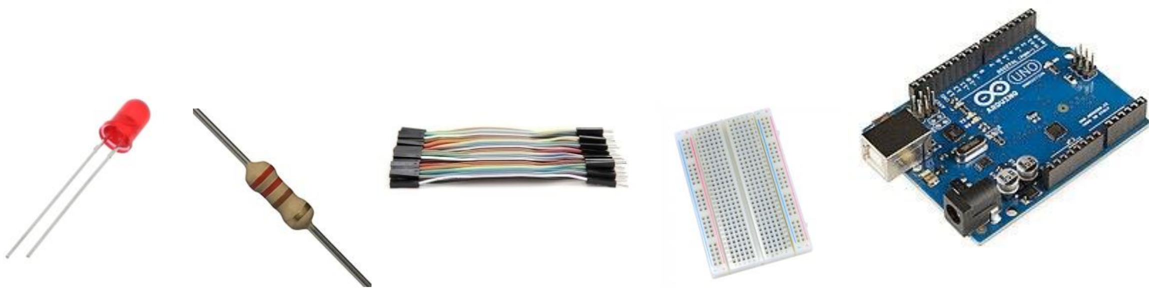

Figure 1.4. Matériel utilisé pour contrôler une LED 5mm avec l'Arduino

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).

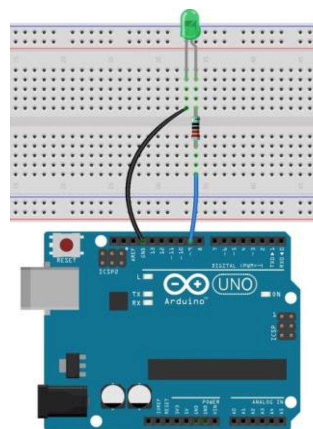

Figure 1.5. Schéma de branchement de la LED sur l'Arduino

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 télé-verse 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.

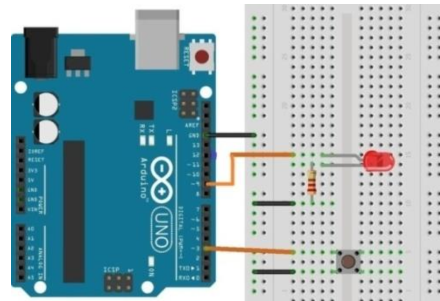

Figure 1.8. Schéma du branchement de la LED et le bouton poussoir sur l'Arduino

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 :

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

```
float  tableau[5][7] ;
```

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.      case 3 :
8.             action1() ;
9.             break ;
10.     case 5 :
11.            action2() ;
12.            break ;
13.     case 10 :
14.            action3() ;
15.            break ;
16. }
```

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

```
1.  switch(variable)
2.  {
3.      case 3 :
4.      case 11 :
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 ***do while***, 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.