

ARDUINO : PREMIERS PAS EN INFORMATIQUE EMBARQUÉE

Eskimon, olyte

19 février 2016

Table des matières

1	Introduction	7
1.0.1	Apprentissage des bases	7
2	Découverte de l'Arduino	9
2.1	Présentation d'Arduino	9
2.1.1	Qu'est-ce que c'est ?	9
2.1.2	Pourquoi choisir Arduino ?	15
2.1.3	Les cartes Arduino	19
2.1.4	Liste d'achat	21
2.2	Quelques bases élémentaires	24
2.2.1	Le courant, la tension et la masse	24
2.2.2	La résistance et sa loi !	27
2.2.3	Le microcontrôleur	29
2.2.4	Les bases de comptage (2,10 et 16)	31
2.3	Le logiciel	34
2.3.1	Installation	34
2.3.2	Interface du logiciel	38
2.3.3	Approche et utilisation du logiciel	40
2.4	Le matériel	44
2.4.1	Présentation de la carte	44
2.4.2	Installation	47
2.4.3	Fonctionnement global	57
2.5	Le langage Arduino (1/2)	61
2.5.1	La syntaxe du langage	61
2.5.2	Les variables	64
2.5.3	Les conditions	71
2.6	Le langage Arduino (2/2)	79
2.6.1	Les boucles	79
2.6.2	Les fonctions	83
2.6.3	Les tableaux	88
3	Gestion des entrées / sorties	95
3.1	Notre premier programme !	95
3.1.1	La diode électroluminescente	95
3.1.2	Par quoi on commence ?	99
3.1.3	Créer le programme : les bons outils !	101
3.1.4	Comment tout cela fonctionne ?	109
3.2	Introduire le temps	115
3.2.1	Comment faire ?	115
3.2.2	Faire clignoter un groupe de LED	118
3.2.3	Réaliser un chenillard	123

3.2.4	Fonction millis()	127
3.3	TP Feux de signalisation routière	129
3.3.1	Préparation	129
3.3.2	Énoncé de l'exercice	131
3.3.3	Correction !	133
3.4	Un simple bouton	134
3.4.1	Qu'est-ce qu'un bouton ?	134
3.4.2	Récupérer l'appui du bouton	141
3.4.3	Interagir avec les LED	146
3.4.4	Les interruptions matérielles	152
3.5	Afficheurs 7 segments	154
3.5.1	Première approche : côté électronique	155
3.5.2	Afficher son premier chiffre !	161
3.5.3	Techniques d'affichage	162
3.5.4	Utilisation du décodeur BCD	166
3.5.5	Utiliser plusieurs afficheurs	171
3.5.6	Contraintes des évènements	180
3.6	[TP] Parking	184
3.6.1	Consigne	184
3.6.2	Correction !	184
4	La communication avec Arduino	195
4.1	Généralités sur la voie série	195
4.1.1	Communiquer, pourquoi ?	195
4.1.2	La norme RS232	197
4.1.3	Connexion série entre Arduino et ...	203
4.1.4	Au delà d'Arduino avec la connexion série	208
4.2	Envoyer et recevoir des données sur la voie série	210
4.2.1	Préparer la voie série	210
4.2.2	Envoyer des données	213
4.2.3	Recevoir des données	219
4.2.4	[Exercice] Attention à la casse !	222
4.3	[TP] Baignade interdite !	224
4.3.1	Sujet du TP sur la voie série	224
4.3.2	Correction !	227
4.3.3	Améliorations	239
4.4	[Annexe] Ordinateur et voie série dans un autre langage de programmation	240
4.4.1	En C++ avec Qt	240
4.4.2	En C# (.Net)	247
4.4.3	En Python	252
5	Les grandeurs analogiques	257
5.1	Les entrées analogiques de l'Arduino	257
5.1.1	Un signal analogique : petits rappels	257
5.1.2	Les convertisseurs analogiques -> numérique ou CAN	259
5.1.3	Lecture analogique, on y vient...	268
5.1.4	Exemple d'utilisation	271
5.1.5	Une meilleure précision ?	276

5.2	[TP] Vu-mètre à LED	278
5.2.1	Consigne	278
5.2.2	Correction !	279
5.2.3	Amélioration	283
5.3	Et les sorties “analogiques”, enfin... presque !	284
5.3.1	Convertir des données binaires en signal analogique	284
5.3.2	La PWM de l’Arduino	287
5.3.3	Transformation PWM -> signal analogique	297
5.3.4	Modifier la fréquence de la PWM	305
5.4	[Exercice] Une animation “YouTube”	309
5.4.1	Énoncé	309
5.4.2	Solution	310
6	Les capteurs et l’environnement autour d’Arduino	315
6.1	Généralités sur les capteurs	315
6.1.1	Capteur et Transducteur	315
6.1.2	Un capteur, ça capte !	317
6.1.3	Les caractéristiques d’un capteur	321
6.2	Différents types de mesures	323
6.2.1	Tout Ou Rien, un capteur qui sait ce qu’il veut	323
6.2.2	Capteurs à résistance de sortie variable	328
6.2.3	Capteurs à tension de sortie variable	338
6.2.4	Étalonner son capteur	345
6.3	Des capteurs plus évolués	354
6.3.1	Capteur à sortie en modulation de largeur d’impulsion (PWM)	354
6.3.2	Capteur à signal de sortie de fréquence variable	358
6.3.3	Capteur utilisant un protocole de communication	360
7	Le mouvement grâce aux moteurs	363
7.1	Le moteur à courant continu	363
7.1.1	Un moteur, ça fait quoi au juste ?	363
7.1.2	Alimenter un moteur	384
7.1.3	Piloter un moteur	392
7.1.4	Et Arduino dans tout ça ?	407
7.2	Un moteur qui a de la tête : le Servomoteur	415
7.2.1	Principe du servomoteur	415
7.2.2	La commande d’un servomoteur	423
7.2.3	Arduino et les servomoteurs	427
7.2.4	L’électronique d’asservissement	430
7.2.5	Un peu d’exercice !	434
7.2.6	Tester un servomoteur “non-standard”	438
7.3	A petits pas, le moteur pas-à-pas	440
7.3.1	Les différents moteurs pas-à-pas et leur fonctionnement	440
7.3.2	Se servir du moteur	448
7.3.3	Utilisation avec Arduino	458
8	L’affichage, une autre manière d’interagir	465
8.1	Les écrans LCD	465
8.1.1	Un écran LCD c’est quoi ?	465

8.1.2	Quel écran choisir ?	468
8.1.3	Comment on s'en sert ?	470
8.2	Votre premier texte sur le LCD !	475
8.2.1	Ecrire du texte sur le LCD	476
8.2.2	Se déplacer sur l'écran	479
8.2.3	Créer un caractère	484
8.3	[TP] Supervision avec un LCD	486
8.3.1	Consigne	486
8.3.2	Correction	487
9	Internet of Things : Arduino sur Internet	493
9.1	Découverte de l'Ethernet sur Arduino	493
9.1.1	Un réseau informatique c'est quoi ?	493
9.1.2	Le shield Ethernet	494
9.1.3	Un peu de vocabulaire	496
9.2	Arduino et Ethernet : client	498
9.2.1	Client et requêtes HTTP	498
9.2.2	Utilisation du shield comme client	499
9.2.3	Exercice, lire l'uptime de Eskimon.fr	512
9.3	Arduino et Ethernet : serveur	512
9.3.1	Préparer l'Arduino	512
9.3.2	Répondre et servir des données	514
9.3.3	Agir sur une requête plus précise	517
9.3.4	Sortir de son réseau privé	527
9.3.5	Faire une interface pour dialoguer avec son Arduino	528
10	Conclusion	533
10.1	Aller plus loin	533
10.2	Remerciements	533

1 Introduction

Bienvenue à toutes et à tous pour un tutoriel sur l'électronique et l'informatique ensemble ! :)

Ce que nous allons apprendre aujourd'hui est un mélange d'électronique et de programmation. On va en effet parler d'informatique embarquée qui est un sous-domaine de l'électronique et qui a l'habileté d'unir la puissance de la programmation à la puissance de l'électronique.

Nous allons, dans un premier temps, voir ce qu'est l'électronique et la programmation. Puis nous enchaînerons sur la prise en main du système Arduino, un système d'informatique embarquée grand public. Enfin, je vous ferai un cours très rapide sur le langage Arduino, mais il posera les bases de la programmation. Une fois ces étapes préliminaires achevées, nous pourrons entamer notre premier programme et faire un pas dans l'informatique embarquée.

1.0.1 Apprentissage des bases

Le cours est composé de façon à ce que les bases essentielles soient regroupées dans les premières parties. C'est-à-dire, pour commencer la lecture, vous devrez lire les parties 1 et 2. Ensuite, les parties 3 et 4 sont également essentielles et sont à lire dans l'ordre.

Après cela, vous aurez acquis toutes les bases nécessaires pour poursuivre la lecture sereinement. C'est seulement alors que vous pourrez sélectionner les chapitres selon les connaissances que vous souhaitez acquérir.

2 Découverte de l'Arduino

Dans cette première partie, nous ferons nos premiers pas avec Arduino. Nous allons avant tout voir de quoi il s'agit exactement, essayer de comprendre comment cela fonctionne, puis installerons le matériel et le logiciel pour ensuite enchaîner sur l'apprentissage du langage de programmation nécessaire au bon fonctionnement de la carte Arduino. Soyez donc attentif afin de bien comprendre tout ce que je vais vous expliquer. Sans les bases, vous n'irez pas bien loin... ;)

2.1 Présentation d'Arduino

Comment faire de l'électronique en utilisant un langage de programmation ? La réponse, c'est le projet Arduino qui l'apporte. Vous allez le voir, celui-ci a été conçu pour être accessible à tous par sa simplicité. Mais il peut également être d'usage professionnel, tant les possibilités d'applications sont nombreuses.

2.1.1 Qu'est-ce que c'est ?

Une équipe de développeurs composée de *Massimo Banzi*, *David Cuartielles*, *Tom Igoe*, *Gianluca Martino*, *David Mellis* et *Nicholas Zambetti* a imaginé un projet répondant au doux nom de **Arduino** et mettant en œuvre une petite carte électronique programmable et un logiciel multiplateforme, qui puisse être accessible à tout un chacun dans le but de créer facilement des systèmes électroniques. Étant donné qu'il y a des débutants parmi nous, commençons par voir un peu le vocabulaire commun propre au domaine de l'électronique et de l'informatique.

2.1.1.0.0.1 Une carte électronique Une **carte électronique** est un support plan, flexible ou rigide, généralement composé d'**epoxy** ou de fibre de verre. Elle possède des pistes électriques disposées sur une, deux ou plusieurs couches (en surface et/ou en interne) qui permettent la mise en relation électrique des composants électroniques. Chaque piste relie tel composant à tel autre, de façon à créer un système électronique qui fonctionne et qui réalise les opérations demandées.

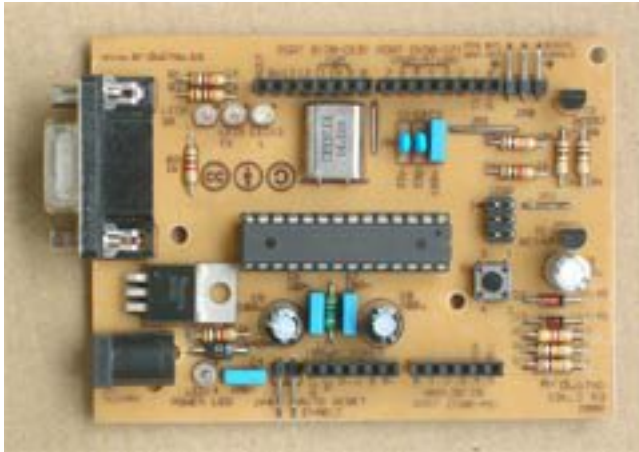


Figure : Exemple de carte électronique : Arduino Severino - (CC-BY-SA, arduino.cc)



Figure : Carte Arduino Duemilanove - (CC-BY-SA, arduino.cc)

Évidemment, tous les composants d'une carte électronique ne sont pas forcément reliés entre eux. Le câblage des composants suit un plan spécifique à chaque carte électronique, qui se nomme le **schéma électronique**.

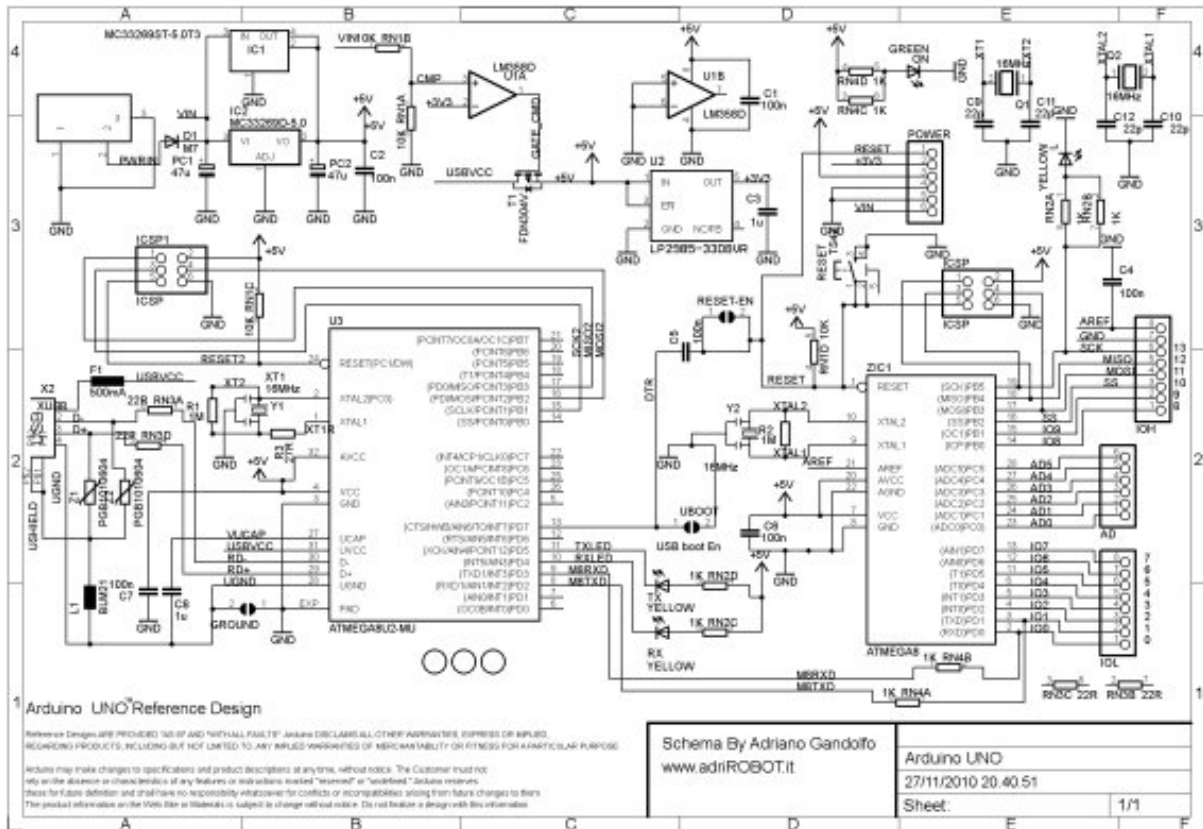


Figure : Exemple de schéma électronique - carte Arduino Uno - (CC-BY-SA, arduino.cc)

Enfin, avant de passer à la réalisation d'une carte électronique, il est nécessaire de transformer le schéma électronique en un **schéma de câblage**, appelé **typon**.

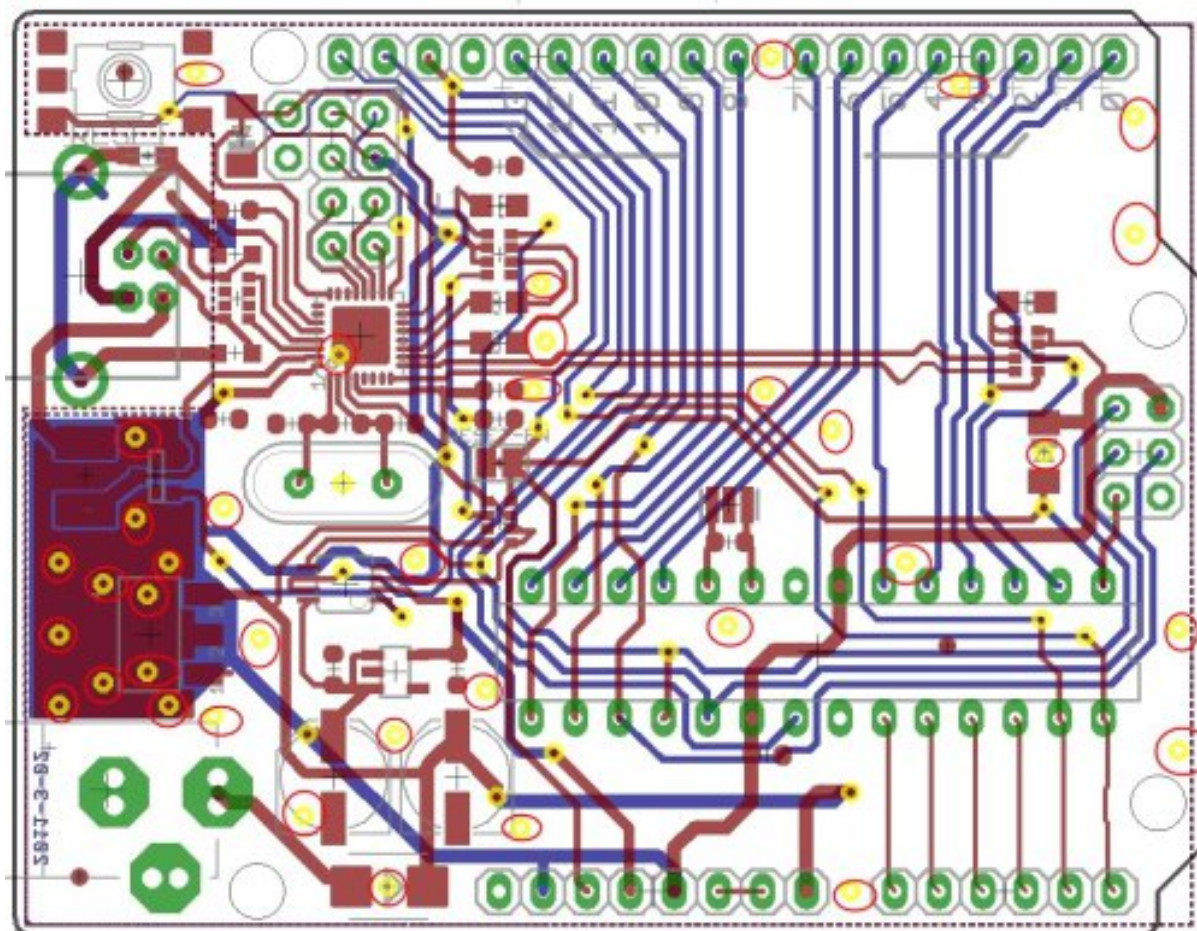


Figure : Exemple de typon - carte Arduino - (CC-BY-SA, arduino.cc)

[[question]] | Une fois que l'on a une carte électronique, on fait quoi avec ?

Eh bien une fois que la carte électronique est faite, nous n'avons plus qu'à la tester et l'utiliser ! Dans notre cas, avec Arduino, nous n'aurons pas à fabriquer la carte et encore moins à la concevoir. Elle existe, elle est déjà prête à l'emploi et nous n'avons plus qu'à l'utiliser. Et pour cela, vous allez devoir apprendre comment l'utiliser, ce que je vais vous montrer dans ce tutoriel.

2.1.1.0.0.2 Programmable ? J'ai parlé de **carte électronique programmable** au début de ce chapitre. Mais savez-vous ce que c'est exactement ? Non pas vraiment. Alors voyons ensemble de quoi il s'agit. La carte Arduino est une carte électronique *qui ne sait rien faire* sans qu'on lui dise *quoi faire*. Pourquoi ? Eh bien c'est dû au fait qu'elle est **programmable**. Cela signifie qu'elle a besoin d'un **programme** pour fonctionner. ##### Un programme Un programme est une liste d'instructions qui est exécutée par un système. Par exemple votre navigateur internet, avec lequel vous lisez probablement ce cours, est un programme. On peut analogiquement faire référence à une liste de course :

Chaque élément de cette liste est une **instruction** qui vous dit : "Va chercher le lait" ou "Va chercher le pain", etc. Dans un programme le fonctionnement est similaire :

- Attendre que l'utilisateur rentre un site internet à consulter
- Rechercher sur internet la page demandée
- Afficher le résultat

Tel pourrait être le fonctionnement de votre navigateur internet. Il va attendre que vous lui de-


- Lait
 - Pain
 - Steak
 - Epinards
 - ...
- 

Figure 2.1 – Exemple, une liste de course

mandiez quelque chose pour aller le chercher et ensuite vous le montrer. Eh bien, tout aussi simplement que ces deux cas, une carte électronique programmable suit une liste d'instructions pour effectuer les opérations demandées par le programme.

[[question]] | Et on les trouve où ces programmes ? Comment on fait pour le mettre dans la carte ?
o_O

Des programmes, on peut en trouver de partout. Mais restons concentrés sur Arduino. Le programme que nous allons mettre dans la carte Arduino, c'est nous qui allons le réaliser. Oui, vous avez bien lu : nous allons programmer cette carte Arduino. Bien sûr, ce ne sera pas aussi simple qu'une liste de course, mais rassurez-vous cependant car nous allons réussir quand même ! Je vous montrerai comment y parvenir, puisque avant tout c'est un des objectifs de ce tutoriel. Voici un exemple de programme :

```
// définition de la broche 2 de la carte en tant que variable
const int led_rouge = 2;

// fonction d'initialisation de la carte
void setup()
{
    // initialisation de la broche 2 comme étant une sortie
    pinMode(led_rouge, OUTPUT);
}

void loop()
{
    // allume la LED
    digitalWrite(led_rouge, LOW);
    // fait une pause de 1 seconde
    delay(1000);
    // éteint la LED
    digitalWrite(led_rouge, HIGH);
    // fait une pause de 1 seconde
```

```
    delay(1000);  
}
```

Code : Un exemple de programme simple

Vous le voyez comme moi, il s'agit de plusieurs lignes de texte, chacune étant une instruction. Ce langage ressemble à un véritable baragouin et ne semble vouloir *a priori* rien dire du tout... Et pourtant, c'est ce que nous saurons faire dans quelques temps ! Car nous apprendrons le **langage informatique** utilisé pour programmer la carte Arduino. Je ne m'attarde pas sur les détails, nous aurons amplement le temps de revenir sur le sujet plus tard. Pour répondre à la deuxième question, nous allons avoir besoin d'un logiciel...

2.1.1.0.0.3 Et un logiciel ? Bon, je ne vais pas vous faire le détail de ce qu'est un logiciel, vous savez sans aucun doute de quoi il s'agit. Ce n'est autre qu'un programme informatique exécuté sur un ordinateur. Oui, pour programmer la carte Arduino, nous allons utiliser un programme ! En fait, il va s'agir d'un **compilateur**. Alors qu'est-ce que c'est exactement ? ##### Un compilateur En informatique, ce terme désigne un logiciel qui est capable de traduire un langage informatique, ou plutôt un programme utilisant un langage informatique, vers un langage plus approprié afin que la machine qui va le lire puisse le comprendre. C'est un peu comme si le patron anglais d'une firme Chinoise donnait des instructions en anglais à un de ses ouvriers chinois. L'ouvrier ne pourrait comprendre ce qu'il doit faire. Pour cela, il a besoin que l'on traduise ce que lui dit son patron. C'est le rôle du **traducteur**. Le compilateur va donc traduire les instructions du programme précédent, écrites en langage texte, vers un langage dit "machine". Ce langage utilise uniquement des 0 et des 1. Nous verrons plus tard pourquoi. Cela pourrait être imagé de la façon suivante :



Figure 2.2 – Le rôle du compilateur

Donc, pour traduire le langage texte vers le langage machine (avec des 0 et des 1), nous aurons besoin de ce fameux compilateur. Et pas n'importe lequel, il faut celui qui soit capable de traduire le *langage texte Arduino* vers le *langage machine Arduino*. Et oui, sinon rien ne va fonctionner. Si vous mettez un traducteur Français vers Allemand entre notre patron anglais et son ouvrier chinois, ça ne fonctionnera pas mieux que s'ils discutaient directement. Vous comprenez ?

[[question]] | Et pourquoi on doit utiliser un traducteur, on peut pas simplement apprendre le langage machine directement ?

Comment dire... non ! Non parce que le langage machine est quasiment impossible à utiliser tel quel. Par exemple, comme il est composé de 0 et de 1, si je vous montre ça : "0001011100111010101000111", vous serez incapable, tout comme moi, de dire ce que cela signifie ! Et même si je vous dis que

la suite "01000001" correspond à la lettre "A", je vous donne bien du courage pour coder rien qu'une phrase! :P Bref, oubliez cette idée. C'est quand même plus facile d'utiliser des mots anglais (car oui nous allons être obligés de faire un peu d'anglais pour programmer, mais rien de bien compliqué rassurez-vous) que des suites de 0 et de 1. Vous ne croyez pas? ##### Envoyer le programme dans la carte Là, je ne vais pas vous dire grand chose car c'est l'environnement de développement qui va gérer tout ça. Nous n'aurons qu'à apprendre comment utiliser ce dernier et il se débrouillera tout seul pour envoyer le programme dans la carte. Nah! Nous n'aurons donc qu'à créer le programme sans nous soucier du reste.

2.1.2 Pourquoi choisir Arduino ?

2.1.2.1 Que va-t-on faire avec ?

Avec Arduino, nous allons commencer par apprendre à programmer puis à utiliser des composants électroniques. En fin de compte, nous saurons créer des systèmes électroniques plus ou moins complexes. Mais ce n'est pas tout... ##### D'abord, Arduino c'est... ... une carte électronique programmable et un logiciel gratuit :

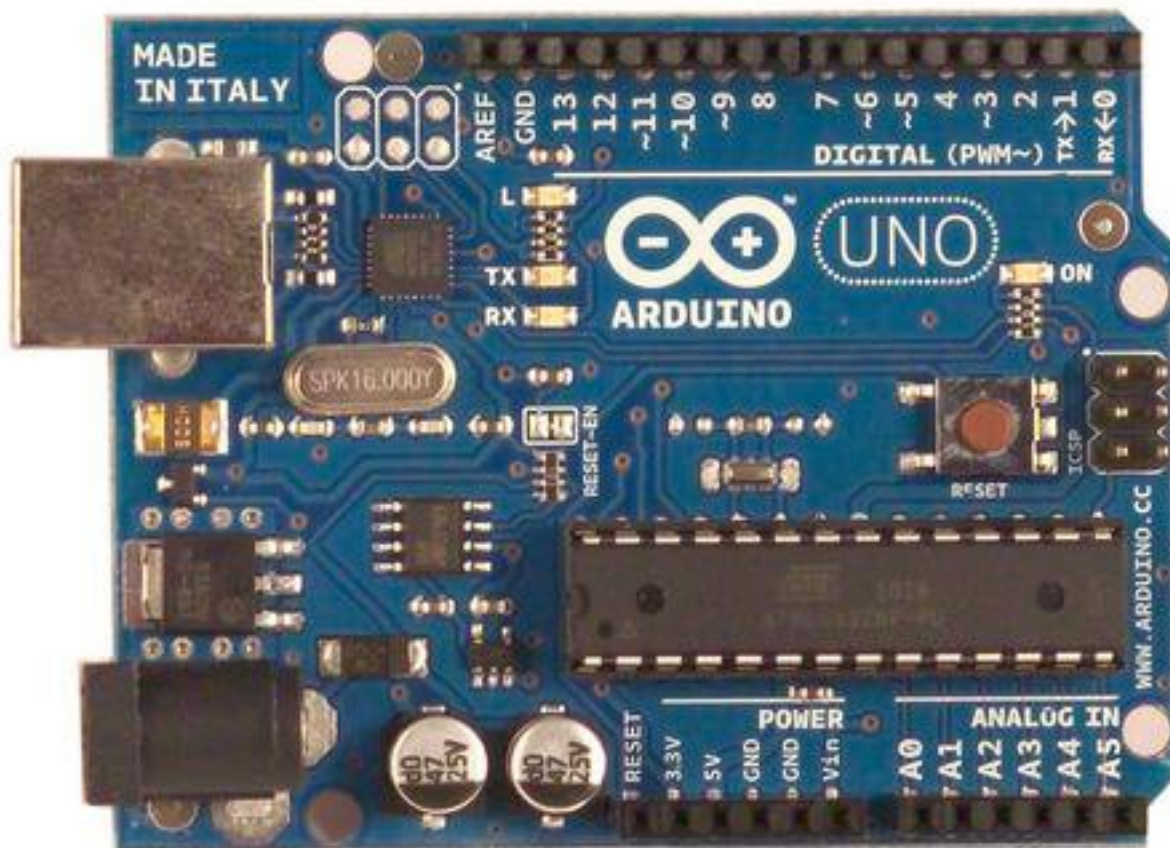


Figure : Carte Arduino Uno - (CC-BY-SA, arduino.cc)

2.1.2.1.1 Mais aussi

- Un prix dérisoire étant donné l'étendue des applications possibles. On comptera 20 euros pour la carte que l'on va utiliser dans le cours. Le logiciel est fourni gratuitement !
- Une compatibilité sous toutes les plateformes, à savoir : Windows, Linux et Mac OS.
- Une communauté ultra développée ! Des milliers de forums d'entraide, de présentations de projets, de propositions de programmes et de bibliothèques, ...
- Un site en anglais arduino.cc et un autre en français arduino.cc où vous trouverez tout de la référence Arduino, le matériel, des exemples d'utilisations, de l'aide pour débiter, des explications sur le logiciel et le matériel, etc.
- Une liberté quasi absolue. Elle constitue en elle-même deux choses :
 - Le logiciel : gratuit et open source, développé en Java, dont la simplicité d'utilisation relève du savoir cliquer sur la souris.
 - Le matériel : cartes électroniques dont les schémas sont en libre circulation sur internet.

[[attention]] | Cette liberté a une condition : le nom « Arduino » ne doit être employé que pour les cartes « officielles ». En somme, vous ne pouvez pas fabriquer votre propre carte sur le modèle Arduino et lui assigner le nom « Arduino ».

2.1.2.1.2 Et enfin, les applications possibles Voici une liste non exhaustive des applications possibles réalisées grâce à Arduino :

- contrôler des appareils domestiques
- donner une “intelligence” à un robot
- réaliser des jeux de lumières
- permettre à un ordinateur de communiquer avec une carte électronique et différents capteurs
- télécommander un appareil mobile (modélisme)
- etc.

Il y a une infinité d'autres utilisations, vous pouvez simplement chercher sur votre moteur de recherche préféré ou sur Youtube le mot “Arduino” pour découvrir les milliers de projets réalisés avec !

2.1.2.1.3 Arduino dans ce tutoriel Je vais quand même rappeler les principaux objectifs de ce cours. Nous allons avant tout découvrir Arduino dans son ensemble et apprendre à l'utiliser. Dans un premier temps, il s'agira de vous présenter ce qu'est Arduino, comment cela fonctionne globalement, pour ensuite entrer un peu plus dans le détail. Nous allons alors apprendre à utiliser le langage Arduino pour pouvoir créer des programmes très simples pour débiter. Nous enchaînerons ensuite avec les différentes fonctionnalités de la carte et ferons de petits TP qui vous permettront d'assimiler chaque notion abordée. Dès lors que vous serez plutôt à l'aise avec toutes les bases, nous nous rapprocherons de l'utilisation de composants électroniques plus ou moins complexes et finirons par un plus “gros” TP alliant la programmation et l'électronique. De quoi vous mettre de l'eau à la bouche ! :P

2.1.2.2 Arduino à l'école

Pédagogiquement, Arduino a aussi pas mal d'atouts. En effet, ses créateurs ont d'abord pensé ce projet pour qu'il soit facile d'accès. Il permet ainsi une très bonne approche de nombreux domaines et ainsi d'apprendre plein de choses assez simplement. ##### Des exemples

Voici quelques exemples d'utilisation possible :

- Simuler le fonctionnement des portes logiques
- Permettre l'utilisation de différents capteurs
- Mettre en œuvre et faciliter la compréhension d'un réseau informatique
- Se servir d'Arduino pour créer des maquettes animées montrant le fonctionnement des collisions entre les plaques de la croûte terrestre, par exemple ^^
- Donner un exemple concret d'utilisation des matrices avec un clavier alphanumérique 16 touches ou plus
- Être la base pour des élèves ayant un TPE à faire pour le BAC
- ...

De plus, énormément de ressources et tutoriels (mais souvent en anglais) se trouvent sur internet, ce qui offre une autonomie particulière à l'apprenant.

2.1.2.2.1 Des outils existant : Enfin, pour terminer de vous convaincre d'utiliser Arduino pour découvrir le monde merveilleux de l'embarqué, il existe différents outils qui peuvent être utilisés avec Arduino. Je vais en citer deux qui me semblent être les principaux : **Ardublock** est un outil qui se greffe au logiciel Arduino et qui permet de programmer avec des blocs. Chaque bloc est une instruction. On peut aisément faire des programmes avec cet outil et même des plutôt complexes. Cela permet par exemple de se concentrer sur ce que l'on doit faire avec Arduino et non se concentrer sur Arduino pour ensuite ce que l'on doit comprendre avec. Citons entre autre la simulation de porte logique : il vaut mieux créer des programmes rapidement sans connaître le langage pour comprendre plus facilement comment fonctionne une porte logique. Et ce n'est qu'un exemple. Cela permet aussi à de jeunes enfants de commencer à programmer sans de trop grandes complications.

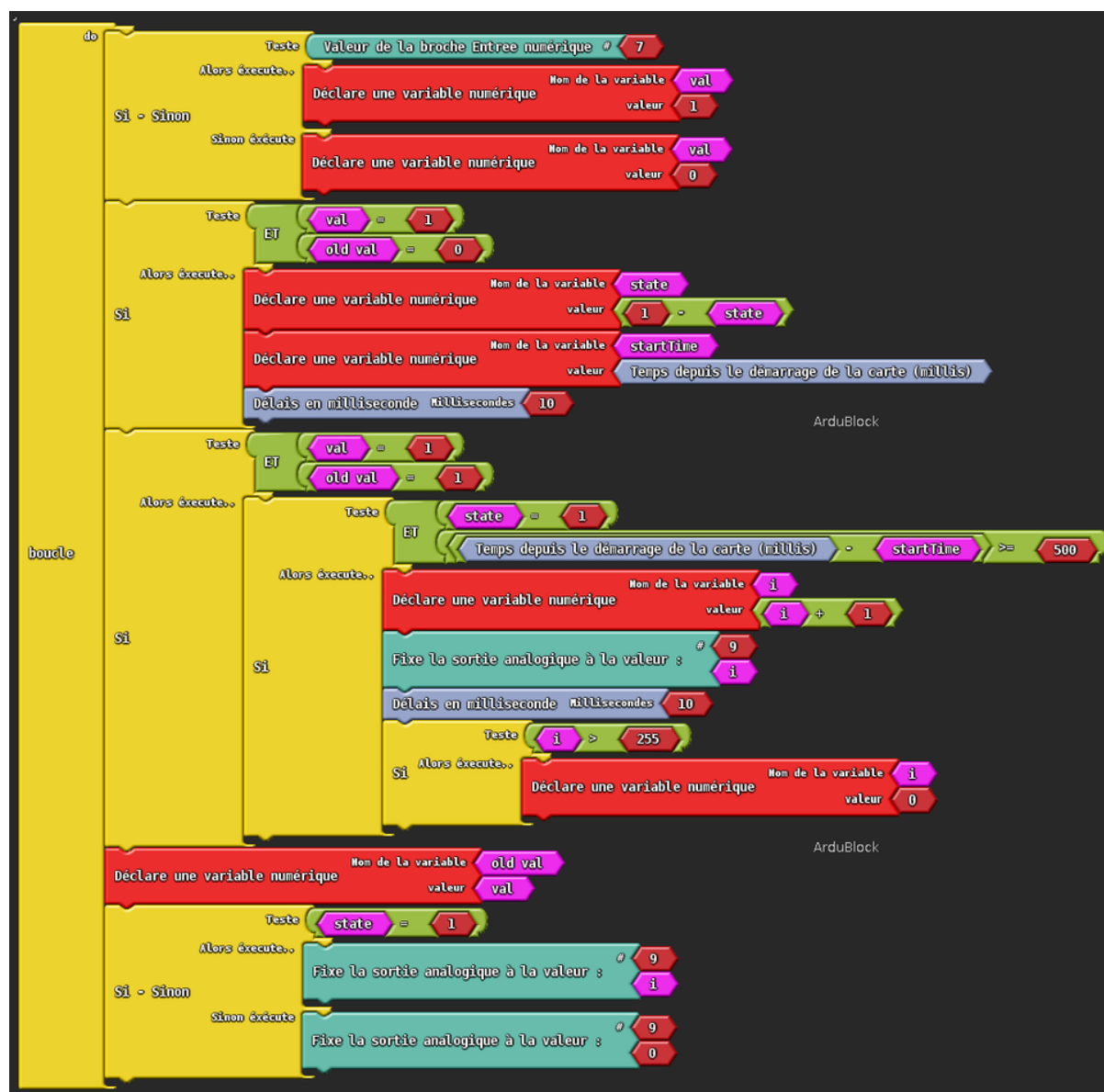


Figure 2.3 – Exemple de programme avec Ardublock

Processing est une autre plateforme en lien avec Arduino. Là il n'y a pas de matériel, uniquement un logiciel. Il permet entre autre de créer des interfaces graphiques avec un langage de programmation très similaire à celui d'Arduino. Par contre, cela demande un niveau un peu plus élevé

pour pouvoir l'utiliser, même si cela reste simple dans l'ensemble.

Voilà un exemple de ce que j'avais réalisé avec Processing pour faire communiquer mon ordinateur avec ma carte Arduino :



Figure 2.4 – Une interface réalisée avec Processing

J'espère avoir été assez convaincant afin que vous franchissiez le pas et ayez du plaisir à apprendre ! :)

2.1.3 Les cartes Arduino

Le matériel que j'ai choisi d'utiliser tout au long de ce cours n'a pas un prix excessif et, je l'ai dit, tourne aux alentours de 25 € TTC. Il existe plusieurs magasins en ligne et en boutiques qui vendent des cartes Arduino. Je vais vous en donner quelques-uns, mais avant, il va falloir différencier certaines choses.

2.1.3.1 Les fabricants

Le projet Arduino est libre et les schémas des cartes circulent librement sur internet. D'où la mise en garde que je vais faire : il se peut qu'un illustre inconnu fabrique *lui-même* ses cartes Arduino. Cela n'a rien de mal en soi, s'il veut les commercialiser, il peut. Mais s'il est malhonnête, il peut vous vendre un produit défectueux. Bien sûr, tout le monde ne cherchera pas à vous arnaquer. Mais la prudence est de rigueur. Faites donc attention où vous achetez vos cartes. ##### Les types de cartes

Il y a trois types de cartes :

- Les dites « officielles », qui sont fabriquées en Italie par le fabricant officiel : *Smart Projects*.



Figure 2.5 – Logo Arduino

- Lesdits « compatibles », qui ne sont pas fabriqués par *Smart Projects*, mais qui sont totalement compatibles avec les Arduino officielles.
- Les « autres », fabriquées par diverses entreprises et commercialisées sous un nom différent (Freeduino, Seeduino, Femtoduino, ...).

2.1.3.2 Les différentes cartes

Des cartes Arduino il en existe beaucoup ! Voyons celles qui nous intéressent... **Les cartes Uno et Duemilanove** . Nous choisirons d'utiliser la carte portant le nom de « Uno » ou « Duemilanove ». Ces deux versions sont presque identiques.



Figure : Carte Arduino Duemilanove - (CC-BY-SA, arduino.cc)



Figure : Carte Arduino Uno - (CC-BY-SA, arduino.cc)

La carte Mega . La carte Arduino Mega est une autre carte qui offre toutes les fonctionnalités de la carte précédente, mais avec des fonctionnalités supplémentaires. On retrouve notamment un nombre d'entrées et de sorties plus important ainsi que plusieurs liaisons séries. Bien sûr, le prix

est plus élevé : > 40 €!

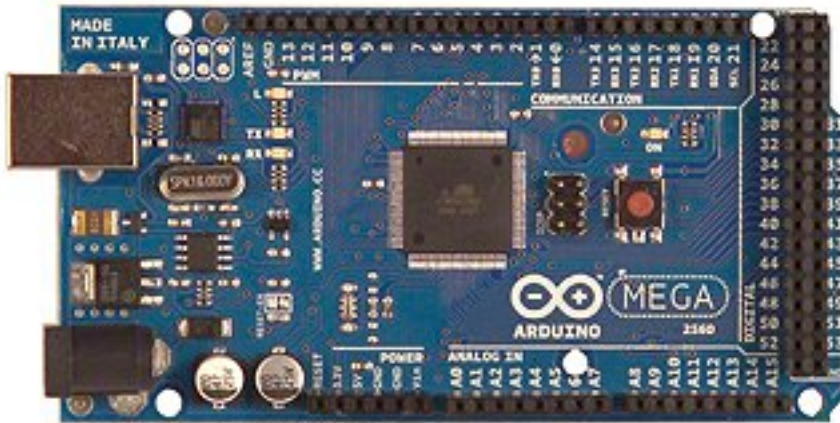


Figure : Carte Arduino Mega

- (CC-BY-SA, [arduino.cc](https://www.arduino.cc))

Les autres cartes. Il existe encore beaucoup d'autres cartes, je vous laisse vous débrouiller pour trouver celle qui conviendra à vos projets. Cela dit, je vous conseil dans un premier temps d'utiliser la carte Arduino Uno ou Duemilanove d'une part car elle vous sera largement suffisante pour débiter et d'autre part car c'est avec celle-ci que nous présentons le cours.

2.1.3.3 Où acheter ?

Il existe sur le net une multitude de magasins (des vendeurs professionnels, des importateurs mais aussi des petits détaillants de quartier) qui proposent des cartes Arduino.

[[question]] | J'ai vu des cartes officielles "édition SMD/CMS". Ça à l'air bien aussi, c'est quoi la différence ? Je peux m'en servir ?

Il n'y a pas de différence ! Enfin presque... "SMD" signifie **Surface Mount Device**, en français on appelle ça des "CMS" pour **Composants Montés en Surface**. Ces composants sont soudés directement sur le cuivre de la carte, il ne la traverse pas comme les autres. Pour les cartes Arduino, on retrouve le composant principal en édition SMD dans ces cartes. La carte est donc la même, aucune différence pour le tuto. Les composants sont les mêmes, seule l'allure "physique" est différente. Par exemple, ci-dessus la "Mega" est en SMD et la Uno est "classique".

2.1.4 Liste d'achat

Tout au long du cours, nous allons utiliser du matériel en supplément de la carte. Rassurez-vous le prix est bien moindre. Je vous donne cette liste, cela vous évitera d'acheter en plusieurs fois. Vous allez devoir me croire sur parole sur leur intérêt. Nous découvrirons comment chaque composant fonctionne et comment les utiliser tout au long du tutoriel. :)

[[attention]] | Attention, cette liste ne contient que les composants en quantités minimales strictes. Libre à vous de prendre plus de LED et de résistances par exemple (au cas où vous en perdriez ou détruisez...). Pour ce qui est des prix, j'ai regardé sur différents sites grand public (donc pas Farnell par exemple), ils peuvent donc paraître plus élevés que la normale dans la mesure où ces sites amortissent moins sur des ventes à des clients fidèles qui prennent tout en grande quantité...

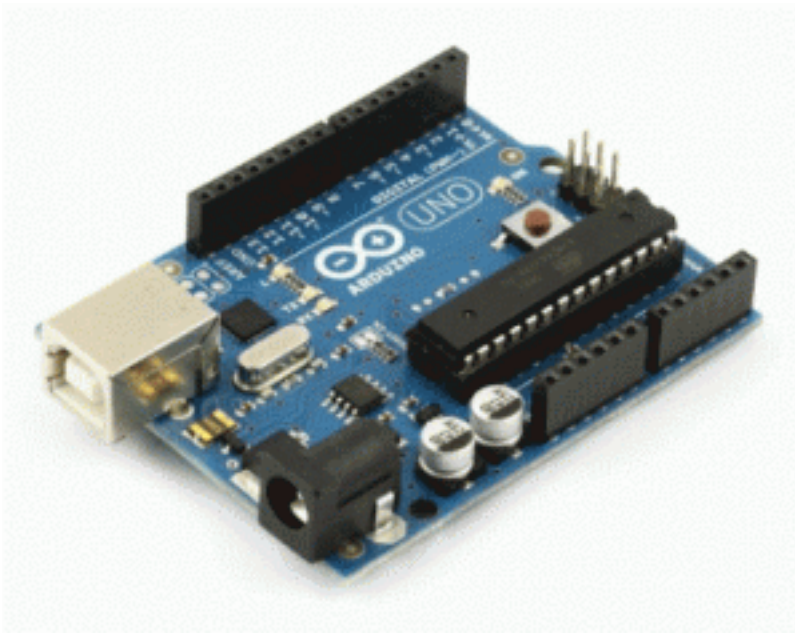
2 Découverte de l'Arduino

Avant que j'oublie, quatre éléments n'apparaîtront pas dans la liste et sont indispensables :

->

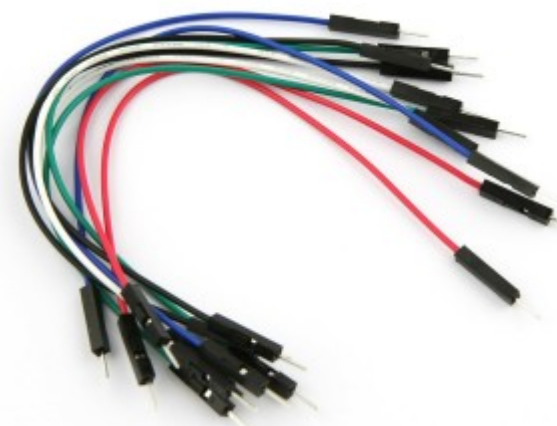
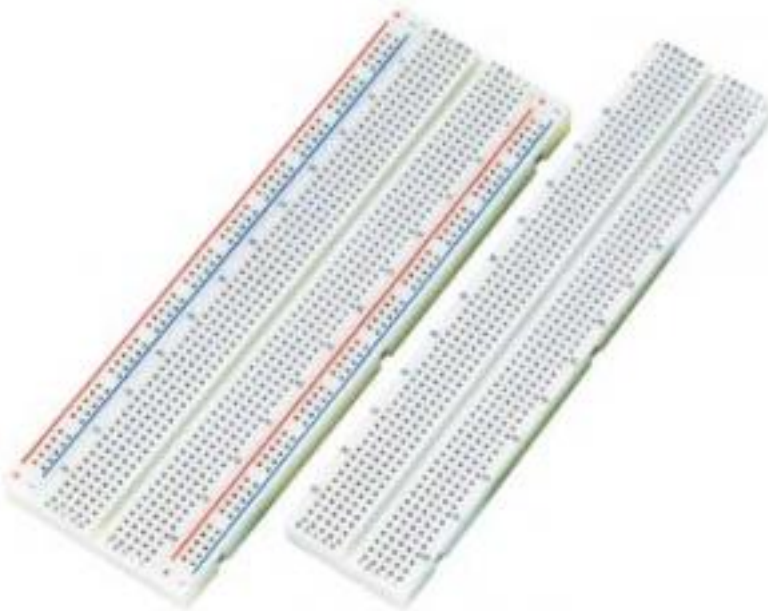
Une Arduino Uno ou Duemilanove

Un câble USB A mâle/B mâle



Une BreadBoard (plaque d'essai)









Un lot de fils pour brancher le tout !



<-

2.1.4.1 Liste Globale

Voici donc la liste du matériel nécessaire pour suivre le cours. Libre à vous de tout acheter ou non.

Désignation	Quantité	Photo	Description
LED rouge	7		Ce composant est une sorte de lampe un peu spécial. Nous nous en servons principalement pour faire de la signalisation.
LED verte	3		
LED jaune (ou orange)	2		
Résistance (entre 220 et 470 Ohm)	10		La résistance est un composant de base qui s'oppose au passage du courant. On s'en sert pour limiter des courants maximums mais aussi pour d'autres choses.
Résistance (entre 2.2 et 4.7 kOhm)	2		
Résistance (10 kOhm)	2		
Bouton Poussoir	2		Un bouton poussoir sert à faire passer le courant lorsqu'on appuie dessus ou au contraire garder le circuit "éteint" lorsqu'il est relâché.
Transistor (2N2222 ou BC547)	2		Le transistor sert à plein de chose. Il peut être utilisé pour faire de l'amplification (de courant ou de tension) mais aussi comme un interrupteur commandé électriquement.
Afficheur 7 segments (anode commune)	2		Un afficheur 7 segments est un ensemble de LEDs (cf. ci-dessus) disposées géométriquement pour afficher des chiffres.
Décodeur BCD (MC14543)	1		Le décodeur BCD (Binaire Codé Décimal) permet piloter des afficheurs 7 segments en limitant le nombre de fils de données (4 au lieu de 7).
Condensateur (10 nF)	2		Le condensateur est un composant de base. Il sert à plein de chose. On peut se le représenter comme un petit réservoir à électricité.
Condensateur 1000 µF	1		Celui-ci est un plus gros réservoir que le précédent

2.1.4.2 Les revendeurs

De nombreux revendeurs existent sur internet, allant du très professionnel avec un grand choix au petit détaillant de quartier sans oublier les grands importateurs chinois aux tarifs imbattables si vous savez être patient.

2.1.4.3 Les kits

Enfin, il existe des kits tout prêts chez certains revendeurs. Nous n'en conseillons aucun pour plusieurs raisons. Tout d'abord, pour ne pas faire trop de publicité et rester conforme avec la charte du site. Ensuite, car il est difficile de trouver un kit "complet". Ils ont tous des avantages et des inconvénients mais aucun (au moment de la publication de ces lignes) ne propose absolument tous les composants que nous allons utiliser. Nous ne voulons donc pas que vous reveniez vous plaindre sur les forums car nous vous aurions fait dépenser votre argent inutilement !

[[erreur]] | Cela étant dit, merci de **ne pas nous spammer de MP** pour que l'on donne notre avis sur tel ou tel kit ! | Usez des forums pour cela, il y a toujours quelqu'un qui sera là pour vous aider. | Et puis nous n'avons pas les moyens de tous les acheter et tester leur qualité !

2.2 Quelques bases élémentaires

En attendant que vous achetiez votre matériel, je vais vous présenter les bases de l'électronique et de la programmation. Cela vous demandera tout de même une bonne concentration pour essayer de comprendre des concepts pas évidents en soi.

[[information]] | La première partie de ce chapitre ne fait que reprendre quelques éléments du [cours sur l'électronique](#), que vous pouvez consulter pour de plus amples explications. ;)

2.2.1 Le courant, la tension et la masse

Pour faire de l'électronique, il est indispensable de connaître sur le bout des doigts ce que sont les grandeurs physiques. Alors, avant de commencer à voir lesquelles on va manipuler, voyons un peu ce qu'est une grandeur physique. Une **grandeur physique** est quelque chose qui se mesure. Plus précisément, il s'agit d'un élément mesurable, grâce à un appareil ou dispositif de mesure, régit par les lois de la physique. Par exemple, la pression atmosphérique est une grandeur physique, ou bien la vitesse à laquelle circule une voiture. En électronique cependant, nous ne mesurons pas ces grandeurs-là, nous avons nos propres grandeurs, qui sont : **le courant** et **la tension**.

2.2.1.0.1 La source d'énergie L'énergie que l'on va manipuler (courant et tension) provient d'un **générateur**. Par exemple, on peut citer : la pile électrique, la batterie électrique, le secteur électrique. Cette énergie qui est fournie par le générateur est restituée à un ou plusieurs **récepteurs**. Le récepteur, d'après son nom, reçoit de l'énergie. On dit qu'il la **consomme**. On peut citer pour exemples : un chauffage d'appoint, un sèche-cheveux, une perceuse.

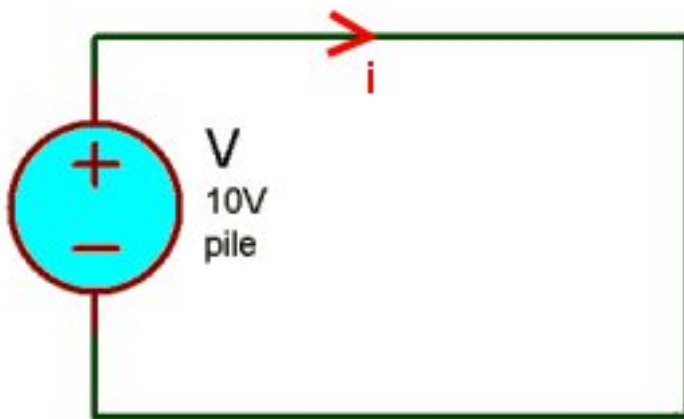
2.2.1.1 Le courant électrique

2.2.1.1.1 Charges électriques Les charges électriques sont des grandeurs physiques mesurables. Elles constituent la matière en elle-même. Dans un atome, qui est élément primaire de la matière, il y a trois charges électriques différentes : les charges **positives**, **négatives** et **neutres**, appelées respectivement **protons**, **électrons** et **neutrons**. Bien, maintenant nous pouvons définir le courant : il s'agit d'un **déplacement ordonné de charges électriques**.

2.2.1.1.2 Conductibilité des matériaux La notion de conductibilité est importante à connaître car elle permet de comprendre pas mal de phénomènes. On peut définir la **conductibilité** comme étant la capacité d'un matériau à se laisser traverser par un courant électrique. De ces matériaux, on peut distinguer quatre grandes familles :

- les isolants : leurs propriétés empêchent le passage d'un courant électrique (plastique, bois, verre)
- les semi-conducteurs : ce sont des isolants, mais qui laissent passer le courant dès lors que l'on modifie légèrement leur structure interne (diode, transistor, LED)
- les conducteurs : pour eux, le courant peut passer librement à travers tout en opposant une faible résistance selon le matériau utilisé (or, cuivre, métal en général)
- les supraconducteurs : ce sont des types bien particuliers qui, à une température extrêmement basse, n'opposent quasiment aucune résistance au passage d'un courant électrique

2.2.1.1.3 Sens du courant Le courant électrique se déplace selon un sens de circulation. Un générateur électrique, par exemple une pile, produit un courant. Et bien ce courant va circuler du pôle positif vers le pôle négatif de la pile si, et seulement si, ces deux pôles sont reliés entre eux par un fil métallique ou un autre conducteur. C'est que l'on appelle le **sens conventionnel** du courant. On note le courant par une flèche qui indique le sens conventionnel de circulation du courant :



->

<-

2.2.1.1.4 Intensité du courant [[attention]] | L'intensité du courant est la vitesse à laquelle circule ce courant. Tandis que le courant est un déplacement ordonné de charges électriques. Voilà un point à ne pas confondre.

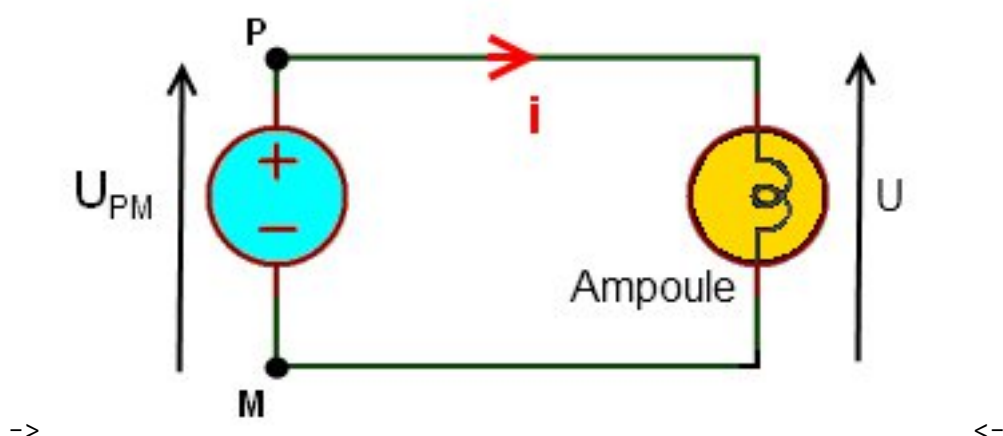
| On mesure la vitesse du courant, appelée **intensité**, en **Ampères** (noté **A**) avec un *Ampèremètre*. En général, en électronique de faible puissance, on utilise principalement le milli-Ampère (**mA**) et le micro-Ampère (**µA**), mais jamais bien au-delà. C'est tout ce qu'il faut savoir sur le courant,

pour l'instant.

2.2.1.2 Tension

Autant le courant se déplace, ou du moins est un déplacement de charges électriques, autant la **tension** est quelque chose de **statique**. Pour bien définir ce qu'est la tension, sachez qu'on la compare à la pression d'un fluide. Par exemple, lorsque vous arrosez votre jardin (ou une plante, comme vous préférez) avec un tuyau d'arrosage, eh bien dans ce tuyau, il y a une certaine pression exercée par l'eau fournie par le robinet. Cette pression permet le déplacement de l'eau dans le tuyau, donc crée un courant. Mais si la pression n'est pas assez forte, le courant ne sera lui non plus pas assez fort. Pour preuve, vous n'avez qu'à pincer le tuyau pour constater que le courant ne circule plus. On appelle ce "phénomène de pression" : la **tension**. Je n'en dis pas plus car ce serait vous embrouiller. ;)

2.2.1.2.1 Notation et unité La tension est mesurée en **Volts** (notée **V**) par un *Voltmètre*. On utilise principalement le Volt, mais aussi son sous-multiple qui est le milli-Volt (**mV**). On représente la tension, d'une pile par exemple, grâce à une flèche toujours orientée dans le sens du courant aux bornes d'un générateur et toujours opposée au courant, aux bornes d'un récepteur :



2.2.1.2.2 La différence de potentiel Sur le schéma précédent, on a au point M une tension de 0V et au point P, une tension de 5V. Prenons notre Voltmètre et mesurons la tension aux bornes du générateur. La borne COM du Voltmètre doit être reliée au point M et la borne "+" au point P. Le potentiel au point P, soustrait par le potentiel au point M vaut : $U_P - U_M = 5 - 0 = 5V$. On dit que la **différence de potentiel** entre ces deux points est de 5V. Cette mesure se note donc : U_{PM} . Si on inverse le sens de branchement du Voltmètre, la borne "+" est reliée au point M et la borne COM au point P. La mesure que l'on prend est la différence de tension (= potentiel) entre le point M et le point P : $U_M - U_P = 0 - 5 = -5V$ Cette démonstration un peu surprenante vient du fait que la masse est arbitraire.

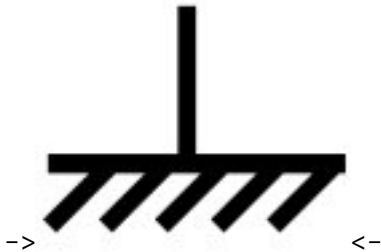
2.2.1.3 La masse

Justement, parlons-en ! La **masse** est, en électronique, un point de référence.

2.2.1.3.1 Notion de référentiel Quand on prend une mesure, en général, on la prend entre deux points bien définis. Par exemple, si vous vous mesurez, vous prenez la mesure de la plante de vos pieds jusqu'au sommet de votre tête. Si vous prenez la plante de vos pieds pour référence (c'est-à-dire le chiffre zéro inscrit sur le mètre), vous lirez 1m70 (par exemple). Si vous inversez, non pas la tête, mais le mètre et que le chiffre zéro de celui-ci se retrouve donc au sommet de votre tête, vous serez obligé de lire la mesure à -1m70. Eh bien, ce chiffre zéro est la référence qui vous permet de vous mesurer. En électronique, cette référence existe, on l'appelle la **masse**.

2.2.1.3.2 Qu'est ce que c'est ? La masse, c'est un référentiel. En électronique, on voit la masse d'un montage comme étant le zéro Volt (0V). C'est le point qui permet de mesurer une bonne partie des tensions présentes dans un montage.

2.2.1.3.3 Représentation et notation Elle se représente par ce symbole, sur un schéma électronique :



Vous ne le verrez pas souvent dans les schémas de ce cours, pour la simple raison qu'elle est présente sur la carte que l'on va utiliser sous un autre nom : **GND**. GND est une abréviation du terme anglais "Ground" qui veut dire terre/sol. Donc, pour nous et tous les montages que l'on réalisera, ce sera le point de référence pour la mesure des tensions présentes sur nos circuits et le zéro Volt de tous nos circuits.

2.2.1.3.4 Une référence arbitraire Pour votre culture, sachez que la masse est quelque chose d'arbitraire. Je l'ai bien montré dans l'exemple au début de ce paragraphe. On peut changer l'emplacement de cette référence et, par exemple, très bien dire que le 5V est la masse. Ce qui aura pour conséquence de modifier l'ancienne masse en -5V.

2.2.2 La résistance et sa loi !

En électronique, il existe plein de composants qui ont chacun une ou plusieurs fonctions. Nous allons voir quels sont ces composants dans le cours, mais pas tout de suite. Car, maintenant, on va aborder la résistance qui est LE composant de base en électronique.

2.2.2.0.1 Présentation C'est le composant le plus utilisé en électronique. Sa principale fonction est de réduire l'intensité du courant (mais pas uniquement). Ce composant se présente sous la forme d'un petit boîtier fait de divers matériaux et repéré par des anneaux de couleur indiquant la valeur de la résistance.

2.2.2.0.2 Symbole Le symbole de la résistance ressemble étrangement à la forme de son boîtier :



Figure 2.7 – Photo de résistance



Figure 2.8 – Symbole de la résistance

2.2.2.0.3 Loi d'ohm Le courant traversant une résistance est régi par une formule assez simple qui se nomme **la loi d'Ohm** :

$$I = \frac{U}{R}$$

- **I** : intensité qui traverse la résistance en Ampères, notée A
- **U** : tension aux bornes de la résistance en Volts, notée V
- **R** : valeur de la résistance en Ohms, notée Ω

En général, on retient mieux la formule sous cette forme : $U = R \times I$

2.2.2.0.4 Unité L'unité de la résistance est l'**ohm**. On le note avec le symbole grec oméga majuscule : Ω .

2.2.2.0.5 Le code couleur La résistance possède une suite d'anneaux de couleurs différentes sur son boîtier. Ces couleurs servent à expliciter la valeur de la résistance sans avoir besoin d'écrire en chiffre dessus (car vous avez déjà essayé d'écrire sur un cylindre :P?) Le premier anneau représente le chiffre des centaines, le second celui des dizaines et le troisième celui des unités. Enfin, après un petit espace vient celui du coefficient multiplicateur. Avec ces quatre anneaux et un peu d'entraînement vous pouvez alors deviner la valeur de la résistance en un clin d'oeil ;). Ce tableau vous permettra de lire ce code qui correspond à la valeur de la résistance :

->

Couleur	Chiffre	Coefficient multiplicateur	Puissance	Tolérance
Noir	0	1	10^0	-
Brun	1	10	10^1	$\pm 1 \%$
Rouge	2	100	10^2	$\pm 2 \%$
Orange	3	1000	10^3	-
Jaune	4	10 000	10^4	-
Vert	5	100 000	10^5	$\pm 0.5 \%$
Bleu	6	1 000 000	10^6	$\pm 0.25 \%$
Violet	7	10 000 000	10^7	$\pm 0.10 \%$
Gris	8	100 000 000	10^8	$\pm 0.05 \%$

Couleur	Chiffre	Coefficient multiplicateur	Puissance	Tolérance
Blanc	9	1 000 000 000	10^9	-
-	-	-	-	-
Or	0.1	0.1	10^{-1}	$\pm 5 \%$
Argent	0.01	0.01	10^{-2}	$\pm 10 \%$
(absent)	-	-	-	$\pm 20 \%$

Table 2.3 – Tableau du code couleurs des résistances

<-

2.2.3 Le microcontrôleur

Nous avons déjà un peu abordé le sujet dans la présentation du cours. Je vous ai expliqué “brièvement” comment fonctionnait un programme et surtout ce que c’était! ^^ Bon, dès à présent, je vais rentrer un petit peu plus dans le détail en vous introduisant des notions basées sur le matériel étroitement lié à la programmation. Nous allons en effet aborder le microcontrôleur dans un niveau de complexité supérieur à ce que je vous avais introduit tout à l’heure. Oh, rien de bien insurmontable, soyez sans craintes. ;)

2.2.3.1 La programmation en électronique

Aujourd’hui, l’électronique est de plus en plus composée de composants numériques programmables. Leur utilisation permet de simplifier les schémas électroniques et par conséquent, réduire le coût de fabrication d’un produit. Il en résulte des systèmes plus complexes et performants pour un espace réduit.

2.2.3.1.1 Comment programmer de l’électronique? Pour faire de l’électronique programmée, il faut un ordinateur et un **composant programmable**. Il existe tout plein de variétés différentes de composants programmables, à noter : les microcontrôleurs, les circuits logiques programmables, ... Nous, nous allons programmer des microcontrôleurs. Mais, à ce propos, vous ai-je dit qu’est ce que c’était qu’un microcontrôleur?

[[question]] | Qu’est ce que c’est ?

Je l’ai dit à l’instant, le microcontrôleur est un composant électronique programmable. On le programme par le biais d’un ordinateur grâce à un langage informatique, souvent propre au type de microcontrôleur utilisé. Je n’entrerais pas dans l’utilisation poussée de ces derniers car le niveau est rudement élevé et la compréhension difficile.

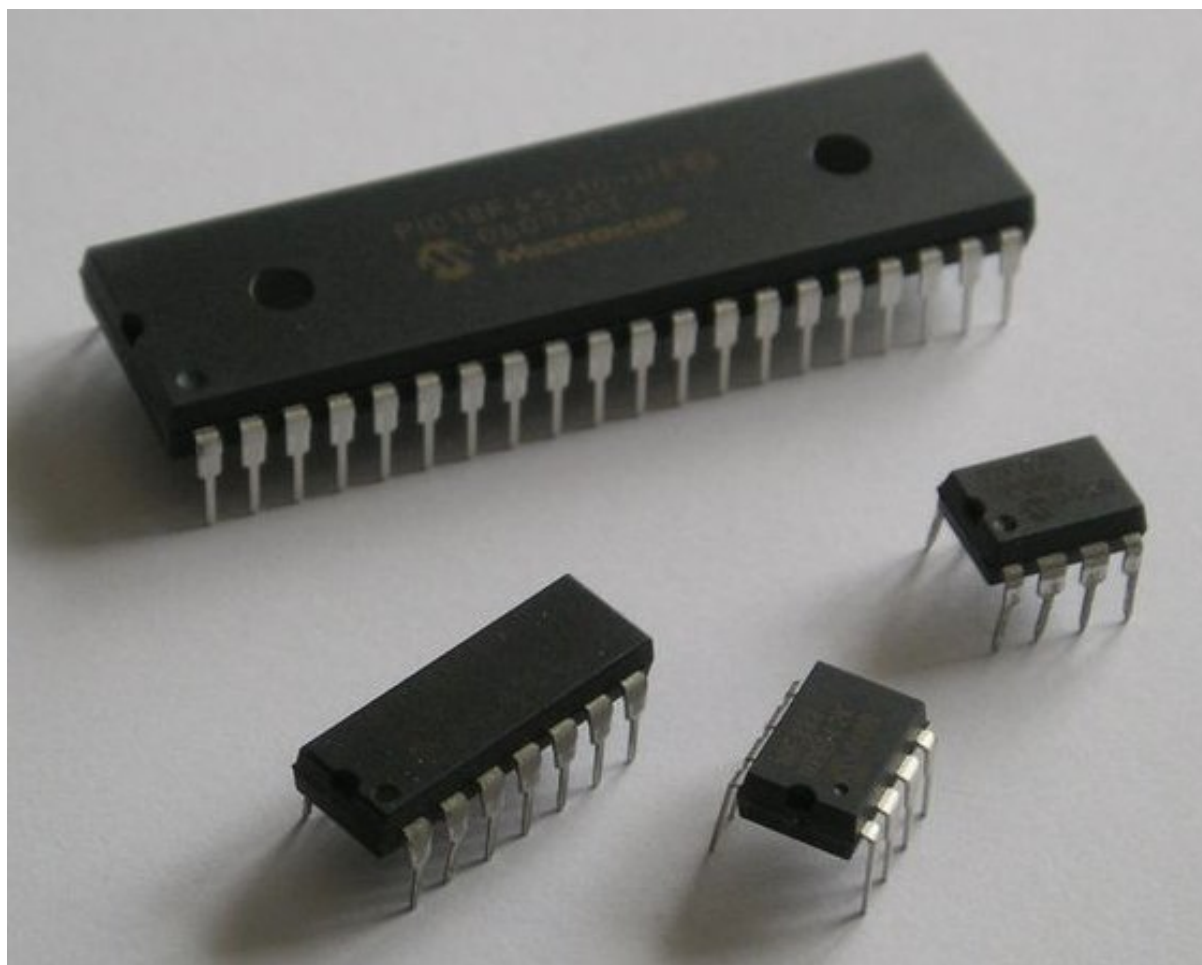


Figure : Des microcontrôleurs de différentes tailles (licence CC-0)

C'est donc le microcontrôleur qui va être le cerveau de la carte Arduino, pour en revenir à nos moutons. C'est lui que nous allons programmer. On aura le temps d'en rediscuter, pour l'instant je veux uniquement vous présenter les éléments principaux qui le composent.

2.2.3.1.2 Composition des éléments internes d'un micro-contrôleur Un microcontrôleur est constitué par un ensemble d'éléments qui ont chacun une fonction bien déterminée. Il est en fait composé des mêmes éléments que sur la carte mère d'un ordinateur. Si l'on veut, c'est un ordinateur (sans écran, sans disque dur, sans lecteur de disque) dans un espace très restreint. Parmi les différents éléments d'un microcontrôleur typique, je vais vous présenter ceux qui vont nous être utiles.

La mémoire

La mémoire du microcontrôleur sert à plusieurs choses. On peut aisément citer le stockage du programme et de données autres que le programme. Il existe cinq types de mémoire :

- La mémoire Flash : c'est celle qui contiendra le programme à exécuter (celui que vous allez créer!). Cette mémoire est effaçable et ré-inscriptible (c'est la même que celle d'une clé USB par exemple).
- RAM : c'est la mémoire dite "vive", elle va contenir les variables de votre programme. Elle est dite "volatile" car elle s'efface si on coupe l'alimentation du micro-contrôleur (comme sur un ordinateur).
- EEPROM : c'est le "disque dur" du microcontrôleur. Vous pourrez y enregistrer des infos

qui ont besoin de survivre dans le temps, même si la carte doit être arrêtée et coupée de son alimentation. Cette mémoire ne s'efface pas lorsque l'on éteint le microcontrôleur ou lorsqu'on le reprogramme.

- Les registres : c'est un type particulier de mémoire utilisé par le processeur. Nous n'en parlerons pas tout de suite.
- La mémoire cache : c'est une mémoire qui fait la liaison entre les registres et la RAM. Nous n'en parlerons également pas tout de suite.

[[information]] | [Une annexe au tutoriel](#) vous fournit plus de détails concernant les mémoires utilisables dans vos programmes.

Le processeur

C'est le composant principal du microcontrôleur. C'est lui qui va exécuter le programme que nous lui donnerons à traiter. On le nomme souvent le CPU.

Diverses choses

Nous verrons plus en détail l'intérieur d'un microcontrôleur, mais pas tout de suite, c'est bien trop compliqué. Je ne voudrais pas perdre la moitié des visiteurs en un instant ! :P

2.2.3.1.3 Fonctionnement Avant tout, pour que le microcontrôleur fonctionne, il lui faut une alimentation ! Cette alimentation se fait en générale par du +5V. D'autres ont besoin d'une tension plus faible, du +3,3V (c'est le cas de la Arduino Due par exemple). En plus d'une alimentation, il a besoin d'un signal d'horloge. C'est en fait une succession de 0 et de 1 ou plutôt une succession de tensions 0V et 5V. Elle permet en outre de cadencer le fonctionnement du microcontrôleur à un rythme régulier. Grâce à elle, il peut introduire la notion de temps en programmation. Nous le verrons plus loin. Bon, pour le moment, vous n'avez pas besoin d'en savoir plus. Passons à autre chose.

*[RAM] : en anglais 'Random Access Memory' [EEPROM] : en anglais 'Electrically-Erasable Programmable Read-Only Memory' * [CPU] : pour Central Processing Unit*

2.2.4 Les bases de comptage (2,10 et 16)

2.2.4.1 Les bases du de comptage

[[question]] | On va apprendre à compter ? o_o

Non, je vais simplement vous expliquer ce que sont les **bases de comptage**. C'est en fait un **système de numération** qui permet de compter en utilisant des caractères de numérations, on appelle ça des **chiffres**.

2.2.4.1.1 Cas simple, la base 10 La base 10, vous la connaissez bien, c'est celle que l'on utilise tous les jours pour compter. Elle regroupe un ensemble de 10 chiffres : 0,1,2,3,4,5,6,7,8,9. Avec ces chiffres, on peut créer une infinité de nombres (ex : 42, 89, 12872, 14.56, 9.3, etc...). Cependant, voyons cela d'un autre œil...

- L'*unité* sera représenté par un chiffre multiplié par 10 à la puissance 0.
- La *dizaine* sera représenté par un chiffre multiplié par 10 à la puissance 1.
- La *centaine* sera représenté par un chiffre multiplié par 10 à la puissance 2.

- [...]
- Le *million* sera représenté par un chiffre multiplié par 10 à la puissance 6.
- etc...

En généralisant, on peut donc dire qu'un nombre (composé de chiffres) est la somme des chiffres multipliés par 10 à une certaine puissance. Par exemple, si on veut écrire 1024, on peut l'écrire : $1 \times 1000 + 0 \times 100 + 2 \times 10 + 4 \times 1 = 1024$ ce qui est équivalent à écrire : $1 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 = 1024$. Eh bien c'est ça, compter en base 10 ! Vous allez mieux comprendre avec la partie suivante.

2.2.4.1.2 Cas informatique, la base 2 et la base 16 En informatique, on utilise beaucoup les bases 2 et 16. Elles sont composées des chiffres suivants :

- pour la **base 2** : les chiffres 0 et 1.
- pour la **base 16** : on retrouve les chiffres de la base 10, plus quelques lettres : 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F

On appelle la base 2, la **base binaire**. Elle représente des états logiques 0 ou 1. Dans un signal numérique, ces états correspondent à des niveaux de tension. En électronique numérique, très souvent il s'agira d'une tension de 0V pour un état logique 0 ; d'une tension de 5V pour un état logique 1. On parle aussi de niveau HAUT ou BAS (in english : HIGH or LOW). Elle existe à cause de la conception physique des ordinateurs. En effet, ces derniers utilisent des millions de transistors, utilisés pour traiter des données binaires, donc deux états distincts uniquement (0 ou 1). Pour compter en base 2, ce n'est pas très difficile si vous avez saisi ce qu'est une base. Dans le cas de la base 10, chaque chiffre était multiplié par 10 à une certaine puissance en partant de la puissance 0. Eh bien en base 2, plutôt que d'utiliser 10, on utilise 2. Par exemple, pour obtenir 11 en base 2 on écrira : 1011... En effet, cela équivaut à faire : $1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$ soit : $1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$

[[information]] | Un chiffre en base 2 s'appelle un **bit**. Un regroupement de 8 bits s'appelle un **octet**. Ce vocabulaire est très important donc retenez-le !

La base 16, ou **base hexadécimale** est utilisée en programmation, notamment pour représenter des octets facilement. Reprenons nos bits. Si on en utilise quatre, on peut représenter des nombres de 0 (0000) à 15 (1111). Ça tombe bien, c'est justement la portée d'un nombre hexadécimale ! En effet, comme dit plus haut il va de 0 (0000 ou 0) à F (1111 ou 15), ce qui représente 16 "chiffres" en hexadécimal. Grâce à cela, on peut représenter "simplement" des octets, en utilisant juste deux chiffres hexadécimaux.

2.2.4.1.3 Les notations Ici, rien de très compliqué, je vais simplement vous montrer comment on peut noter un nombre en disant à quelle base il appartient.

- Base binaire : $(10100010)_2$
- Base décimale : $(162)_{10}$
- Base hexadécimale : $(A2)_{16}$

À présent, voyons les différentes méthodes pour passer d'une base à l'autre grâce aux **conversions**.

2.2.4.2 Conversions

Souvent, on a besoin de convertir les nombres dans des bases différentes. On retrouvera deux méthodes, bonnes à savoir l'une comme l'autre. La première vous apprendra à faire les conversions

“à la main”, vous permettant de bien comprendre les choses. La seconde, celle de la calculatrice, vous permettra de faire des conversions sans vous fatiguer.

2.2.4.2.1 Décimale - Binaire Pour convertir un nombre décimal (en base 10) vers un nombre binaire (en base 2, vous suivez c'est bien!), il suffit de savoir diviser par ... 2! Ça ira? Prenez votre nombre, puis divisez le par 2. Divisez ensuite le quotient obtenu par 2... puis ainsi de suite jusqu'à avoir un quotient nul. Il vous suffit alors de lire les restes de bas en haut pour obtenir votre nombre binaire... Par exemple le nombre 42 s'écrira 101010 en binaire. Voilà un schéma de démonstration de cette méthode :

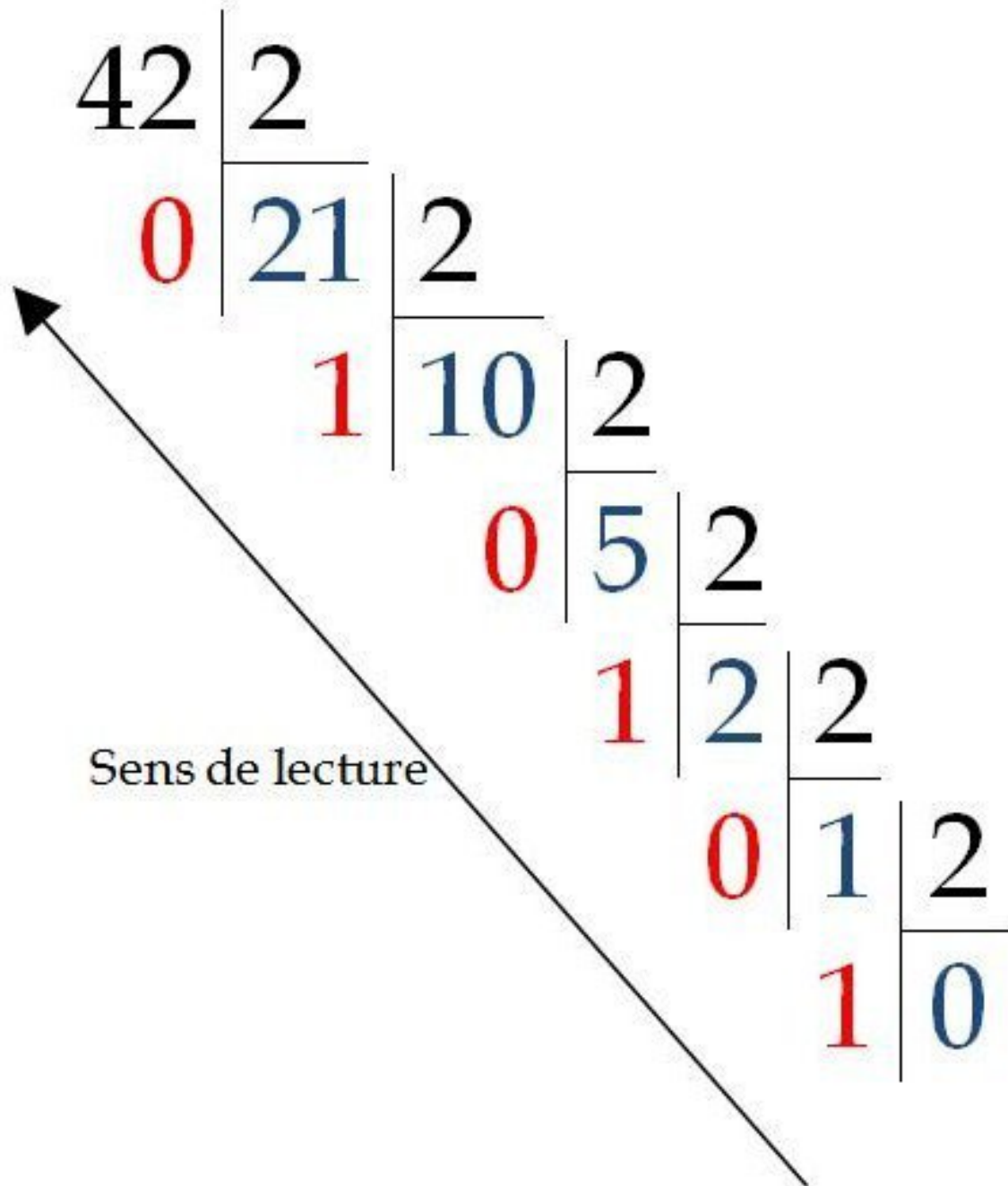


Figure : On garde les restes (en rouge) et on lit le résultat de bas en haut.

2.2.4.2.2 Binaire - Hexadécimal La conversion de binaire à l'hexadécimal est la plus simple à réaliser. Tout d'abord, commencez à regrouper les bits par blocs de quatre en commençant par la droite. S'il n'y a pas assez de bits à gauche pour faire le dernier groupe de quatre, on rajoute des zéros. Prenons le nombre 42, qui s'écrit en binaire, on l'a vu, **101010**, on obtiendra deux groupes de 4 bits qui seront **0010 1010**. Ensuite, il suffit de calculer bloc par bloc pour obtenir un chiffre hexadécimal en prenant en compte la valeur de chaque bit. Le premier bit, de poids faible (tout à droite), vaudra par exemple $A (1 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 = 10 : A \text{ en hexadécimal})$. Ensuite, l'autre bloc vaudra simplement $2 (0 \times 8 + 0 \times 4 + 1 \times 2 + 0 \times 1 = 2)$. Donc 42 en base décimale vaut 2A en base hexadécimale, ce qui s'écrit aussi $(42)_{10} = (2A)_{16}$. Pour passer de hexadécimal à binaire, il suffit de faire le fonctionnement inverse en s'aidant de la base décimale de temps en temps. La démarche à suivre est la suivante :

- Je sépare les chiffres un par un (on obtient 2 et A)
- Je "convertis" leurs valeurs en décimal (ce qui nous fait 2 et 10)
- Je met ces valeurs en binaire (et on a donc 0010 1010)

2.2.4.2.3 Décimal - Hexadécimal Ce cas est plus délicat à traiter, car il nécessite de bien connaître la table de multiplication par 16. :euh : Comme vous avez bien suivi les explications précédentes, vous comprenez comment faire ici... Mais comme je suis nul en math, je vous conseillerais de faire un passage par la base binaire pour faire les conversions !

2.2.4.2.4 Méthode rapide Pour cela, je vais dans *Démarrer / Tous les programmes / Accessoires / Calculatrice* (ou en faisant une petite recherche dans le menu Démarrer ou directement depuis l'écran d'accueil de Windows 8). Qui a dit que j'étais fainéant ? :colere2 :

[[information]] | Pour obtenir cet affichage, il vous faudra peut-être utiliser le menu *Affichage* puis sélectionner *Programmeur*.

Sur le côté gauche, il y a des options à cocher pour afficher le nombre entré dans la base que l'on veut. Présentement, je suis en base 10 (décimale - bouton *Déc*). Si je clique sur *Hex* :

Je vois que mon nombre 42 a été converti en : **2A**. Et maintenant, si je clique sur *Bin* :

Notre nombre a été converti en : **00101010** !

2.3 Le logiciel

Afin de vous laisser un peu de temps supplémentaire pour vous procurer votre carte Arduino, je vais vous montrer brièvement comment se présente le logiciel Arduino.

2.3.1 Installation

Il n'y a pas besoin d'installer le logiciel Arduino sur votre ordinateur puisque ce dernier est une version portable. Regardons ensemble les étapes pour préparer votre ordinateur à l'utilisation de la carte Arduino.

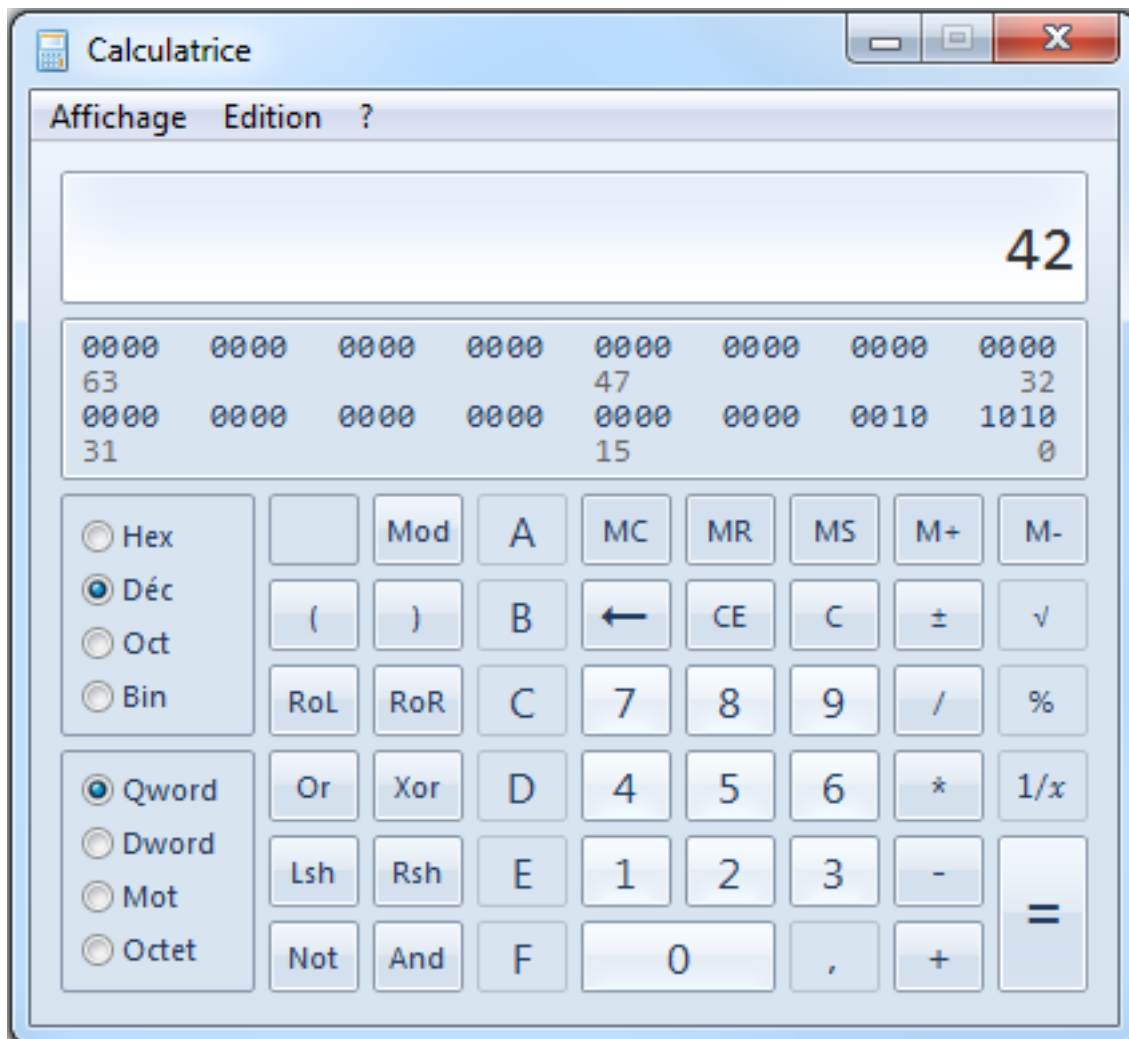


Figure 2.9 – Utilisation de la calculatrice Windows



Figure 2.10 – Sélection de Hex



Figure 2.11 – Sélection de Bin

2.3.1.1 Téléchargement

Pour télécharger le logiciel, il faut se rendre sur [la page de téléchargement du site arduino.cc](http://arduino.cc) . Vous avez deux catégories :

- Download : Dans cette catégorie, vous pouvez télécharger la dernière version du logiciel. Les plateformes Windows, Linux et Mac sont supportées par le logiciel. **C'est donc ici que vous allez télécharger le logiciel.**
- Previous IDE Versions : Dans cette catégorie-là, vous avez toutes les versions du logiciel, sous les plateformes précédemment citées, depuis le début de sa création.

2.3.1.1.1 Sous Windows Pour moi ce sera sous Windows. Je clique sur le lien **Windows** et le fichier apparaît et doit être enregistré ou bon vous semble.

Une fois que le téléchargement est terminé, vous n'avez plus qu'à décompresser le fichier avec un utilitaire de décompression (7-zip, WinRar, ...). À l'intérieur du dossier se trouvent quelques fichiers et l'exécutable du logiciel :

2.3.1.1.2 Mac os Cliquez sur le lien Mac OS. Un fichier **.dmg** apparaît. Enregistrez-le.

Double-cliquez sur le fichier **.dmg** :

On y trouve l'application Arduino (**.app**), mais aussi le driver à installer (**.mpkg**). Procédez à l'installation du driver puis installez l'application en la glissant dans le raccourci du dossier "Applications" qui est normalement présent sur votre ordinateur.

2.3.1.1.3 Sous Linux Rien de plus simple, en allant dans la logithèque, recherchez le logiciel "Arduino". Sinon vous pouvez aussi passer par la ligne de commande :

```
$ sudo apt-get install arduino
```

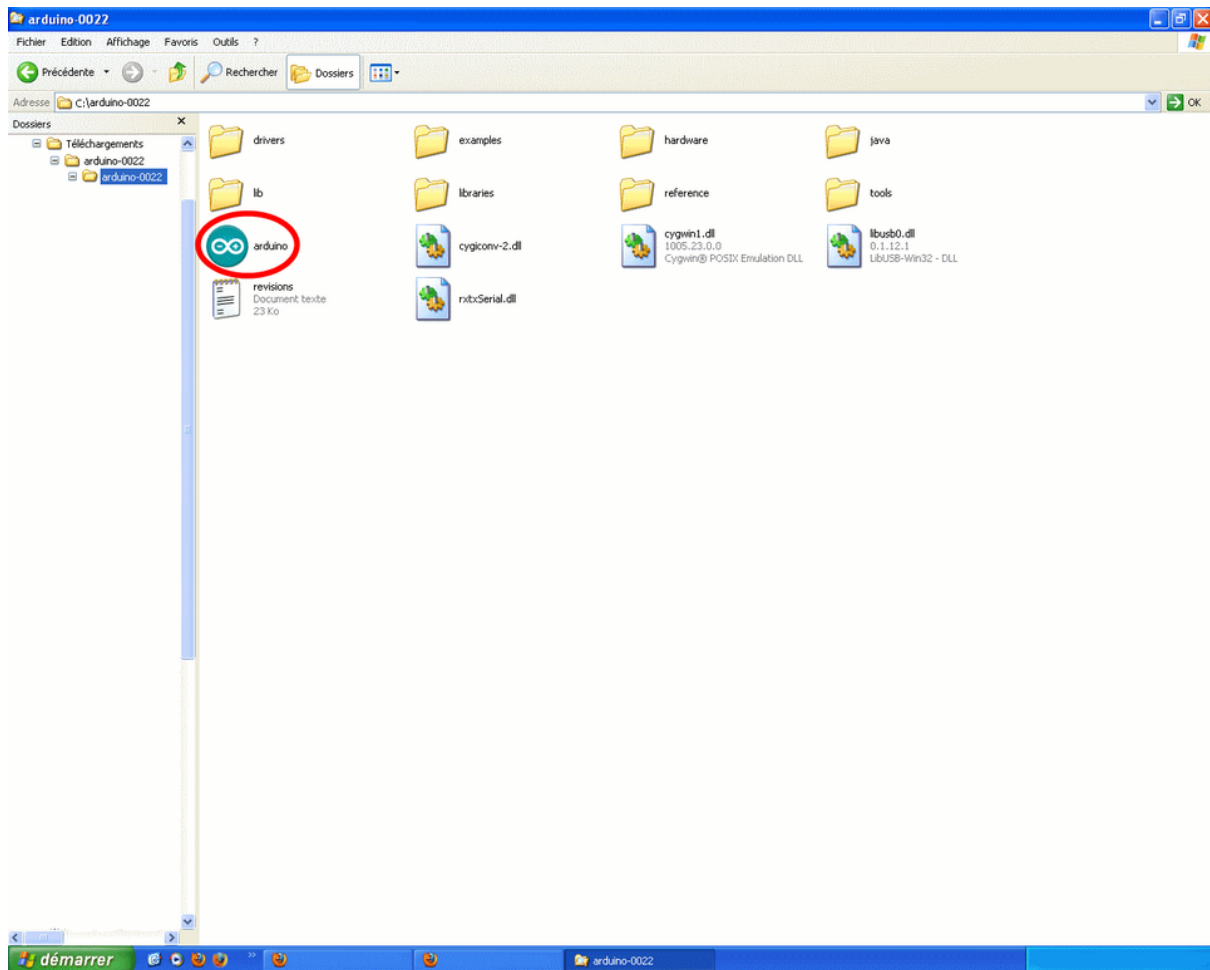


Figure 2.12 – Exécutable du logiciel Arduino

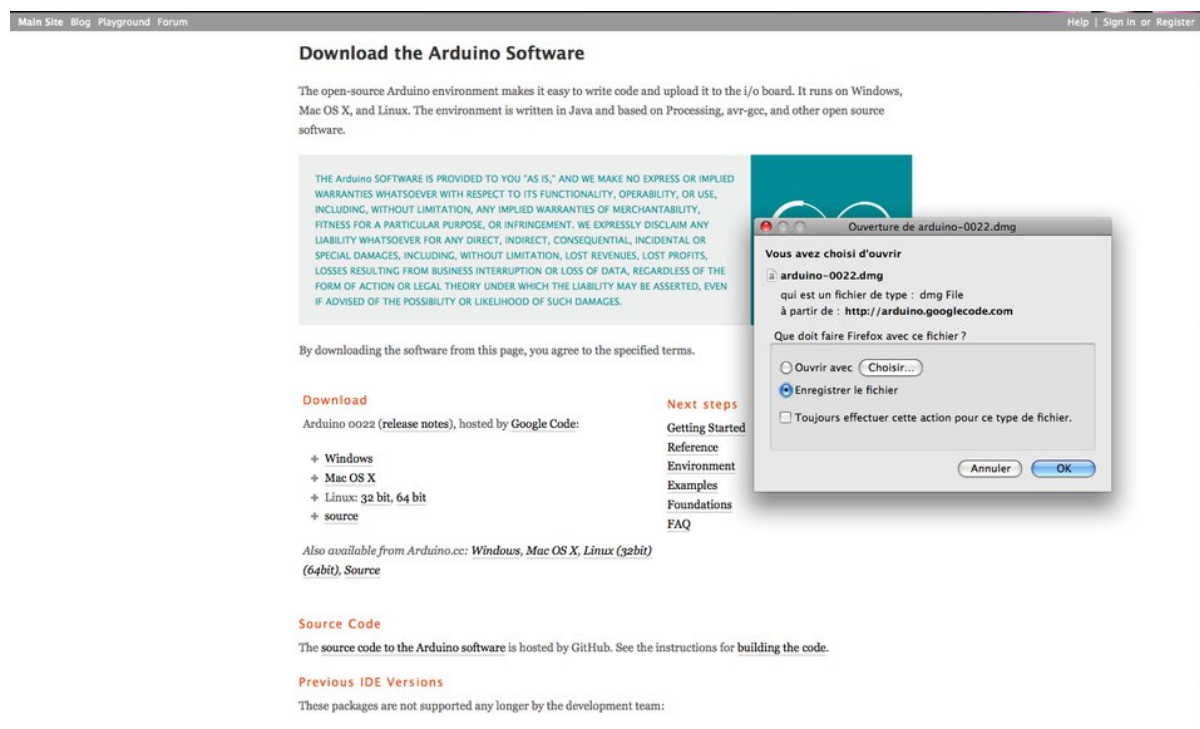


Figure 2.13 – Téléchargement sous Mac OS

Plusieurs dépendances seront installées en même temps.

[[information]] | Je rajoute [un lien](#) qui vous mènera vers la page officielle.

2.3.2 Interface du logiciel

2.3.2.1 Lancement du logiciel

Lançons le logiciel en double-cliquant sur l'icône avec le symbole "infinie" en vert. C'est l'exécutible du logiciel. Après un léger temps de réflexion, une image s'affiche :

Cette fois, après quelques secondes, le logiciel s'ouvre. Une fenêtre se présente à nous :

Ce qui saute aux yeux en premier, c'est la clarté de présentation du logiciel. On voit tout de suite son interface intuitive. Voyons comment se compose cette interface.

2.3.2.2 Présentation du logiciel

J'ai découpé, grâce à mon ami paint.net, l'image précédente en plusieurs parties :

2.3.2.2.1 Correspondance

- Le cadre numéro 1 : ce sont les options de configuration du logiciel
- Le cadre numéro 2 : il contient les boutons qui vont nous servir lorsque l'on va programmer nos cartes
- Le cadre numéro 3 : ce bloc va contenir le programme que nous allons créer
- Le cadre numéro 4 : celui-ci est important, car il va nous aider à corriger les fautes dans notre programme. C'est le **débogueur**.

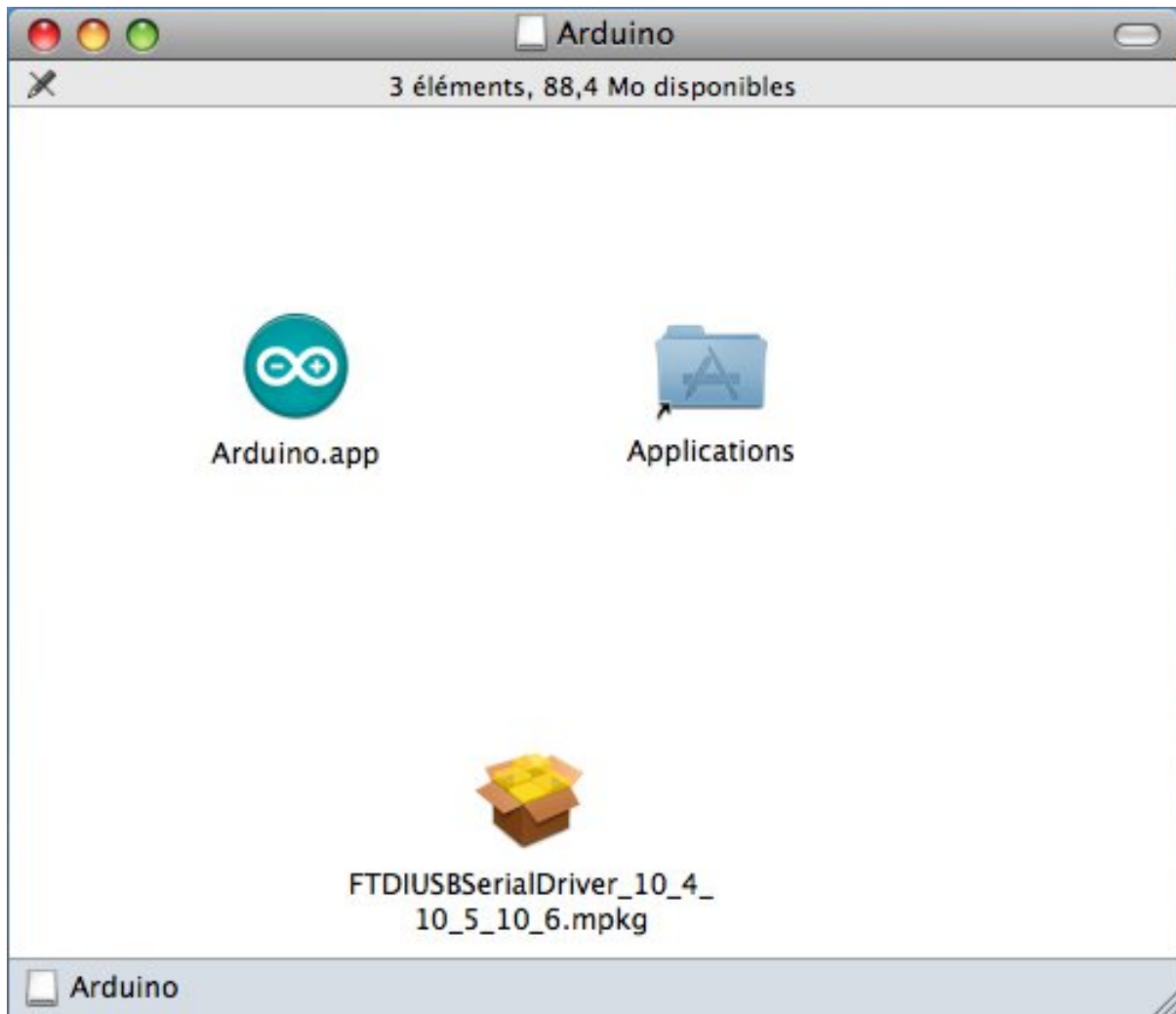


Figure 2.14 – Contenu du téléchargement

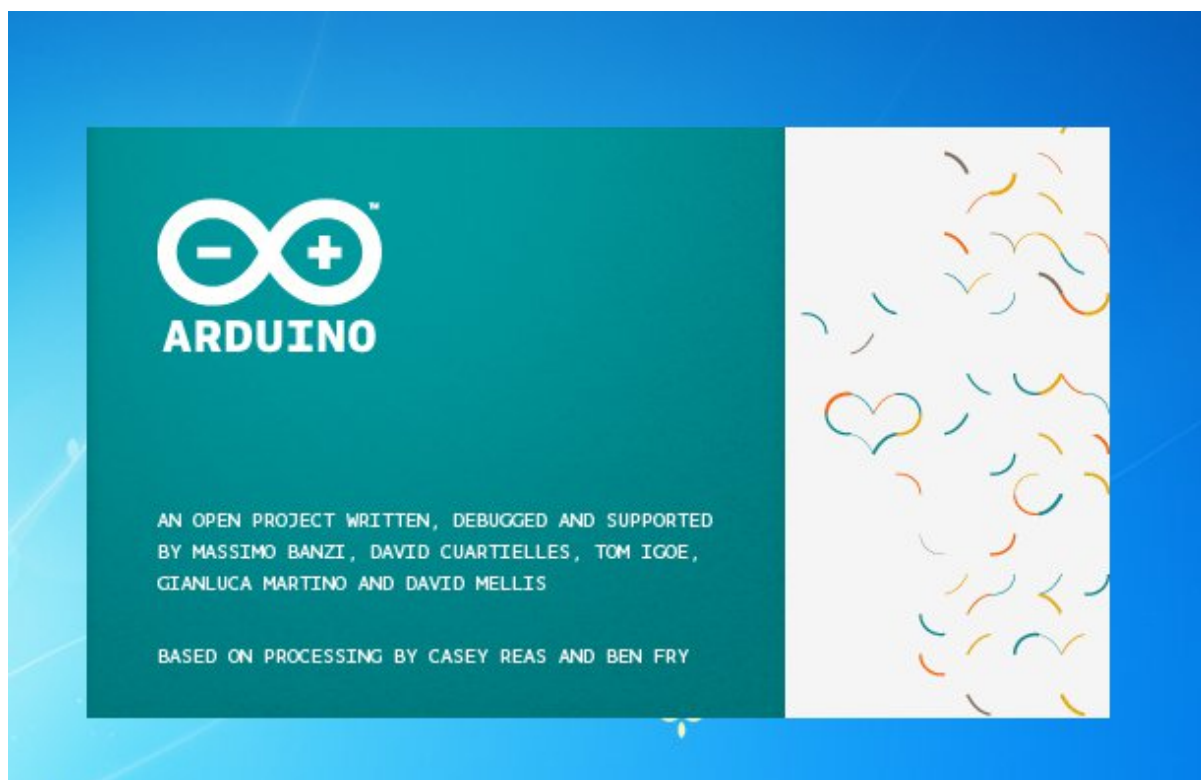


Figure 2.15 – Le splash screen Arduino

2.3.3 Approche et utilisation du logiciel

Attaquons-nous plus sérieusement à l'utilisation du logiciel. La barre des menus est entourée en rouge et numérotée par le chiffre 1.

2.3.3.1 Le menu *File*

C'est principalement ce menu que l'on va utiliser le plus. Il dispose d'un certain nombre de choses qui vont nous être très utiles. Il a été traduit en français progressivement, nous allons donc voir les quelques options qui sortent de l'ordinaire :

- *Carnet de croquis* : Ce menu regroupe les fichiers que vous avez pu faire jusqu'à maintenant (et s'ils sont enregistrés dans le dossier par défaut du logiciel).
- *Exemples (exemples)* : Ceci est important, toute une liste se déroule pour afficher les noms d'exemples de programmes existants ; avec ça, vous pourrez vous aider/inspirer pour créer vos propres programmes ou tester de nouveaux composants.
- *Téléverser* : Permet d'envoyer le programme sur la carte Arduino. Nous y reviendrons ;).
- *Téléverser avec un programmeur* : Idem que ci-dessus, mais avec l'utilisation d'un programmeur (vous n'en n'aurez que très rarement besoin).
- *Préférences* : Vous pourrez régler ici quelques paramètres du logiciel. Le reste des menus n'est pas intéressant pour l'instant, on y reviendra plus tard, avant de commencer à programmer.

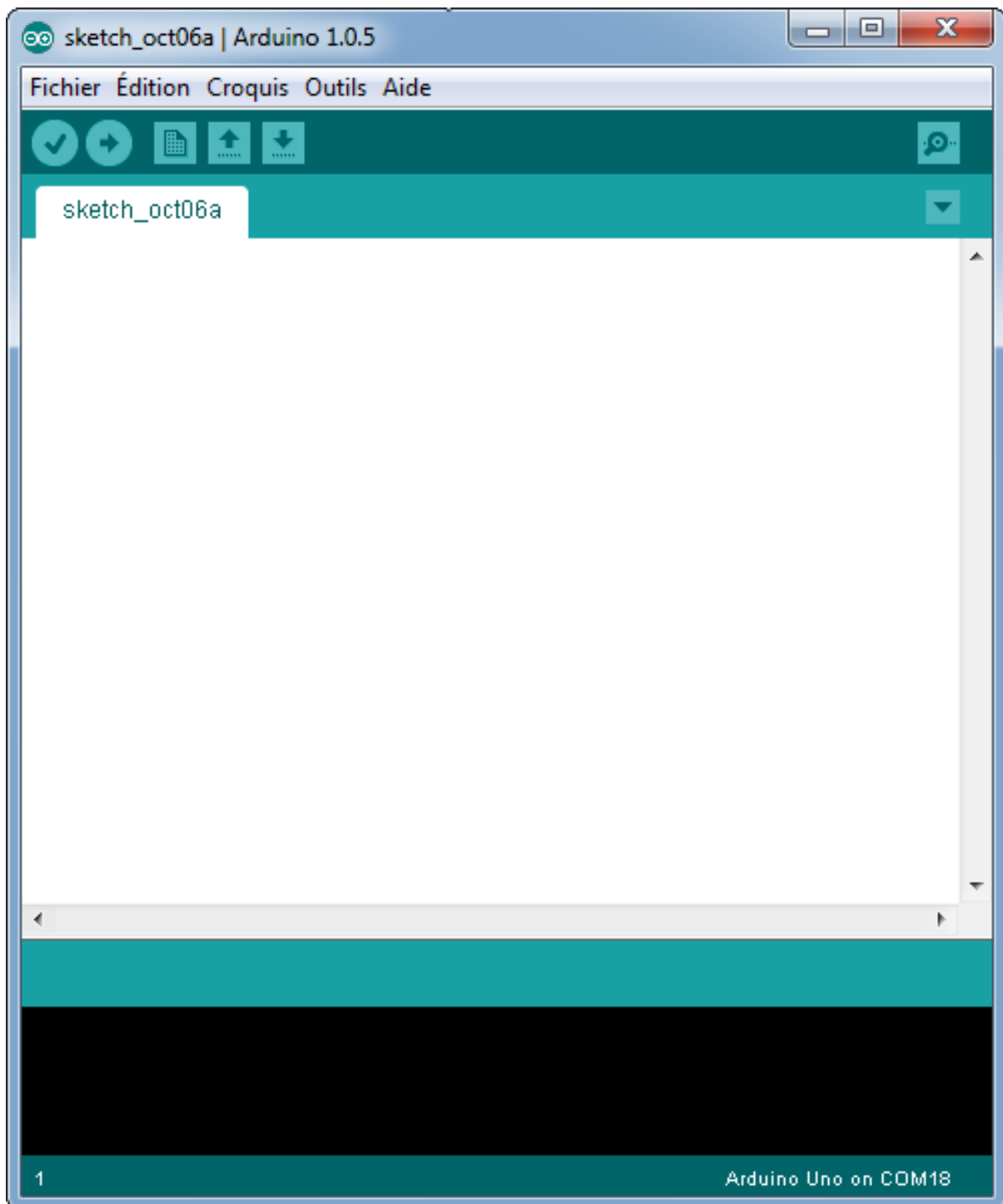


Figure 2.16 – L'interface de l'IDE Arduino

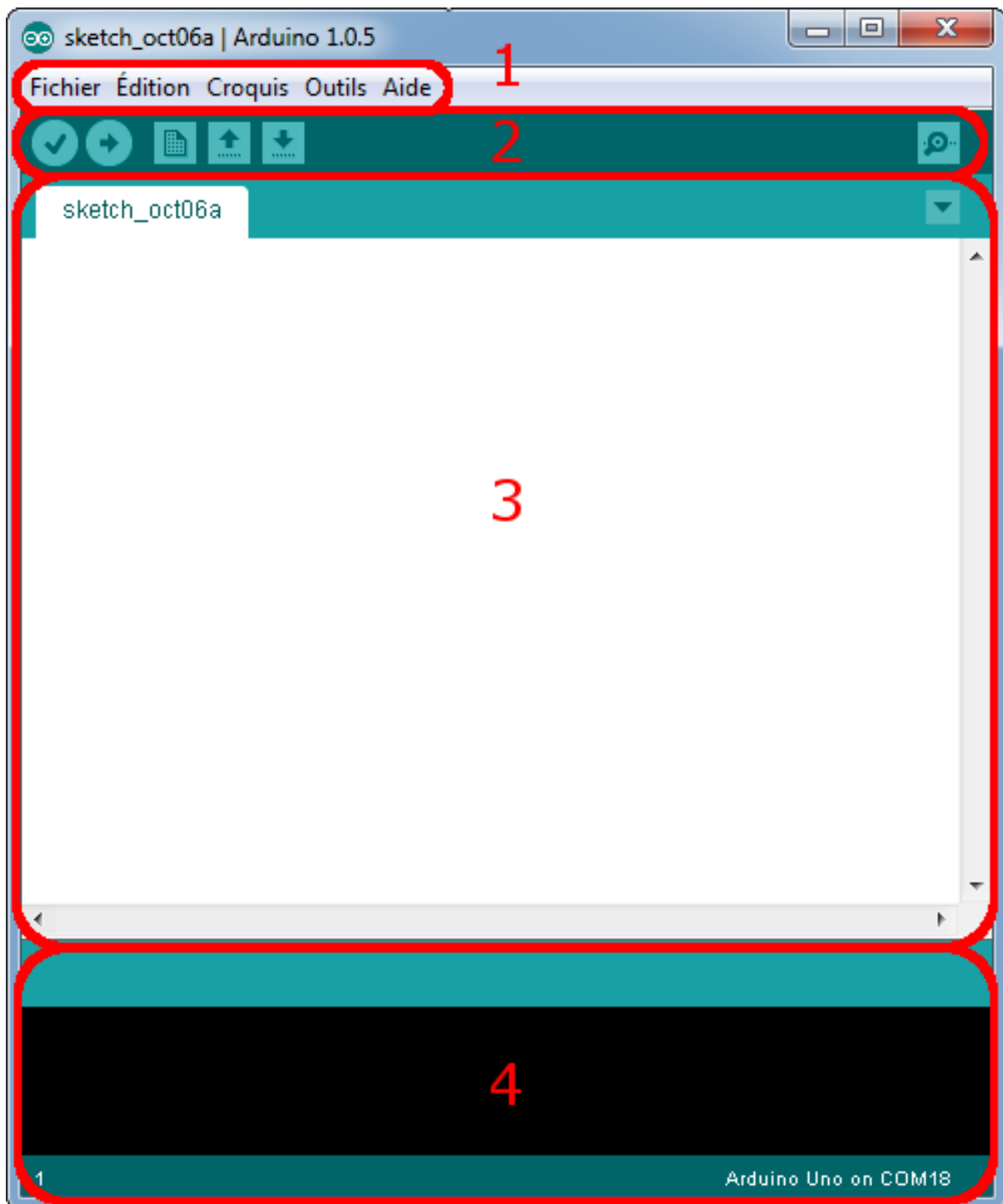


Figure 2.17 – L'interface de l'IDE Arduino en détail

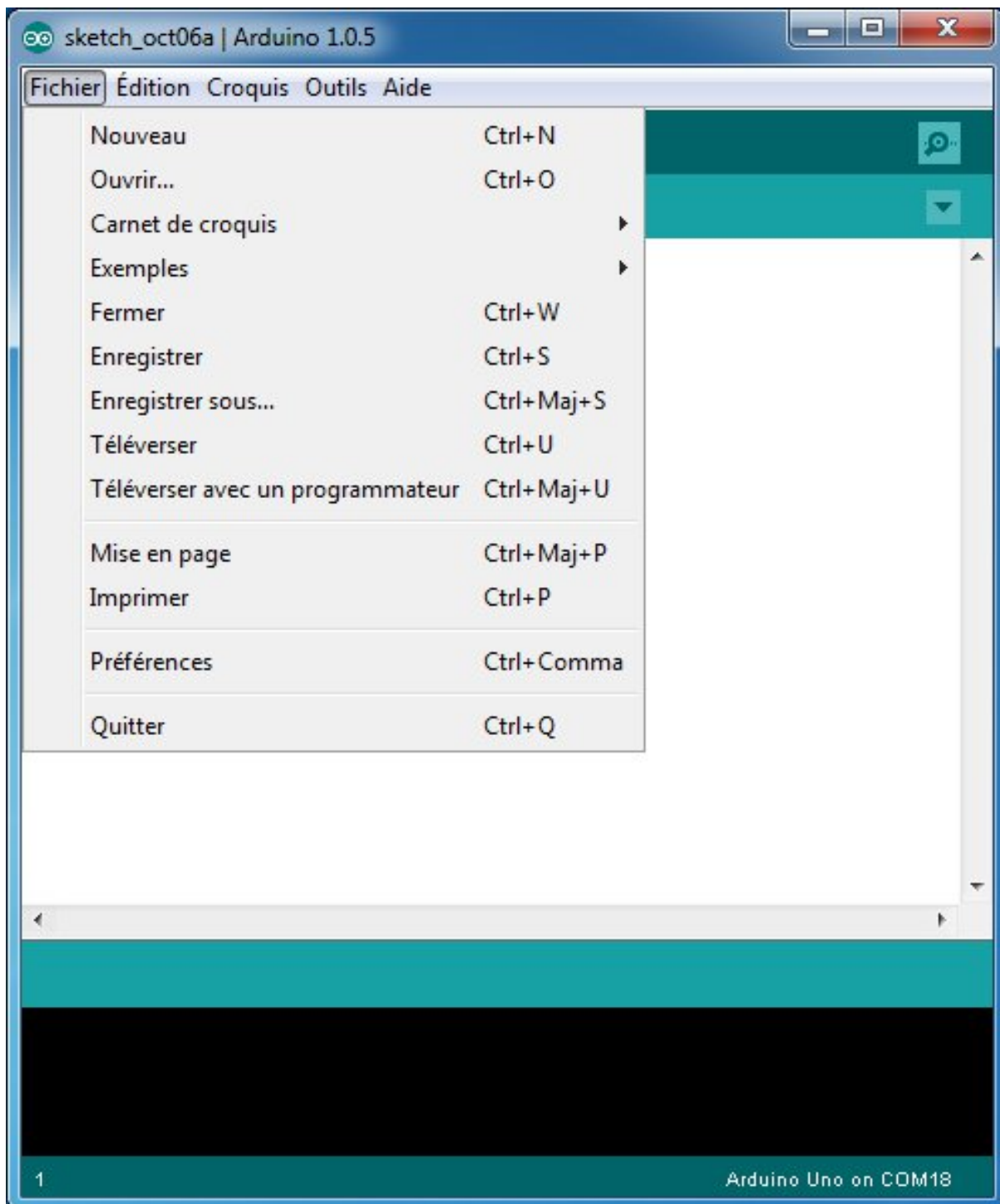


Figure 2.18 – Le menu Fichier

2.3.3.2 Les boutons

Voyons à présent à quoi servent les boutons, encadrés en rouge et numérotés par le chiffre 2.

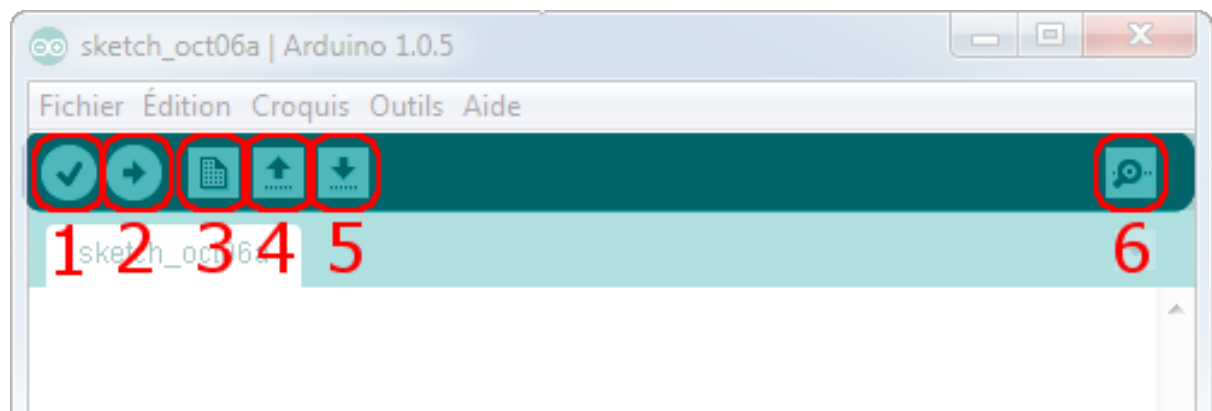


Figure 2.19 – La barre d’outils

- Bouton 1 : Ce bouton permet de vérifier le programme, il actionne un module qui cherche les erreurs dans votre programme
- Bouton 2 : Charge (téléverse) le programme dans la carte Arduino.
- Bouton 3 : Crée un nouveau fichier.
- Bouton 4 : Ouvre un fichier.
- Bouton 5 : Enregistre le fichier.
- Bouton 6 : Ouvre le moniteur série (on verra plus tard ce que c’est ;)).

Enfin, on va pouvoir s’occuper du matériel que vous devriez tous posséder en ce moment même : la carte Arduino!

2.4 Le matériel

Dans ce chapitre, je vais vous montrer brièvement comment se présente la carte Arduino, comment l’installer et son fonctionnement global.

2.4.1 Présentation de la carte

Pour commencer notre découverte de la carte Arduino, je vais vous présenter la carte en elle-même. Nous allons voir comment s’en servir et avec quoi. J’ai représenté en rouge sur cette photo les points importants de la carte :

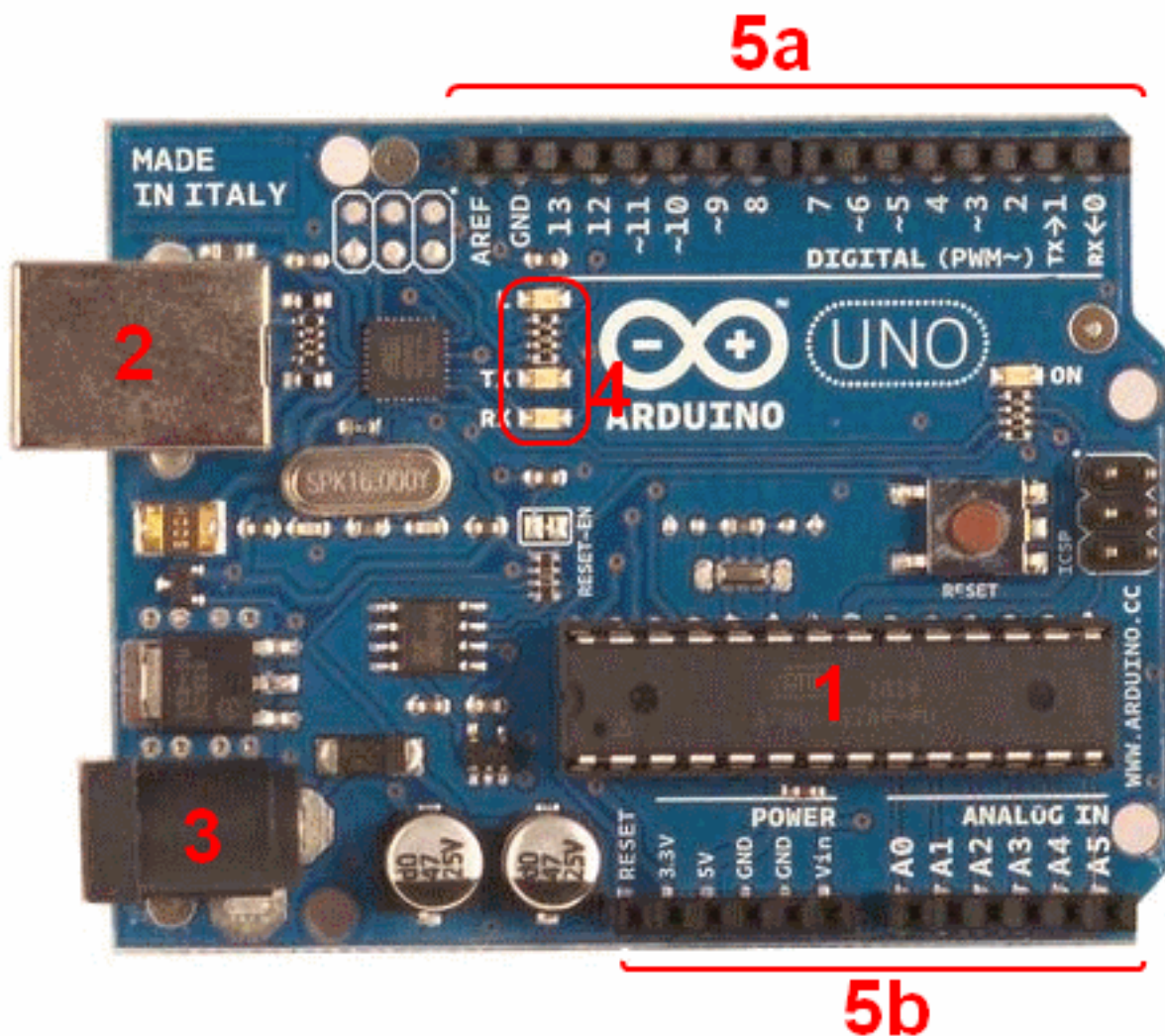


Figure : Présentation de la carte Arduino – (CC-BY-SA, arduino.cc)

2.4.1.1 Constitution de la carte

Voyons quels sont ces points importants et à quoi ils servent.

2.4.1.1.1 Le micro-contrôleur Voilà le cerveau de notre carte (en 1). C'est lui qui va recevoir le programme que vous aurez créé et qui va le stocker dans sa mémoire puis l'exécuter. Grâce à ce programme, il va savoir faire des choses, qui peuvent être : faire clignoter une LED, afficher des caractères sur un écran, envoyer des données à un ordinateur, ... ##### Alimentation

Pour fonctionner, la carte a besoin d'une alimentation. Le microcontrôleur fonctionnant sous 5V, la carte peut être alimentée en 5V par le port USB (en 2) ou bien par une alimentation externe (en 3) qui est comprise entre 7V et 12V. Cette tension doit être continue et peut par exemple être fournie par une pile 9V. Un régulateur se charge ensuite de réduire la tension à 5V pour le bon fonctionnement de la carte. Pas de danger de tout griller donc ! Veuillez seulement à respecter l'intervalle de 7V à 15V (même si le régulateur peut supporter plus, pas la peine de le retrancher

dans ses limites) ##### Visualisation

Les trois “points blancs” entourés en rouge (4) sont en fait des LED dont la taille est de l'ordre du millimètre. Ces LED servent à deux choses :

- Celle tout en haut du cadre : elle est connectée à une broche du microcontrôleur et va servir pour tester le matériel. *Nota* : Quand on branche la carte au PC, elle clignote quelques secondes.
- Les deux LED du bas du cadre : servent à visualiser l'activité sur la voie série (une pour l'émission et l'autre pour la réception). Le téléchargement du programme dans le microcontrôleur se faisant par cette voie, on peut les voir clignoter lors du chargement.

2.4.1.1.2 La connectique La carte Arduino ne possédant pas de composants qui peuvent être utilisés pour un programme, mis à part la LED connectée à la broche 13 du microcontrôleur, il est nécessaire de les rajouter. Mais pour ce faire, il faut les connecter à la carte. C'est là qu'intervient la connectique de la carte (en 5a et 5b). Par exemple, on veut connecter une LED sur une sortie du microcontrôleur. Il suffit juste de la connecter, avec une résistance en série, à la carte, sur les fiches de connexion de la carte.

Cette connectique est importante et a un brochage qu'il faudra respecter. Nous le verrons quand nous apprendrons à faire notre premier programme. C'est avec cette connectique que la carte est “extensible” car l'on peut y brancher tous types de montages et modules ! Par exemple, la carte Arduino Uno peut être étendue avec des shields, comme le « **Shield Ethernet** » qui permet de connecter cette dernière à internet.



Figure : Le

shield Ethernet Arduino - (CC-BY-SA - arduino.cc)

2.4.2 Installation

Afin d'utiliser la carte, il faut l'installer. Normalement, les *drivers* (pilote, en français) sont déjà installés sous GNU/Linux. Sous mac, il suffit de double cliquer sur le fichier *.mkpg* inclus dans le téléchargement de l'application Arduino et l'installation des drivers s'exécute de façon automatique.

2.4.2.1 Sous Windows

Lorsque vous connectez la carte à votre ordinateur sur le port USB, un petit message en bas de l'écran apparaît. Théoriquement, la carte que vous utilisez doit s'installer toute seule. Cependant, si vous êtes sous Windows 7 comme moi, il se peut que cela ne marche pas du premier coup. Dans ce cas, laissez la carte branchée puis allez dans le panneau de configuration. Cliquez ensuite sur "Système" puis, dans le panneau de gauche sélectionnez "gestionnaire de périphériques". Une fois ce menu ouvert, vous devriez voir un composant avec un panneau "attention" jaune. Faites un clic droit sur le composant et cliquez sur "Mettre à jour les pilotes". Dans le nouveau menu, sélectionnez l'option "Rechercher le pilote moi-même". Enfin, il ne vous reste plus qu'à aller sélectionner le bon dossier contenant le driver. Il se trouve dans le dossier d'Arduino que vous avez du décompresser un peu plus tôt et se nomme "drivers" (attention, ne descendez pas jusqu'au dossier "FTDI"). Par exemple, pour moi le chemin sera : le-chemin-du-dossier\arduino-0022\arduino-0022\drivers

[[attention]] Il semblerait qu'il y est des problèmes en utilisant la version française d'Arduino (les drivers sont absents du dossier). Si c'est le cas, il vous faudra télécharger la version originale (anglaise) pour pouvoir installer les drivers.

Après l'installation et une suite de clignotement sur les micro-LED de la carte, celle-ci devrait être fonctionnelle. Une petite LED verte témoigne de la bonne alimentation de la carte :

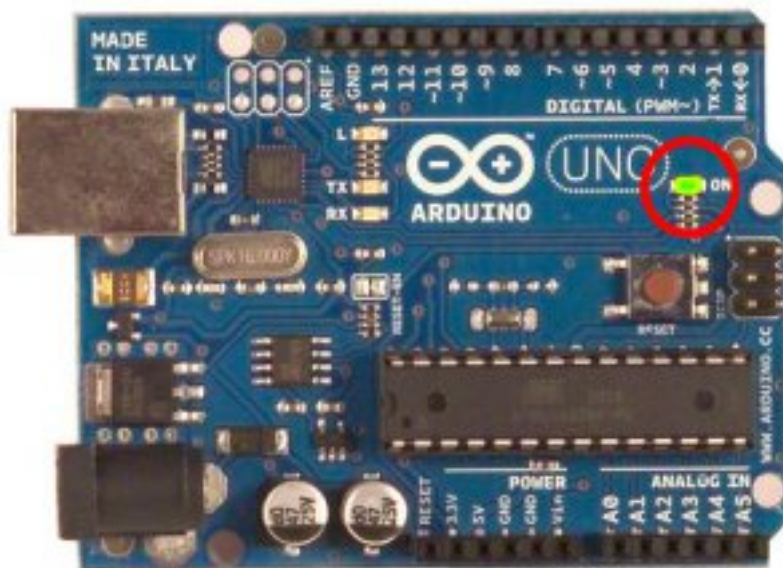


Figure 2.20 – Carte connectée et alimentée

2.4.2.2 Tester son matériel

Avant de commencer à programmer la tête baissée, il faut, avant toutes choses, tester le bon fonctionnement de la carte. Ce serait idiot de programmer la carte et chercher les erreurs dans le programme alors que le problème vient de la carte! ^^ Nous allons tester notre matériel en chargeant un programme qui fonctionne dans la carte.

[[question]] |Mais, on n'en a pas encore fait de programmes? o_O

Tout juste! Mais le logiciel Arduino contient des exemples de programmes. Et bien ce sont ces exemples que nous allons utiliser pour tester la carte.

2.4.2.2.1 1ère étape : ouvrir un programme Nous allons choisir un exemple tout simple qui consiste à faire clignoter une LED. Son nom est *Blink* et vous le trouverez dans la catégorie *Basics* :

Une fois que vous avez cliqué sur *Blink*, une nouvelle fenêtre va apparaître. Elle va contenir le programme *Blink*. Vous pouvez fermer l'ancienne fenêtre qui va ne nous servir plus à rien.

2.4.2.2.2 2e étape Avant d'envoyer le programme *Blink* vers la carte, il faut dire au logiciel quel est le nom de la carte et sur quel port elle est branchée. **Choisir la carte que l'on va programmer.** Ce n'est pas très compliqué, le nom de votre carte est indiqué sur elle. Pour nous, il s'agit de la carte "Uno". Allez dans le menu *Tools* ("outils" en français) puis dans *Board* ("carte" en français). Vérifiez que c'est bien le nom "Arduin Uno" qui est coché. Si ce n'est pas le cas, cochez-le.

Choisissez le port de connexion de la carte. Allez dans le menu *Tools*, puis *Serial port*. Là, vous choisissez le port COMX, X étant le numéro du port qui est affiché. Ne choisissez pas COM1 car il n'est quasiment jamais connecté à la carte. Dans mon cas, il s'agit de COM5 :

Pour trouver le port de connexion de la carte, vous pouvez aller dans le **gestionnaire de périphériques** qui se trouve dans le **panneau de configuration**. Regardez à la ligne **Ports (COM et LPT)** et là, vous devriez avoir **Arduino Uno (COMX)**. Aller, une image pour le plaisir :

2.4.2.2.3 Dernière étape Très bien. Maintenant, il va falloir envoyer le programme dans la carte. Pour ce faire, il suffit de cliquer sur le bouton *Téléverser*, en jaune-orangé sur la photo :

Vous verrez tout d'abord le message "Compilation du croquis en cours..." pour vous informer que le programme est en train d'être compilé en langage machine avant d'être envoyé. Ensuite vous aurez ceci :

En bas de l'image, vous voyez le texte : "Téléversement...", cela signifie que le logiciel est en train d'envoyer le programme dans la carte. Une fois terminé, il affiche un autre message :

Le message affiché : "Téléversement terminé" signale que le programme a bien été chargé dans la carte. Si votre matériel fonctionne, vous devriez avoir une LED sur la carte qui clignote :

[[attention]] |Si vous n'obtenez pas ce message mais plutôt un truc en rouge, pas d'inquiétude, le matériel n'est pas forcément défectueux!

En effet, plusieurs erreurs sont possibles :

- l'IDE recompile avant d'envoyer le code, vérifiez la présence d'erreurs.
- La voie série est peut-être mal choisie, vérifiez les branchements et le choix de la voie série.

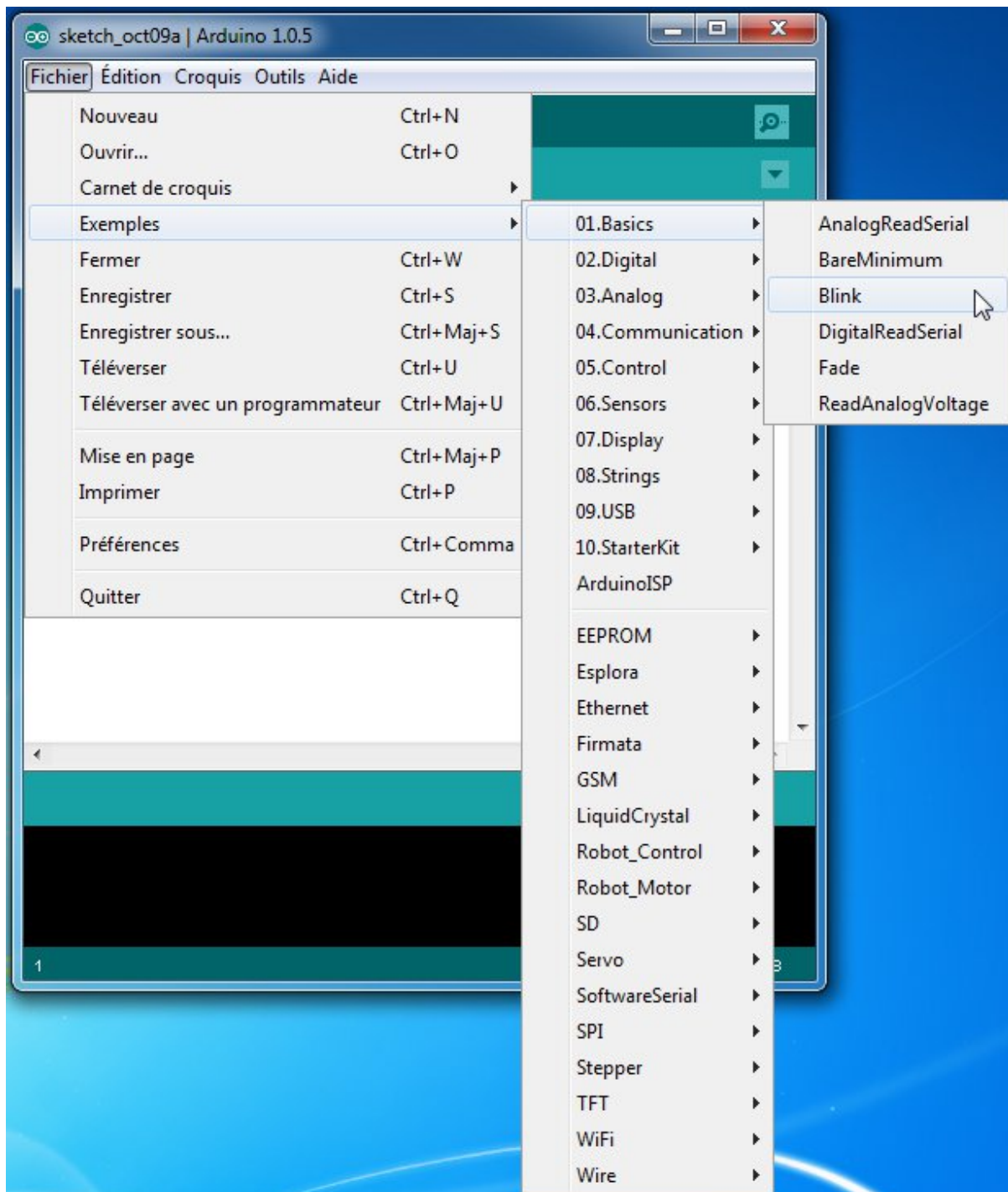


Figure 2.21 – Menu de choix d'un exemple

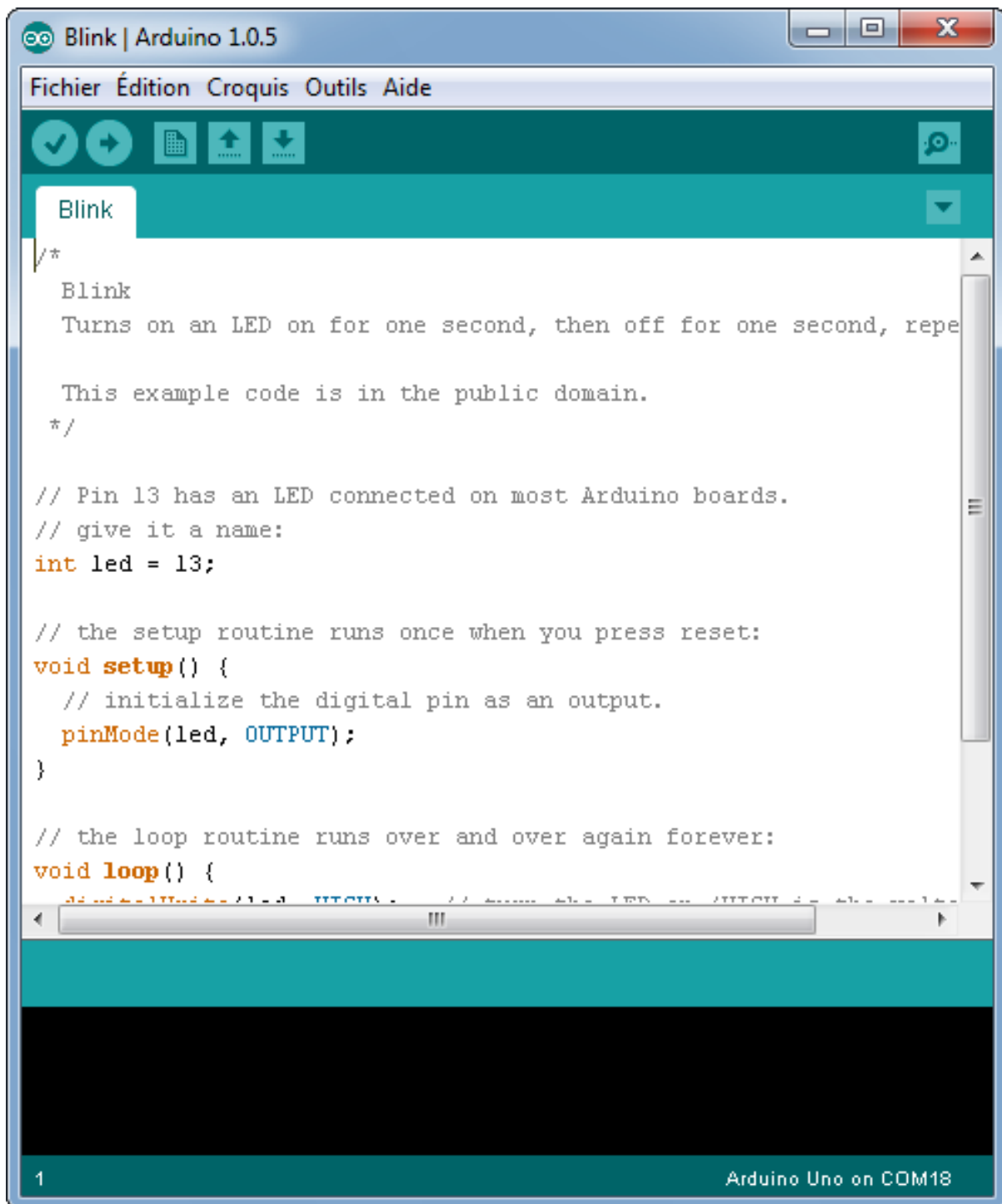


Figure 2.22 – Fenêtre du programme blink

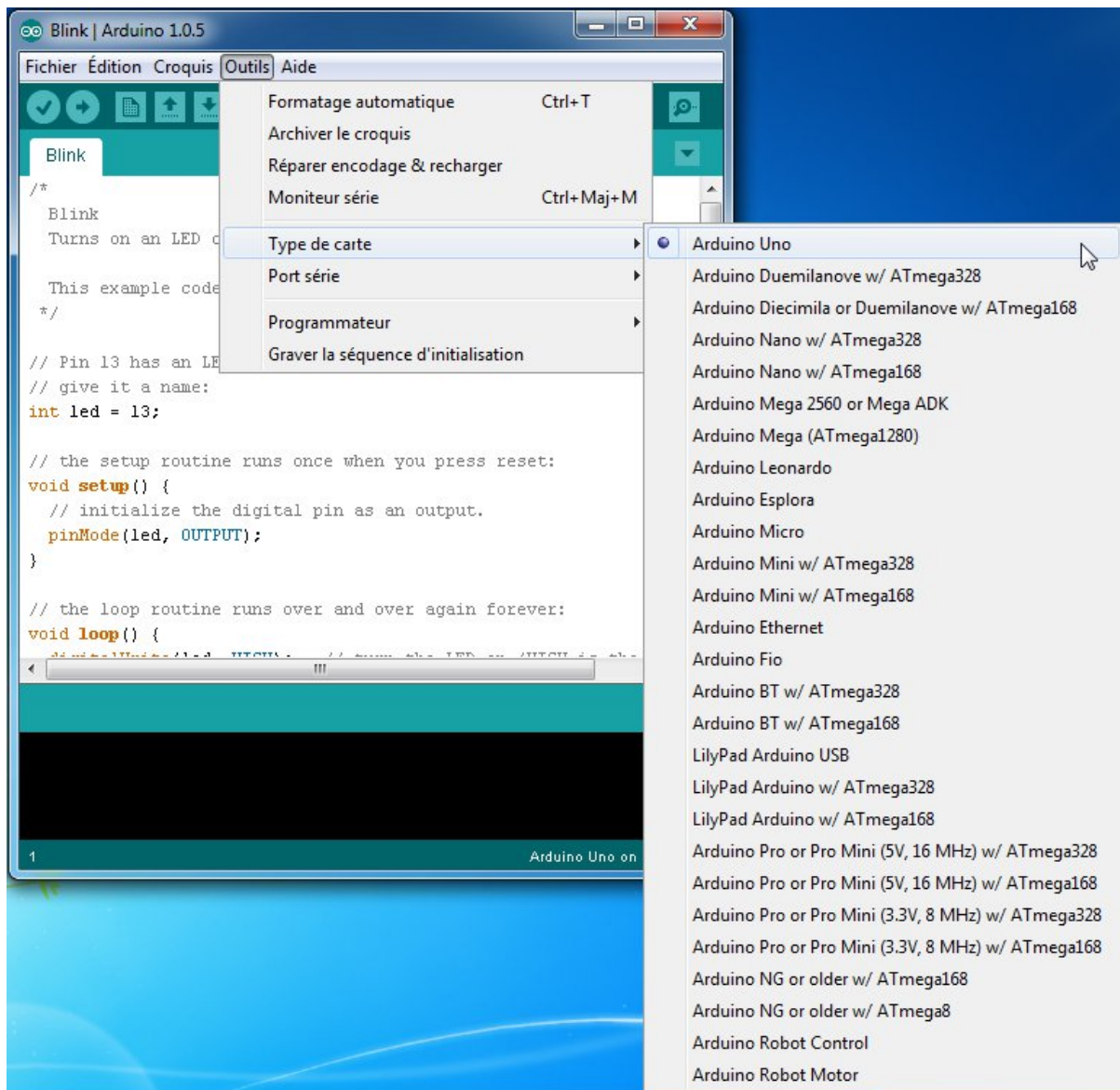


Figure 2.23 – Menu de sélection de la carte cible

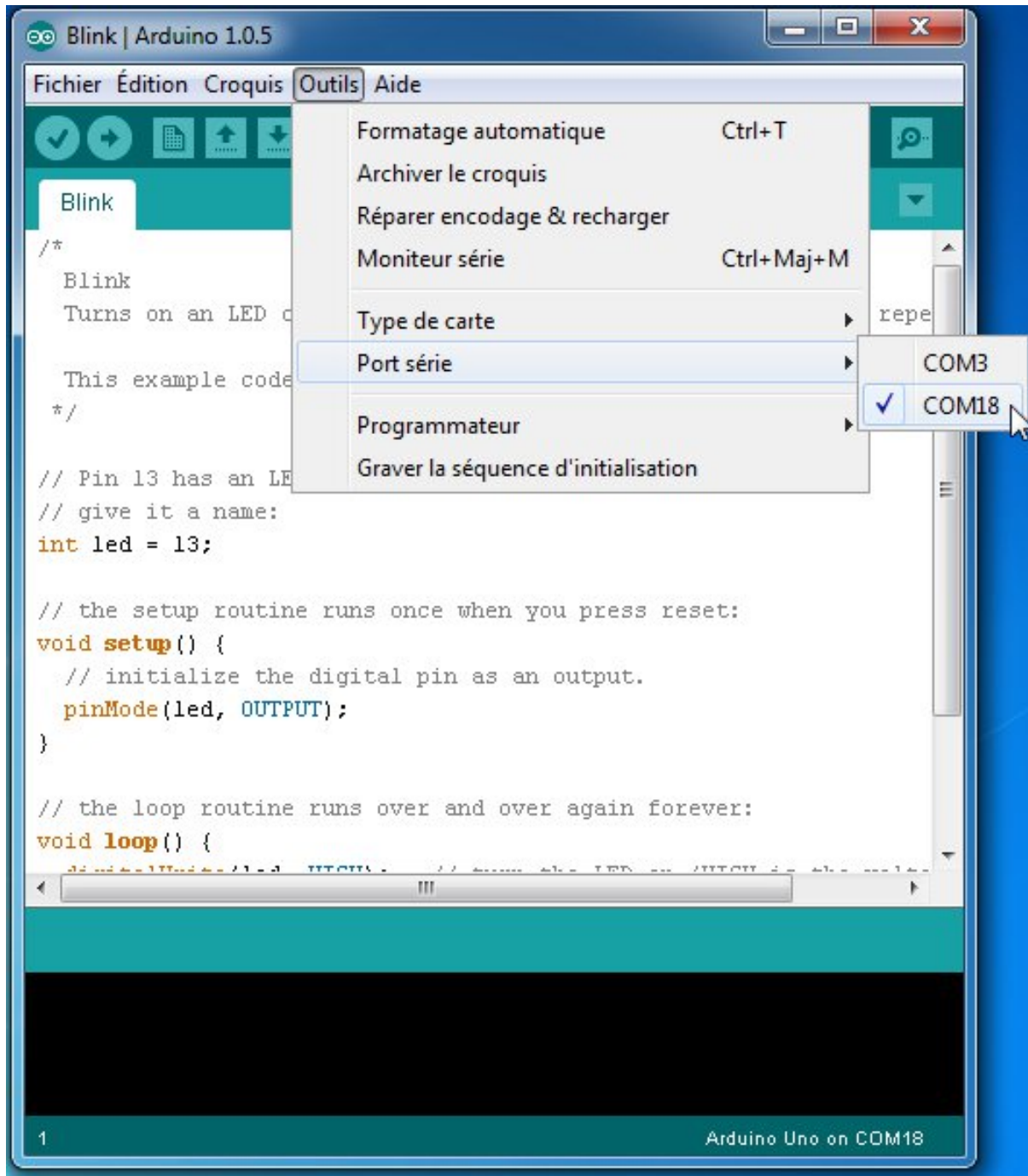


Figure 2.24 – Menu de sélection du port com

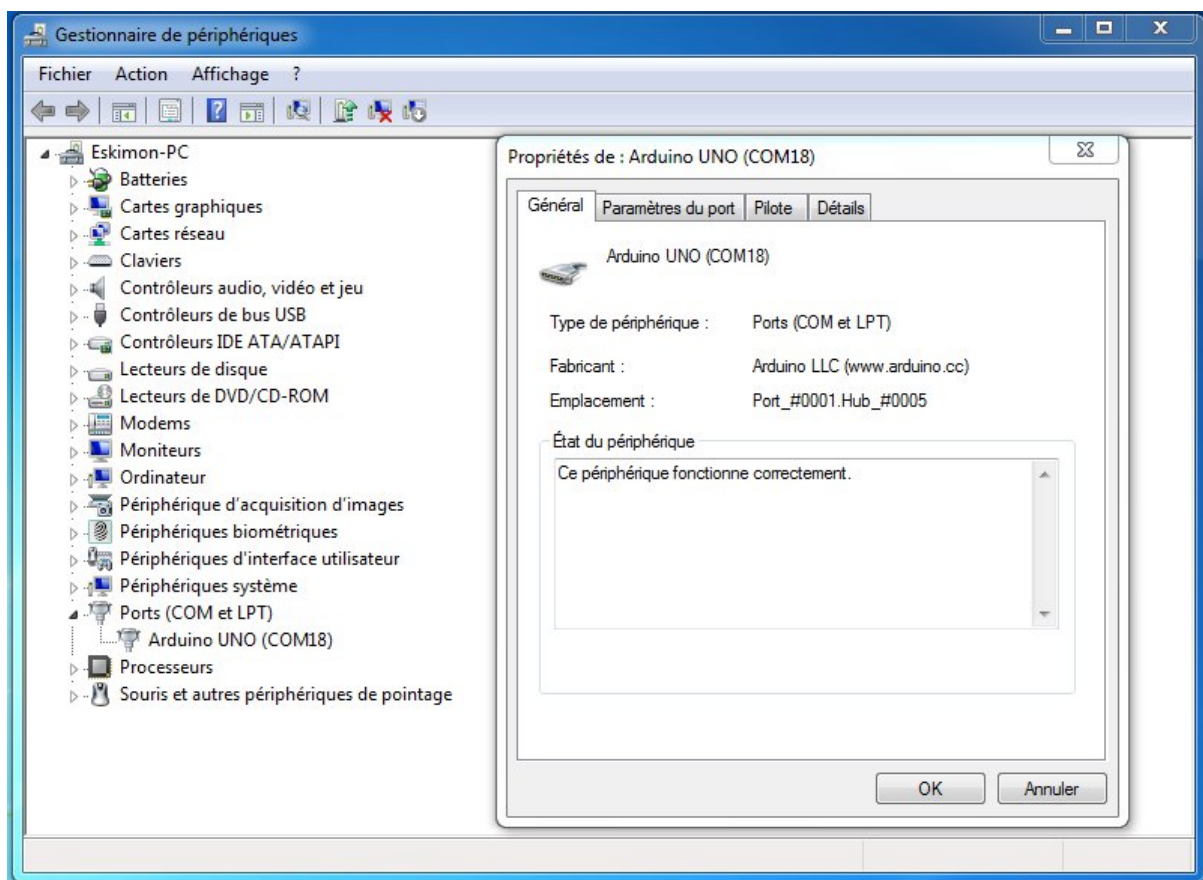


Figure 2.25 – Le panneau de configuration Windows et la carte Arduino

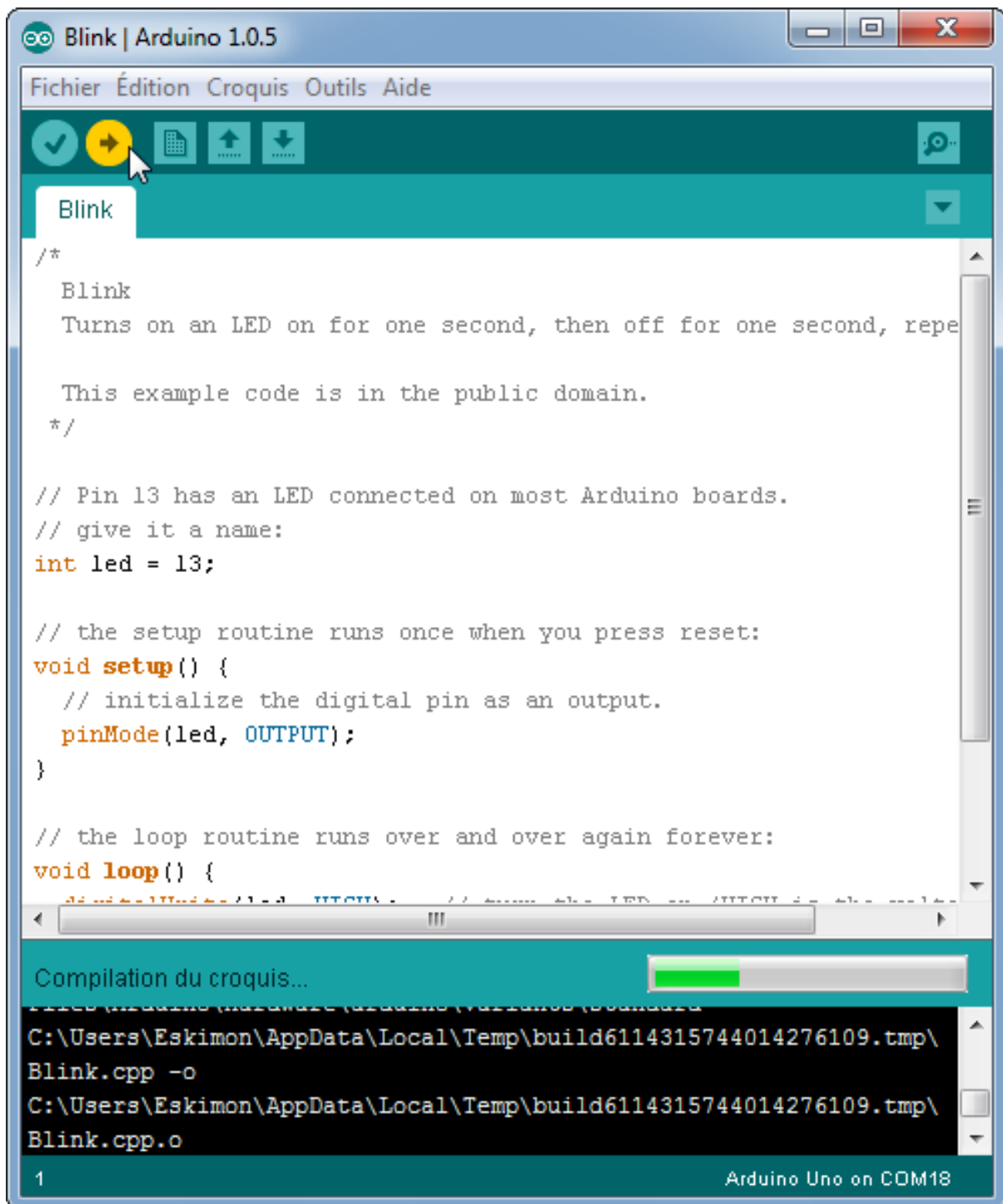


Figure 2.26 – Compilation en cours...

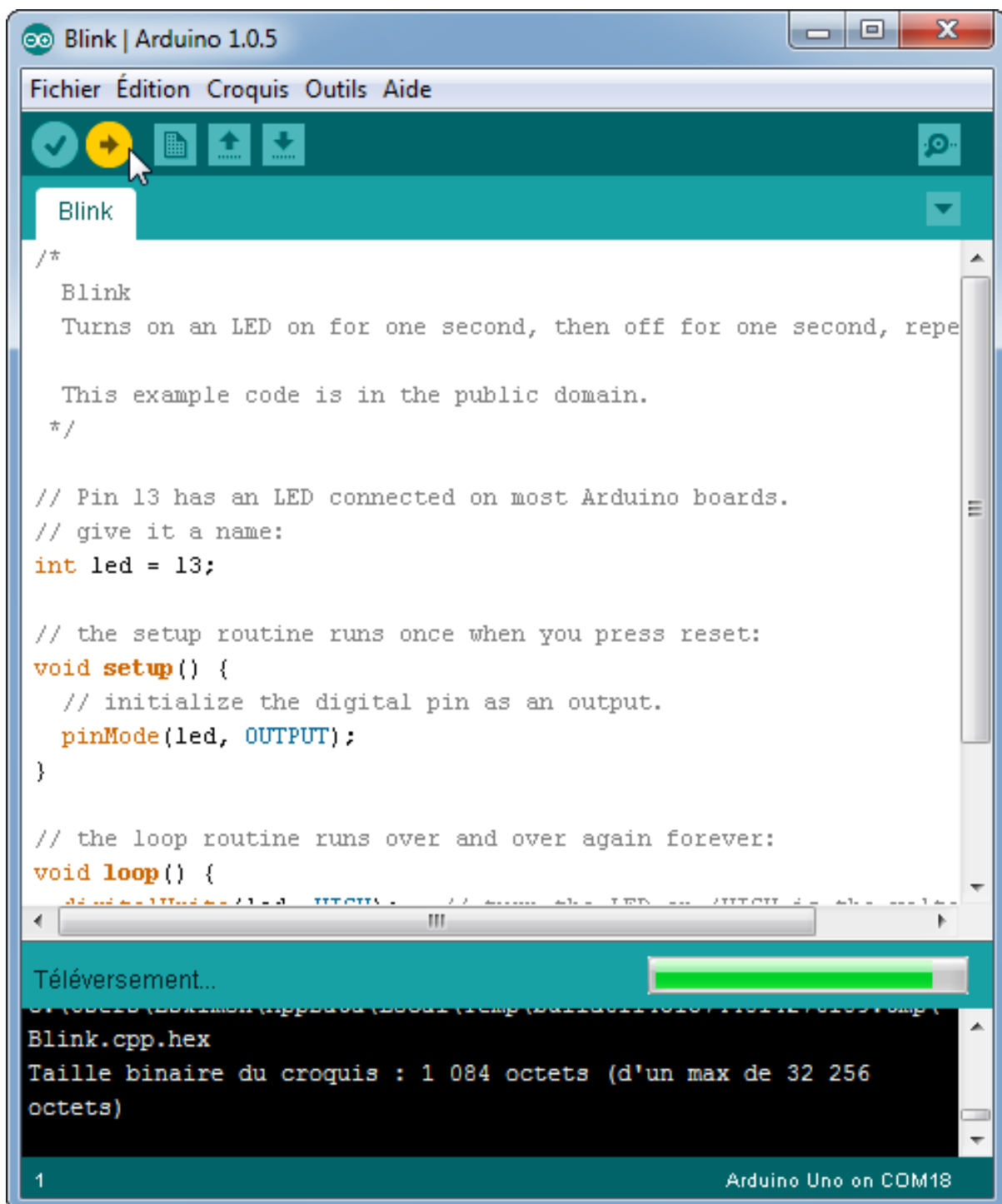


Figure 2.27 – Chargement du programme en cours...

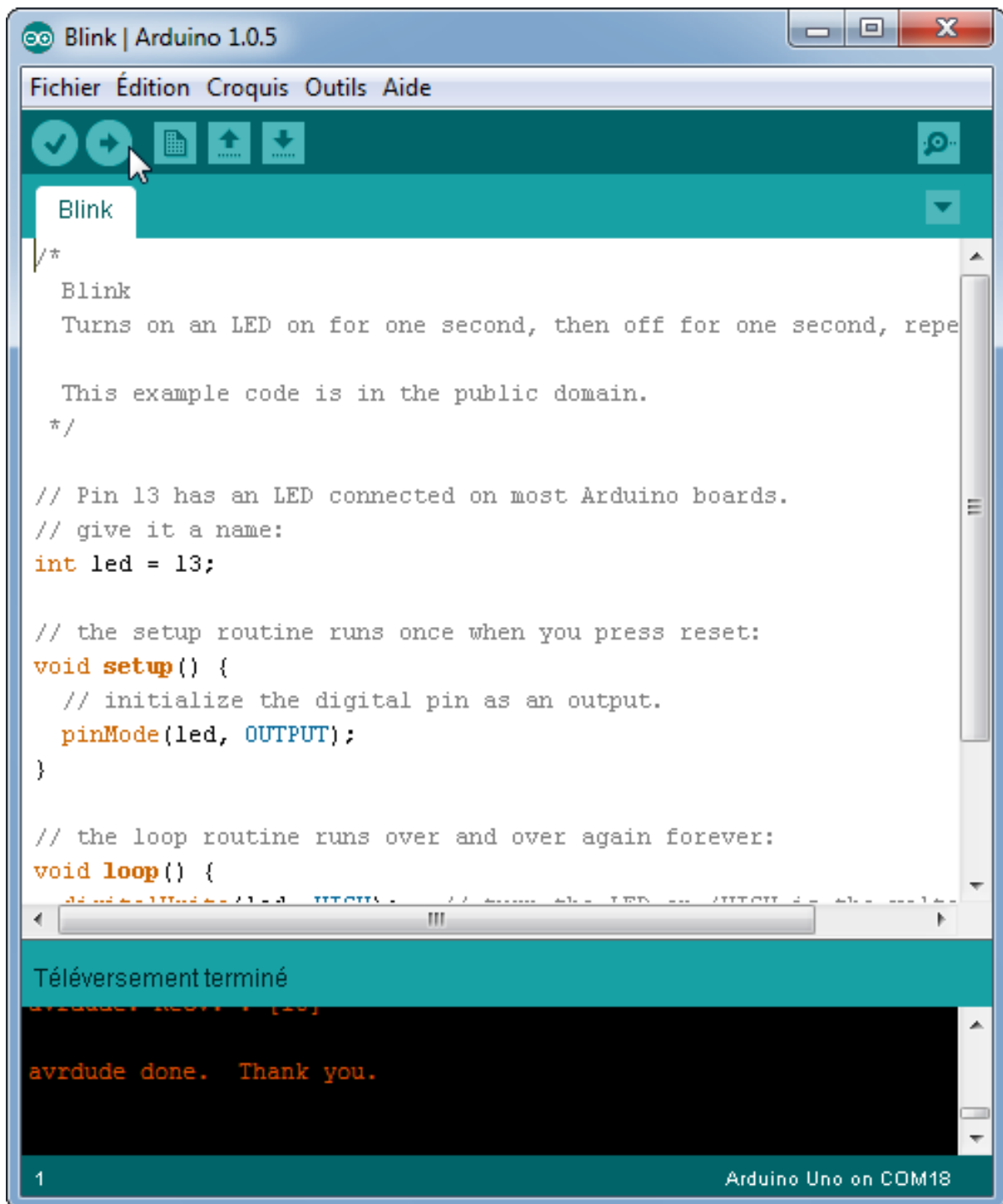


Figure 2.28 – Le chargement est terminé!

- L'IDE est codé en JAVA, il peut-être capricieux et bugger de temps en temps (surtout avec la voie série...) : réessayez l'envoi!



Figure 2.29 – LED sur la carte qui clignote

2.4.3 Fonctionnement global

Nous avons vu précédemment ce qu'était une carte électronique programmable. Nous avons également vu de quels éléments se basait une carte électronique pour fonctionner (schéma électronique, schéma de câblage). Je viens de vous présenter la carte, de quoi elle est principalement constituée. Enfin, je vous ai montré comment l'utiliser de manière à faire clignoter une petite lumière. Dorénavant, nous allons voir comment elle fonctionne de façon globale et répondre à quelques questions qui pourraient vous trotter dans la tête : "Comment la carte sait qu'il y a une LED de connectée ?", "Et comment sait-elle que c'est sur telle broche ?", "Et le programme, où est-ce qu'il se trouve et sous quelle forme ?", "Comment la carte fait pour comprendre ce qu'elle doit faire ?", ... De nombreuses questions, effectivement! :P #### Partons du programme

2.4.3.0.1 Le contenu Le contenu du programme, donc le programme en lui-même, est ce qui va définir chaque action que va exécuter la carte Arduino. Mais ce n'est pas tout! Dans le programme il y a plusieurs zones, que nous verrons plus en détail tout au long de la lecture de ce cours, qui ont chacune un rôle particulier. Voici leur présentation accompagnée d'un exemple.

- La première zone sert principalement (je ne vais pas m'étendre) à dire à la carte de **garder en mémoire quelques informations** qui peuvent être : l'emplacement d'un élément

connecté à la carte, par exemple une LED en broche 13, ou bien une valeur quelconque qui sera utile dans le programme :

```
// déclaration de variables globales (broches...)  
const int ledPin = 13;
```

- La zone secondaire est l'endroit où l'on va **initialiser certains paramètres** du programme. Par exemple, on pourra dire à la carte qu'elle devra communiquer avec l'ordinateur ou simplement lui dire ce qu'elle devra faire de la LED qui est connectée sur sa broche 13. On peut encore faire d'autres choses, mais nous le verrons plus tard.

```
void setup()  
{  
    // Declaration de la broche en sortie  
    pinMode(ledPin, OUTPUT);  
}
```

- La dernière zone est la **zone principale où se déroulera le programme**. Tout ce qui va être écrit dans cette zone sera exécuté par la carte, ce sont les actions que la carte fera. Par exemple, c'est ici qu'on pourra lui dire de faire clignoter la LED sur sa broche 13. On pourra également lui demander de faire une opération telle que 2+2 ou bien d'autres choses encore!

```
void loop() {  
    digitalWrite(13, HIGH); // led a l'etat haut  
    delay(1000);           // attendre 1 seconde  
    digitalWrite(13, LOW); // led a l'etat bas  
    delay(1000);           // attendre 1 seconde  
}
```

Code : Exemple de boucle

En conclusion, tout (vraiment tout !) ce que va faire la carte est inscrit dans le programme. Sans programme, la carte ne sert à **rien** ! C'est grâce au programme que la carte Arduino va savoir qu'une LED est connectée sur sa broche 13 et ce qu'elle va devoir faire avec, allumer et éteindre la LED alternativement pour la faire clignoter.

2.4.3.0.2 Et l'envoi Le programme est envoyé dans la carte lorsque vous cliquez sur le bouton



. Le logiciel Arduino va alors vérifier si le programme ne contient pas d'erreur et ensuite le compiler (le traduire) pour l'envoyer dans la carte :

L'envoi du programme est géré par votre ordinateur : le programme passe, sous forme de 0 et de 1, dans le câble USB qui relie votre ordinateur à votre carte et arrive dans la carte. Le reste se passe dans la carte elle-même...

2.4.3.1 Réception du programme

Le programme rentre donc dans la carte en passant en premier par le connecteur USB de celle-ci. Il va alors subir une petite transformation qui permet d'adapter le signal électrique correspondant au programme (oui car le programme transite dans le câble USB sous forme de signal électrique)

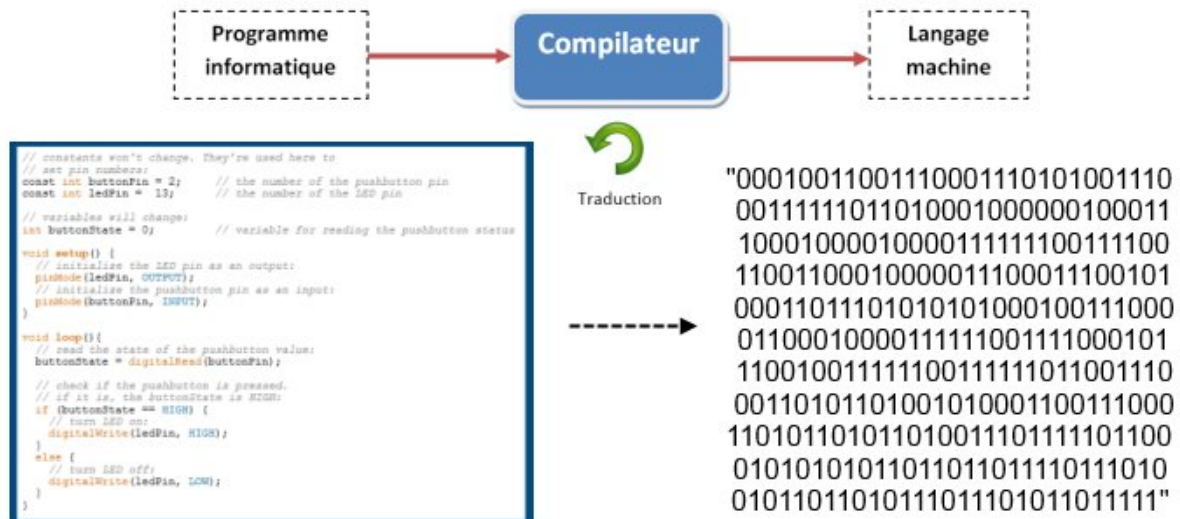


Figure 2.30 – Au départ, le programme est sous forme de texte, puis il est transformé

vers un signal plus approprié pour le microcontrôleur. On passe ainsi d'un signal codé pour la norme USB à un signal codé pour une simple voie série (que l'on étudiera plus tard d'ailleurs). Puis, ce "nouveau" signal est alors intercepté par le microcontrôleur.

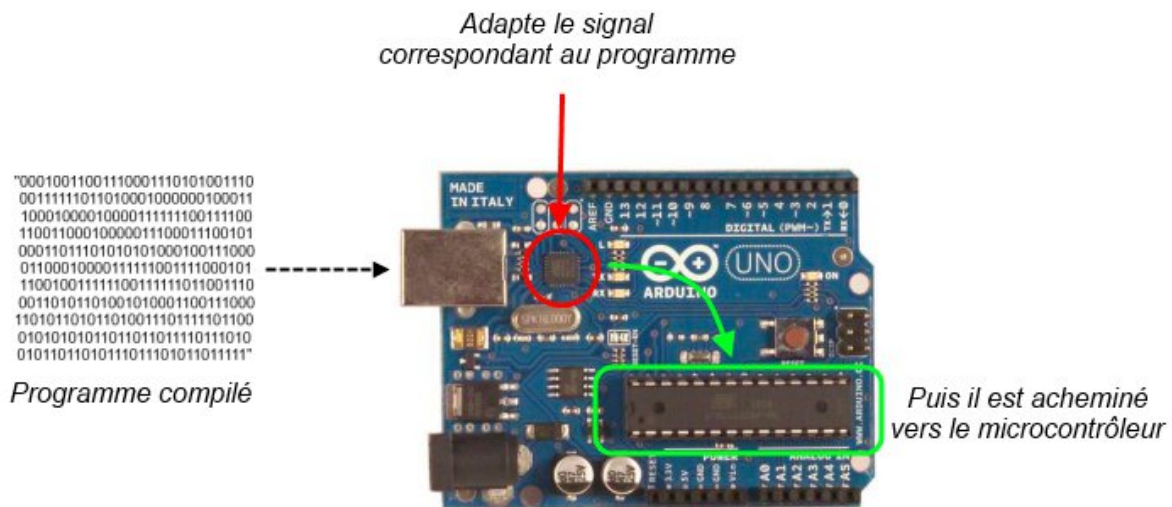


Figure 2.31 – Réception du programme

Tout le reste se passe alors...

2.4.3.2 À l'intérieur du microcontrôleur

2.4.3.2.1 L'emplacement du programme Le microcontrôleur reçoit le programme sous forme de signal électrique sur ses broches Tx et Rx, d'ailleurs disponible sur les broches de la carte (cf. image). Une fois qu'il est reçu, il est intégralement stocké dans une mémoire de type Flash que l'on appellera "la mémoire de programme". Ensuite, lorsque la carte démarre "normalement" (qu'aucun programme n'est en train d'être chargé), le cerveau va alors gérer les données

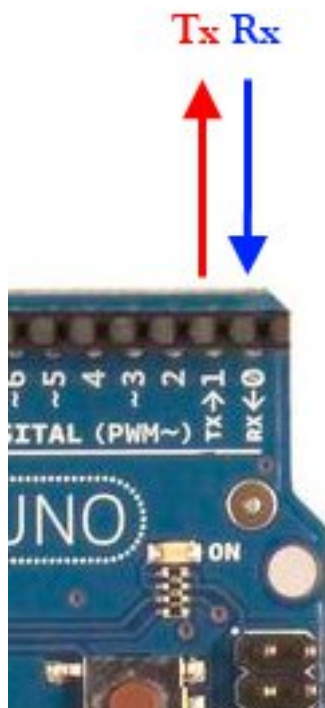


Figure 2.32 – À l'intérieur du microcontrôleur

et les répartir dans les différentes mémoires :

- La **mémoire programme** est celle qui va servir à savoir où l'on en est dans le programme, à quelle instruction on est rendu. C'est à dire, en quelque sorte, pointer sur des morceaux des zones 2 et 3 que l'on a vu dans le précédent exemple de programme.
- La **mémoire de données**, aussi appelé "RAM" (comme dans votre ordinateur) va stocker les variables telles que le numéro de la broche sur laquelle est connectée une LED, ou bien une simple valeur comme un chiffre, un nombre, des caractères, etc.

Voici un petit synoptique qui vous montre un peu l'intérieur du microcontrôleur (c'est très simplifié) :

2.4.3.2.2 Démarrage du microcontrôleur Lorsque le microcontrôleur démarre, il va commencer par lancer un bout de code particulier : le *bootloader*. C'est ce dernier qui va surveiller si un nouveau programme arrive sur la voie USB et s'il faut donc changer l'ancien en mémoire par le nouveau. Si rien n'arrive, il donne la main à votre programme, celui que vous avez créé. Ce dernier va alors défiler, instruction par instruction. Chaque fois qu'une nouvelle variable sera nécessaire, elle sera mise en RAM pour que l'on ait une mémoire de cette dernière (et supprimée lorsqu'elle n'est plus nécessaire). Sinon, les instructions vont se suivre une par une, dans l'ordre que vous les avez écrites.

Maintenant que vous connaissez et comprenez le fonctionnement global de la carte Arduino, nous allons pouvoir apprendre les bases du langage Arduino et à terme nous amuser à réaliser des programmes. Allez, hop, chapitre suivant ! :D

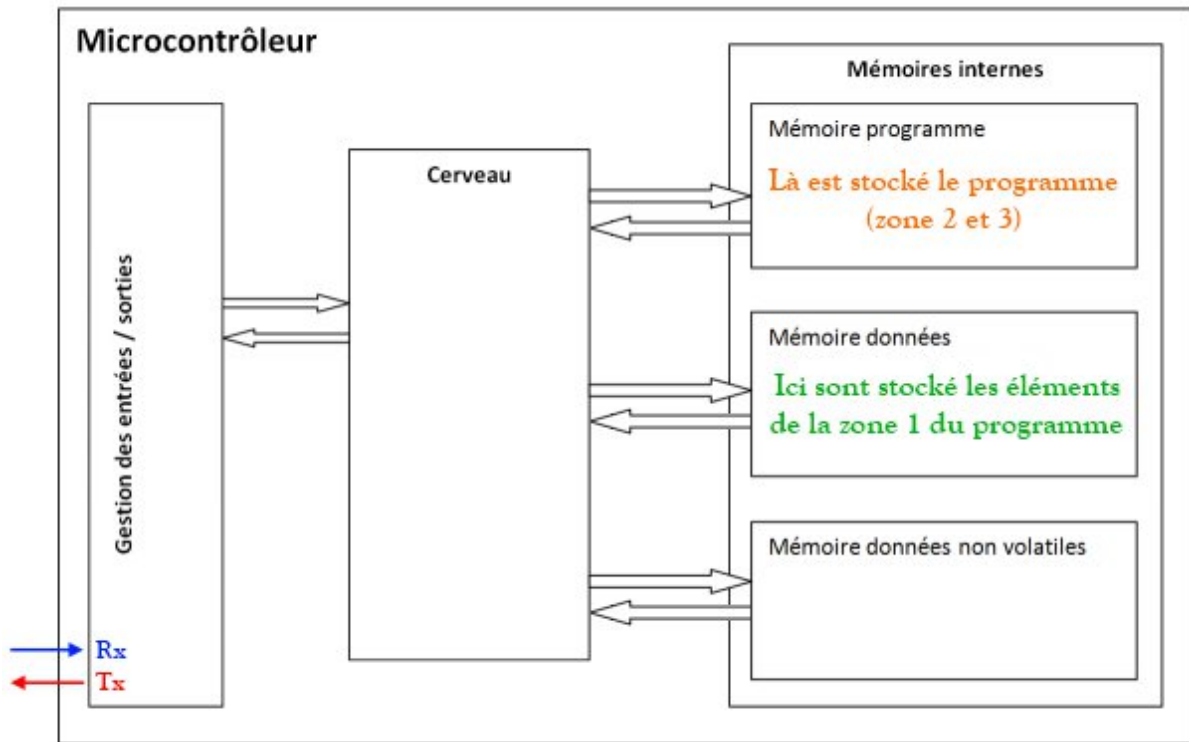


Figure 2.33 – Intérieur du microcontrôleur

2.5 Le langage Arduino (1/2)

A présent que vous avez une vision globale sur le fonctionnement de la carte Arduino, nous allons pouvoir apprendre à programmer avant de nous lancer dans la réalisation de programmes très simples pour débuter ! Pour pouvoir programmer notre carte, il nous faut trois choses :

- Un ordinateur
- Une carte Arduino
- Et connaître le langage Arduino

C'est ce dernier point qu'il nous faut acquérir. Le but même de ce chapitre est de vous apprendre à programmer avec le langage Arduino. Cependant, ce n'est qu'un support de cours que vous pourrez parcourir lorsque vous devrez programmer tout seul votre carte. En effet, c'est en manipulant que l'on apprend, ce qui implique que votre apprentissage en programmation sera plus conséquent dans les prochains chapitres que dans ce cours même.

[[i]] | Le langage Arduino est très proche du C et du C++. | Pour ceux dont la connaissance de ces langages est fondée, ne vous sentez pas obligé de lire les deux chapitres sur le langage Arduino. Bien qu'il y ait des points quelques peu importants.

2.5.1 La syntaxe du langage

La syntaxe d'un langage de programmation est l'ensemble des *règles d'écriture* liées à ce langage. On va donc voir dans ce sous-chapitre les règles qui régissent l'écriture du langage Arduino.

2.5.1.1 Le code minimal

Avec Arduino, nous devons utiliser un *code minimal* lorsque l'on crée un programme. Ce code permet de diviser le programme que nous allons créer en deux grosses parties.

```
// fonction d'initialisation de la carte
void setup()
{
    // contenu de l'initialisation
}

// fonction principale, elle se répète (s'exécute) à l'infini
void loop()
{
    // contenu de votre programme
}
```

Code : Le code minimal d'un programme Arduino

Vous avez donc devant vous le code minimal qu'il faut insérer dans votre programme. Mais que peut-il bien signifier pour quelqu'un qui n'a jamais programmé ?

2.5.1.1.1 La fonction setup Dans ce code se trouvent deux fonctions. Les fonctions sont en fait *des portions de code*.

```
// fonction d'initialisation de la carte
void setup()
{
    // contenu de l'initialisation
    // on écrit le code à l'intérieur
}
```

Code : Zoom sur la fonction setup

Cette fonction **setup()** est appelée *une seule fois* lorsque le programme commence. C'est pourquoi c'est dans cette fonction que l'on va écrire le code qui n'a besoin d'être exécuté une seule fois. On appelle cette fonction : **fonction d'initialisation**. On y retrouvera la mise en place des différentes sorties et quelques autres réglages. C'est un peu le check-up de démarrage. Imaginez un pilote d'avion dans sa cabine qui fait l'inventaire :P : - *patte 2 en sortie, état haut ?* - OK - *timer 3 à 15 millisecondes ?* - OK ...

Une fois que l'on a initialisé le programme il faut ensuite créer son "cœur", autrement dit le programme en lui même.

```
// fonction principale, elle se répète (s'exécute) à l'infini
void loop()
{
    // contenu de votre programme
}
```

Code : Zoom sur la fonction principale

C'est donc dans cette fonction `loop()` où l'on va écrire le contenu du programme. Il faut savoir que cette fonction est appelée en permanence, c'est-à-dire qu'elle est exécutée une fois, puis lorsque son exécution est terminée, on la ré-exécute et encore et encore. On parle de **boucle infinie**.

[[i]] | A titre informatif, on n'est pas obligé d'écrire quelque chose dans ces deux fonctions. En revanche, il est **obligatoire** de les écrire, même si elles ne contiennent aucun code !

2.5.1.1.2 Les instructions [[q]] | Dans ces fonctions, on écrit quoi ?

C'est justement l'objet de ce paragraphe. Dans votre liste pour le dîner de ce soir, vous écrivez les tâches importantes qui vous attendent. Ce sont des **instructions**. Les instructions sont des lignes de code qui disent au programme : "fait ceci, fait cela, ..." C'est tout bête mais très puissant car c'est ce qui va orchestrer notre programme.

2.5.1.1.3 Les points virgules Les points virgules terminent les instructions. Si par exemple je dis dans mon programme : "appelle la fonction `*couperDuSaucisson*` *je dois mettre un point virgule après l'appel de cette fonction*.

[[e]] | Les points virgules (;) sont synonymes d'erreurs car il arrive très souvent de les oublier à la fin des instructions. |Par conséquent le code ne marche pas et la recherche de l'erreur peut nous prendre un temps conséquent ! Donc faites bien attention.

2.5.1.1.4 Les accolades Les accolades sont les "conteneurs" du code du programme. Elles sont propres aux fonctions, aux conditions et aux boucles. Les instructions du programme sont écrites à l'intérieur de ces accolades.

Parfois elles ne sont pas obligatoires dans les *conditions* (nous allons voir plus bas ce que c'est), mais je recommande de les **mettre tout le temps** ! Cela rendra plus lisible votre programme.

2.5.1.1.5 Les commentaires Pour finir, on va voir ce qu'est un commentaire. J'en ai déjà mis dans les exemples de codes. Ce sont des lignes de codes qui seront ignorées par le programme. Elles ne servent en rien lors de l'exécution du programme.

[[q]] | Mais alors c'est inutile ? o_O

Non car cela va nous permettre à nous et aux programmeurs qui liront votre code (s'il y en a) de savoir ce que signifie la ligne de code que vous avez écrite. C'est très important de mettre des commentaires et cela permet aussi de reprendre un programme laissé dans l'oubli plus facilement ! Si par exemple vous connaissez mal une instruction que vous avez écrite dans votre programme, vous mettez une ligne de commentaire pour vous rappeler la prochaine fois que vous lirez votre programme ce que la ligne signifie.

```
// cette ligne est un commentaire sur UNE SEULE ligne
```

Code : Ligne unique de commentaire

```
/*cette ligne est un commentaire, sur PLUSIEURS lignes
qui sera ignoré par le programme, mais pas par celui qui lit le code */
```

Code : Commentaire sur plusieurs lignes

2.5.1.1.6 Les accents `[[a]]` | Il est formellement interdit de mettre des accents en programmation. Sauf dans les commentaires.

2.5.2 Les variables

Nous l'avons vu, dans un microcontrôleur, il y a plusieurs types de mémoire. Nous nous occuperons seulement de la mémoire "vive" (RAM) et de la mémoire "morte" (EEPROM). Je vais vous poser une énigme. Imaginons que vous avez connecté un bouton poussoir sur une broche de votre carte Arduino. Comment allez-vous stocker l'état du bouton (appuyé ou éteint) ?

2.5.2.1 Une variable, qu'est ce que c'est ?

Une **variable est un nombre**. Ce nombre est stocké dans un espace de la mémoire vive (RAM) du microcontrôleur. La manière qui permet de les stocker est semblable à celle utilisée pour ranger des chaussures : dans un casier numéroté.

->

Cha	ussur	es ra	ngées	dans	des	cases	numé	rotée	s
1	2	3	4	5	6	7	8	9	10
11	12	13	14	15	16	17	18	19	20
21	22	23	24	25	26	27	28	29	30
31	32	33	34	35	36	37	38	39	40
41	42	43	44	45	46	47	48	49	50
51	52	53	54	55	56	57	58	59	60

Table 2.4 – Un tableau est un ensemble de case <-

`[[q]]` | Une variable est un nombre, c'est tout ? o_o

Ce nombre a la particularité de changer de valeur. Étrange n'est-ce pas ? Et bien pas tant que ça, car une variable est en fait le **conteneur** du nombre en question. Et ce conteneur va être stocké dans une case de la mémoire. Si on matérialise cette explication par un schéma, cela donnerait :

-> **nombre => variable => mémoire** <-

— le symbole "=>" signifiant : "est contenu dans..."

2.5.2.1.1 Le nom d'une variable Le nom de variable accepte quasiment tous les caractères sauf :

- . (le point)
- , (la virgule)
- é,à,ç,è (les accents)

Bon je vais pas tous les donner, il n'accepte que l'alphabet alphanumérique (`[a-z]`, `[A-Z]`, `[0-9]`) et `_` (underscore). Il ne doit **pas** commencer par un chiffre.

2.5.2.2 Définir une variable

Si on donne un nombre à notre programme, il ne sait pas si c'est une variable ou pas. Il faut le lui indiquer. Pour cela, on donne un **type** aux variables. Oui, car il existe plusieurs types de variables ! Par exemple la variable "x" vaut 4 :

```
x = 4 ;
```

Code : Assignation à une variable

Et bien ce code ne fonctionnerait pas car il ne suffit pas ! En effet, il existe une multitude de nombres : les nombres entiers, les nombres décimaux, ... C'est pour cela qu'il faut assigner une variable à un type. Voilà les types de variables les plus répandus :

->

Type	Quel nombre il stocke ?	Valeurs maximales du nombre stocké	Nombre sur X bits	Nombre d'octets
int	entier	-32 768 à +32 767	16 bits	2 octets
long	entier	-2 147 483 648 à +2 147 483 647	32 bits	4 octets
char	entier	-128 à +127	8 bits	1 octets
float	décimale	-3.4×10^{38} à $+3.4 \times 10^{38}$	32 bits	4 octets
double	décimale	-3.4×10^{38} à $+3.4 \times 10^{38}$	32 bits	4 octets

Table 2.5 – Les types de variables

<-

Par exemple, si notre variable "x" ne prend que des valeurs entières, on utilisera les types **int**, **long**, ou **char**. Si maintenant la variable "x" ne dépasse pas la valeur 64 ou 87, alors on utilisera le type **char**.

```
char x = 0 ;
```

Code : Assignation avec un type

[[i]] | Si en revanche x = 260, alors on utilisera le type supérieur (qui accepte une plus grande quantité de nombre) à **char**, autrement dit **int** ou **long**.

[[q]] | Mais t'es pas malin, pour éviter les dépassements de valeur ont met tout dans des double ou long !

Oui, mais NON. Un microcontrôleur, ce n'est pas un ordinateur 2GHz multicore, 4Go de RAM ! Ici on parle d'un système qui fonctionne avec un CPU à 16MHz (soit 0,016 GHz) et 2 Ko de SRAM pour la mémoire vive. Donc deux raisons font qu'il faut choisir ses variables de manière judicieuse :

- La RAM n'est pas extensible, quand y en a plus, il y en a plus !
- Le processeur est de type 8 bits (sur Arduino UNO), donc il est optimisé pour faire des traitements sur des variables de taille 8 bits, un traitement sur une variable 32 bits prendra donc (beaucoup) plus de temps !

Si à présent notre variable "x" ne prend jamais une valeur négative (-20, -78, ...), alors on utilisera un type **non-signé**. C'est à dire, dans notre cas, un **char** dont la valeur n'est plus de -128 à

+127, mais de 0 à 255. Voici le tableau des types non signés, on repère ces types par le mot **unsigned** (de l'anglais : non-signé) qui les précède :

->

Type	Quel nombre il stocke ?	Valeurs maximales du nombre stocké	Nombre sur X bits	Nombre
unsigned char	entier non négatif	0 à 255	8 bits	1 octet
unsigned int	entier non négatif	0 à 65 535	16 bits	2 octets
unsigned long	entier non négatif	0 à 4 294 967 295	32 bits	4 octets

Table 2.6 – Les types non signés

<-

Une des particularités du langage Arduino est qu'il accepte un nombre plus important de types de variables. Je vous les liste dans ce tableau :

->

Type	Quel nombre il stocke ?	Valeurs maximales du nombre stocké	Nombre sur X bits	Nombre d'octets
byte	entier non négatif	0 à 255	8 bits	1 octet
word	entier non négatif	0 à 65535	16 bits	2 octets
boolean	entier non négatif	0 à 1	1 bit	1 octet

Table 2.7 – Les types propres à Arduino

<-

[[i]] | Pour votre information, vous pouvez retrouver ces tableaux sur [cette page](#).

2.5.2.3 Les variables booléennes

Les variables **booléennes** sont des variables qui ne peuvent prendre *que deux valeurs* : ou VRAI ou FAUX. Elles sont utilisées notamment dans les boucles et les conditions. Nous verrons pourquoi. Une variable booléenne peut être définie de plusieurs manières :

```
// variable est fausse car elle vaut FALSE, du terme anglais "faux"
boolean variable = FALSE;
// variable est vraie car elle vaut TRUE, du terme anglais "vrai"
boolean variable = TRUE;
```

Code : Les variables booléennes

Quand une variable vaut "0", on peut considérer cette variable comme une variable booléenne, elle est donc fausse. En revanche, lorsqu'elle vaut "1" ou n'importe quelle valeurs différente de zéro, on peut aussi la considérer comme une variable booléenne, elle est donc vraie. Voilà un exemple :

```
// variable est fausse car elle vaut 0
int variable = 0;
// variable est vraie car elle vaut 1
int variable = 1;
// variable est vraie car sa valeur est différente de 0
int variable = 42;
```

Code : Un booléen codé avec des entiers

Le langage Arduino accepte aussi une troisième forme d'écriture (qui lui sert pour utiliser les broches de sorties du microcontrôleur) :

```
// variable est à l'état logique bas (= traduction de "low"), donc 0
int variable = LOW;
// variable est à l'état logique haut (= traduction de "high"), donc 1
int variable = HIGH;
```

Code : l'utilisation des constantes LOW et HIGH

Nous nous servons de cette troisième écriture pour allumer et éteindre des lumières...

2.5.2.4 Les opérations "simples"

On va voir à présent les opérations qui sont possibles avec le langage Arduino (addition, multiplication, ...).

Je vous vois tout de suite dire : "Mais pourquoi on fait ça, on l'a fait en primaire! :mad :)" Et bien parce que c'est quelque chose d'essentiel, car on pourra ensuite faire des opérations avec des variables. Vous verrez, vous changerez d'avis après avoir lu la suite! ;)

2.5.2.4.1 L'addition Vous savez ce que c'est, pas besoin d'explications. Voyons comment on fait cette opération avec le langage Arduino. Prenons la même variable que tout à l'heure :

```
// définition de la variable x
int x = 0;

// on change la valeur de x par une opération simple
x = 12 + 3;
// x vaut maintenant 12 + 3 = 15
```

Code : L'addition

Faisons maintenant une addition de variables :

```
// définition de la variable x et assignation à la valeur 38
int x = 38;
int y = 10;
int z = 0;
// faisons une addition
// on a donc z = 38 + 10 = 48
z = x + y;
```

Code : Addition de deux variables

2.5.2.4.2 La soustraction On peut reprendre les exemples précédents, en faisant une soustraction :

```
/définition de la variable x
int x = 0;

// on change la valeur de x par une opération simple
x = 12 - 3;
// x vaut maintenant 12 - 3 = 9
```

Code : La soustraction

```
int x = 38; // définition de la variable x et assignation à la valeur 38
int y = 10;
int z = 0;

z = x - y; // on a donc z = 38 - 10 = 28
```

Code : Soustraction de deux variables

2.5.2.4.3 La multiplication

```
int x = 0;
int y = 10;
int z = 0;

x = 12 * 3; // x vaut maintenant 12 * 3 = 36

z = x * y; // on a donc z = 36 * 10 = 360

// on peut aussi multiplier (ou une autre opération) un nombre et une variable :
z = z * ( 1 / 10 ); // soit z = 360 * 0.1 = 36
```

Code : la multiplication

2.5.2.4.4 La division

```
float x = 0;
float y = 15;
float z = 0;

x = 12 / 2; // x vaut maintenant 12 / 2 = 6

z = y / x; // on a donc z = 15 / 6 = 2.5
```

Code : La division

Attention cependant, si vous essayer de stocker le résultat d'une division dans une variable de type char, int ou long, le résultat sera stocké sous la forme d'un entier arrondi au nombre inférieur. Par exemple dans le code précédent si on met z dans un int on aura :

```
float x = 0;
float y = 15;
int z = 0;

x = 12 / 2; // x vaut maintenant 12 / 2 = 6

z = y / x; // on a donc z = 15 / 6 = 2!
```

Code : Arrondi au nombre inférieur

2.5.2.4.5 Le modulo Après cette brève explication sur les opérations de base, passons à quelque chose de plus sérieux. Le modulo est une opération de base, certes moins connue que les autres. Cette opération permet d'obtenir le reste d'une division.

```
18 % 6 // le reste de l'opération est 0, car il y a 3*6 dans 18 donc 18 - 18 = 0
18 % 5 // le reste de l'opération est 3, car il y a 3*5 dans 18 donc 18 - 15 = 3
```

Code : Le modulo

Le modulo est utilisé grâce au symbole %. C'est tout ce qu'il faut retenir. Autre exemple :

```
int x = 24;
int y = 6;
int z = 0;

z = x % y; // on a donc z = 24 % 6 = 0 (car 6 * 4 = 24)
```

Code : le modulo entre deux variables

[[i]] | Le modulo ne peut-être fait que sur des nombres entiers

2.5.2.5 Quelques opérations bien pratiques

Voyons un peu d'autres opérations qui facilitent parfois l'écriture du code.

2.5.2.5.1 L'incrément Derrière ce nom barbare se cache une simple opération d'addition.

```
var = 0;
var++; // c'est cette ligne de code qui nous intéresse
```

Code : Incrémenter

“var++ ;” revient à écrire : “var = var + 1 ;” En fait, on ajoute le chiffre 1 à la valeur de var. Et si on répète le code un certain nombre de fois, par exemple 30, et bien on aura var = 30.

2.5.2.5.2 La décrémentation C'est l'inverse de l'incrément. Autrement dit, on enlève le chiffre 1 à la valeur de var.

```
var = 30;
var--; // décrémentation de var
```

Code : La décrémentation

2.5.2.5.3 Les opérations composées Parfois il devient assez lassant de réécrire les mêmes chose et l'on sait que les programmeurs sont des gros fainéants ! :P Il existe des raccourcis lorsque l'on veut effectuer une opération sur une même variable :

```
int x, y;

x += y; // correspond à x = x + y;
x -= y; // correspond à x = x - y;
x *= y; // correspond à x = x * y;
x /= y; // correspond à x = x / y;
```

Code : des opérations composées

Avec un exemple, cela donnerait :

```
int var = 10;

// opération 1
var = var + 6;
var += 6; // var = 16

// opération 2
var = var - 6;
var -= 6; // var = 4

// opération 3
var = var * 6;
var *= 6; // var = 60

// opération 4
var = var / 5;
var /= 5; // var = 2
```

Code : Opérations composées : détails

2.5.2.6 L'opération de bascule (ou "inversion d'état")

Un jour, pour le projet du BAC, je devais (ou plutôt "je voulais") améliorer un code qui servait à programmer un module d'une centrale de gestion domestique. Mon but était d'afficher un choix à l'utilisateur sur un écran. Pour ce faire, il fallait que je réalise une **bascule programmée** (c'est comme ça que je la nomme maintenant). Et après maintes recherches et tests, j'ai réussi à trouver ! Et il s'avère que cette "opération", si l'on peut l'appeler ainsi, est très utile dans certains cas. Nous l'utiliserons notamment lorsque l'on voudra faire clignoter une lumière. Sans plus attendre, voilà cette astuce :

```
// on définit une variable x qui ne peut prendre que la valeur 0 ou 1
// (soit vraie ou fausse)
boolean x = 0;

x = 1 - x; // c'est la toute l'astuce du programme !
```

Code : La bascule

Analysons cette instruction. A chaque exécution du programme (oui, j'ai omis de vous le dire, il se répète jusqu'à l'infini), la variable x va changer de valeur :

- 1^{er} temps : $x = 1 - x$ soit $x = 1 - 0$ donc $x = 1$
- 2^e temps : $x = 1 - x$ or x vaut maintenant 1 donc $x = 1 - 1$ soit $x = 0$
- 3^e temps : x vaut 0 donc $x = 1 - 0$ soit $x = 1$

Ce code se répète donc et à chaque répétition, la variable x change de valeur et passe de 0 à 1, de 1 à 0, de 0 à 1, etc. Il agit bien comme une bascule qui change la valeur d'une variable booléenne. En mode console cela donnerait quelque chose du genre (n'essayez pas cela ne marchera pas, c'est un exemple) :

```
x = 0
x = 1
x = 0
x = 1
x = 0
...
```

Code : Le résultat de la bascule

Mais il existe d'autres moyens d'arriver au même résultat. Par exemple, en utilisant l'opérateur '!' qui signifie "not" ("non"). Ainsi, avec le code suivant on aura le même fonctionnement :

```
x = !x;
```

Code : La bascule avec l'opérateur !

Puisqu'à chaque passage x devient "pas x " donc si x vaut 1 son contraire sera 0 et s'il vaut 0, il deviendra 1.

2.5.3 Les conditions

2.5.3.1 Qu'est-ce qu'une condition ?

C'est un choix que l'on fait entre plusieurs propositions. En informatique, les conditions servent à tester des variables. Par exemple : *Vous faites une recherche sur un site spécialisé pour acheter une nouvelle voiture. Vous imposez le prix de la voiture qui doit être inférieur à 5000€ (c'est un petit budget ^^). Le programme qui va gérer ça va faire appel à un **test conditionnel**. Il va éliminer tous les résultats de la recherche dont le prix est supérieur à 5000€.*

2.5.3.2 Quelques symboles

Pour tester des variables, il faut connaître quelques symboles. Je vous ai fait un joli tableau pour que vous vous repérez bien :

->

Symbole	A quoi il sert	Signification
==	Ce symbole, composé de deux égales, permet de tester l'égalité entre deux variables	... est égale à .

Symbole	A quoi il sert	Signification
<	Celui-ci teste l'infériorité d'une variable par rapport à une autre	...est inférieur
>	Là c'est la supériorité d'une variable par rapport à une autre	...est supérieur
<=	teste l'infériorité ou l'égalité d'une variable par rapport à une autre	...est inférieur
>=	teste la supériorité ou l'égalité d'une variable par rapport à une autre	...est supérieur
!=	teste la différence entre deux variables	...est différent

Table 2.8 – Les symboles conditionnels

<-

“Et si on s'occupait des conditions ? Ou bien sinon on va tranquillement aller boire un bon café ?”

Cette phrase implique un choix : le premier choix est de s'occuper des conditions. Si l'interlocuteur dit oui, alors il s'occupe des conditions. Mais s'il dit non, alors il va boire un bon café. Il a donc l'obligation d'effectuer une action sur les deux proposées. En informatique, on parle de **condition**. “si la condition est vraie”, on fait une action. En revanche “si la condition est fausse”, on exécute une autre action.

2.5.3.3 If...else

La première condition que nous verrons est la condition if...else. Voyons un peu le fonctionnement.

2.5.3.3.1 if On veut tester la valeur d'une variable. Prenons le même exemple que tout à l'heure. Je veux tester si la voiture est inférieure à 5000€.

```
int prix_voiture = 4800; // variable : prix de la voiture définit à 4800€
```

D'abord on définit la variable “prix_voiture”. Sa valeur est de 4800€. Ensuite, on doit tester cette valeur. Pour tester une condition, on emploie le terme *if* (de l'anglais “si”). Ce terme doit être suivi de parenthèses dans lesquelles se trouveront les variables à tester. Donc entre ces parenthèses, nous devons tester la variable prix_voiture afin de savoir si elle est inférieure à 5000€.

```
if(prix_voiture < 5000)
{
    // la condition est vraie, donc j'achète la voiture
}
```

Code : Le test d'une condition

On peut lire cette ligne de code comme ceci : “**si** la variable *prix_voiture* est inférieure à 5000, on exécute le code qui se trouve entre les accolades.

[[a]] | Les instructions qui sont *entre* les accolades ne seront exécutées que si la condition testée est *vraie* !

Le “schéma” à suivre pour tester une condition est donc le suivant :


```
if(/* contenu de la condition à tester */)
{
    // instructions à exécuter si la condition est vraie
}
```

Code : Syntaxe d'une condition

2.5.3.3.2 else On a pour l'instant testé que si la condition est vraie. Maintenant, nous allons voir comment faire pour que d'autres instructions soient exécutées si la condition est fausse. Le terme *else* de l'anglais "sinon" implique notre deuxième choix si la condition est fausse. *Par exemple, si le prix de la voiture est inférieur à 5000€, alors je l'achète. Sinon, je ne l'achète pas.* Pour traduire cette phrase en ligne de code, c'est plus simple qu'avec un *if*, il n'y a pas de parenthèses à remplir :

```
int prix_voiture = 5500;

if(prix_voiture < 5000)
{
    // la condition est vraie, donc j'achète la voiture
}
else
{
    // la condition est fausse, donc je n'achète pas la voiture
}
```

Code : Si "", alors "", sinon ""

[[i]] | Le *else* est généralement utilisé pour les conditions dites **de défaut**. C'est lui qui a le pouvoir sur toutes les conditions, c'est-à-dire que si aucune condition n'est vraie, on exécute les instructions qu'il contient.

[[i]] | Le *else* n'est pas obligatoire, on peut très bien mettre plusieurs *if* à la suite.

Le "schéma" de principe à retenir est le suivant :

```
else // si toutes les conditions précédentes sont fausses...
{
    // ...on exécute les instructions entre ces accolades
}
```

Code : Syntaxe du *else*

2.5.3.3.3 else if [[q]] | A ce que je vois, on a pas trop le choix : soit la condition est vraie, soit elle est fausse. Il n'y a pas d'autres possibilités ? o_O

Bien sûr que l'on peut tester d'autres conditions ! Pour cela, on emploie le terme *else if* qui signifie "sinon si..." *Par exemple, SI le prix de la voiture est inférieur à 5000€ je l'achète ; SINON SI elle est égale à 5500€ mais qu'elle a l'option GPS en plus, alors je l'achète ; SINON je ne l'achète pas.* Le *sinon si* s'emploie comme le *if* :

```
int prix_voiture = 5500;

if(prix_voiture < 5000)
```

```
{
    // la condition est vraie, donc j'achète la voiture
}
else if(prix_voiture == 5500)
{
    // la condition est vraie, donc j'achète la voiture
}
else
{
    // la condition est fausse, donc je n'achète pas la voiture
}
```

Code : Utilisation de `else if`

A retenir donc, si la première condition est fausse, on teste la deuxième, si la deuxième est fausse, on teste la troisième, etc. “Schéma” de principe du *else*, idem au *if* :

```
else if(/* test de la condition */) // si elle est vraie...
{
    // ...on exécute les instructions entre ces accolades
}
```

Code : Syntaxe du `else if`

[[a]] | Le “else if” ne peut pas être utilisée toute seule, il faut obligatoirement qu’il y ait un “if” avant !

2.5.3.4 Les opérateurs logiques

Et si je vous posais un autre problème ? Comment faire pour savoir si la voiture est inférieure à 5000€ ET si elle est grise ? :twisted :

[[q]] | C’est vrai ça, si je veux que la voiture soit grise en plus d’être inférieure à 5000€, comment je fais ?

Il existe des opérateurs qui vont nous permettre de tester cette condition ! Voyons quels sont ses opérateurs puis testons-les !

->

Opérateur	Signification
&&	... ET ...
	... OU ...
!	NON

Table 2.9 – les opérateurs logiques

<-

2.5.3.4.1 ET Reprenons ce que nous avons testé dans le *else if* : *Si la voiture vaut 5500€ ET qu'elle a l'option GPS en plus, ALORS je l'achète*. On va utiliser un *if* et un opérateur logique qui sera le *ET* :

```
int prix_voiture = 5500;
int option_GPS = TRUE;

/* l'opérateur && lie les deux conditions qui doivent être
vraies ensemble pour que la condition soit remplie */
if(prix_voiture == 5500 && option_GPS)
{
    // j'achète la voiture si la condition précédente est vraie
}

```

Code : Conjonction de deux conditions

2.5.3.4.2 OU On peut reprendre la condition précédente et la première en les assemblant pour rendre le code beaucoup moins long.

[[i]] | Et oui, les programmeurs sont des flemmards! :P

Rappelons quelles sont ces conditions :

```
int prix_voiture = 5500;
int option_GPS = TRUE;

if(prix_voiture < 5000)
{
    // la condition est vraie, donc j'achète la voiture
}
else if(prix_voiture == 5500 && option_GPS)
{
    // la condition est vraie, donc j'achète la voiture
}
else
{
    // la condition est fausse, donc je n'achète pas la voiture
}

```

Vous voyez bien que l'instruction dans le *if* et le *else if* est la même. Avec un opérateur logique, qui est le *OU*, on peut rassembler ces conditions :

```
int prix_voiture = 5500;
int option_GPS = TRUE;

if((prix_voiture < 5000) || (prix_voiture == 5500 && option_GPS))
{
    // la condition est vraie, donc j'achète la voiture
}
else
{
    // la condition est fausse, donc je n'achète pas la voiture
}

```

Code : Utilisation du OU logique

Lisons la condition testée dans le if : “SI le prix de la voiture est inférieur à 5000€ OU SI le prix de la voiture est égal à 5500€ ET la voiture à l’option GPS en plus, ALORS j’achète la voiture”.

[[e]] | Attention aux parenthèses qui sont à bien placer dans les conditions, ici elles n’étaient pas nécessaires, mais elles aident à mieux lire le code. ;)

2.5.3.4.3 NON [[q]] | Moi j’aimerais tester “si la condition est fausse j’achète la voiture”. Comment faire ?

~~Toi-t-as-un-souci~~ Il existe un dernier opérateur logique qui se prénomme NON. Il permet en effet de tester si la condition est fausse :

```
int prix_voiture = 5500;

if(!(prix_voiture < 5000))
{
    // la condition est vraie, donc j'achète la voiture
}
```

Code : L’opérateur *négation*

Se lit : “SI le prix de la voiture N’EST PAS inférieur à 5000€, alors j’achète la voiture”. On s’en sert avec le caractère ! (point d’exclamation), généralement pour tester des variables booléennes. On verra dans les boucles que ça peut grandement simplifier le code.

2.5.3.5 Switch

Il existe un dernier test conditionnel que nous n’avons pas encore abordé, c’est le *switch*. Voilà un exemple :

```
int options_voiture = 0;

if(options_voiture == 0)
{
    // il n'y a pas d'options dans la voiture
}
else if(options_voiture == 1)
{
    // la voiture a l'option GPS
}
else if(options_voiture == 2)
{
    // la voiture a l'option climatisation
}
else if(options_voiture == 3)
{
    // la voiture a l'option vitre automatique
}
else if(options_voiture == 4)
```

```

{
    // la voiture a l'option barres de toit
}
else if(options_voiture == 5)
{
    // la voiture a l'option  siège éjectable
}
else
{
    // retente ta chance;-)
}

```

Code : Un grand nombre de `else if`

Ce code est indigeste ! C'est infâme ! Grottesque ! Pas beau ! En clair, il faut trouver une solution pour changer cela. Cette solution existe, c'est le *switch*. Le *switch*, comme son nom l'indique, va tester la variable jusqu'à la fin des valeurs qu'on lui aura données. Voici comment cela se présente :

```

int options_voiture = 0;

switch (options_voiture)
{
    case 0 :
        // il n'y a pas d'options dans la voiture
        break;
    case 1 :
        // la voiture a l'option GPS
        break;
    case 2 :
        // la voiture a l'option climatisation
        break;
    case 3 :
        // la voiture a l'option vitre automatique
        break;
    case 4 :
        // la voiture a l'option barres de toit
        break;
    case 5 :
        // la voiture a l'option siège éjectable
        break;
    default :
        // retente ta chance;-)
        break;
}

```

Code : Utilisation de `switch`

Si on testait ce code, en réalité cela ne fonctionnerait pas car il n'y a pas d'instruction pour afficher à l'écran, mais nous aurions quelque chose du genre :

il n'y a pas d'options dans la voiture

Si option_voiture vaut maintenant 5 :

la voiture a l'option siège éjectable

[[e]] | L'instruction **break** est **nécessaire**, car si vous ne la mettez pas, l'ordinateur, ou plutôt la carte Arduino, va exécuter toutes les instructions. | Pour éviter cela, on met cette instruction **break**, qui vient de l'anglais "casser/arrêter" pour dire à la carte Arduino qu'il faut arrêter de tester les conditions car on a trouvé la valeur correspondante.

2.5.3.6 La condition ternaire ou condensée

Cette condition est en fait une simplification d'un test if...else. Il n'y a pas grand-chose à dire dessus, par conséquent un exemple suffira : Ce code :

```
int prix_voiture = 5000;
int achat_voiture = FALSE;

if(prix_voiture == 5000) // si c'est vrai
{
    achat_voiture = TRUE; // on achète la voiture
}
else // sinon
{
    achat_voiture = FALSE; // on n'achète pas la voiture
}
```

Est équivalent à celui-ci :

```
int prix_voiture = 5000;
int achat_voiture = FALSE;

achat_voiture= (prix_voiture == 5000) ? TRUE : FALSE;
```

Code : Utilisation de la condition ternaire

Cette ligne :

```
achat_voiture= (prix_voiture == 5000) ? TRUE : FALSE;
```

Se lit comme ceci : "Est-ce que le prix de la voiture est égal à 5000€? SI oui, alors j'achète la voiture SINON je n'achète pas la voiture"

[[i]] | Bon, vous n'êtes pas obligé d'utiliser cette condition ternaire, c'est vraiment pour les gros flemmards juste pour simplifier le code, mais pas forcément la lecture de ce dernier.

Nous n'avons pas encore fini avec le langage Arduino. Je vous invite donc à passer à la partie suivante pour poursuivre l'apprentissage de ce langage.

2.6 Le langage Arduino (2/2)

[[q]] | J'ai une question. Si je veux faire que le code que j'ai écrit se répète, je suis obligé de le recopier autant de fois que je veux? Ou bien il existe une solution? o_O

Voilà une excellente question qui introduit le chapitre que vous allez commencer à lire car c'est justement l'objet de ce chapitre. Nous allons voir comment faire pour qu'un bout de code se répète. Puis nous verrons, ensuite, comment organiser notre code pour que celui-ci devienne plus lisible et facile à déboguer. Enfin, nous apprendrons à utiliser les tableaux qui nous seront très utiles. Voilà le programme qui vous attend! ;)

2.6.1 Les boucles

[[q]] |Qu'est-ce qu'une boucle?

En programmation, une **boucle** est une instruction qui permet de répéter un bout de code. Cela va nous permettre de faire se répéter un bout de programme ou un programme entier. Il existe deux types principaux de boucles :

- La **boucle conditionnelle**, qui teste une condition et qui exécute les instructions qu'elle contient tant que la condition testée est vraie.
- La **boucle de répétition**, qui exécute les instructions qu'elle contient, un nombre de fois prédéterminé.

2.6.1.1 La boucle while

Problème : **Je veux que le volet électrique de ma fenêtre se ferme automatiquement quand la nuit tombe. Nous ne nous occuperons pas de faire le système qui ferme le volet à l'arrivée de la nuit. La carte Arduino dispose d'un capteur qui indique la position du volet (ouvert ou fermé). Ce que nous cherchons à faire : c'est créer un bout de code qui fait descendre le volet tant qu'il n'est pas fermé. Pour résoudre le problème posé, il va falloir que l'on utilise une boucle.**

```
/* ICI, un bout de programme permet de faire les choses suivantes :
1. un capteur détecte la tombée de la nuit et la levée du jour
   - Si c'est la nuit, alors on doit fermer le volet
   - Sinon, si c'est le jour, on doit ouvrir le volet
2. le programme lit l'état du capteur qui indique si le volet est ouvert ou fermé
3. enregistrement de cet état dans la variable de type String : position_volet
   - Si le volet est ouvert, alors : position_volet = "ouvert";
   - Sinon, si le volet est fermé : position_volet = "ferme";
*/

while(position_volet == "ouvert")
{
    // instructions qui font descendre le volet
}
```

Code : La boucle while

2.6.1.1.1 Comment lire ce code ? En anglais, le mot **while** signifie “tant que”. Donc si on lit la ligne :

```
while(position_volet == "ouvert") { /* instructions */ }
```

Il faut la lire : “TANT QUE la position du volet est **ouvert**”, on boucle/répète les instructions de la boucle (entre les accolades).

2.6.1.1.2 Construction d'une boucle while Voilà donc la syntaxe de cette boucle qu'il faut retenir :

```
while(/* condition à tester */)
{
    // les instructions entre ces accolades sont répétées
    // tant que la condition est vraie
}
```

Code : Syntaxe de la boucle while

2.6.1.1.3 Un exemple Prenons un exemple simple, réalisons un compteur !

```
// variable compteur qui va stocker le nombre de fois que la boucle
int compteur = 0;
// aura été exécutée

// tant que compteur est différent de 5, on boucle
while(compteur != 5)
{
    compteur++; // on incrémente la variable compteur à chaque tour de boucle
}
```

Code : Un petit compteur

Si on teste ce code (dans la réalité rien ne s'affiche, c'est juste un exemple pour vous montrer), cela donne :

```
compteur = 0
compteur = 1
compteur = 2
compteur = 3
compteur = 4
compteur = 5
```

Code : Résultat de notre compteur

Donc au départ, la variable **compteur** vaut 0, on exécute la boucle et on incrémente **compteur**. Mais **compteur** ne vaut pour l'instant que 1, donc on ré-exécute la boucle. Maintenant **compteur** vaut 2. On répète la boucle, ... jusqu'à 5. Si **compteur** vaut 5, la boucle n'est pas ré-exécutée et on continue le programme. Dans notre cas, le programme se termine.

2.6.1.2 La boucle do...while

Cette boucle est similaire à la précédente. Mais il y a une différence qui a son importance ! En effet, si on prête attention à la place la condition dans la boucle **while**, on s'aperçoit qu'elle est testée avant de rentrer dans la boucle. Tandis que dans une boucle do...while, la condition est testée seulement lorsque le programme est rentré dans la boucle :

```
do
{
    // les instructions entre ces accolades sont répétées
    // TANT QUE la condition est vrai
}while(/* condition à tester */);
```

Code : Syntaxe de la boucle do...while

[[i]] | Le mot **do** vient de l'anglais et se traduit par **faire**. |Donc la boucle do...while signifie "faire les instructions, tant que la condition testée est fautive". |Tandis que dans une boucle while on pourrait dire : "tant que la condition est fautive, fais ce qui suit".

[[q]] | Qu'est-ce que ça change ?

Et bien, dans une **while**, si la condition est fautive dès le départ, on entrera jamais dans cette boucle. A l'inverse, avec une boucle **do...while**, on entre dans la boucle *puis* on test la condition. Reprenons notre compteur :

```
// variable compteur = 5
int compteur = 5;

do
{
    compteur++; // on incrémente la variable compteur à chaque tour de boucle
}while(compteur < 5); // tant que compteur est inférieur à 5, on boucle
```

Code : un compteur avec do...while

Dans ce code, on définit dès le départ la valeur de **compteur** à 5. Or, le programme va rentrer dans la boucle alors que la condition est fautive. Donc **la boucle est au moins exécutée une fois** ! Et ce quelle que soit la véracité de la condition. En test cela donne :

```
compteur = 6
```

Code : Résultat de la boucle do...while

2.6.1.2.1 Concaténation Une boucle est une instruction qui a été répartie sur plusieurs lignes. Mais on peut l'écrire sur une seule ligne :

```
// variable compteur = 5
int compteur = 5;

do{compteur++;}while(compteur < 5);
```

Code : La boucle sur une seule ligne

[[e]] | C'est pourquoi il ne faut pas oublier le point virgule à la fin (après le while). |Alors que dans une simple boucle **while** le point virgule **ne doit pas** être mis !

2.6.1.3 La boucle for

Voilà une boucle bien particulière. Ce qu'elle va nous permettre de faire est assez simple. Cette boucle est exécutée X fois. Contrairement aux deux boucles précédentes, on doit lui donner trois paramètres.

```
for(int compteur = 0; compteur < 5; compteur++)
{
    // code à exécuter
}
```

Code : la boucle for

2.6.1.3.1 Fonctionnement

```
for(int compteur = 0; compteur < 5; compteur++)
```

D'abord, on crée la boucle avec le terme **for** (signifie “pour que”). Ensuite, entre les parenthèses, on doit donner trois **paramètres** qui sont :

- la création et l'assignation de la variable à une valeur de départ
- suivit de la définition de la condition à tester
- suivit de l'instruction à exécuter

Donc, si on lit cette ligne : “POUR compteur allant de 0 jusque 5, on incrémente compteur”. De façon plus concise, la boucle est exécutée autant de fois qu'il sera nécessaire à **compteur** pour arriver à 5. Donc ici, le code qui se trouve à l'intérieur de la boucle sera exécuté 5 fois.

2.6.1.3.2 A retenir La structure de la boucle :

```
for(/*initialisation de la variable*/; /*condition à laquelle la boucle s'arrête*/; /*
```

Code : syntaxe de la boucle for

2.6.1.4 La boucle infinie

La boucle infinie est très simple à réaliser, d'autant plus qu'elle est parfois très utile. Il suffit simplement d'utiliser une **while** et de lui assigner comme condition une valeur qui ne change jamais. En l'occurrence, on met souvent le chiffre 1.

```
while(1)
{
    // instructions à répéter jusqu'à l'infinie
}
```

Code : La boucle infinie

On peut lire : “TANT QUE la condition est égale à 1, on exécute la boucle”. Et cette condition sera toujours remplie puisque “1” n'est pas une variable mais bien un chiffre. Également, il est possible de mettre tout autre chiffre entier, ou bien le booléen “TRUE” :

```
while(TRUE)
{
    // instructions à répéter jusqu'à l'infinie
}
```

Code : La boucle infinie avec un booléen

[[a]] | Cela ne fonctionnera pas avec la valeur `0`. |En effet, `0` signifie “condition fausse” donc la boucle s’arrêtera aussitôt...

[[i]] | La fonction `loop()` se comporte comme une boucle infinie, puisqu’elle se répète après avoir fini d’exécuter ses tâches.

2.6.2 Les fonctions

Dans un programme, les lignes sont souvent très nombreuses. Il devient alors impératif de séparer le programme en petits bouts afin d’améliorer la lisibilité de celui-ci, en plus d’améliorer le fonctionnement et de faciliter le débogage. Nous allons voir ensemble ce qu’est une fonction, puis nous apprendrons à les créer et les appeler.

2.6.2.1 Qu’est-ce qu’une fonction ?

Une **fonction** est un “conteneur” mais différent des variables. En effet, une variable ne peut contenir qu’un nombre, tandis qu’une fonction peut contenir un *programme entier* ! Par exemple ce code est une fonction :

```
void setup()
{
    // instructions
}
```

Code : un exemple de fonction

En fait, lorsque l’on va programmer notre carte Arduino, on va écrire notre programme dans des fonctions. Pour l’instant nous n’en connaissons que 2 : `setup()` et `loop()`. Dans l’exemple précédent, à la place du commentaire, on peut mettre des instructions (conditions, boucles, variables, ...). Ce sont ces instructions qui vont constituer le programme en lui même. Pour être plus concret, une fonction est un bout de programme qui permet de réaliser une tâche bien précise. Par exemple, pour mettre en forme un texte, on peut colorier un *mot* en bleu, mettre le *mot* en gras ou encore grossir ce *mot*. A chaque fois, on a utilisé une fonction :

- **gras**, pour mettre le mot en gras
- **colorier**, pour mettre le mot en bleu
- **grossir**, pour augmenter la taille du mot

En programmation, on va utiliser des fonctions. Alors ces fonctions sont “réparties dans deux grandes familles”. Ce que j’entends par là, c’est qu’il existe des fonctions toutes prêtes dans le langage Arduino et d’autres que l’on va devoir *créer nous même*. C’est ce dernier point qui va nous intéresser.

[[e]] | On ne peut pas écrire un programme sans mettre de fonctions à l’intérieur ! |On est obligé d’utiliser la fonction `setup()` et `loop()` (même si on ne met rien dedans). |Si vous écrivez des instructions en dehors d’une fonction, le logiciel Arduino refusera systématiquement de compiler

votre programme. Il n'y a que les variables globales que vous pourrez déclarer en dehors des fonctions.

[[q]] | J'ai pas trop compris à quoi ça sert ? o_O

L'utilité d'une fonction réside dans sa capacité à simplifier le code et à le séparer en "petits bouts" que l'on assemblera ensemble pour créer le programme final. Si vous voulez, c'est un peu comme les jeux de construction en plastique : chaque pièce a son propre mécanisme et réalise une fonction. Par exemple une roue permet de rouler ; un bloc permet de réunir plusieurs autres blocs entre eux ; un moteur va faire avancer l'objet créé... Et bien tous ces éléments seront assemblés entre eux pour former un objet (voiture, maison, ...). Tout comme, les fonctions seront assemblées entre elles pour former un programme. On aura par exemple la fonction : "mettre au carré un nombre" ; la fonction : "additionner a + b" ; etc. Qui au final donnera le résultat souhaité.

2.6.2.2 Fabriquer une fonction

Pour fabriquer une fonction, nous avons besoin de savoir trois choses :

- Quel est le **type** de la fonction que je souhaite créer ?
- Quel sera son **nom** ?
- Quel(s) **paramètre(s)** prendra-t-elle ?

2.6.2.2.1 Nom de la fonction Pour commencer, nous allons, en premier lieu, choisir le nom de la fonction. Par exemple, si votre fonction doit récupérer la température d'une pièce fournie par un capteur de température : vous appellerez la fonction **lireTemperaturePiece**, ou bien **lire_temperature_piece**, ou encore **lecture_temp_piece**. Bon, des noms on peut lui en donner plein, mais soyez logique quant au choix de ce dernier. Ce sera plus facile pour comprendre le code que si vous l'appellez **tmp** (pour température ;)).

[[a]] | Un nom de fonction explicite garantit une lecture rapide et une compréhension aisée du code. | Un lecteur doit savoir ce que fait la fonction juste grâce à son nom, sans lire le contenu !

2.6.2.2.2 Les types et les paramètres Les fonctions ont pour but de découper votre programme en différentes unités logiques. Idéalement, le programme principal ne devrait utiliser que des appels de fonctions, en faisant un minimum de traitement. Afin de pouvoir fonctionner, elles utilisent, la plupart du temps, des "choses" en **entrées** et renvoient "quelque chose" en **sortie**. Les entrées seront appelées des **paramètres de la fonction** et la sortie sera appelée **valeur de retour**.

[[i]] | Notez qu'une fonction ne peut renvoyer qu'un seul résultat à la fois. Notez également qu'une fonction ne renvoie pas obligatoirement un résultat. Elle n'est pas non plus obligée d'utiliser des paramètres.

2.6.2.2.3 Les paramètres Les paramètres servent à nourrir votre fonction. Ils servent à donner des informations au traitement qu'elle doit effectuer. Prenons un exemple concret. Pour changer l'état d'une sortie du microcontrôleur, Arduino nous propose la fonction suivante : `digitalWrite(pin, value)` (<http://arduino.cc/en/Reference/DigitalWrite>). Ainsi, la référence nous explique que la fonction a les caractéristiques suivantes :

- paramètre **pin** : le numéro de la broche à changer

- paramètre **value** : l'état dans lequel mettre la broche (HIGH, (haut, +5V) ou LOW (bas, masse))
- retour : pas de retour de résultat

Comme vous pouvez le constater, l'exemple est explicite sans lire le code de la fonction. Son nom, `digitalWrite` ("écriture numérique" pour les anglophobes), signifie qu'on va changer l'état d'une broche **numérique** (donc pas analogique). Ses paramètres ont eux aussi des noms explicites, **pin** pour la broche à changer et **value** pour l'état à lui donner. Lorsque vous aller créer des fonctions, c'est à vous de voir si elles ont besoin de paramètres ou non. Par exemple, vous voulez faire une fonction qui met en pause votre programme, vous pouvez faire une fonction `Pause()` et déterminera la durée pendant laquelle le programme sera en pause. On obtiendra donc, par exemple, la syntaxe suivante : `void Pause(char duree)`. Pour résumer un peu, on a le choix de créer des **fonctions vides**, donc sans paramètres, ou bien des **fonctions "typées"** qui acceptent *un ou plusieurs* paramètres.

[[q]] | Mais c'est quoi ça "void" ?

J'y arrive ! Souvenez vous, un peu plus haut je vous expliquais qu'une fonction pouvait retourner une valeur, la fameuse valeur de sortie, je vais maintenant vous expliquer son fonctionnement.

2.6.2.3 Le type void

On vient de voir qu'une fonction pouvait accepter des paramètres et éventuellement renvoyer quelque chose. Mais ce n'est pas obligatoire. En effet, si l'on reprend notre fonction "Pause", elle ne renvoie rien car ce n'est pas nécessaire de signaler quoi que ce soit. Dans ce cas, on préfixera le nom de notre fonction avec le mot-clé "void". La syntaxe utilisée est la suivante :

```
void nom_de_la_fonction()
{
    // instructions
}
```

Code : une fonction sans valeur de retour

On utilise donc le type `void` pour dire que la fonction n'aura pas de retour. Une fonction de type `void` ne peut donc pas retourner de valeur. Par exemple :

```
void fonction()
{
    int var = 24;
    return var; // ne fonctionnera pas car la fonction est de type void
}
```

Code : Impossible pour une fonction `void` de retourner un entier

Ce code ne fonctionnera pas, parce que la fonction `int`. Ce qui est impossible ! Le compilateur le refusera et votre code final ne sera pas généré. Vous connaissez d'ailleurs déjà au moins deux fonctions qui n'ont pas de retour... Et oui, la fonction "setup" et la fonction "loop";). Il n'y en a pas plus à savoir. ;)

2.6.2.4 Les fonctions “typées”

Là, cela devient légèrement plus intéressant. En effet, si on veut créer une fonction qui calcule le résultat d'une addition de deux nombres (ou un calcul plus complexe), il serait bien de pouvoir renvoyer directement le résultat plutôt que de le stocker dans une variable qui a une portée globale et d'accéder à cette variable dans une autre fonction. En clair, l'appel de la fonction nous donne directement le résultat. On peut alors faire “ce que l'on veut” avec ce résultat (le stocker dans une variable, l'utiliser dans une fonction, lui faire subir une opération, ...)

2.6.2.4.1 Comment créer une fonction typée ? En soit, cela n'a rien de compliqué, il faut simplement remplacer long, ...) Voilà un exemple :

```
int maFonction()
{
    int resultat = 44; // déclaration de ma variable resultat
    return resultat;
}
```

Code : Une fonction typée “entier”

Notez que je n'ai pas mis les deux fonctions principales, à savoir loop(), mais elles sont obligatoires! Lorsqu'elle sera appelée, la fonction resultat. Voyez cet exemple :

```
int calcul = 0;

void loop()
{
    calcul = 10 * maFonction();
}

int maFonction()
{
    int resultat = 44; // déclaration de ma variable resultat
    return resultat;
}
```

Code : Appel d'une fonction typée

Dans la fonction calcul = 10 * 44; Ce qui nous donne : calcul = 440. Bon ce n'est qu'un exemple très simple pour vous montrer le fonctionnement. Plus tard, lorsque vous serez au point, vous utiliserez certainement cette combinaison de façon plus complexe. ;)

[[a]] | Comme cet exemple est très simple, je n'ai pas inscrit la valeur retournée par la fonction maFonction() dans une variable, mais il est préférable de le faire. Du moins, lorsque c'est utile, ce qui n'est pas le cas ici.

2.6.2.5 Les fonctions avec paramètres

C'est bien gentil tout ça, mais maintenant vous allez voir quelque chose de bien plus intéressant. Voilà un code, nous verrons ce qu'il fait après :

```

int x = 64;
int y = 192;

void loop()
{
    maFonction(x, y);
}

int maFonction(int param1, int param2)
{
    int somme = 0;
    somme = param1 + param2;
    // somme = 64 + 192 = 255

    return somme;
}

```

Code : Une fonction avec deux paramètres

[[q]] | Que se passe-t-il ?

J'ai défini trois variables : `maFonction()` est "typée" et accepte des **paramètres**. Lisons le code du début :

- On déclare nos variables
- La fonction `maFonction()` que l'on a créée

C'est sur ce dernier point que l'on va se pencher. En effet, on a donné à la fonction des paramètres. Ces paramètres servent à "nourrir" la fonction. Pour faire simple, on dit à la fonction : "**Voilà deux paramètres, je veux que tu t'en serves pour faire le calcul que je veux**" Ensuite arrive le prototype de la fonction.

[[q]] | Le prototype... de quoi tu parles ?

Le prototype c'est le "titre complet" de la fonction. Grâce à elle on connaît le **nom** de la fonction, le **type** de la valeur retournée, et le type des différents **paramètres**.

```
int maFonction(int param1, int param2)
```

Code : le prototype de la fonction.

La fonction récupère dans des variables les paramètres que l'on lui a envoyés. Autrement dit, dans la variable `y`. Soit : `param2 = y = 192`. Pour finir, on utilise ces deux variables créées "à la volée" dans le prototype de la fonction pour réaliser le calcul souhaité (une somme dans notre cas).

[[i]] | On parle aussi parfois de signature, pour désigner le nom et les paramètres de la fonction.

[[q]] | A quoi ça sert de faire tout ça ? Pourquoi on utilise pas simplement les variables `x` et `y` dans la fonction ?

Cela va nous servir à simplifier notre code. Mais pas seulement ! Par exemple, vous voulez faire plusieurs opérations différentes (addition, soustraction, etc.) et bien au lieu de créer plusieurs fonctions, on ne va en créer qu'une qui les fait toutes ! Mais, afin de lui dire quelle opération faire, vous lui donnerez un paramètre lui disant : "**Multiplie ces deux nombres**" ou bien "**additionne ces deux nombres**". Ce que cela donnerait :

```
unsigned char operation = 0;
int x = 5;
int y = 10;

void loop()
{
    // le paramètre "opération" donne le type d'opération à faire
    maFonction(x, y, operation);
}

int maFonction(int param1, int param2, int param3)
{
    int resultat = 0;
    switch(param3)
    {
        case 0 : // addition, resultat = 15
            resultat = param1 + param2;
            break;
        case 1 : // soustraction, resultat = -5
            resultat = param1 - param2;
            break;
        case 2 : // multiplication, resultat = 50
            resultat = param1 * param2;
            break;
        case 3 : // division, resultat = 0 (car nombre entier)
            resultat = param1 / param2;
            break;
        default :
            resultat = 0;
            break;
    }

    return resultat;
}
```

Code : Une fonction générique

2.6.3 Les tableaux

Comme son nom l'indique, cette partie va parler des tableaux.

[[q]] | Quel est l'intérêt de parler de cette surface ennuyeuse qu'utilisent nos chers enseignants ?

Eh bien détrompez-vous, en informatique un tableau ça n'a rien à voir ! Si on devait (beaucoup) résumer, un tableau est une grosse variable. Son but est de **stocker des éléments de mêmes types en les mettant dans des cases**. Par exemple, un prof qui stocke les notes de ses élèves. Il utilisera un tableau de float (nombre à virgule), avec une case par élèves. Nous allons utiliser cet exemple tout au long de cette partie. Voici quelques précisions pour bien tout comprendre :

- chaque élève sera identifié par un numéro allant de 0 (le premier élève) à 19 (le vingtième élève de la classe)
- on part de 0 car en informatique la première valeur dans un tableau est 0 !

2.6.3.1 Un tableau en programmation

Un tableau, tout comme sous Excel, c'est un ensemble constitué de cases, lesquels vont contenir des informations. En programmation, ces informations seront des **nombres**. Chaque case d'un tableau contiendra une valeur. En reprenant l'exemple des notes des élèves, le tableau répertoriant les notes de chaque élève ressemblerait à ceci :

->

élève 0	élève 1	élève 2	[...]	élève n-1	élève n
10	15,5	8	[...]	18	7

Table 2.10 – Un tableau en informatique

<-

2.6.3.1.1 A quoi ça sert ? On va principalement utiliser des tableaux lorsque l'on aura besoin de stocker des informations sans pour autant créer une variable pour chaque information. Toujours avec le même exemple, au lieu de créer une variable `eleve2` et ainsi de suite pour chaque élève, on inscrit les notes des élèves dans un tableau.

[[q]] | Mais, concrètement c'est quoi un tableau : une variable ? une fonction ?

Ni l'un, ni l'autre. En fait, on pourrait comparer cela avec un index qui pointe vers les valeurs de variables qui sont contenus dans chaque case du tableau. Un petit schéma pour simplifier :

->

élève 0	élève 1
variable dont on ne connaît pas le nom mais qui stocke une valeur	idem, mais variable différente de la case pr

<-

Par exemple, cela donnerait :

->

élève 0	élève 1
variable note_eleve0	variable note_eleve1

<-

Avec notre exemple :

->

élève 0	élève 1
10	15,5

<-

Soit, lorsque l'on demandera la valeur de la case 1 (correspondant à la note de l'élève 1), le tableau nous renverra le nombre : 15,5. Alors, dans un premier temps, on va voir comment déclarer un tableau et l'initialiser. Vous verrez qu'il y a différentes manières de procéder. Après, on finira par apprendre comment utiliser un tableau et aller chercher des valeurs dans celui-ci. Et pour finir, on terminera ce chapitre par un exemple. Il y a encore du boulot ! ;)

2.6.3.2 Déclarer un tableau

Comme expliqué plus tôt, un tableau contient des éléments *de même type*. On le déclare donc avec un type semblable, et une taille représentant le nombre d'éléments qu'il contiendra. Par exemple, pour notre classe de 20 étudiants :

```
float notes[20];
```

Code : déclarer un tableau de 20 cases

On veut stocker des notes, donc des valeurs décimales entre 0 et 20. On va donc créer un tableau de float (car c'est le type de variable qui accepte les nombres à virgule, souvenez-vous ! ;)). Dans cette classe, il y a 20 élèves (de 0 à 19) donc le tableau contiendra 20 éléments. Si on voulait faire un tableau de 100 étudiants dans lesquels on recense leurs nombres d'absence, on ferait le tableau suivant :

```
char absence[100];
```

Code : un tableau de char

2.6.3.3 Accéder et modifier une case du tableau

Pour accéder à une case d'un tableau, il suffit de connaître l'**indice** de la case à laquelle on veut accéder. L'indice c'est le numéro de la case qu'on veut lire/écrire. Par exemple, pour lire la valeur de la case 10 (donc indice 9 car on commence à 0) :

```
float notes[20]; // notre tableau
float valeur; // une variable qui contiendra une note
```

```
// valeur contient désormais la note du dixième élève
valeur = notes[9];
```

Code : Accéder à une valeur

Ce code se traduit par l'enregistrement de la valeur contenue dans la dixième case du tableau, dans une variable nommée `valeur`. A présent, si on veut aller modifier cette même valeur, on fait comme avec une variable normale, il suffit d'utiliser l'opérateur '=' :

```
notes[9] = 10,5; // on change la note du dixième élève
```

Code : Modifier une valeur

En fait, on procède de la même manière que pour changer la valeur d'une variable, car, je vous l'ai dit, chaque case d'un tableau est une variable qui contient une valeur ou non.

[[a]] | Faites attention aux indices utilisés. Si vous essayez de lire/écrire dans une case de tableau trop loin (indice trop grand, par exemple 987362598412 :P), le comportement pourrait devenir imprévisible. Car en pratique vous modifieriez des valeurs qui seront peut-être utilisées par le système pour autre chose. Ce qui pourrait avoir de graves conséquences !

[[i]] | Vous avez sûrement rencontré des crashes de programme sur votre ordinateur, ils sont souvent dû à la modification de variable qui n'appartiennent pas au programme, donc l'OS "tue" ce programme qui essaye de manipuler des trucs qui ne lui appartiennent pas.

*[OS] : Operating System

2.6.3.4 Initialiser un tableau

Au départ, notre tableau était vide :

```
float notes[20];
// on créer un tableau dont le contenu est vide, on sait simplement qu'il contiendra 20 cases
```

Ce que l'on va faire, c'est **initialiser** notre tableau. On a la possibilité de remplir chaque case *une par une* ou bien utiliser une boucle qui remplira le tableau à notre place. Dans le premier cas, on peut mettre la valeur que l'on veut dans chaque case du tableau, tandis qu'avec la deuxième solution, on remplira les cases du tableau avec la même valeur, bien que l'on puisse le remplir avec des valeurs différentes mais c'est un peu plus compliqué. Dans notre exemple des notes, on part du principe que l'examen n'est pas passé, donc tout le monde à 0. :P Pour cela, on parcourt toutes les cases en leur mettant la valeur 0 :

```
char i=0; // une variable que l'on va incrémenter
float notes[20]; // notre tableau

void setup()
{
    // boucle for qui remplira le tableau pour nous
    for(i = 0; i < 20; i++)
    {
        notes[i] = 0; // chaque case du tableau vaudra 0
    }
}
```

Code : Initialisation d'un tableau

[[i]] | L'initialisation d'un tableau peut se faire directement lors de sa création, comme ceci :

```
float note[] = {0,0,0,0 /*, etc.*/ };
// Le tableau aura alors autant de case que de nombre passé en paramètres
```

Code : Initialisation à la déclaration

2.6.3.5 Exemple de traitement

[[q]] | Bon c'est bien beau tout ça, on a des notes coincées dans un tableau, on en fait quoi? :roll :
Excellente question, et ça dépendra de l'usage que vous en aurez :)! Voyons des cas d'utilisations pour notre tableau de notes (en utilisant des fonctions;)).

2.6.3.5.1 La note maximale Comme le titre l'indique, on va rechercher la note maximale (le meilleur élève de la classe). La fonction recevra en paramètre le tableau de float, le nombre d'éléments dans ce tableau et renverra la meilleure note.

```
float meilleurNote(float tableau[], int nombreEleve)
{
    int i = 0;
    int max = 0; // variable contenant la future meilleure note

    for(i=0; i<nombreEleve, i++)
    {
        // si la note lue est meilleure que la meilleure actuelle
        if(tableau[i] > max)
        {
            // alors on l'enregistre
            max = tableau[i];
        }
    }
    // on retourne la meilleure note
    return max;
}
```

Code : Recherche de la note maximale

Ce que l'on fait, pour lire un tableau, est exactement la même chose que lorsqu'on l'initialise avec une boucle for.

[[i]] | Il est tout à fait possible de mettre la valeur de la case recherché dans une variable :

```
int valeur = tableau[5]; // on enregistre la valeur de la case 6 du tableau dans une v
```

Voilà, ce n'était pas si dur, vous pouvez faire pareil pour chercher la valeur minimale afin vous entraîner!

2.6.3.5.2 Calcul de moyenne Ici, on va chercher la moyenne des notes. La signature de la fonction sera exactement la même que celle de la fonction précédente, à la différence du nom! Je vous laisse réfléchir, voici la signature de la fonction, le code est plus bas mais essayez de le trouver vous-même avant :

```
float moyenneNote(float tableau[], int nombreEleve)
```

Une solution : [[s]] | cpp | float moyenneNote(float tableau[], int nombreEleve)
| { | int i = 0; | double total = 0; // addition de toutes les notes |
float moyenne = 0; // moyenne des notes | for(i = 0; i < nombreEleve; i++)

```
| { | total = total + tableau[i]; | } | moyenne = total / nombreEleve; |  
return moyenne; | } | |Code : Calcul de la moyenne des notes
```

On en termine avec les tableaux, on verra peut être plus de choses en pratique. :)

Maintenant vous pouvez pleurer, de joie bien sûr, car vous venez de terminer la première partie !
A présent, faisons place à la pratique...

3 Gestion des entrées / sorties

Maintenant que vous avez acquis assez de connaissances en programmation et quelques notions d'électronique, on va se pencher sur l'utilisation de la carte Arduino. Je vais vous parler des entrées et des sorties de la carte et vous aider à créer votre premier programme !

3.1 Notre premier programme !

Vous voilà enfin arrivé au moment fatidique où vous allez devoir programmer ! Mais avant cela, je vais vous montrer ce qui va nous servir pour ce chapitre.

En l'occurrence, apprendre à utiliser une LED et la référence, présente sur le site arduino.cc qui vous sera très utile lorsque vous aurez besoin de faire un programme utilisant une notion qui n'est pas traitée dans ce cours.

*[LED] : Light Emitting Device/Diode

3.1.1 La diode électroluminescente

3.1.1.1 DEL / LED ?

La question n'est pas de savoir quelle abréviation choisir mais plutôt de savoir ce que c'est. Une **DEL / LED** : Diode Electro-Luminescente, ou bien "Light Emitting Diode" en anglais. Il s'agit d'un composant électronique qui crée de la lumière quand il est parcouru par un courant électrique. Je vous en ai faits acheter de différentes couleurs. Vous pouvez, pour ce chapitre, utiliser celle que vous voudrez, cela m'est égal. ;) Vous voyez, ci-dessous sur votre droite, la photo d'une DEL de couleur rouge. La taille n'est pas réelle, sa "tête" (en rouge) ne fait que 5mm de diamètre. C'est ce composant que nous allons essayer d'allumer avec notre carte Arduino. Mais avant, voyons un peu comment il fonctionne.

[[i]] J'appellerai la diode électroluminescente, tout au long du cours, une LED. [Une LED est en fait une **diode** qui émet de la lumière.]Je vais donc vous parler du fonctionnement des diodes en même temps que celui des LED.

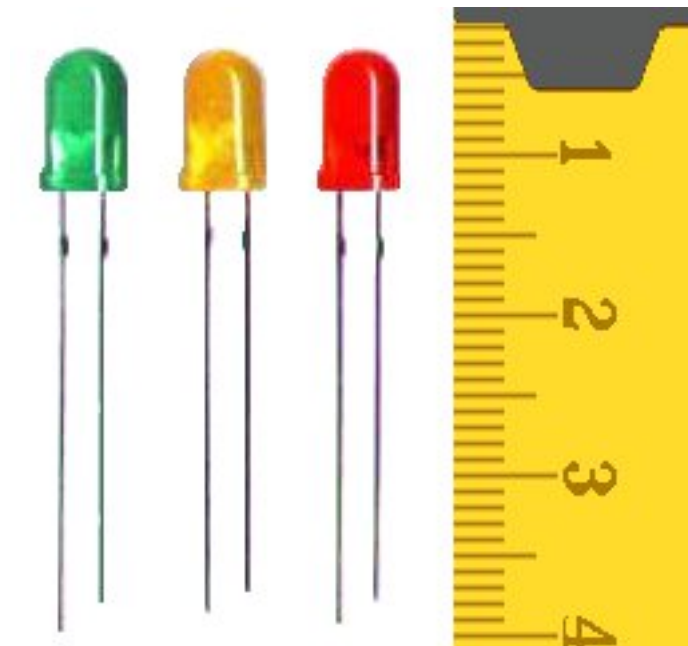


Figure : Des LED / DEL - (CC-BY-SA, Sa-

peraud)

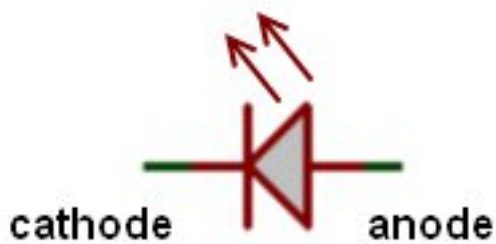
3.1.1.1.1 Symbole Sur un schéma électronique, chaque composant est repéré par un symbole qui lui est propre. Celui de la diode est le suivant :



->

<-

Celui de la LED est :



->

<-

Il y a donc très peu de différence entre les deux. La LED est simplement une diode qui émet de la lumière, d'où les flèches sur son symbole.

3.1.1.1.2 Astuce mnémotechnique Pour ce souvenir de quel côté est l'anode ou la cathode, voici un moyen mnémotechnique simple et en image :) ...

->



K comme K-thode

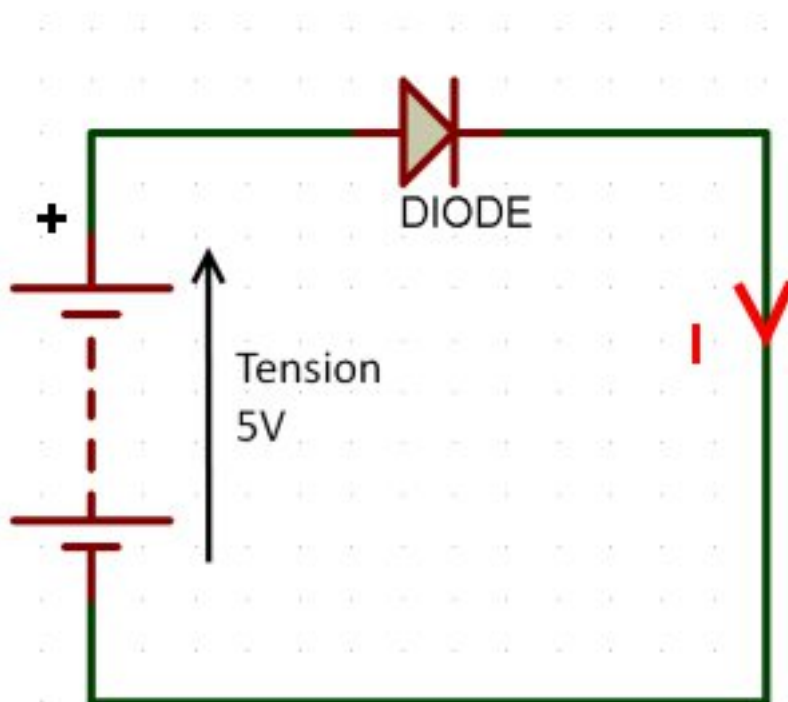
A comme A-node

Table 3.1 – Moyen mnémotechnique pour se souvenir de l’anode et de la cathode

<-

3.1.1.2 Fonctionnement

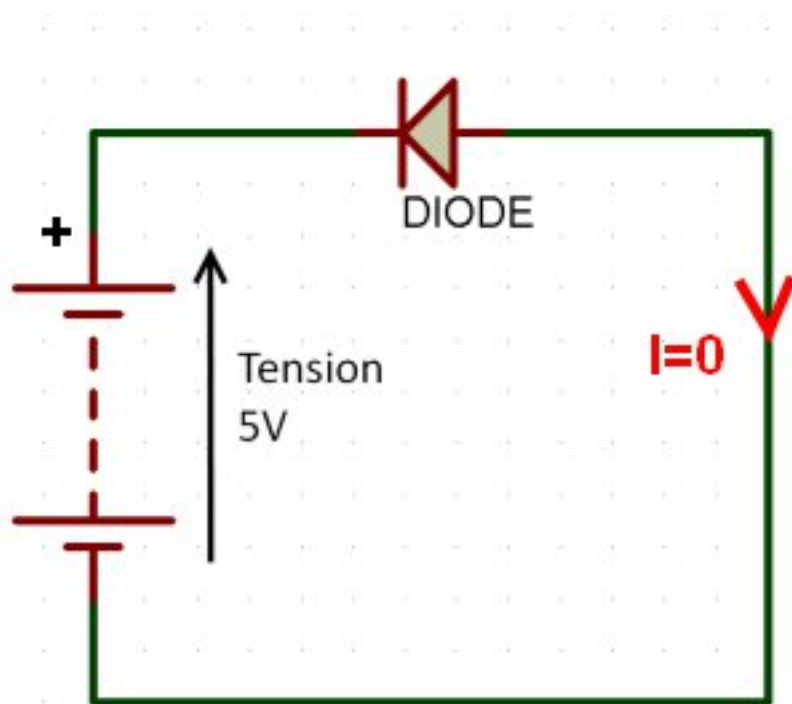
3.1.1.2.1 Polarisation directe On parle de **polarisation** lorsqu’un composant électronique est utilisé dans un circuit électronique de la “bonne manière”. En fait lorsqu’il est polarisé, c’est qu’on l’utilise de la façon souhaitée. Pour polariser la diode, on doit faire en sorte que le courant la parcourt de l’anode vers la cathode. Autrement dit, la tension doit être plus élevée à l’anode qu’à la cathode.



->

<-

3.1.1.2.2 Polarisation inverse La polarisation inverse d'une diode est l'opposé de la polarisation directe. Pour créer ce type de montage, il suffit simplement, dans notre cas, de "retourner" la diode enfin la brancher "à l'envers". Dans ce cas, le courant ne passe pas.



[[i]] |Note : une diode polarisée en inverse ne grillera pas si elle est utilisée dans de bonnes conditions. |En fait, elle fonctionne de "la même façon" pour le courant positif et négatif.

3.1.1.2.3 Utilisation [[a]] |Si vous ne voulez pas faire partir votre première diode en fumée, je vous conseille de lire les prochaines lignes attentivement :P

En électronique, deux paramètres sont à prendre en compte : le courant et la tension. Pour une diode, deux tensions sont importantes. Il s'agit de la tension maximum en polarisation directe, et la tension maximum en polarisation inverse. Ensuite, pour un bon fonctionnement des LED, le courant a lui aussi son importance.

3.1.1.2.4 La tension maximum directe Lorsque l'on utilise un composant, on doit prendre l'habitude d'utiliser la "datasheet" ("documentation technique" en anglais) qui nous donne toutes les caractéristiques sur le composant. Dans cette datasheet, on retrouvera quelque chose appelé "Forward Voltage", pour la diode. Cette indication représente la chute de tension aux bornes de la diode lorsque du courant la traverse en sens direct. Pour une diode classique (type 1N4148), cette tension sera d'environ 1V. Pour une LED, on considérera plutôt une tension de 1,2 à 1,6V.

[[i]] |Bon, pour faire nos petits montages, on ne va pas chipoter, mais c'est la démarche à faire lorsque l'on conçoit un schéma électrique et que l'on dimensionne ses composants.

3.1.1.2.5 La tension maximum inverse Cette tension représente la différence maximum admissible entre l'anode et la cathode lorsque celle-ci est branchée "à l'envers". En effet, si vous mettez une tension trop importante à ces bornes, la jonction ne pourra pas le supporter et partira en fumée. En anglais, on retrouve cette tension sous le nom de "Reverse Voltage" (ou même

“Breakdown Voltage”). Si l’on reprend la diode 1N4148, elle sera comprise entre 75 et 100V. Au-delà de cette tension, la jonction casse et la diode devient inutilisable. Dans ce cas, la diode devient soit un court-circuit, soit un circuit ouvert. Parfois cela peut causer des dommages importants dans nos appareils électroniques ! Quoi qu’il en soit, on ne manipulera jamais du 75V ! :P

3.1.1.2.6 Le courant de passage Le courant qui traverse une LED a son importance. Si l’on branche directement la LED sur une pile, elle va s’allumer, puis tôt ou tard finira par s’éteindre... définitivement. En effet, si on ne limite pas le courant traversant la LED, elle prendra le courant maximum, et ça c’est pas bon car ce n’est pas le courant maximum qu’elle peut supporter. Pour limiter le courant, on place une résistance avant (ou après) la LED. Cette résistance, soigneusement calculée, lui permettra d’assurer un fonctionnement optimal.

[[q]] | Mais comment on la calcule cette résistance ?

Simplement avec la formule de base, la loi d’Ohm. ;) Petit rappel :

$$U = R * I$$

Dans le cas d’une LED, on considère, en général, que l’intensité la traversant doit être de 20 mA. Si on veut être rigoureux, il faut aller chercher cette valeur dans le datasheet. On a donc $I = 20mA$. Ensuite, on prendra pour l’exemple une tension d’alimentation de 5V (en sortie de l’Arduino, par exemple) et une tension aux bornes de la LED de 1,2V en fonctionnement normal. On peut donc calculer la tension qui sera aux bornes de la résistance : $U_r = 5 - 1,2 = 3,8V$. Enfin, on peut calculer la valeur de la résistance à utiliser :

$$R = \frac{U}{I} R = \frac{3,8}{0,02} R = 190\Omega$$

Et voilà, vous connaissez la valeur de la résistance à utiliser pour être sûr de ne pas griller des LED à tour de bras. :) À votre avis, vaut-il mieux utiliser une résistance de plus forte valeur ou de plus faible valeur ?

[[secret]] | **Réponse :** | Si on veut être sûr de ne pas détériorer la LED à cause d’un courant trop fort, on doit placer une résistance dont la valeur est plus grande que celle calculée. Autrement, la diode recevra le courant maximal pouvant la traverser. |

3.1.2 Par quoi on commence ?

3.1.2.0.1 Le but Le but de ce premier programme est... de vous faire programmer ! :P Non, je ne rigole pas ! Car c’est en pratiquant la programmation que l’on retient le mieux les commandes utilisées. De plus, en faisant des erreurs, vous vous forgerez de bonnes bases qui vous seront très utiles ensuite, lorsqu’il s’agira de gagner du temps. Mais avant tout, c’est aussi parce que ce tuto est centré sur la programmation que l’on va programmer !

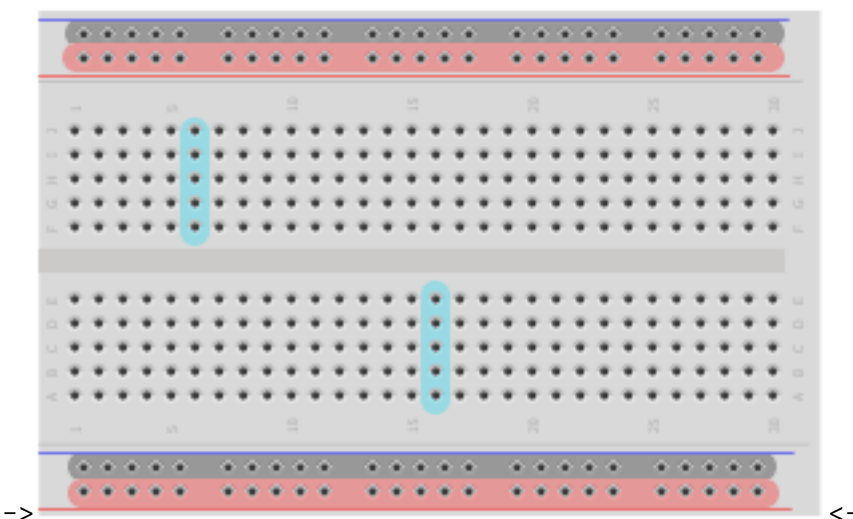
3.1.2.0.2 Objectif L’objectif de ce premier programme va consister à allumer une LED. C’est nul me direz vous. J’en conviens. Cependant, vous verrez que ce n’est pas très simple. Bien entendu, je n’allais pas créer un chapitre entier dont le but ultime aurait été d’allumer une LED ! Non. Alors j’ai prévu de vous montrer deux trois trucs qui pourront vous aider dès lors que vous voudrez sortir du nid et prendre votre envol vers de nouveaux cieux ! ;)

3.1.2.0.3 Matériel Pour pouvoir programmer, il vous faut, bien évidemment, une carte Arduino et un câble USB pour relier la carte au PC. Mais pour voir le résultat de votre programme, vous aurez besoin d'éléments supplémentaires. Notamment, une LED et une résistance.

3.1.2.1 Un outil formidable : la breadboard !

Je vais maintenant vous présenter un outil très pratique lorsque l'on fait ses débuts en électronique ou lorsque l'on veut tester rapidement/facilement un montage. Cet accessoire s'appelle une **breadboard** (littéralement : planche à pain, techniquement : plaque d'essai sans soudure). Pour faire simple, c'est une plaque pleine de trous !

3.1.2.1.1 Principe de la breadboard Certes la plaque est pleine de trous, mais pas de manière innocente ! En effet, la plupart d'entre eux sont reliés. Voici un petit schéma rapide qui va aider à la compréhension.



Comme vous pouvez le voir sur l'image, j'ai dessiné des zones. Les zones rouges et noires correspondent à l'alimentation. Souvent, on retrouve deux lignes comme celles-ci permettant de relier ses composants aux alimentations nécessaires. Par convention, le noir représente la masse et le rouge est l'alimentation (+5V, +12V, -5V... ce que vous voulez y amener). Habituellement tous les trous d'une même **ligne** sont reliés sur cette zone. Ainsi, vous avez une ligne d'alimentation parcourant tout le long de la carte. Ensuite, on peut voir des zones en bleu. Ces zones sont reliées entre elles par **colonne**. Ainsi, tous les trous sur une même colonne sont reliés entre eux. En revanche, chaque colonne est distincte. En faisant chevaucher des composants sur plusieurs colonnes vous pouvez les connecter entre eux. Dernier point, vous pouvez remarquer un espace coupant la carte en deux de manière symétrique. Cet espace coupe aussi la liaison des colonnes. Ainsi, sur le dessin ci-dessus on peut voir que chaque colonne possède cinq trous reliés entre eux. Cet espace au milieu est normalisé et doit faire la largeur des circuits intégrés standards. En posant un circuit intégré à cheval au milieu, chaque patte de ce dernier se retrouve donc sur une colonne, isolée de la précédente et de la suivante.

[[i]] Si vous voulez voir plus concrètement ce fonctionnement, je vous conseille d'essayer le logiciel **Fritzing**, qui permet de faire des circuits de manière assez simple et intuitive. Vous verrez ainsi comment les colonnes sont séparées les unes des autres. De plus, ce logiciel sera utilisé pour le reste du tuto pour les captures d'écrans des schémas électroniques.

3.1.2.2 Réalisation

Avec le brochage de la carte Arduino, vous devrez connecter la plus grande patte au +5V (broche 5V). La plus petite patte étant reliée à la résistance, elle-même reliée à la broche numéro 2 de la carte. Tout ceci a une importance. En effet, on pourrait faire le contraire, brancher la LED vers la masse et l'allumer en fournissant le 5V depuis la broche de signal. Cependant, les composants comme les microcontrôleurs n'aiment pas trop délivrer du courant, ils préfèrent l'absorber. Pour cela, on préférera donc alimenter la LED en la plaçant au +5V et en mettant la broche de Arduino à la masse pour faire passer le courant. Si on met la broche à 5V, dans ce cas le potentiel est le même de chaque côté de la LED et elle ne s'allume pas ! Ce n'est pas plus compliqué que ça ! ;) Schéma de la réalisation (un exemple de branchement sans breadboard et deux exemples avec) :

3.1.2.3 Créer un nouveau projet

Pour pouvoir programmer notre carte, il faut que l'on crée un nouveau programme. Ouvrez votre logiciel Arduino. Allez dans le menu *File* et choisissez l'option *Save as...* :

Vous arrivez dans cette nouvelle fenêtre :

Tapez le nom du programme, dans mon cas, je l'ai appelé *test_1*. Enregistrez. Vous arrivez dans votre nouveau programme, qui est vide pour l'instant, et dont le nom s'affiche en Haut de la fenêtre et dans un petit onglet :

3.1.2.3.1 Le code minimal Pour commencer le programme, il nous faut un code minimal. Ce code va nous permettre d'initialiser la carte et va servir à écrire notre propre programme. Ce code, le voici :

```
// fonction d'initialisation de la carte
void setup()
{
    // contenu de l'initialisation
}

// fonction principale, elle se répète (s'exécute) à l'infini
void loop()
{
    // contenu de votre programme
}
```

Code : Squelette minimal d'un programme Arduino

3.1.3 Créer le programme : les bons outils !

3.1.3.1 La référence Arduino

3.1.3.1.1 Qu'est ce que c'est ? L'Arduino étant un projet dont la communauté est très active, nous offre sur son site internet une **référence**. Mais qu'est ce que c'est ? Eh bien il s'agit simplement de "la notice d'utilisation" du langage Arduino. Plus exactement, une page internet de leur site est dédiée au référencement de chaque code que l'on peut utiliser pour faire un programme.

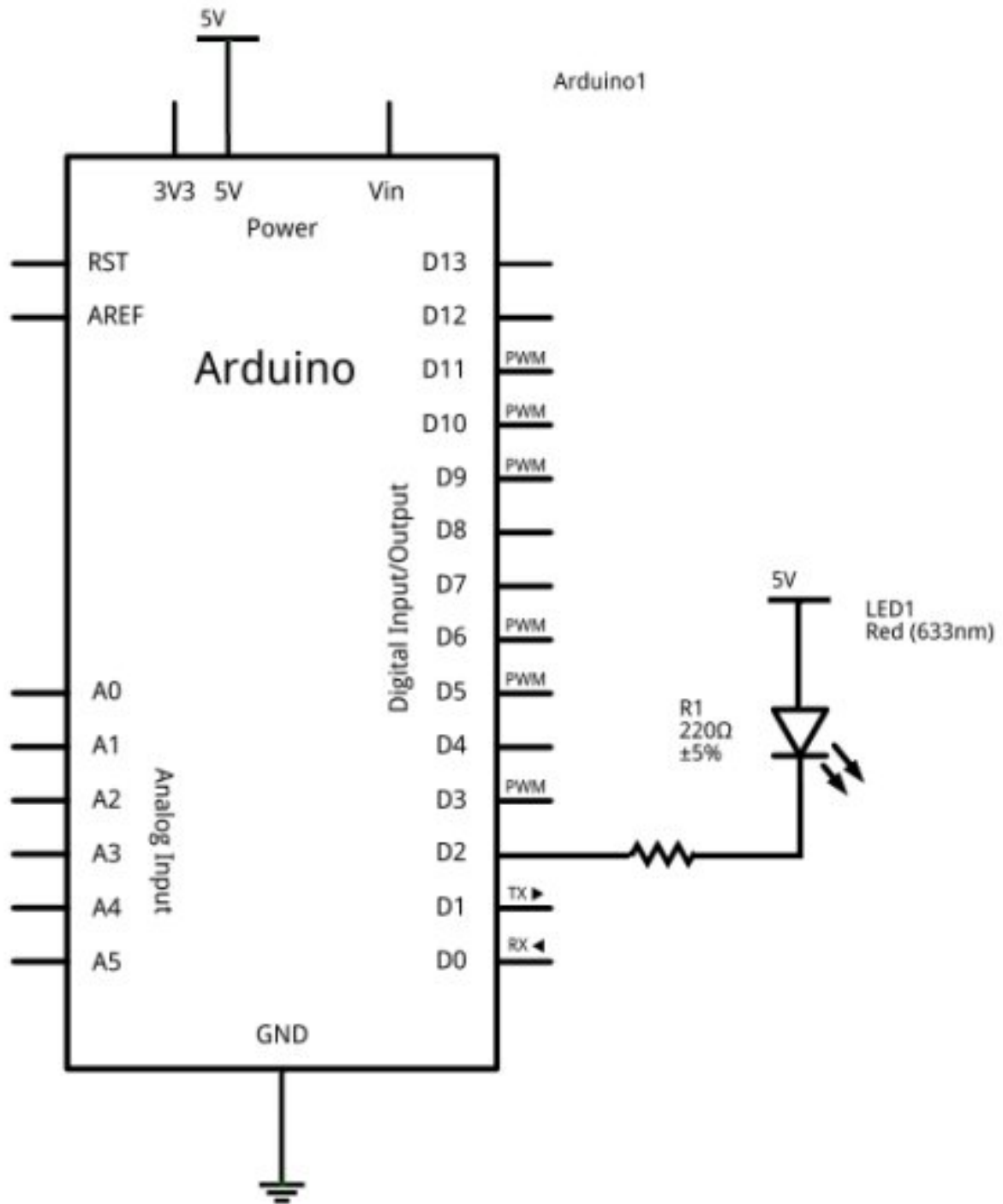


Figure 3.1 – réalisation montage, schéma de la carte

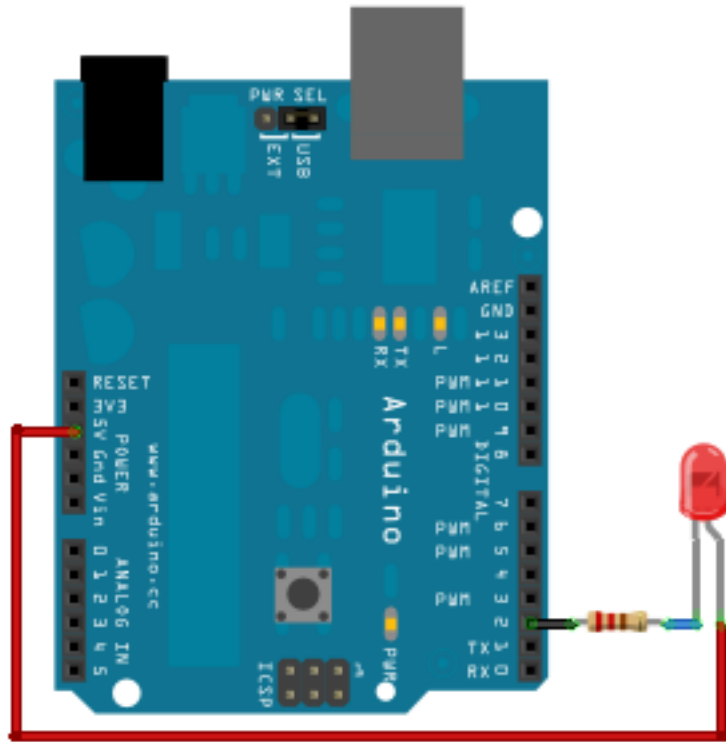


Figure 3.2 – Montage avec une LED et sans breadboard

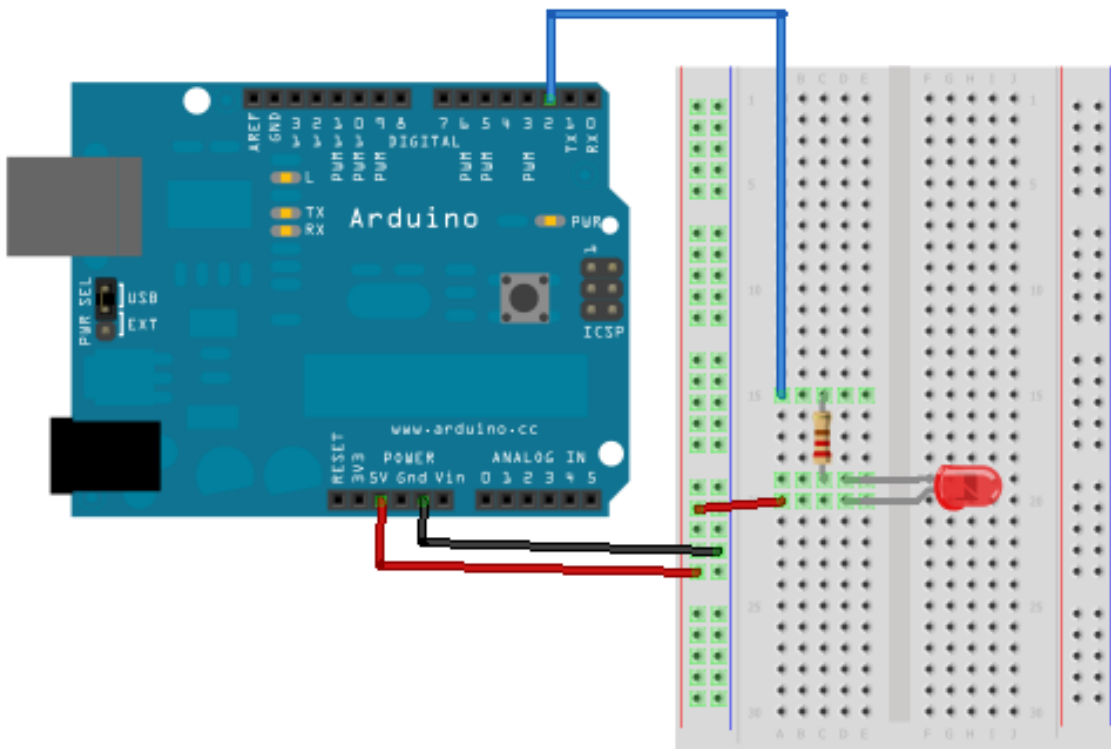


Figure 3.3 – Montage une LED sur breadboard

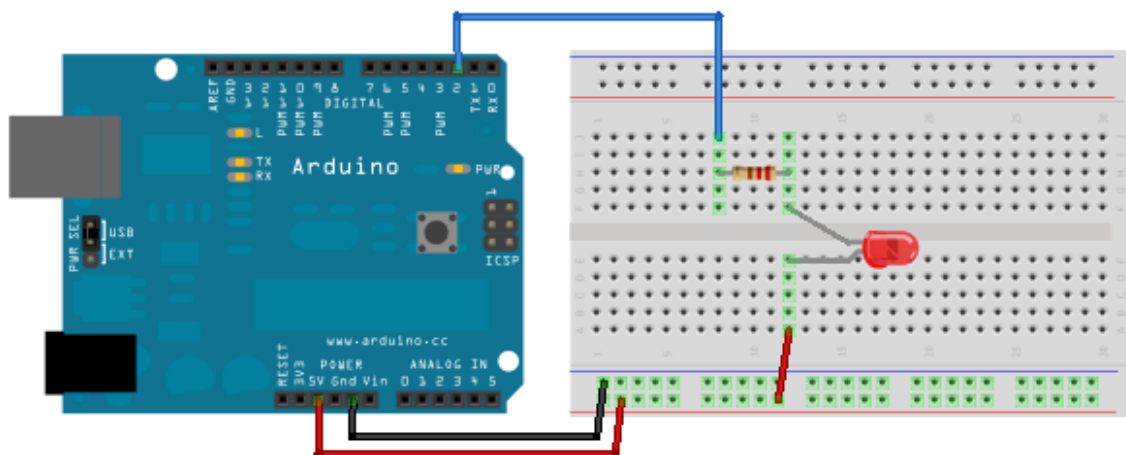


Figure 3.4 – Montage une LED sur breadboard

3.1.3.1.2 Comment l'utiliser ? Pour l'utiliser, il suffit d'aller sur [la page de leur site](#), malheureusement en anglais, mais dont il existe une traduction pas tout à fait complète sur le site Français Arduino. Ce que l'on voit en arrivant sur la page : trois colonnes avec chacune un type d'éléments qui forment les langages Arduino.

- *Structure* : cette colonne référence les éléments de la structure du langage Arduino. On y retrouve les conditions, les opérations, etc.
- *Variables* : comme son nom l'indique, elle regroupe les différents types de variables utilisables, ainsi que certaines opérations particulières
- *Functions* : ici c'est tout le reste, mais surtout les fonctions de lecture/écriture des broches du microcontrôleur (ainsi que d'autres fonctions bien utiles)

[[a]] Il est très important de savoir utiliser la documentation que nous offre Arduino! |Car en sachant cela, vous pourrez faire des programmes sans avoir appris préalablement à utiliser telle fonction ou telle autre. Vous pourrez devenir les maitres du monde!!! |Euh, non, je crois pas en fait... ^^

3.1.3.2 Allumer notre LED

3.1.3.2.1 1ère étape Il faut avant tout définir les broches du micro-contrôleur. Cette étape constitue elle-même deux sous étapes. La première étant de créer une variable définissant la broche utilisée, ensuite, définir si la broche utilisée doit être une entrée du micro-contrôleur ou une sortie. Premièrement, donc, définissons la broche utilisée du microcontrôleur :

```
const int led_rouge = 2; // définition de la broche 2 de la carte en tant que variable
```

Le terme `const` signifie que l'on définit la variable comme étant constante. Par conséquent, on change la nature de la variable qui devient alors constante et sa valeur ne pourra jamais être changée. Le terme `int` correspond à un type de variable. En définissant une variable de ce type, elle peut stocker un nombre allant de -2147483648 à $+2147483647$! Cela nous suffit amplement! ;) Nous sommes donc en présence d'une variable, nommée `led_rouge`, qui est en fait une constante, qui peut prendre une valeur allant de -2147483648 à $+2147483647$. Dans notre cas, cette variable, pardon constante, est assignée à 2. Le chiffre 2.

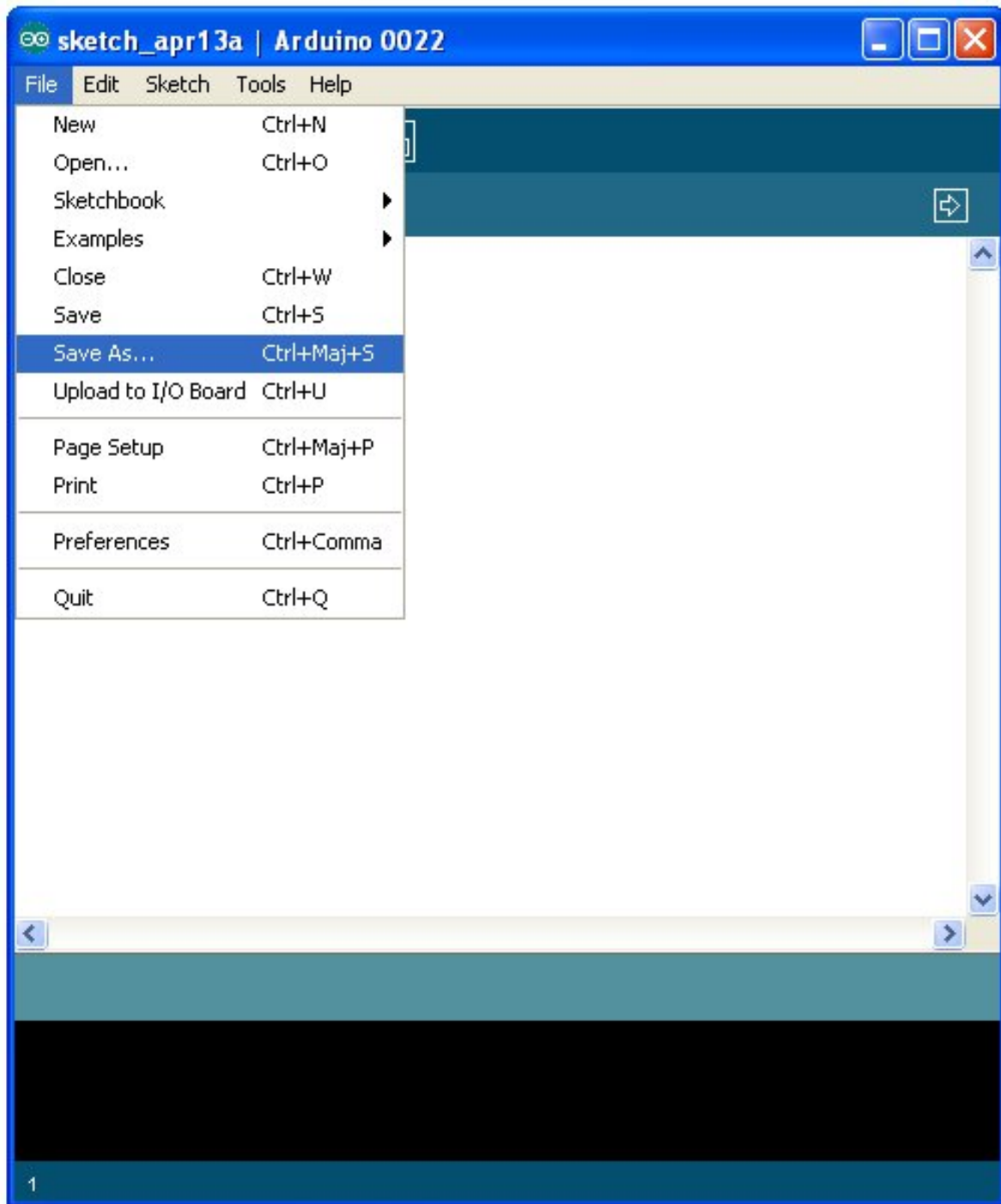


Figure 3.5 – Enregistrer sous...

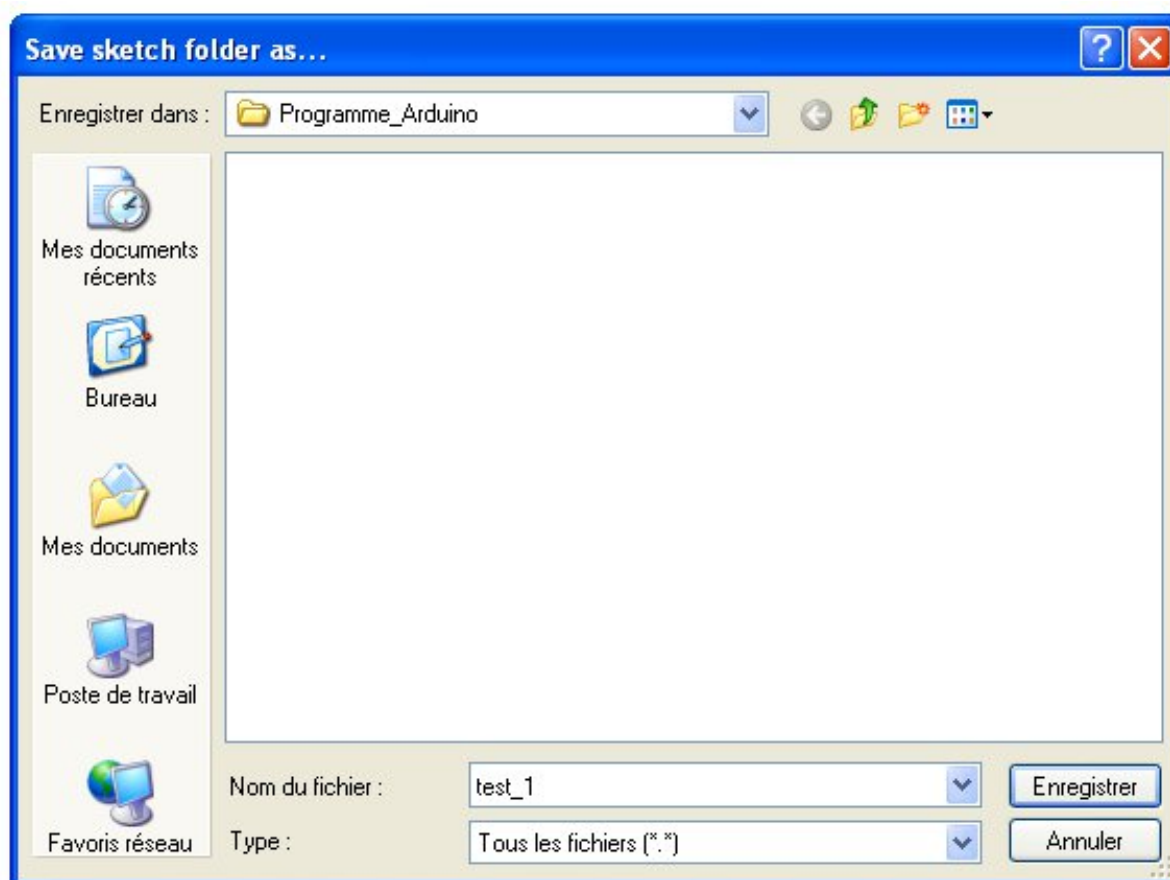


Figure 3.6 – Enregistrer

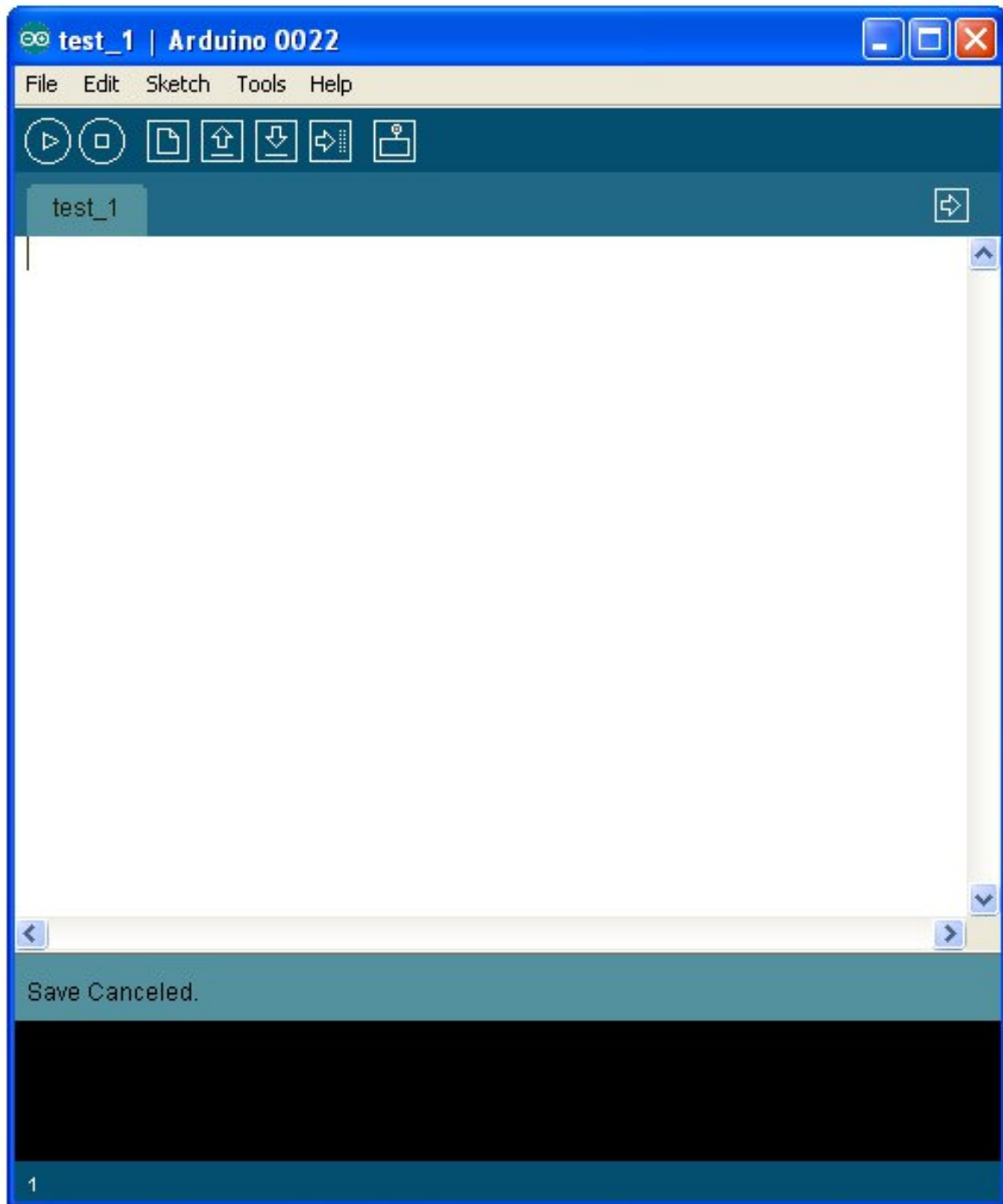


Figure 3.7 – Votre nouveau programme !

[[i]] |Lorsque votre code sera compilé, le micro-contrôleur saura ainsi que sur sa broche numéro 2, il y a un élément connecté.

Bon, cela ne suffit pas de définir la broche utilisée. Il faut maintenant dire si cette broche est une **entrée** ou une **sortie**. Oui, car le micro-contrôleur a la capacité d'utiliser certaines de ses broches en entrée ou en sortie. C'est fabuleux ! En effet, il suffit simplement d'interchanger UNE ligne de code pour dire qu'il faut utiliser une broche en entrée (récupération de données) ou en sortie (envoi de données). Cette ligne de code justement, parlons-en ! Elle doit se trouver dans la fonction `setup()`. Dans la référence, ce dont nous avons besoin se trouve dans la catégorie **Functions**, puis dans **Digital I/O**. I/O pour Input/Output, ce qui signifie dans la langue de Molière : Entrée/Sortie. La fonction se trouve être `pinMode()`. Pour utiliser cette fonction, il faut lui envoyer deux paramètres :

- Le nom de la variable que l'on a défini à la broche
- Le type de broche que cela va être (entrée ou sortie)

```
// fonction d'initialisation de la carte
void setup()
{
    // initialisation de la broche 2 comme étant une sortie
    pinMode(led_rouge, OUTPUT);
}
```

Code : Initialisation de la sortie

Ce code va donc définir la `led_rouge` (qui est la broche numéro 2 du micro-contrôleur) en sortie, car `OUTPUT` signifie en français : *sortie*. Maintenant, tout est prêt pour créer notre programme. Voici le code quasiment complet :

```
// définition de la broche 2 de la carte en tant que variable
const int led_rouge = 2;

// fonction d'initialisation de la carte
void setup()
{
    // initialisation de la broche 2 comme étant une sortie
    pinMode(led_rouge, OUTPUT);
}

// fonction principale, elle se répète (s'exécute) à l'infini
void loop()
{
    // contenu de votre programme
}
```

3.1.3.2.2 2e étape Cette deuxième étape consiste à créer le contenu de notre programme. Celui qui va aller remplacer le commentaire dans la fonction `loop()`, pour réaliser notre objectif : allumer la LED ! Là encore, on ne claque pas des doigts pour avoir le programme tout prêt ! :P Il faut retourner chercher dans la référence Arduino ce dont on a besoin.

[[q]] |Oui, mais là, on ne sait pas ce que l'on veut ? o_O

On cherche une fonction qui va nous permettre d’allumer cette LED. Il faut donc que l’on se débrouille pour la trouver. Et avec notre niveau d’anglais, on va facilement trouver. Soyons un peu logique, si vous le voulez bien. Nous savons que c’est une fonction qu’il nous faut (je l’ai dit il y a un instant), on regarde donc dans la catégorie **Functions** de la référence. Si on garde notre esprit logique, on va s’occuper d’allumer une LED, donc de dire quel est l’état de sortie de la broche numéro 2 où laquelle est connectée notre LED. Donc, il est fort à parier que cela se trouve dans **Digital I/O**. Tiens, il y a une fonction suspecte qui se prénomme `digitalWrite()`. En français, cela signifie “écriture numérique”. C’est donc l’écriture d’un état logique (0 ou 1). Quelle se trouve être la première phrase dans la description de cette fonction ? Celle-ci : “Write a HIGH or a LOW value to a digital pin”. D’après notre niveau bilingue, on peut traduire par : *Écriture d’une valeur HAUTE ou une valeur BASSE sur une sortie numérique*. Bingo ! C’est ce que l’on recherchait ! Il faut dire que je vous ai un peu aidé. ^^

[[q]] | Ça signifie quoi “valeur HAUTE ou valeur BASSE” ?

En électronique numérique, un niveau haut correspondra à une tension de +5V et un niveau dit bas sera une tension de 0V (généralement la masse). Sauf qu’on a connecté la LED au pôle positif de l’alimentation, donc pour qu’elle s’allume, il faut qu’elle soit reliée au 0V. Par conséquent, on doit mettre un état bas sur la broche du microcontrôleur. Ainsi, la différence de potentiel aux bornes de la LED permettra à celle-ci de s’allumer. Voyons un peu le fonctionnement de `digitalWrite()` en regardant dans sa syntaxe. Elle requiert deux paramètres. Le nom de la broche que l’on veut mettre à un état logique et la valeur de cet état logique. Nous allons donc écrire le code qui suit, d’après cette syntaxe :

```
digitalWrite(led_rouge, LOW); // écriture en sortie (broche 2) d'un état BAS
```

Si on teste le code entier :

```
// définition de la broche 2 de la carte en tant que variable
const int led_rouge = 2;

// fonction d'initialisation de la carte
void setup()
{
    // initialisation de la broche 2 comme étant une sortie
    pinMode(led_rouge, OUTPUT);
}

// fonction principale, elle se répète (s'exécute) à l'infini
void loop()
{
    // écriture en sortie (broche 2) d'un état BAS
    digitalWrite(led_rouge, LOW);
}
```

Code : Allumage de la led

On voit s’éclairer la LED !!! C’est fantastique ! o_o

3.1.4 Comment tout cela fonctionne ?

[[q]] | Et comment ça se passe à l’intérieur ?? o_o | Je comprends pas comment le microcontrôleur

fait pour tout comprendre et tout faire. |Je sais qu'il utilise les 0 et les 1 du programme qu'on lui a envoyé, mais comment il sait qu'il doit aller chercher le programme, le lire, l'exécuter, etc. ?

Eh bien, eh bien ! En voilà des questions ! Je vais essayer d'y répondre simplement, sans entrer dans le détail qui est quand même très compliqué. Bon, si vous êtes prêt, c'est parti ! D'abord, tout se passe dans le cerveau du microcontrôleur...

3.1.4.1 Le démarrage

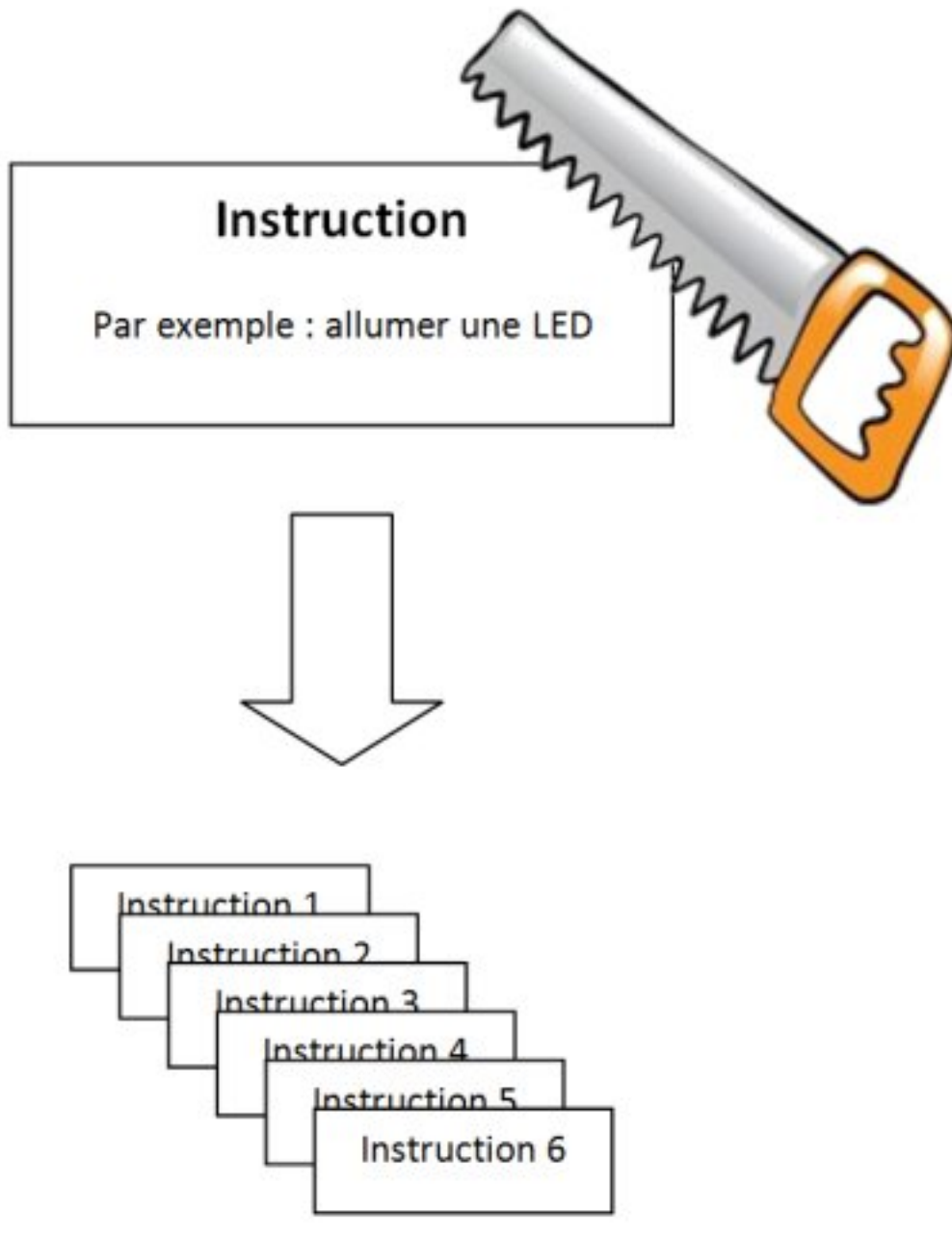
Un peu comme vous démarreriez un ordinateur, la carte Arduino aussi démarre. Alors c'est un peu transparent parce qu'elle démarre dans deux cas principaux : le premier c'est lorsque vous la branchez sur le port USB ou une sur autre source d'alimentation ; le deuxième c'est lorsque le compilateur a fini de charger le programme dans la carte, il la redémarre. Et au démarrage de la carte, il se passe des trucs.

3.1.4.1.1 Chargez ! Vous vous souvenez du chapitre où je vous présentais un peu le fonctionnement global de la carte ? Oui, **celui-là**. Je vous parlais alors de l'exécution du programme. Au démarrage, la carte (après un petit temps de vérification pour voir si le compilateur ne lui charge pas un nouveau programme) commence par aller charger les variables en mémoire de données. C'est un petit mécanisme électronique qui va simplement faire en sorte de copier les variables inscrites dans le programme vers la mémoire de données. En l'occurrence, dans le programme que l'on vient de créer, il n'y a qu'une variable et elle est constante en plus. Ce ne sera donc pas bien long à mettre ça en mémoire ! Ensuite, vient la lecture du programme. Et là, que peut-il bien se passer à l'intérieur du microcontrôleur ? En fait, ce n'est pas très compliqué (sur le principe :P).

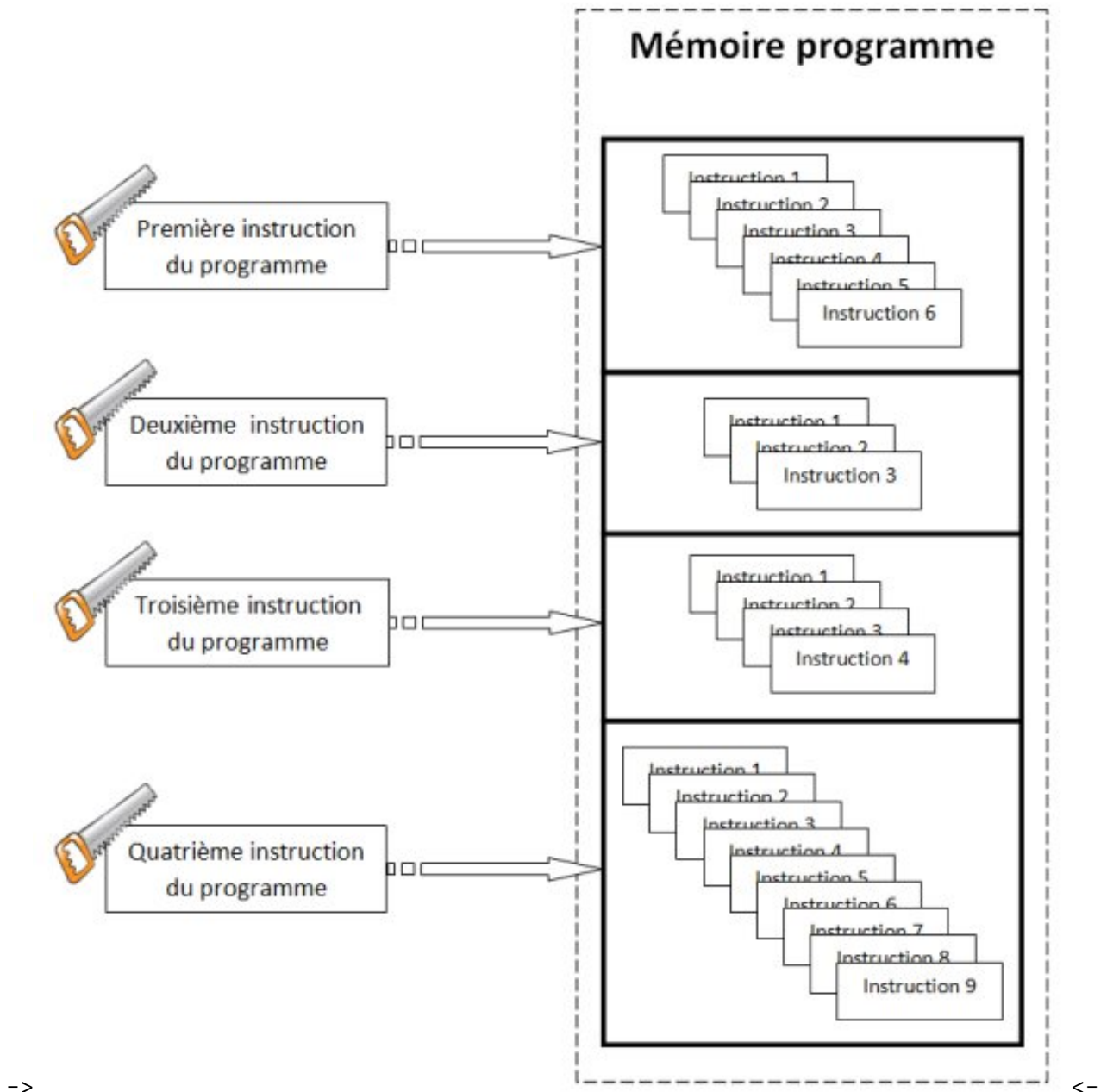
3.1.4.1.2 La vraie forme du programme À présent, le cerveau du microcontrôleur va aller lire la première instruction du programme, celle qui se trouve dans la fonction `set up()`. Sauf que, l'instruction n'est plus sous la même forme. Non, cette fois-ci je ne parle pas des 0 et des 1, mais bien d'une transformation de l'instruction. C'est le compilateur qui a découpé chaque instruction du programme en plusieurs petites instructions beaucoup plus simples.

[[q]] |Et pourquoi cela ? Le microcontrôleur ne sait pas faire une instruction aussi simple que de déclarer une broche en sortie ou allumer une LED ? o_o

Oui. C'est pourquoi il a besoin que le programme soit non plus sous forme de "grandes instructions" comme on l'a écrit, mais bien sous forme de plusieurs petites instructions. Et cela est dû au fait qu'il ne sait exécuter que des instructions très simples !



Bien entendu, il n'y a pas de limite à six instructions, il peut y en avoir beaucoup plus ou beaucoup moins ! Donc, en mémoire de programme, là où le programme de la carte est stocké, on va avoir plutôt quelque chose qui ressemble à ça :



Chaque grande instruction est découpée en petites instructions par le compilateur et est ensuite stockée dans la mémoire de programme. Pour être encore plus détaillé, chaque instruction agit sur un *registre*. Un registre, c'est la forme la plus simplifiée de la mémoire en terme de programmation. On en trouve plusieurs, par exemple le registre des timers ou celui des entrées/sorties du port A (ou B, ou C) ou encore des registres généraux pour manipuler les variables. Par exemple, pour additionner 3 à la variable 'a' le microcontrôleur fera les opérations suivantes :

1. chargement de la variable 'a' dans le registre général 8 (par exemple) depuis la RAM
2. chargement de la valeur 3 dans le registre général 9
3. mise du résultat de "registre 8 + registre 9" dans le registre 8
4. changement de la valeur de 'a' en RAM depuis le registre 8

3.1.4.2 Et l'exécution du programme

À présent que l'on a plein de petites instructions, qu'avons nous de plus ? Pas grand chose me direz-vous. Le Schmilblick n'a guère avancé... ^^ Pour comprendre, il faut savoir que le microcontrôleur ne sait faire que quelques instructions. Ces instructions sont encore plus simple que d'allumer une LED ! Il peut par exemple faire des opérations logiques (ET, OU, NON, décalage de bits, ...), des opérations numériques (addition et soustraction, les multiplications et divisions sont faites avec des opérations du types décalage de bits) ou encore copier et stocker des données. Il sait en faire, donc, mais pas tant que ça. Tout ce qu'il sait faire est régi par son **jeu d'instructions**. C'est-à-dire qu'il a une liste des instructions possibles qu'il sait exécuter et il s'y tient. Le compilateur doit donc absolument découper chaque instruction du programme en instructions que le microcontrôleur sait exécuter.

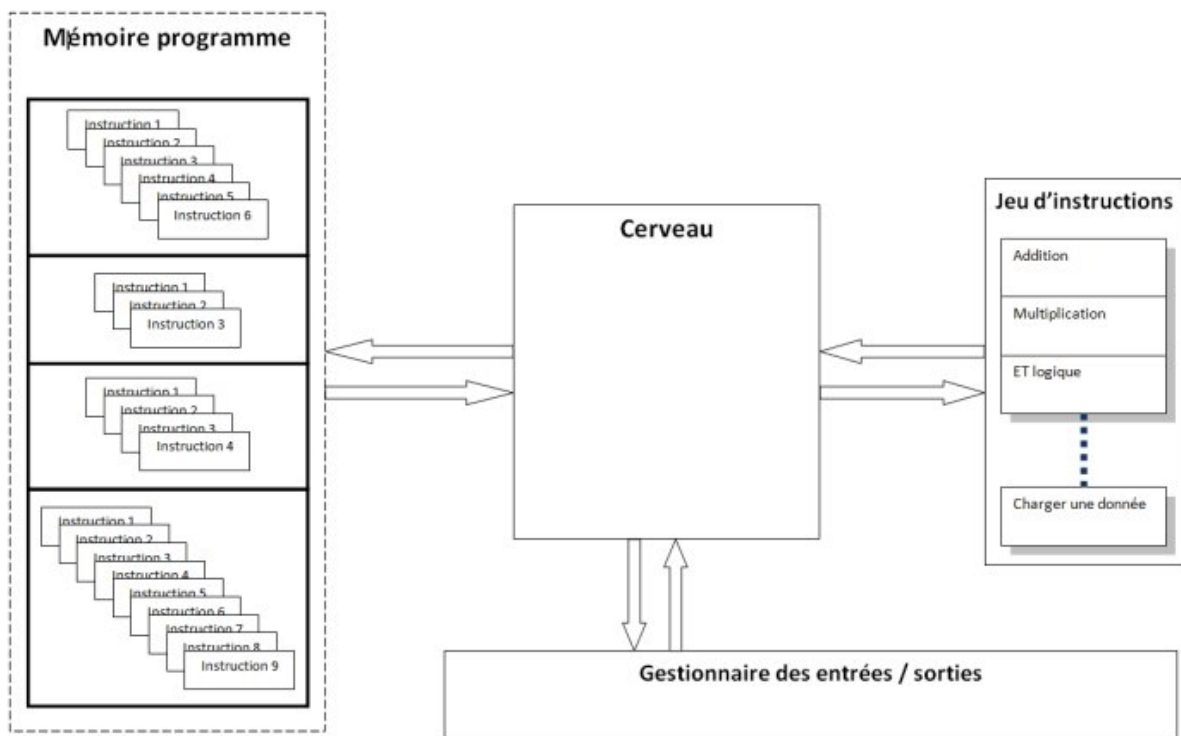


Figure 3.8 – Interaction entre les éléments du microcontrôleur

Le cerveau du microcontrôleur va aller lire le programme, il compare ensuite chaque instruction à son registre d'instructions et les exécute. Pour allumer une LED, il fera peut-être un ET logique, chargera une donnée, fera une soustraction, ... on ne sait pas mais il va y arriver. Et pour terminer, il communiquera à son gestionnaire d'entrées/sortie pour lui informer qu'il faut activer tel transistor interne pour mettre une tension sur telle broche de la carte pour ainsi allumer la LED qui y est connectée.

[[q]] |Waoou ! Et ça va vite tout ça ?

Extrêmement vite ! Cela dit, tout dépend de sa vitesse d'exécution...

3.1.4.3 La vitesse d'exécution

Le microcontrôleur est capable de faire un très grand nombre d'opérations par seconde. Ce nombre est défini par sa vitesse, entre autre. Sur la carte Arduino Duemilanove ou Uno, il y a un composant, que l'on appelle un **quartz**, qui va définir à quelle vitesse va aller le microcontrôleur. Ce quartz permet de **cadencer** le microcontrôleur. C'est en fait une **horloge** qui permet au microcontrôleur de se repérer. À chaque top de l'horloge, le microcontrôleur va faire quelque chose. Ce quelque chose peut, par exemple, être l'exécution d'une instruction, ou une lecture en mémoire. Cependant, chaque action ne dure pas qu'un seul top d'horloge. Suivant l'action réalisée, cela peut prendre plus ou moins de temps (de nombre de "coups d'horloge").

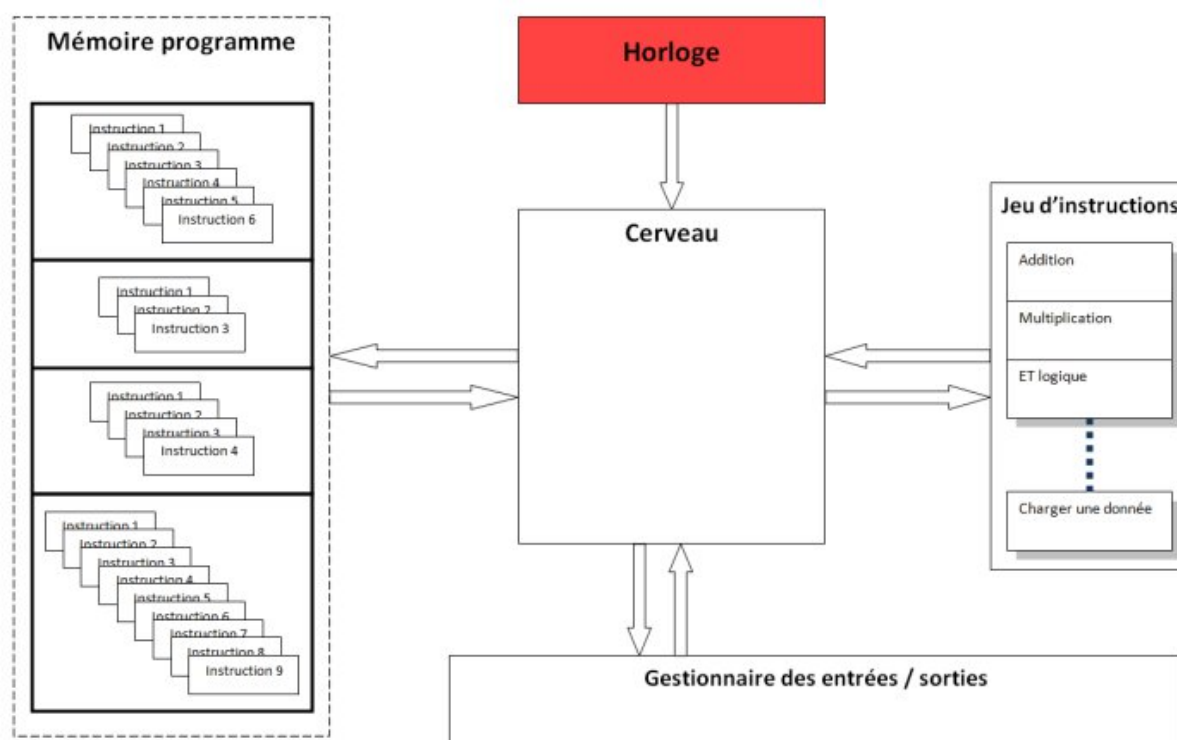


Figure 3.9 – L'horloge dans le système

La carte Arduino atteint au moins le million d'instructions par seconde ! Cela peut paraître énorme, mais comme je le disais, si il y a des instructions qui prennent beaucoup de temps, eh bien il se peut qu'elle n'exécute qu'une centaine d'instructions en une seconde. Tout dépend du temps pris par une instruction à être exécutée. Certaines opérations sont aussi parallélisées. Par exemple, le microcontrôleur peut faire une addition d'un côté pour une variable et en même temps il va mesurer le nombre de coups d'horloge pour faire s'incrémenter un compteur pour gérer un timer. Ces opérations sont **réellement** faites en parallèle, ce n'est pas un faux multi-tâche comme sur un ordinateur. Ici les deux registres travaillent en même temps. Le nombre de la fin ? 62.5 nanosecondes. C'est le temps qu'il faut au microcontrôleur d'Arduino pour faire une instruction la plus simple possible (en prenant en compte l'Arduino Uno et son quartz à 16MHz).

À présent, vous savez utiliser les sorties du micro-contrôleur, nous allons donc pouvoir passer aux choses sérieuses et faire clignoter notre LED !

3.2 Introduire le temps

C'est bien beau d'allumer une LED, mais si elle ne fait rien d'autre, ce n'est pas très utile. Autant la brancher directement sur une pile (avec une résistance tout de même ! :P). Alors voyons comment rendre intéressante cette LED en la faisant clignoter ! Ce que ne sait pas faire une pile... Pour cela il va nous falloir introduire la notion de temps. Eh bien devinez quoi ? Il existe une fonction toute prête là encore ! Je ne vous en dis pas plus, passons à la pratique !

3.2.1 Comment faire ?

3.2.1.0.1 Trouver la commande... Je vous laisse chercher un peu par vous-même, cela vous entrainera ! :pirate : ... Pour ceux qui ont fait l'effort de chercher et n'ont pas trouvé (à cause de l'anglais ?), je vous donne la fonction qui va bien : on va utiliser : `delay()`. Petite description de la fonction : elle va servir à mettre en pause le programme pendant un temps prédéterminé.

3.2.1.0.2 Utiliser la commande La fonction admet un paramètre qui est le temps pendant lequel on veut mettre en pause le programme. Ce temps doit être donné en millisecondes. C'est-à-dire que si vous voulez arrêter le programme pendant une seconde, il va falloir donner à la fonction ce même temps, écrit en millisecondes, soit 1000ms. La fonction est simple à utiliser :

```
// on fait une pause du programme pendant 1000ms, soit 1 seconde
delay(1000);
```

Rien de plus simple donc. Pour 20 secondes de pause, il aurait fallu écrire :

```
// on fait une pause du programme pendant 20000ms, soit 20 secondes
delay(20000);
```

3.2.1.0.3 Mettre en pratique : faire clignoter une LED Du coup, si on veut faire clignoter notre LED, on peut utiliser cette fonction. Voyons un peu le schéma de principe du clignotement d'une LED :

Vous le voyez, la LED s'allume. Puis, on fait intervenir la fonction `delay()`, qui va mettre le programme en pause pendant un certain temps. Ensuite, on éteint la LED. On met en pause le programme. Puis on revient au début du programme. On recommence et ainsi de suite. C'est cette suite de commandes qui forme le processus faisant clignoter la LED.

[[i]] | Dorénavant, prenez l'habitude de faire ce genre de schéma lorsque vous faites un programme. | Cela aide grandement la réflexion, croyez moi ! ;) C'est le principe de perdre du temps pour en gagner. Autrement dit : **l'organisation** !

Maintenant, il faut que l'on traduise ce schéma, portant le nom d'**organigramme**, en code.

Il suffit pour cela de remplacer les phrases dans chaque cadre par une ligne de code. Par exemple, "on allume la LED", va être traduit par l'instruction que l'on a vue dans le chapitre précédent :

```
digitalWrite(led_rouge, LOW); // allume la LED
```

Ensuite, on traduit le cadre suivant, ce qui donne :

```
// fait une pause de 1 seconde (= 1000ms)
delay(1000);
```

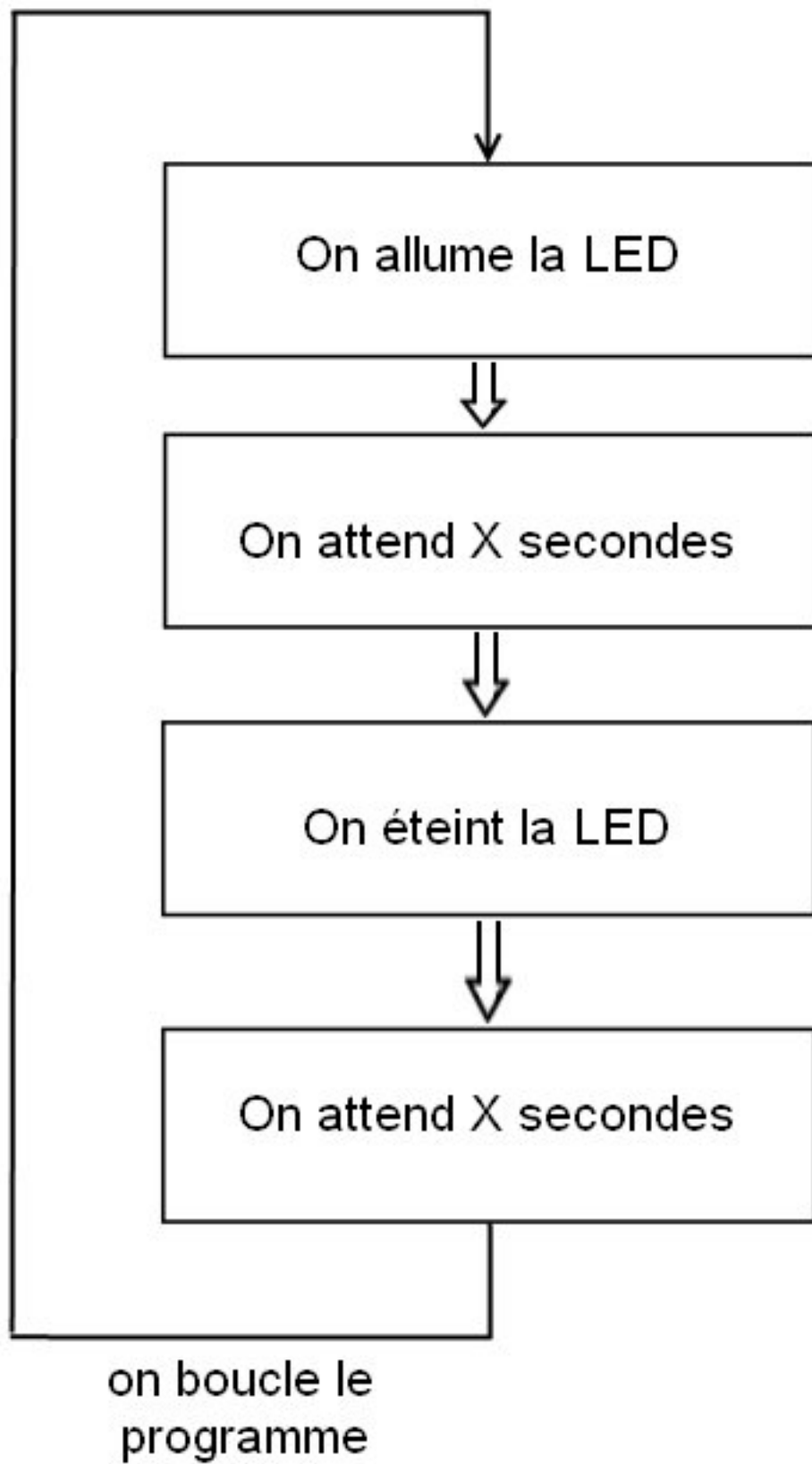


Figure 3.10 – Schéma de principe du clignotement

Puis, on traduit la ligne suivante :

```
// éteint la LED
digitalWrite(led_rouge, HIGH);
```

Enfin, la dernière ligne est identique à la deuxième, soit :

```
// fait une pause de 1 seconde
delay(1000);
```

On se retrouve avec le code suivant :

```
// allume la LED
digitalWrite(led_rouge, LOW);
// fait une pause de 1 seconde
delay(1000);
// éteint la LED
digitalWrite(led_rouge, HIGH);
// fait une pause de 1 seconde
delay(1000);
```

La fonction qui va boucler à l'infini le code précédent est la fonction `loop()`. On inscrit donc le code précédent dans cette fonction :

```
void loop()
{
    // allume la LED
    digitalWrite(led_rouge, LOW);
    // fait une pause de 1 seconde
    delay(1000);
    // éteint la LED
    digitalWrite(led_rouge, HIGH);
    // fait une pause de 1 seconde
    delay(1000);
}
```

Et on n'oublie pas de définir la broche utilisée par la LED, ainsi que d'initialiser cette broche en tant que sortie. Cette fois, le code est terminé !

```
// définition de la broche 2 de la carte en tant que variable
const int led_rouge = 2;

// fonction d'initialisation de la carte
void setup()
{
    // initialisation de la broche 2 comme étant une sortie
    pinMode(led_rouge, OUTPUT);
}

void loop()
{
    // allume la LED
```

```
digitalWrite(led_rouge, LOW);  
// fait une pause de 1 seconde  
delay(1000);  
// éteint la LED  
digitalWrite(led_rouge, HIGH);  
// fait une pause de 1 seconde  
delay(1000);  
}
```

Vous n'avez plus qu'à charger le code dans la carte et admirer ~~mon~~ votre travail ! La LED clignote ! Libre à vous de changer le temps de clignotement : vous pouvez par exemple éteindre la LED pendant 40ms et l'allumer pendant 600ms :

```
[[secret]]|cpp | // définition de la broche 2 de la carte en tant que variable  
| const int led_rouge = 2; | // fonction d'initialisation de la carte |  
void setup() | { | // initialisation de la broche 2 comme étant une sortie  
| pinMode(led_rouge, OUTPUT); | } | void loop() | { | // allume la LED |  
digitalWrite(led_rouge, LOW); | // fait une pause de 600 ms | delay(600);  
| // éteint la LED | digitalWrite(led_rouge, HIGH); | // fait une pause de  
40 ms | delay(40); | } |
```

Et hop, une petite vidéo d'illustration !

->!(<https://www.youtube.com/watch?v=YAOakcEoIfk>)<-

3.2.2 Faire clignoter un groupe de LED

Vous avouerez facilement que ce n'était pas bien difficile d'arriver jusque-là. Alors, à présent, accentuons la difficulté. Notre but : faire clignoter *un groupe* de LED.

3.2.2.0.1 Le matériel et les schémas Ce groupe de LED sera composé de six LED, nommées L1, L2, L3, L4, L5 et L6. Vous aurez par conséquent besoin d'un nombre identique de résistances. Le schéma de la réalisation :

3.2.2.0.2 Le programme Le programme est un peu plus long que le précédent, car il ne s'agit plus d'allumer une seule LED, mais six ! Voilà l'organigramme que va suivre notre programme :

Cet organigramme n'est pas très beau, mais il a le mérite d'être assez lisible. Nous allons essayer de le suivre pour créer notre programme. Traduction des six premières instructions :

```
digitalWrite(L1, LOW); // notez que le nom de la broche a changé  
digitalWrite(L2, LOW); // et ce pour toutes les LED connectées  
digitalWrite(L3, LOW); // au micro-contrôleur  
digitalWrite(L4, LOW);  
digitalWrite(L5, LOW);  
digitalWrite(L6, LOW);
```

Ensuite, on attend 1,5 seconde :

```
delay(1500);
```

Puis on traduit les six autres instructions :

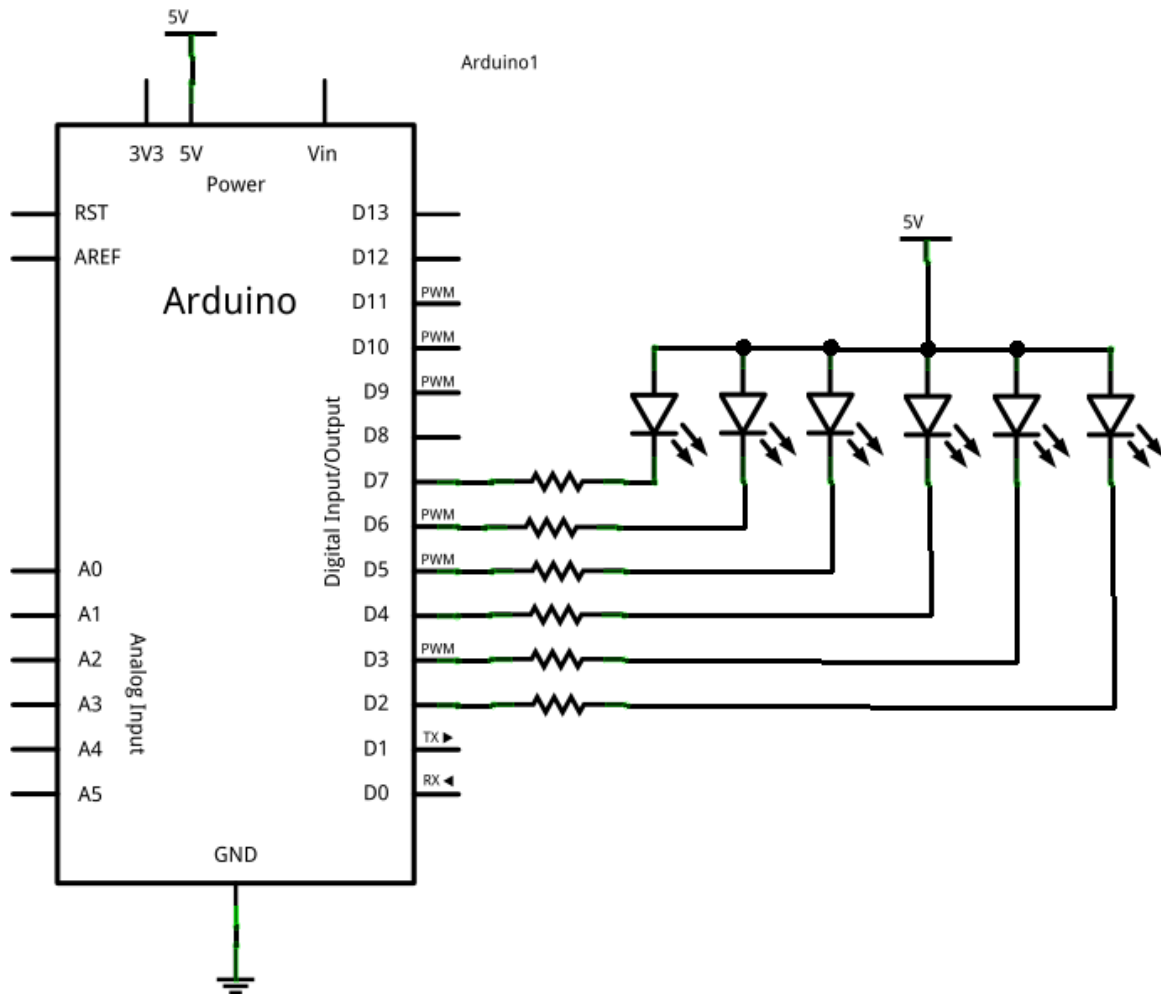


Figure 3.11 – Schéma avec 6 leds

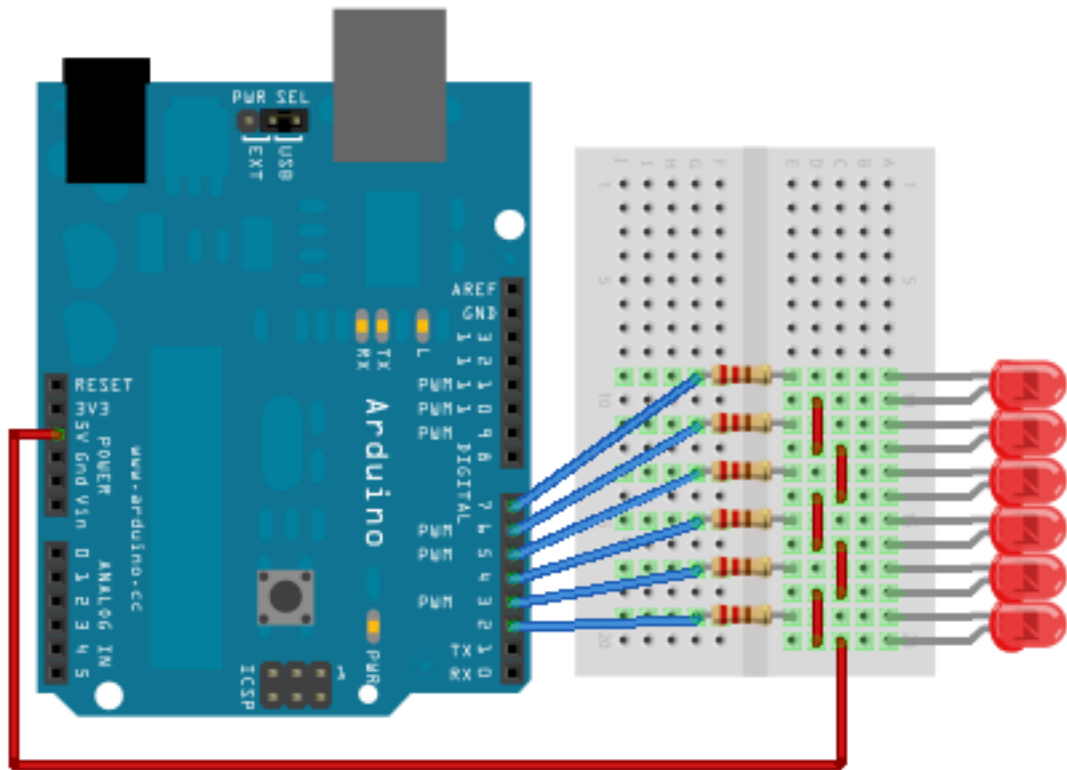


Figure 3.12 – Montage avec 6 leds

```
digitalWrite(L1, HIGH); // on éteint les LED
digitalWrite(L2, HIGH);
digitalWrite(L3, HIGH);
digitalWrite(L4, HIGH);
digitalWrite(L5, HIGH);
digitalWrite(L6, HIGH);
```

Enfin, la dernière ligne de code, disons que nous attendrons 4,32 secondes :

```
delay(4320);
```

Tous ces bouts de code sont à mettre à la suite et dans la fonction loop() pour qu'ils se répètent.

```
void loop()
{
    digitalWrite(L1, LOW); // allumer les LED
    digitalWrite(L2, LOW);
    digitalWrite(L3, LOW);
    digitalWrite(L4, LOW);
    digitalWrite(L5, LOW);
    digitalWrite(L6, LOW);

    delay(1500);           // attente du programme de 1,5 secondes

    digitalWrite(L1, HIGH); // on éteint les LED
```

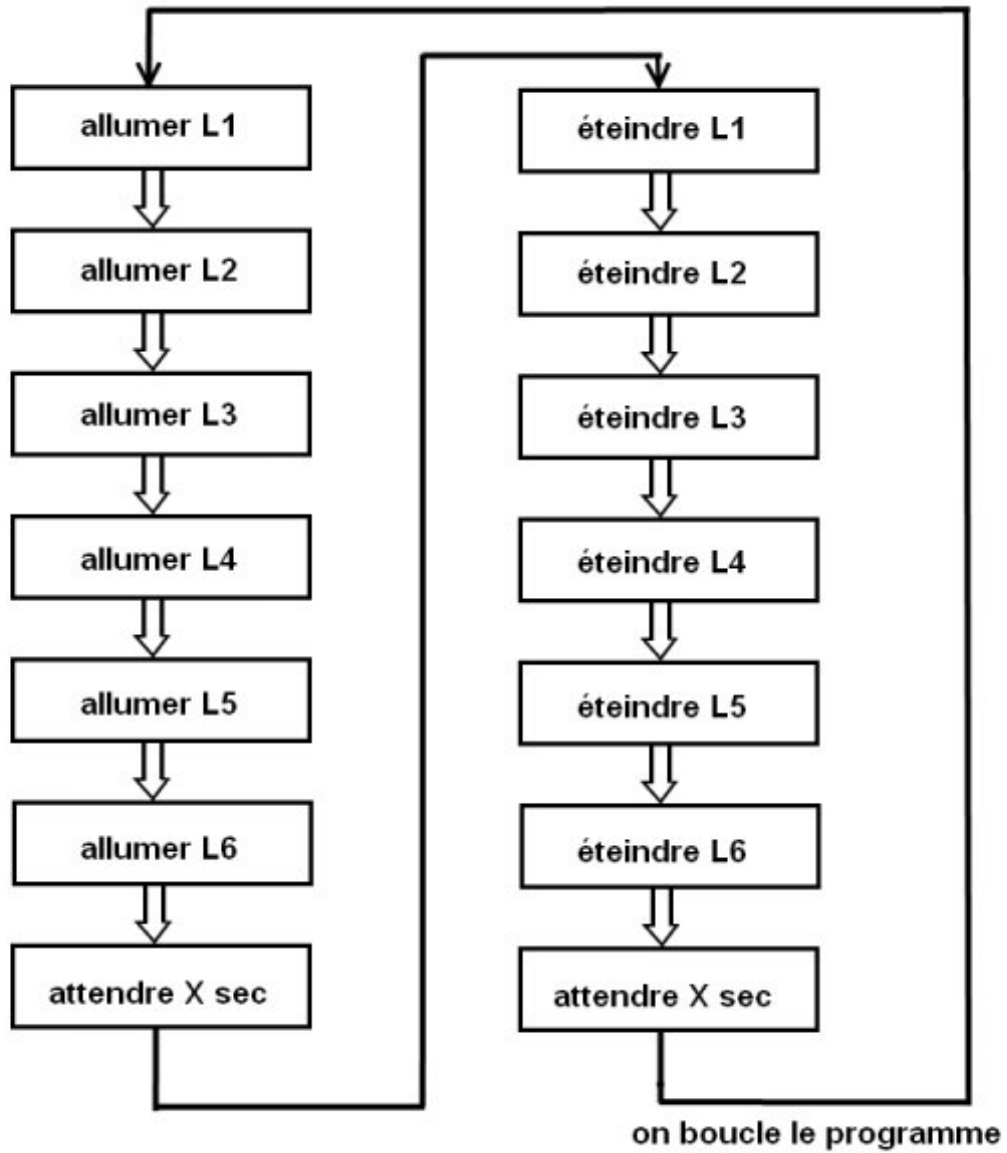



Figure 3.13 – Description du programme

```
    digitalWrite(L2, HIGH);
    digitalWrite(L3, HIGH);
    digitalWrite(L4, HIGH);
    digitalWrite(L5, HIGH);
    digitalWrite(L6, HIGH);

    delay(4320);           // attente du programme de 4,32 secondes
}
```

Code : la boucle complète

Je l'ai mentionné dans un de mes commentaires entre les lignes du programme, les noms attribués aux broches sont à changer. En effet, car si on définit des noms de variables identiques, le compilateur n'aimera pas ça et vous affichera une erreur. En plus, le micro-contrôleur ne pourrait pas exécuter le programme car il ne saurait pas quelle broche mettre à l'état HAUT ou BAS. Pour définir les broches, on fait la même chose qu'à notre premier programme :

```
const int L1 = 2; // broche 2 du micro-contrôleur se nomme maintenant : L1
const int L2 = 3; // broche 3 du micro-contrôleur se nomme maintenant : L2
const int L3 = 4; // ...
const int L4 = 5;
const int L5 = 6;
const int L6 = 7;
```

Code : Définition des broches

Maintenant que les broches utilisées sont définies, il faut dire si ce sont des entrées ou des sorties :

```
pinMode(L1, OUTPUT); // L1 est une broche de sortie
pinMode(L2, OUTPUT); // L2 est une broche de sortie
pinMode(L3, OUTPUT); // ...
pinMode(L4, OUTPUT);
pinMode(L5, OUTPUT);
pinMode(L6, OUTPUT);
```

3.2.2.0.3 Le programme final Il n'est certes pas très beau, mais il fonctionne :

```
const int L1 = 2; // broche 2 du micro-contrôleur se nomme maintenant : L1
const int L2 = 3; // broche 3 du micro-contrôleur se nomme maintenant : L2
const int L3 = 4; // ...
const int L4 = 5;
const int L5 = 6;
const int L6 = 7;

void setup()
{
    pinMode(L1, OUTPUT); // L1 est une broche de sortie
    pinMode(L2, OUTPUT); // L2 est une broche de sortie
    pinMode(L3, OUTPUT); // ...
    pinMode(L4, OUTPUT);
    pinMode(L5, OUTPUT);
}
```

```

    pinMode(L6, OUTPUT);
}

void loop()
{
    // allumer les LED
    digitalWrite(L1, LOW);
    digitalWrite(L2, LOW);
    digitalWrite(L3, LOW);
    digitalWrite(L4, LOW);
    digitalWrite(L5, LOW);
    digitalWrite(L6, LOW);

    // attente du programme de 1,5 secondes
    delay(1500);

    // on éteint les LED
    digitalWrite(L1, HIGH);
    digitalWrite(L2, HIGH);
    digitalWrite(L3, HIGH);
    digitalWrite(L4, HIGH);
    digitalWrite(L5, HIGH);
    digitalWrite(L6, HIGH);

    // attente du programme de 4,32 secondes
    delay(4320);
}

```

Code : Allumage puis extinction en boucle d'un groupe de leds

Voilà, vous avez en votre possession un magnifique clignotant, que vous pouvez attacher à votre vélo! :P

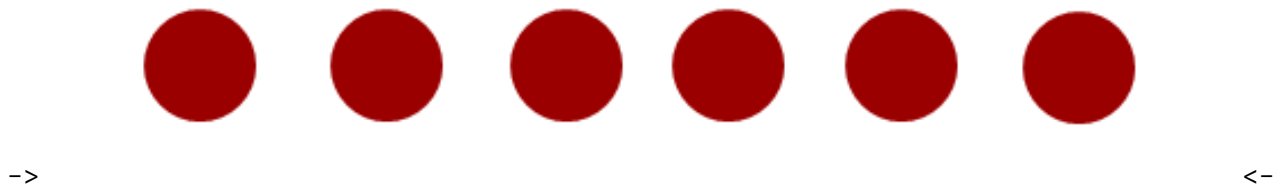
[q] |Une question me chiffonne. Doit-on toujours écrire l'état d'une sortie, ou peut-on faire plus simple?

C'est là un un point intéressant. Si je comprends bien, vous vous demandez comment faire pour remplacer l'intérieur de la fonction `loop()`? C'est vrai que c'est très lourd à écrire et à lire! Il faut en effet s'occuper de définir l'état de chaque LED. C'est rébarbatif, surtout si vous en aviez mis autant qu'il y a de broches disponibles sur la carte! Il y a une solution pour faire ce que vous dites. Nous allons la voir dans quelques chapitres, ne soyez pas impatient!;) En attendant, voici une vidéo d'illustration du clignotement :

->!(<https://www.youtube.com/watch?v=2SgxKU67mn8>)<-

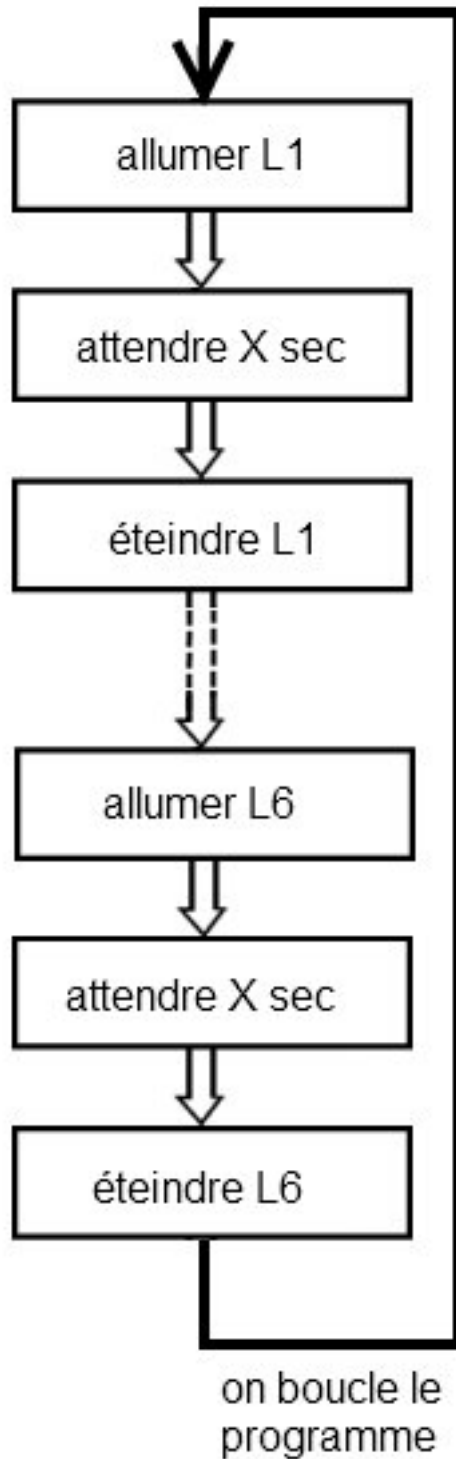
3.2.3 Réaliser un chenillard

3.2.3.0.1 Le but du programme Le but du programme que nous allons créer va consister à réaliser un chenillard. Pour ceux qui ne savent pas ce qu'est un chenillard, je vous ai préparé une petite image .gif animée :



Comme on dit souvent, une image vaut mieux qu'un long discours! :P Voilà donc ce qu'est un chenillard. Chaque LED s'allume alternativement et dans l'ordre. De la gauche vers la droite ou l'inverse, c'est au choix.

3.2.3.0.2 Organigramme Comme j'en ai marre de faire des dessins avec paint.net, je vous laisse réfléchir tout seuls comme des grands à l'organigramme du programme. ... Bon, aller, le voilà cet organigramme! Attention, il n'est pas complet, mais si vous avez compris le principe, le compléter ne vous posera pas de problèmes :



[[secret]] |->

<-

À vous de jouer!

3.2.3.0.3 Le programme Normalement, sa conception ne devrait pas vous poser de problèmes. Il suffit en effet de récupérer le code du programme précédent (“allumer un groupe de LED”) et de le modifier en fonction de notre besoin. Ce code, je vous le donne, avec les commentaires qui vont bien :

3 Gestion des entrées / sorties

```
const int L1 = 2; // broche 2 du micro-contrôleur se nomme maintenant : L1
const int L2 = 3; // broche 3 du micro-contrôleur se nomme maintenant : L2
const int L3 = 4; // ...
const int L4 = 5;
const int L5 = 6;
const int L6 = 7;

void setup()
{
  pinMode(L1, OUTPUT); // L1 est une broche de sortie
  pinMode(L2, OUTPUT); // L2 est une broche de sortie
  pinMode(L3, OUTPUT); // ...
  pinMode(L4, OUTPUT);
  pinMode(L5, OUTPUT);
  pinMode(L6, OUTPUT);
}

// on change simplement l'intérieur de la boucle pour atteindre notre objectif

void loop() // la fonction loop() exécute le code qui suit en le répétant en boucle
{
  digitalWrite(L1, LOW); // allumer L1
  delay(1000);           // attendre 1 seconde
  digitalWrite(L1, HIGH); // on éteint L1
  digitalWrite(L2, LOW); // on allume L2 en même temps que l'on éteint L1
  delay(1000);           // on attend 1 seconde
  digitalWrite(L2, HIGH); // on éteint L2 et
  digitalWrite(L3, LOW); // on allume immédiatement L3
  delay(1000);           // ...
  digitalWrite(L3, HIGH);
  digitalWrite(L4, LOW);
  delay(1000);
  digitalWrite(L4, HIGH);
  digitalWrite(L5, LOW);
  delay(1000);
  digitalWrite(L5, HIGH);
  digitalWrite(L6, LOW);
  delay(1000);
  digitalWrite(L6, HIGH);
}
```

Code : Votre premier chenillard

Vous le voyez, ce code est très lourd et n'est pas pratique. Nous verrons plus loin comment faire en sorte de l'alléger. Mais avant cela, un TP arrive... Au fait, voici un exemple de ce que vous pouvez obtenir !

->!(<https://www.youtube.com/watch?v=hKNqRAi-kKI>)<-

3.2.4 Fonction millis()

Nous allons terminer ce chapitre par un point qui peut être utile, notamment dans certaines situations où l'on ne veut pas arrêter le programme. En effet, si on veut faire clignoter une LED sans arrêter l'exécution du programme, on ne peut pas utiliser la fonction de `delay()` qui met en pause le programme durant le temps défini.

3.2.4.1 Les limites de la fonction delay()

Vous avez probablement remarqué, lorsque vous utilisez la fonction `delay()` tout notre programme s'arrête le temps d'attendre. Dans certains cas ce n'est pas un problème mais dans certains cas ça peut être plus gênant. Imaginons, vous êtes en train de faire avancer un robot. Vous mettez vos moteurs à une vitesse moyenne, tranquille, jusqu'à ce qu'un petit bouton sur l'avant soit appuyé (il clic lorsqu'on touche un mur par exemple). Pendant ce temps-là, vous décidez de faire des signaux en faisant clignoter vos LED. Pour faire un joli clignotement, vous allumez une LED rouge pendant une seconde puis l'éteignez pendant une autre seconde. Voilà par exemple ce qu'on pourrait faire comme code

```
void setup()
{
    pinMode(moteur, OUTPUT);
    pinMode(led, OUTPUT);
    pinMode(bouton, INPUT);
    // on met le moteur en marche (en admettant qu'il soit en marche à HIGH)
    digitalWrite(moteur, HIGH);
    // on allume la LED
    digitalWrite(led, LOW);
}

void loop()
{
    // si le bouton est cliqué (on rentre dans un mur)
    if(digitalRead(bouton)==HIGH)
    {
        // on arrête le moteur
        digitalWrite(moteur, LOW);
    }
    else // sinon on clignote
    {
        digitalWrite(led, HIGH);
        delay(1000);
        digitalWrite(led, LOW);
        delay(1000);
    }
}
```

[[a]] |Attention ce code n'est pas du tout rigoureux voire faux dans son écriture, il sert juste à comprendre le principe!

Maintenant imaginez. Vous roulez, tester que le bouton n'est pas appuyé, donc faites clignoter les LED (cas du else). Le temps que vous fassiez l'affichage en entier s'écoule 2 longues secondes ! Le robot a pu pendant cette éternité se prendre le mur en pleine poire et les moteurs continuent à avancer tête baissée jusqu'à fumer ! Ce n'est pas bon du tout ! Voici pourquoi la fonction millis() peut nous sauver.

3.2.4.2 Découvrons et utilisons millis()

Tout d'abord, quelques précisions à son sujet, avant d'aller s'en servir. À l'intérieur du cœur de la carte Arduino se trouve un chronomètre. Ce chrono mesure l'écoulement du temps depuis le lancement de l'application. Sa granularité (la précision de son temps) est la milliseconde. La fonction millis() nous sert à savoir quelle est la valeur courante de ce compteur. Attention, comme ce compteur est capable de mesurer une durée allant jusqu'à 50 jours, la valeur retournée doit être stockée dans une variable de type "long".

[[q]] |C'est bien gentil mais concrètement on l'utilise comment ?

Eh bien c'est très simple. On sait maintenant "lire l'heure". Maintenant, au lieu de dire "allume-toi pendant une seconde et ne fais surtout rien pendant ce temps", on va faire un truc du genre "Allume-toi, fais tes petites affaires, vérifie l'heure de temps en temps et si une seconde est écoulée, alors réagis!". Voici le code précédent transformé selon la nouvelle philosophie :

```
long temps; // variable qui stocke la mesure du temps
boolean etat_led;
```

```
void setup()
{
    pinMode(moteur, OUTPUT);
    pinMode(led, OUTPUT);
    pinMode(bouton, INPUT);
    // on met le moteur en marche
    digitalWrite(moteur, HIGH);
    // par défaut la LED sera éteinte
    etat_led = 0;
    // on éteint la LED
    digitalWrite(led, etat_led);

    // on initialise le temps
    temps = millis();
}
```

```
void loop()
{
    // si le bouton est cliqué (on rentre dans un mur)
    if(digitalRead(bouton)==HIGH)
    {
        // on arrête le moteur
        digitalWrite(moteur, LOW);
    }
    else // sinon on clignote
```



```

{
  // on compare l'ancienne valeur du temps et la valeur sauvee
  // si la comparaison (l'un moins l'autre) dépasse 1000...
  // ...cela signifie qu'au moins une seconde s'est écoulée
  if((millis() - temps) > 1000)
  {
    etat_led = !etat_led; // on inverse l'état de la LED
    digitalWrite(led, etat_led); // on allume ou éteint
    temps = millis(); // on stocke la nouvelle heure
  }
}
}

```

Code : Clignotement avec `millis`

Et voilà, grâce à cette astuce plus de fonction bloquante. L'état du bouton est vérifié très fréquemment ce qui permet de s'assurer que si jamais on rentre dans un mur, on coupe les moteurs très vite. Dans ce code, tout s'effectue de manière fréquente. En effet, on ne reste jamais bloqué à attendre que le temps passe. À la place, on avance dans le programme et teste souvent la valeur du chronomètre. Si cette valeur est de 1000 itérations supérieures à la dernière valeur mesurée, alors cela signifie qu'une seconde est passée.

[[a]] |Attention, au `if` de la ligne 25 ne faites surtout pas `millis() - temps == 1000`. |Cela signifierait que vous voulez vérifier que 1000 millisecondes EXACTEMENT se sont écoulées, ce qui est très peu probable (vous pourrez plus probablement mesurer plus ou moins mais rarement exactement)

Maintenant que vous savez maîtriser le temps, vos programmes/animations vont pouvoir posséder un peu plus de "vie" en faisant des pauses, des motifs, etc. Impressionnez-moi!

3.3 TP Feux de signalisation routière

Vous voilà arrivé pour votre premier TP, que vous ferez seul ! :twisted : Je vous aiderai quand même un peu. Le but de ce TP va être de réaliser un feu de signalisation routière. Je vous donne en détail tout ce qu'il vous faut pour mener à bien cet objectif.

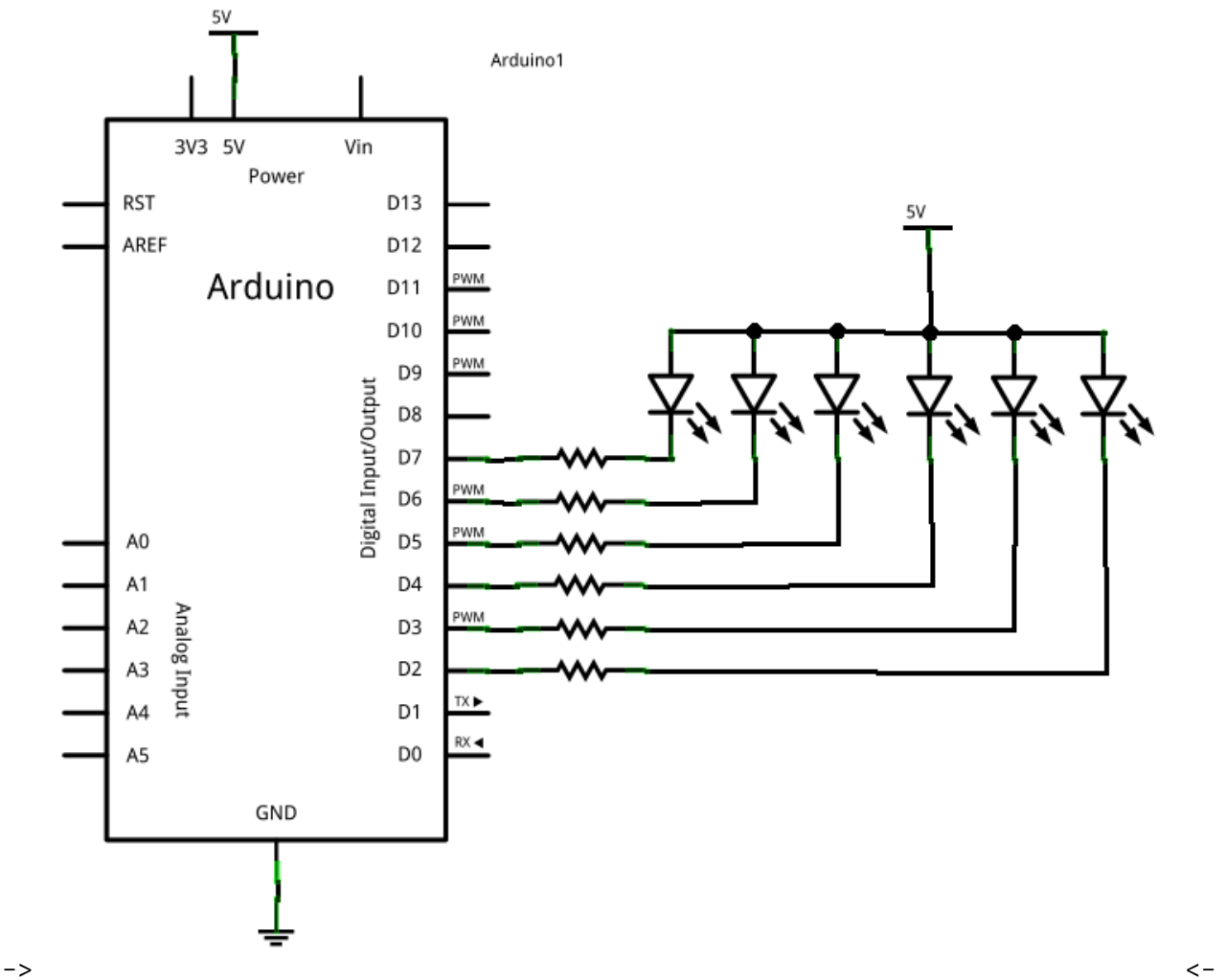
3.3.1 Préparation

Ce dont nous avons besoin pour réaliser ces feux.

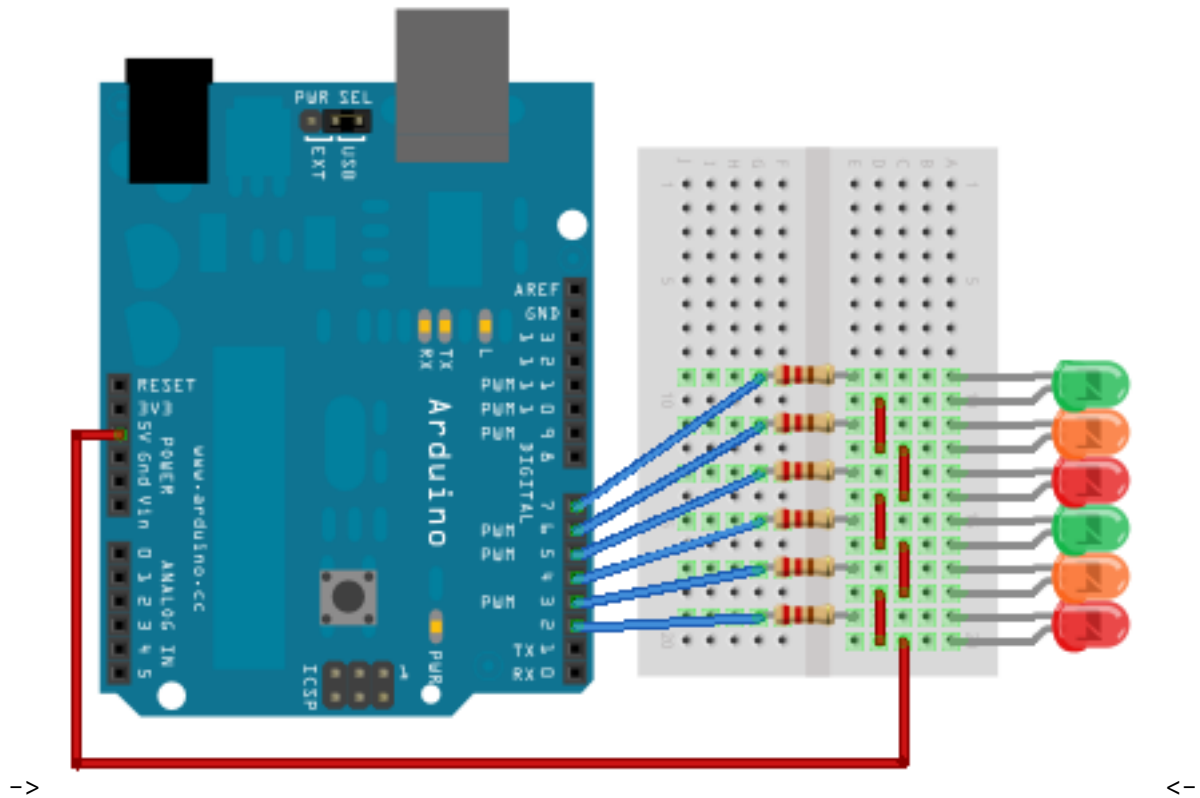
3.3.1.0.1 Le matériel Le matériel est la base de notre besoin. On a déjà utilisé 6 LED et résistances, mais elles étaient pour moi en l'occurrence toutes rouges. Pour faire un feu routier, il va nous falloir 6 LED, mais dont les couleurs ne sont plus les mêmes.

- LED : un nombre de 6, dont 2 **rouges**, 2 **jaune** (ou **orange**) et 2 **vertes** ;
- Résistors : 6 également, de la même valeur que ceux que vous avez utilisés.
- Arduino : une carte Arduino évidemment !

3.3.1.0.2 Le schéma C'est le même que pour le montage précédent, seul la couleur des LED change, comme ceci :



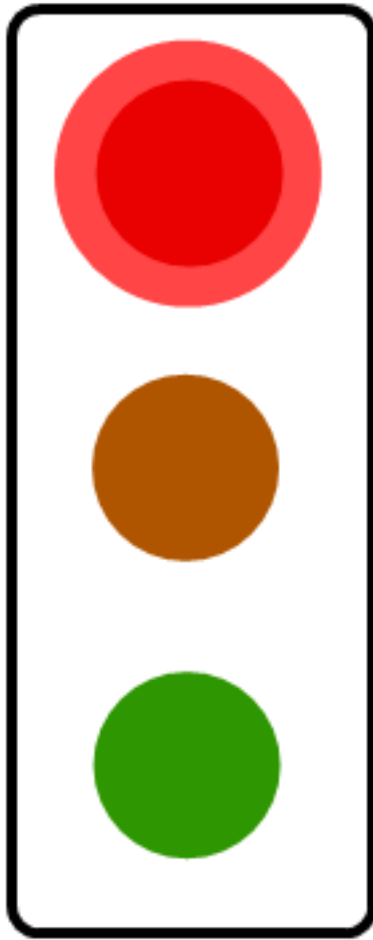
Vous n'avez donc plus qu'à reprendre le dernier montage et changer la couleur de 4 LED, pour obtenir ceci :



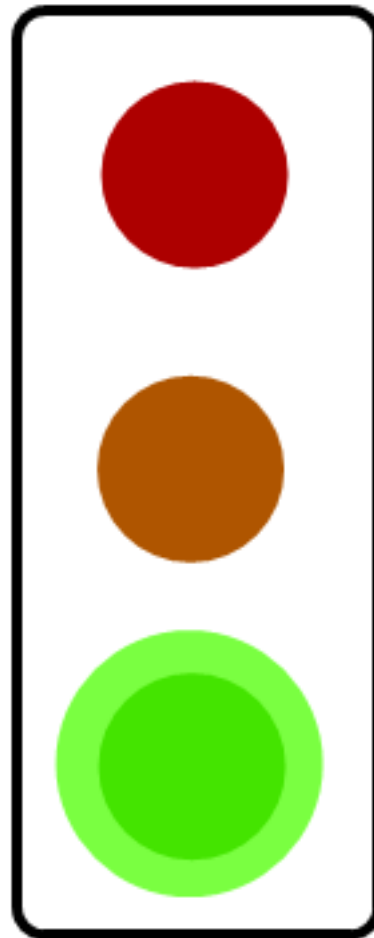
[[e]] |N'oubliez pas de tester votre matériel en chargeant un programme qui fonctionne! |Cela évite de s'acharner à faire un nouveau programme qui ne fonctionne pas à cause d'un matériel défectueux. On est jamais sûr de rien, croyez-moi! ;)

3.3.2 Énoncé de l'exercice

3.3.2.0.1 Le but Je l'ai dit, c'est de réaliser des feux de signalisation. Alors, vu le nombre de LED, vous vous doutez bien qu'il faut réaliser 2 feux. Ces feux devront être synchronisés. Là encore, je vous ai préparé une belle image animée :



feux 1



feux 2

->

<-

3.3.2.0.2 Le temps de la séquence Vous allez mettre un délai de 3 secondes entre le feu vert et le feu orange. Un délai de 1 seconde entre le feu orange et le feu rouge. Et un délai de 3 secondes entre le feu rouge et le feu vert.

3.3.2.0.3 Par où commencer ? D'abord, vous devez faire l'organigramme. Oui je ne vous le donne pas ! Ensuite, vous commencez un nouveau programme. Dans ce programme, vous devez définir quelles sont les broches du micro-contrôleur que vous utilisez. Puis définir si ce sont des entrées, des sorties, ou s'il y a des deux. Pour terminer, vous allez faire le programme complet dans la fonction qui réalise une boucle.

3.3.2.0.4 C'est parti ! Allez, c'est parti ! A vous de m'épater. :P Vous avez théoriquement toutes les bases nécessaires pour réaliser ce TP. En plus on a presque déjà tout fait. Mince, j'en ai trop dit... Pendant ce temps, moi je vais me faire une raclette. ^^ Et voici un résultat possible :

->!(<https://www.youtube.com/watch?v=ByPDSyHhf-g>)<-

3.3.3 Correction !

3.3.3.0.1 Fini ! Vous avez fini ? Votre code ne fonctionne pas, mais vous avez eu beau chercher pourquoi, vous n'avez pas trouvé ?

Très bien. Dans ce cas, vous pouvez lire la correction.

Ceux qui n'ont pas cherché ne sont pas les bienvenus ici ! :diable :

3.3.3.0.2 L'organigramme Cette fois, l'organigramme a changé de forme, c'est une liste. Comment le lire ? De haut en bas ! Le premier élément du programme commence après le début, le deuxième élément, après le premier, etc.

```

— DEBUT
— /* première partie du programme, on s'occupe principalement du deuxième
   feu */
— Allumer led_rouge_feux_1
— Allumer led_verte_feux_2
— Attendre 3 secondes
— Éteindre led_verte_feux_2
— Allumer led_jaune_feux_2
— Attendre 1 seconde
— Éteindre led_jaune_feux_2
— Allumer led_rouge_feux_2
— /* deuxième partie du programme, pour l'instant : led_rouge_feux_1
   et led_rouge_feux_2 sont allumées; on éteint donc la led_rouge_feux_1
   pour allumer la led_verte_feux_1 */
— Attendre 3 secondes
— Éteindre led_rouge_feux_1
— Allumer led_verte_feux_1
— Attendre 3 secondes
— Éteindre led_verte_feux_1
— Allumer led_jaune_feux_1
— Attendre 1 seconde
— Éteindre led_jaune_feux_1
— Allumer led_rouge_feux_1
— FIN

```

Voilà donc ce qu'il faut suivre pour faire le programme. Si vous avez trouvé comme ceci, c'est très bien, sinon il faut s'entraîner car c'est très important d'organiser son code et en plus cela permet d'éviter certaines erreurs !

3.3.3.1 La correction, enfin !

Voilà le moment que vous attendez tous : la correction ! Alors, je prévois tout de suite, le code que je vais vous montrer n'est pas absolu, on peut le faire de différentes manières

3.3.3.1.1 La fonction setup Normalement ici aucune difficulté, on va nommer les broches, puis les placer en sortie et les mettre dans leur état de départ.

```
[[secret]] | cpp | // définition des broches | const int led_rouge_feux_1 =  
2; | const int led_jaune_feux_1 = 3; | const int led_verte_feux_1 = 4; |  
const int led_rouge_feux_2 = 5; | const int led_jaune_feux_2 = 6; | const  
int led_verte_feux_2 = 7; | | void setup() | { | // initialisation en sortie  
de toutes les broches | pinMode(led_rouge_feux_1, OUTPUT); | pinMode(led_jaune_feux_1,  
OUTPUT); | pinMode(led_verte_feux_1, OUTPUT); | pinMode(led_rouge_feux_2,  
OUTPUT); | pinMode(led_jaune_feux_2, OUTPUT); | pinMode(led_verte_feux_2,  
OUTPUT); | | // on initialise toutes les LED éteintes au début du programme  
| // (sauf les deux feux rouges) | digitalWrite(led_rouge_feux_1, LOW);  
| digitalWrite(led_jaune_feux_1, HIGH); | digitalWrite(led_verte_feux_1,  
HIGH); | digitalWrite(led_rouge_feux_2, LOW); | digitalWrite(led_jaune_feux_2,  
HIGH); | digitalWrite(led_verte_feux_2, HIGH); | } | | Code : La fonction setup
```

Vous remarquerez l'utilité d'avoir des variables bien nommées.

3.3.3.1.2 Le code principal Si vous êtes bien organisé, vous ne devriez pas avoir de problème ici non plus ! Point trop de paroles, la solution arrive

```
[[secret]]|cpp | void loop() | { | // première séquence | digitalWrite(led_rouge_feux_1,  
HIGH); | digitalWrite(led_verte_feux_1, LOW); | | delay(3000); | | // deuxième  
séquence | digitalWrite(led_verte_feux_1, HIGH); | digitalWrite(led_jaune_feux_1,  
LOW); | | delay(1000); | | // troisième séquence | digitalWrite(led_jaune_feux_1,  
HIGH); | digitalWrite(led_rouge_feux_1, LOW); | | delay(1000); | | /* deuxième  
partie du programme, on s'occupe du feux numéro 2 */ | | // première séquence  
| digitalWrite(led_rouge_feux_2, HIGH); | digitalWrite(led_verte_feux_2,  
LOW); | | delay(3000); | | // deuxième séquence | digitalWrite(led_verte_feux_2,  
HIGH); | digitalWrite(led_jaune_feux_2, LOW); | | delay(1000); | | // deuxième  
séquence | digitalWrite(led_jaune_feux_2, HIGH); | digitalWrite(led_rouge_feux_2,  
LOW); | | delay(1000); | | /* le programme va reboucler et revenir au début  
*/ | } | | Code : La fonction principale
```

Si ça marche, tant mieux, sinon référez vous à la résolution des problèmes en annexe du cours. Ce TP est donc terminé, vous pouvez modifier le code pour par exemple changer les temps entre chaque séquence, ou bien même modifier les séquences elles-mêmes, ...

Bon, c'était un TP gentillet. L'intérêt est seulement de vous faire pratiquer pour vous "enfoncer dans le crâne" ce que l'on a vu jusqu'à présent.

3.4 Un simple bouton

À la fin de ce chapitre, vous serez capable d'utiliser des boutons ou des interrupteurs pour inter-agir de manière simple avec votre programme.

3.4.1 Qu'est-ce qu'un bouton ?

Derrière ce titre trivial se cache un composant de base très utile, possédant de nombreux détails que vous ignorez peut-être. Commençons donc dès maintenant l'autopsie de ce dernier.

3.4.1.1 Mécanique du bouton

Vous le savez sûrement déjà, un bouton n'est jamais qu'un fil qui est connecté ou non selon sa position. En pratique, on en repère plusieurs, qui diffèrent selon leur taille, leurs caractéristiques électriques, les positions mécaniques possibles, etc.

3.4.1.1.1 Le bouton poussoir normalement ouvert (NO) Dans cette partie du tutoriel, nous allons utiliser ce type de boutons poussoirs (ou BP). Ces derniers ont deux positions :

- **Relâché** : le courant ne passe pas, le circuit est déconnecté ; on dit que le circuit est "ouvert".
- **Appuyé** : le courant passe, on dit que le circuit est **fermé**.

[[a]] | Retenez bien ces mots de vocabulaire !

3.4.1.1.2 Le bouton poussoir normalement fermé (NF) Ce type de bouton est l'opposé du type précédent, c'est-à-dire que lorsque le bouton est relâché, il laisse passer le courant. Et inversement :

- **Relâché** : le courant passe, le circuit est connecté ; on dit que le circuit est "fermé".
- **Appuyé** : le courant ne passe pas, on dit que le circuit est **ouvert**.

3.4.1.1.3 Les interrupteurs À la différence d'un bouton poussoir, l'interrupteur agit comme une bascule. Un appui ferme le circuit et il faut un second appui pour l'ouvrir de nouveau. Il possède donc des états stables (ouvert ou fermé). On dit qu'un interrupteur est **bistable**. Vous en rencontrez tous les jours lorsque vous allumez la lumière ;).

3.4.1.2 L'électronique du bouton

3.4.1.2.1 Symbole Le BP et l'interrupteur ne possèdent pas le même symbole pour les schémas électroniques. Le premier est représenté par une barre qui doit venir faire contact pour fermer le circuit ou défaire le contact pour ouvrir le circuit. Le second est représenté par un fil qui ouvre un circuit et qui peut bouger pour le fermer. Voici leurs symboles, il est important de s'en rappeler :

->

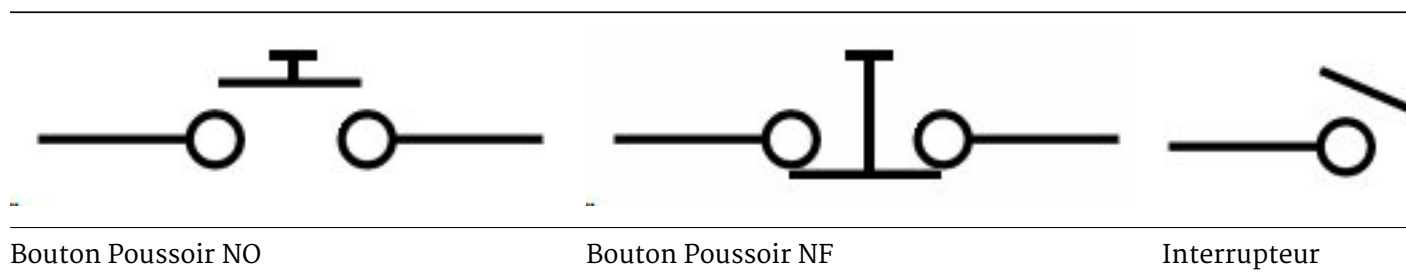


Table 3.2 – Symboles conventionnels des boutons

<-

3.4.1.2.2 Tension et courant

Voici maintenant quelques petites précisions sur les boutons :

- Lorsqu'il est ouvert, la tension à ses bornes ne peut être nulle (ou alors c'est que le circuit n'est pas alimenté). En revanche, lorsqu'il est fermé cette même tension doit être nulle. En effet, aux bornes d'un fil la tension est de 0V.
- Ensuite, lorsque le bouton est ouvert, aucun courant ne peut passer, le circuit est donc déconnecté. Par contre, lorsqu'il est fermé, le courant nécessaire au bon fonctionnement des différents composants le traverse. Il est donc important de prendre en compte cet aspect. Un bouton devant supporter deux ampères ne sera pas aussi gros qu'un bouton tolérant 100 ampères (et pas aussi cher :P).

[[a]] Il est très fréquent de trouver des boutons dans les starters kit.

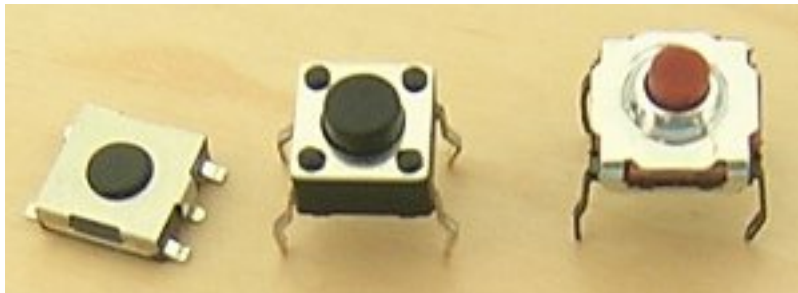


Figure : Quelques boutons pous-

soirs - (CC-BY-SA [Scwerllguy](#))

Souvent ils ont quatre pattes (comme sur l'image ci-dessus). Si c'est le cas, sachez que les broches sont reliées deux à deux. Cela signifie qu'elles fonctionnent par paire. Il faut donc se méfier lorsque vous le branchez sinon vous obtiendrez le même comportement qu'un fil (si vous connectez deux broches reliées). Utilisez un multimètre pour déterminer quelles broches sont distinctes. **Pour ne pas se tromper, on utilise en général deux broches qui sont opposées sur la diagonale du bouton.**

3.4.1.3 Contrainte pour les montages

Voici maintenant un point très important, soyez donc attentif car je vais vous expliquer le rôle d'une résistance de pull-up !

[[q]] |C'est quoi c't'animal, le poule-eup ?

Lorsque l'on fait de l'électronique, on a toujours peur des perturbations (générées par plein de choses : des lampes à proximité, un téléphone portable, un doigt sur le circuit, l'électricité statique, ...). On appelle ça des contraintes de **CEM**. Ces perturbations sont souvent inoffensives, mais perturbent beaucoup les montages électroniques. Il est alors nécessaire de les prendre en compte lorsque l'on fait de l'électronique de signal. Par exemple, dans certains cas on peut se retrouver avec un bit de signal qui vaut 1 à la place de 0, les données reçues sont donc fausses.

Pour contrer ces effets nuisibles, on place en série avec le bouton une résistance de pull-up. Cette résistance sert à "tirer" ("to pull" in english) le potentiel vers le haut (up) afin d'avoir un signal clair sur la broche étudiée. Sur le schéma suivant, on voit ainsi qu'en temps normal le "signal" a un potentiel de 5V. Ensuite, lorsque l'utilisateur appuiera sur le bouton une connexion sera faite avec la masse. On lira alors une valeur de 0V pour le signal. Voici donc un deuxième intérêt de la résistance de pull-up, éviter le court-circuit qui serait généré à l'appui !

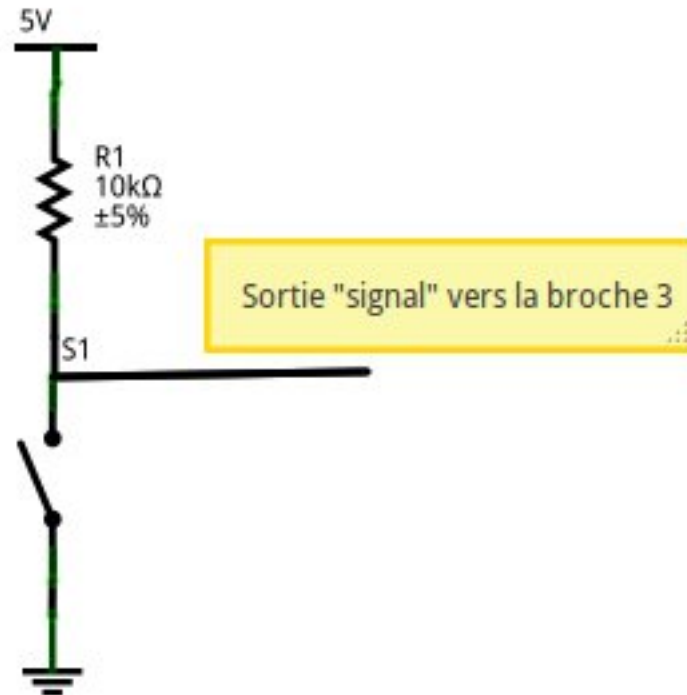


Figure 3.14 – Résistance Pull up

3.4.1.3.1 Filtrer les rebonds Les boutons ne sont pas des systèmes mécaniques parfaits. Du coup, lorsqu'un appui est fait dessus, le signal ne passe pas immédiatement et proprement de 5V à 0V. En l'espace de quelques millisecondes, le signal va "sauter" entre 5V et 0V plusieurs fois avant de se stabiliser. Il se passe le même phénomène lorsque l'utilisateur relâche le bouton.

Ce genre d'effet n'est pas désirable, car il peut engendrer des parasites au sein de votre programme (si vous voulez détecter un appui, les rebonds vont vous en générer une dizaine en quelques millisecondes, ce qui peut-être très gênant dans le cas d'un compteur par exemple). Voilà un exemple de chronogramme relevé lors du relâchement d'un bouton poussoir :

Pour atténuer ce phénomène, nous allons utiliser un condensateur en parallèle avec le bouton. Ce composant servira ici "d'amortisseur" qui absorbera les rebonds (comme sur une voiture avec les cahots de la route). Le condensateur, initialement chargé, va se décharger lors de l'appui sur le bouton. S'il y a des rebonds, ils seront encaissés par le condensateur durant cette décharge. Il se passera le phénomène inverse (charge du condensateur) lors du relâchement du bouton. Ce principe est illustré à la figure suivante :

3.4.1.3.2 Schéma résumé En résumé, voilà un montage que vous pourriez obtenir avec un bouton, sa résistance de pull-up et son filtre anti-rebond sur votre carte Arduino :

3.4.1.4 Les pull-ups internes

Comme expliqué précédemment, pour obtenir des signaux clairs et éviter les courts-circuits, on utilise des résistances de pull-up. Cependant, ces dernières existent aussi en interne du microcontrôleur de l'Arduino, ce qui évite d'avoir à les rajouter par nous-mêmes par la suite. Ces dernières ont une valeur de 20 kilo-Ohms. Elles peuvent être utilisées sans aucune contrainte technique.

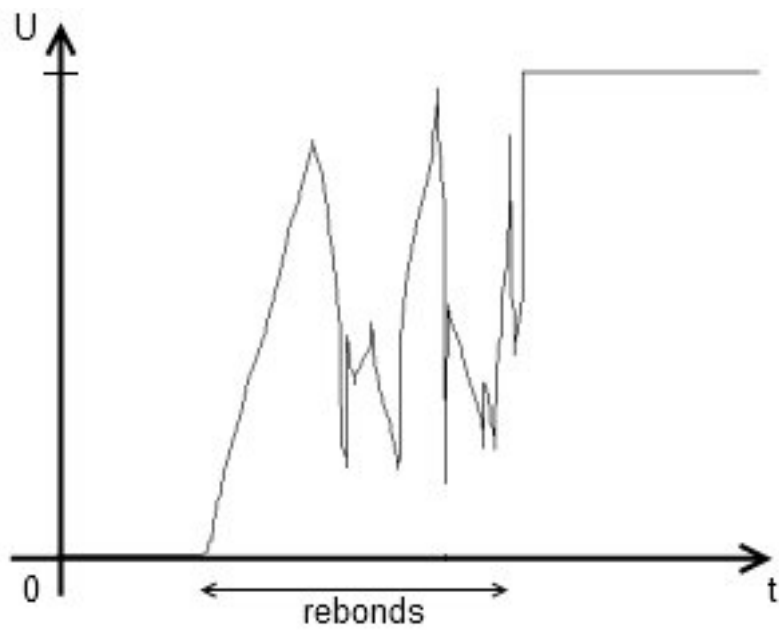


Figure 3.15 – Chronogramme avec rebond

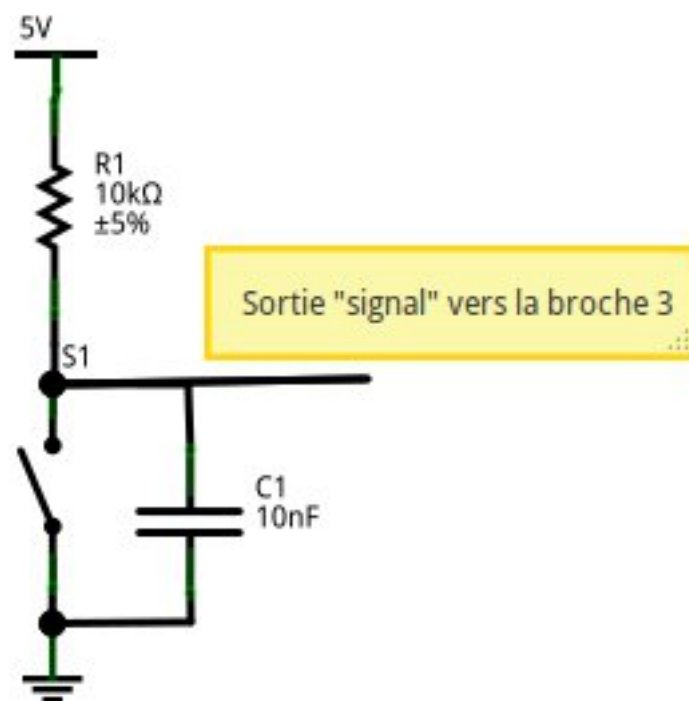


Figure 3.16 – Filtre anti-rebond

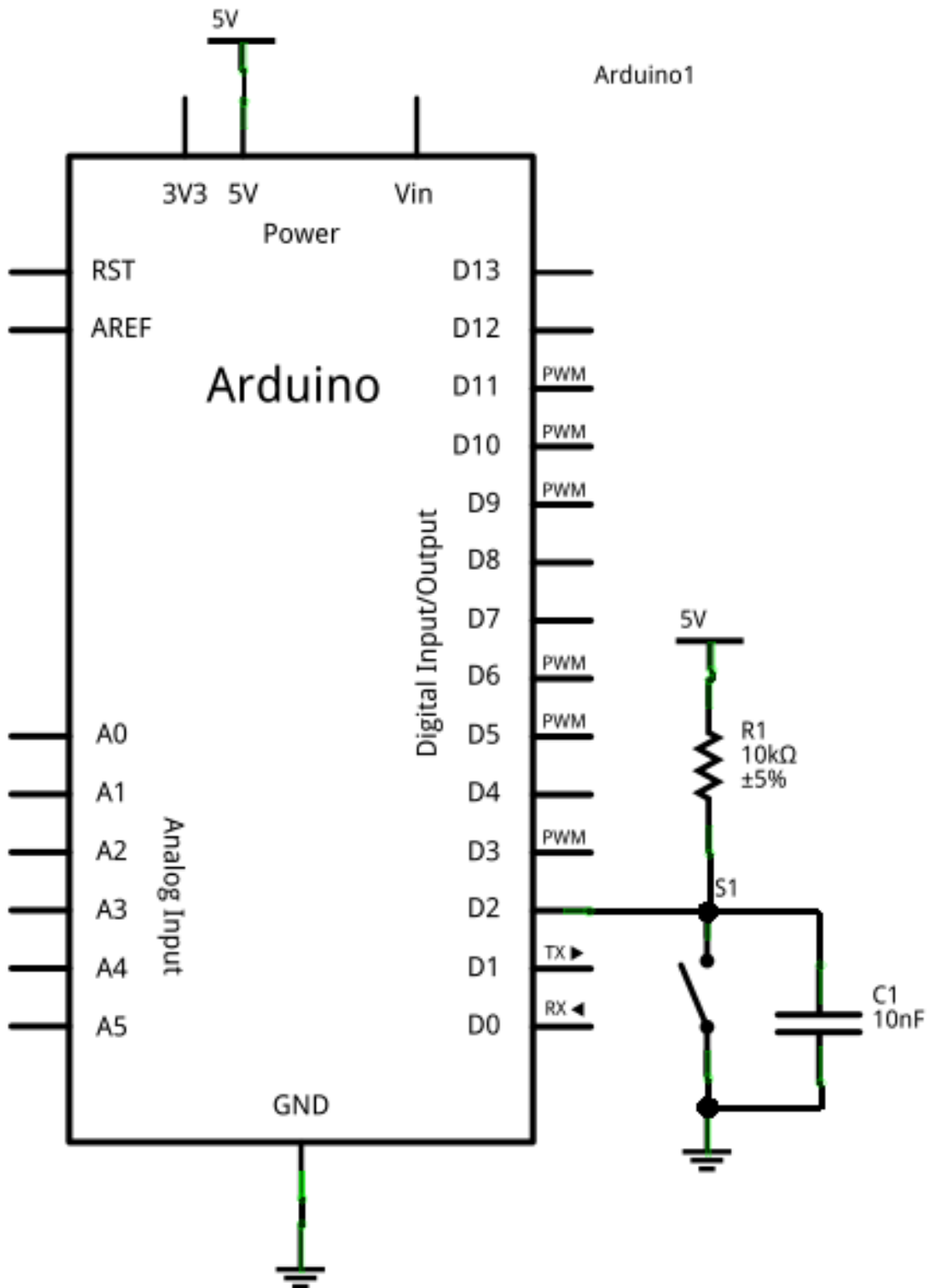


Figure 3.17 – Un bouton + résistance pullup – Schéma

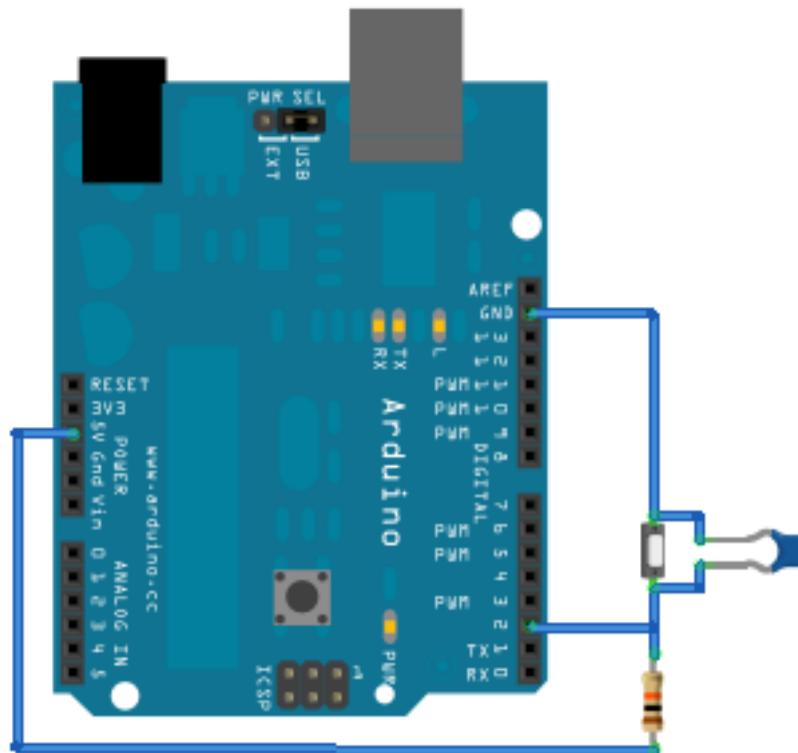


Figure 3.18 – Un bouton + résistance pullup – Montage

Cependant, si vous les mettez en marche, il faut se souvenir que cela équivaut à mettre la broche à l'état haut (et en entrée évidemment). Donc si vous repassez à un état de sortie ensuite, rappelez-vous bien que tant que vous ne l'avez pas changée elle sera à l'état haut. Ce que je viens de dire permet de mettre en place ces dernières dans le logiciel :

```
const int unBouton = 2; // un bouton sur la broche 2

void setup()
{
    // on met le bouton en entrée
    pinMode(unBouton, INPUT);
    // on active la résistance de pull-up en mettant la broche à l'état haut
    // (mais cela reste toujours une entrée)
    digitalWrite(unBouton, HIGH);
}

void loop()
{
    // votre programme
}
```

Code : Initialisation d'un bouton

[[i]] | Depuis la version 1.0.1 d'Arduino, une pull-up peut être simplement mise en oeuvre en utilisant le deuxième argument de `pinMode`. La syntaxe devient `pinMode(unBouton,`

INPUT_PULLUP) et il n'y a plus besoin de faire un `digitalWrite()` ensuite.

3.4.1.4.1 Schéma résumé * [CEM] : Compatibilité ElectroMagnétique

3.4.2 Récupérer l'appui du bouton

3.4.2.1 Montage de base

Pour cette partie, nous allons apprendre à lire l'état d'une entrée numérique. Tout d'abord, il faut savoir qu'une entrée numérique ne peut prendre que deux états, HAUT (HIGH) ou BAS (LOW). L'état haut correspond à une tension de +5V sur la broche, tandis que l'état bas est une tension de 0V. Dans notre exemple, nous allons utiliser un simple bouton. Dans la réalité, vous pourriez utiliser n'importe quel capteur qui possède une sortie numérique. Nous allons donc utiliser :

- Un bouton poussoir (et une résistance de 10k de pull-up et un condensateur anti-rebond de 10nF)
- Une LED (et sa résistance de limitation de courant)
- La carte Arduino

Voici maintenant le schéma à réaliser :

3.4.2.2 Paramétrer la carte

Afin de pouvoir utiliser le bouton, il faut spécifier à Arduino qu'il y a un bouton de connecté sur une de ses broches. Cette broche sera donc une **entrée**. Bien entendu, comme vous êtes de bons élèves, vous vous souvenez que tous les paramétrages initiaux se font dans la fonction `setup()`. Vous vous souvenez également que pour définir le type (entrée ou sortie) d'une broche, on utilise la fonction : `pinMode()`. Notre bouton étant branché sur la pin 2, on écrira :

```
pinMode(2, INPUT);
```

Pour plus de clarté dans les futurs codes, on considérera que l'on a déclaré une variable globale nommée "bouton" et ayant la valeur 2.

Comme ceci :

```
const int bouton = 2;

void setup()
{
    pinMode(bouton, INPUT);
}
```

Voilà, maintenant notre carte Arduino sait qu'il y a quelque chose de connecté sur sa broche 2 et que cette broche est configurée en entrée.

3.4.2.3 Récupérer l'état du bouton

Maintenant que le bouton est paramétré, nous allons chercher à savoir quel est son état (appuyé ou relâché).

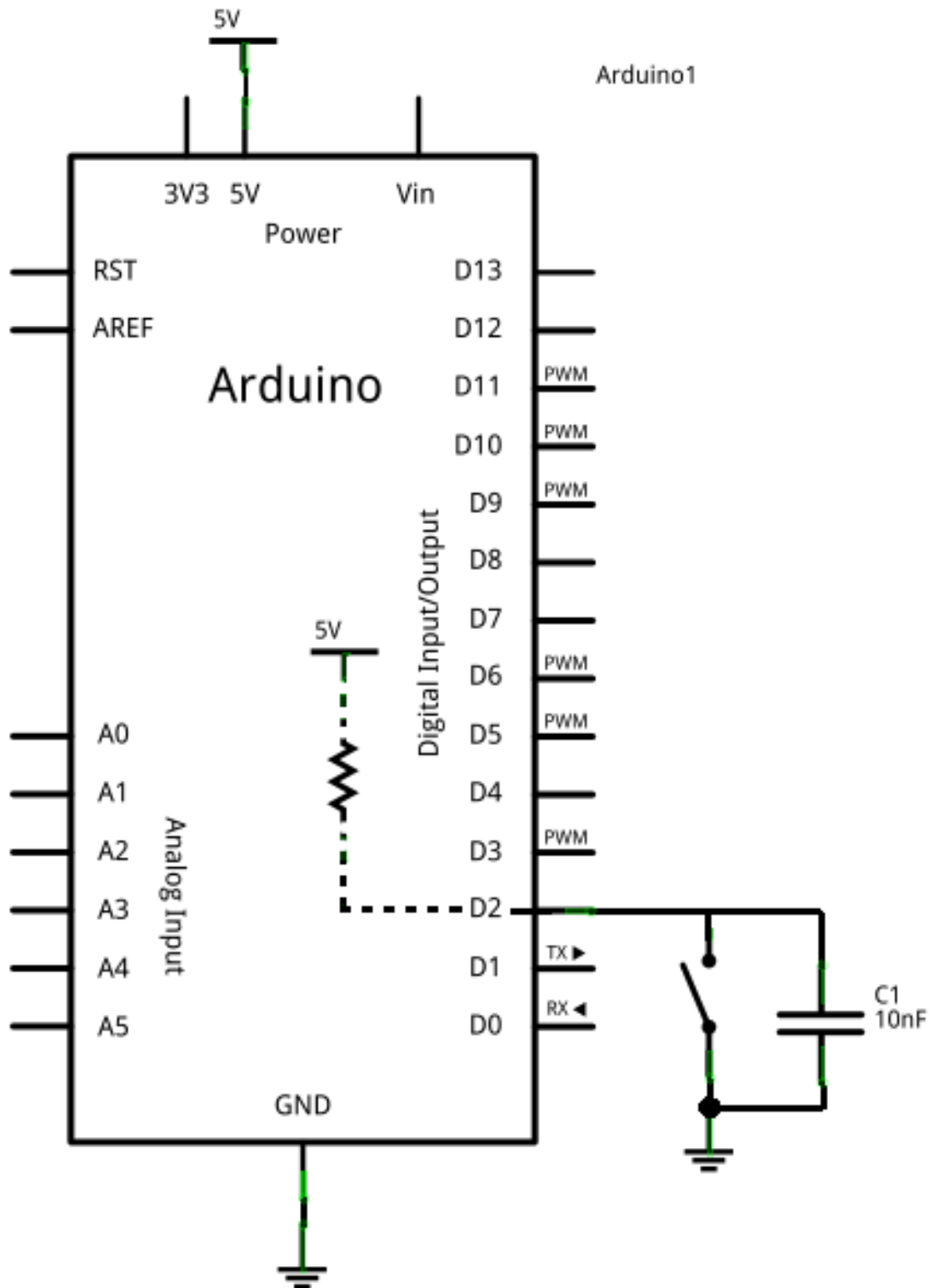


Figure 3.19 – Un bouton + résistance pullup interne - Schéma

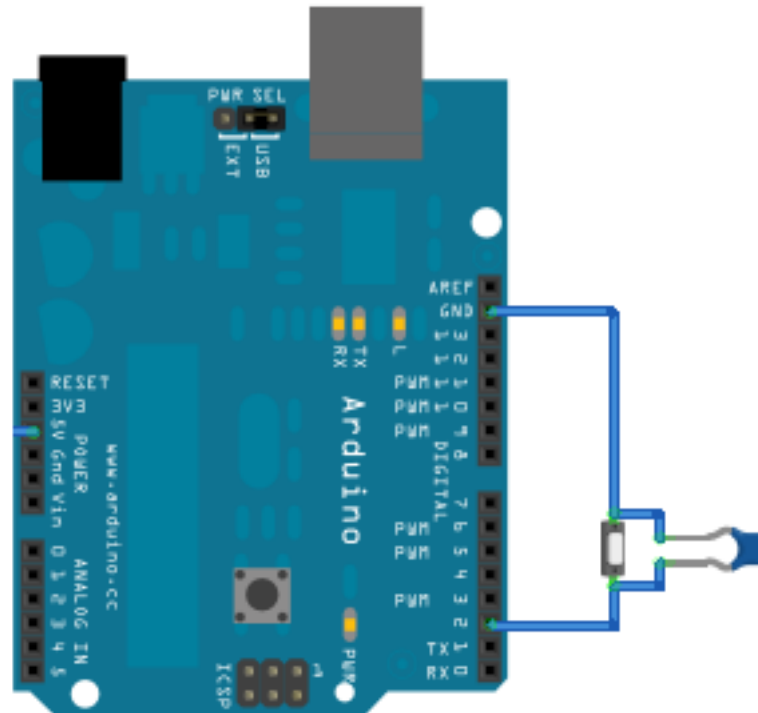


Figure 3.20 – Un bouton + résistance pullup interne – Montage

- S’il est relâché, la tension à ses bornes sera de +5V, donc un état logique HIGH.
- S’il est appuyé, elle sera de 0V, donc LOW.

Un petit tour sur la référence et nous apprenons qu’il faut utiliser la fonction `digitalRead()` pour lire l’état logique d’une entrée logique. Cette fonction prend un paramètre qui est la broche à tester et elle retourne une variable de type `int`. Pour lire l’état de la broche 2 nous ferons donc :

```
int etat;

void loop()
{
    etat = digitalRead(bouton); // Rappel : bouton = 2

    if(etat == HIGH)
        actionRelache(); // le bouton est relâché
    else
        actionAppui(); // le bouton est appuyé
}
```

[[a]] |Observez dans ce code, on appelle deux fonctions qui dépendent de l’état du bouton. |Ces fonctions ne sont pas présentes dans ce code, si vous le testez ainsi, il ne fonctionnera pas. Pour ce faire, vous devrez créer les fonctions `actionAppui()`.

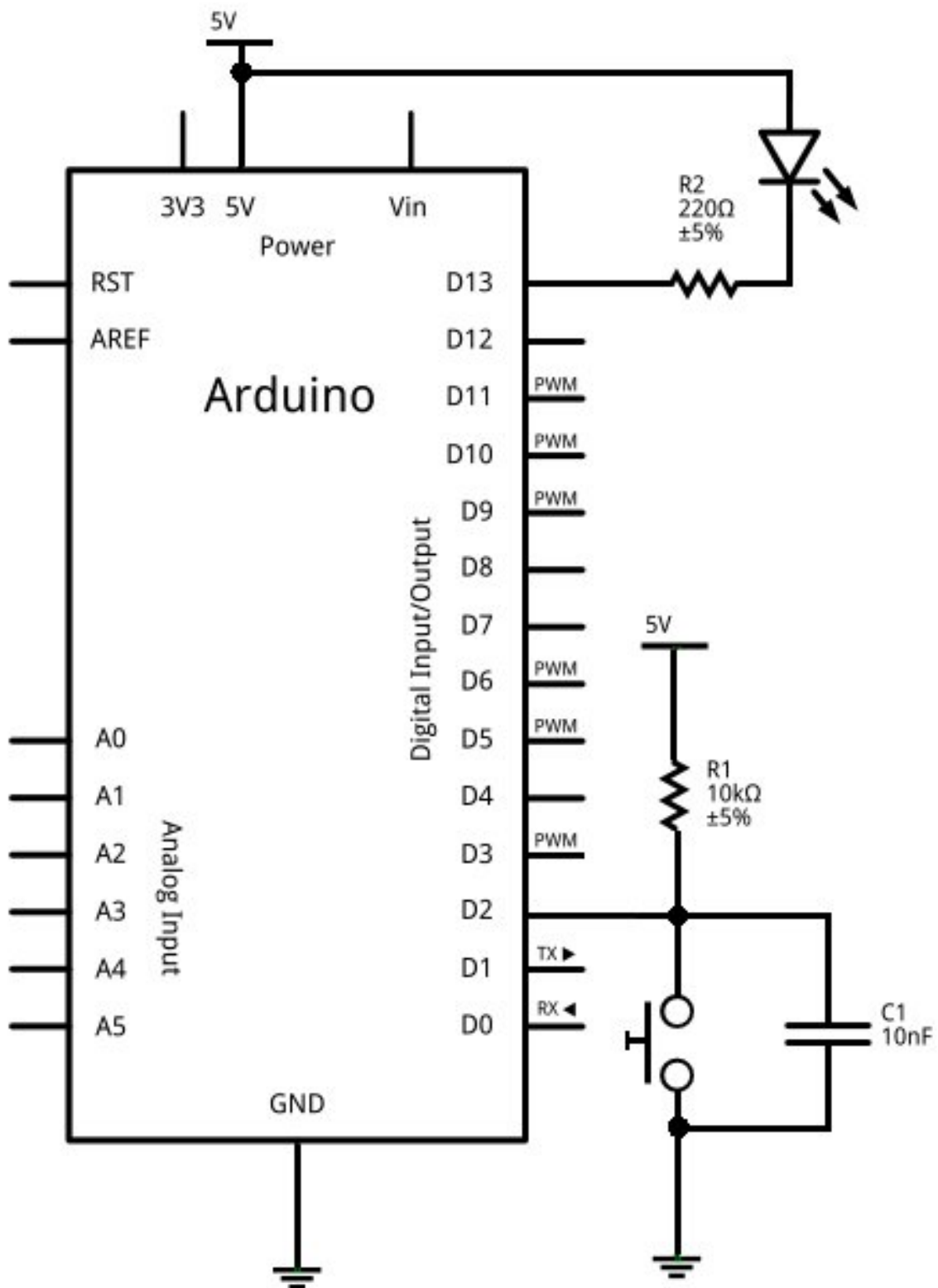


Figure 3.21 - Un bouton et une LED - Schéma

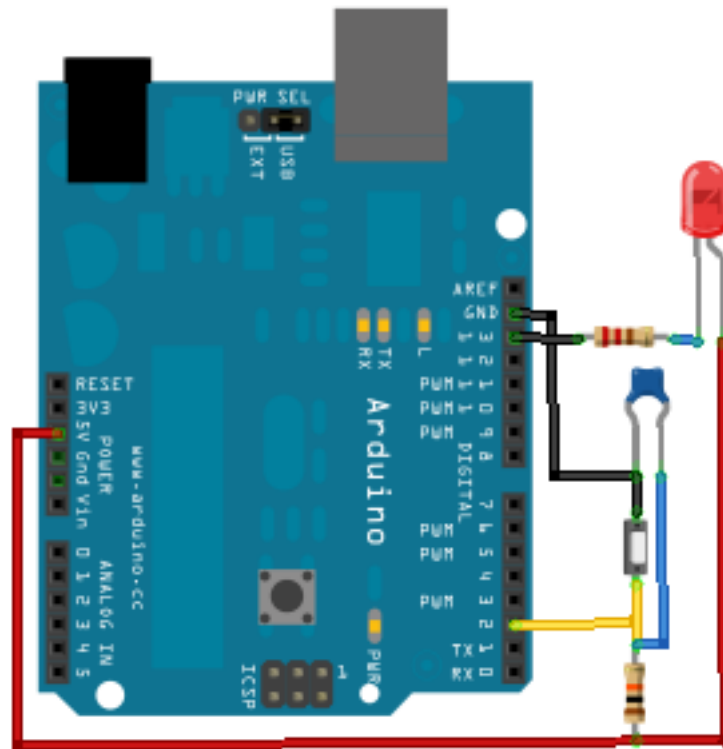


Figure 3.22 – Un bouton et une LED – Montage

3.4.2.4 Test simple

Nous allons passer à un petit test, que *vous* allez faire. Moi je regarde! :diab! :

3.4.2.4.1 But L'objectif de ce test est assez simple : lorsque l'on appuie sur le bouton, la LED doit s'éteindre. Lorsque l'on relâche le bouton, la LED doit s'allumer. Autrement dit, tant que le bouton est **éteint**, la LED est **allumée**.

3.4.2.4.2 Correction Allez, c'est vraiment pas dur, en plus je vous donne le montage dans la première partie... Voici la correction :

```
// le bouton est connecté à la broche 2 de la carte Aduino
const int bouton = 2;
// la LED à la broche 13
const int led = 13;

// variable qui enregistre l'état du bouton
int etatBouton;
```

Code : Les variables globales

```
void setup()
{
    pinMode(led, OUTPUT); // la led est une sortie
```

```
pinMode(bouton, INPUT); // le bouton est une entrée
etatBouton = HIGH; // on initialise l'état du bouton comme "relâché"
}
```

Code : La fonction setup()

```
void loop()
{
  etatBouton = digitalRead(bouton); // Rappel : bouton = 2

  if(etatBouton == HIGH) // test si le bouton a un niveau logique HAUT
  {
    digitalWrite(led, LOW); //le bouton est relâché, la LED est allumée
  }
  else // test si le bouton a un niveau logique différent de HAUT (donc BAS)
  {
    digitalWrite(led, HIGH); //la LED reste éteinte
  }
}
```

Code : La fonction loop()

J'espère que vous y êtes parvenu sans trop de difficultés ! Si oui, passons à l'exercice suivant...

->!(<https://www.youtube.com/watch?v=Eb3Q36zu-S8>)<-

3.4.3 Interagir avec les LED

Nous allons maintenant faire un exemple d'application ensemble.

3.4.3.1 Montage à faire

[[i]] |Pour cet exercice, nous allons utiliser deux boutons et quatre LED de n'importe quelles couleurs.

- Les deux boutons seront considérés actifs (appuyés) à l'état bas (0V) comme dans la partie précédente. Ils seront connectés sur les broches 2 et 3 de l'Arduino.
- Ensuite, les quatre LED seront connectées sur les broches 10 à 13 de l'Arduino.

Voilà donc le montage à effectuer :

3.4.3.2 Objectif : Barregraphe à LED

Dans cet exercice, nous allons faire un mini-barregraphe. Un barregraphe est un afficheur qui indique une quantité, provenant d'une information quelconque (niveau d'eau, puissance sonore, etc.), sous une forme lumineuse. Le plus souvent, on utilise des LED alignées en guise d'affichage. Chaque LED se verra allumée selon un niveau qui sera une fraction du niveau total. Par exemple, si je prends une information qui varie entre 0 et 100, chacune des 4 LED correspondra au quart du maximum de cette variation. Soit $\frac{100}{4} = 25$. En l'occurrence, l'information entrante c'est l'appui des boutons. Par conséquent un appui sur un bouton allume une LED, un appui sur un autre bouton éteint une LED. En fait ce n'est pas aussi direct, il faut incrémenter ou décrémenter la valeur d'une variable et en fonction de cette valeur, on allume telle quantité de LED.

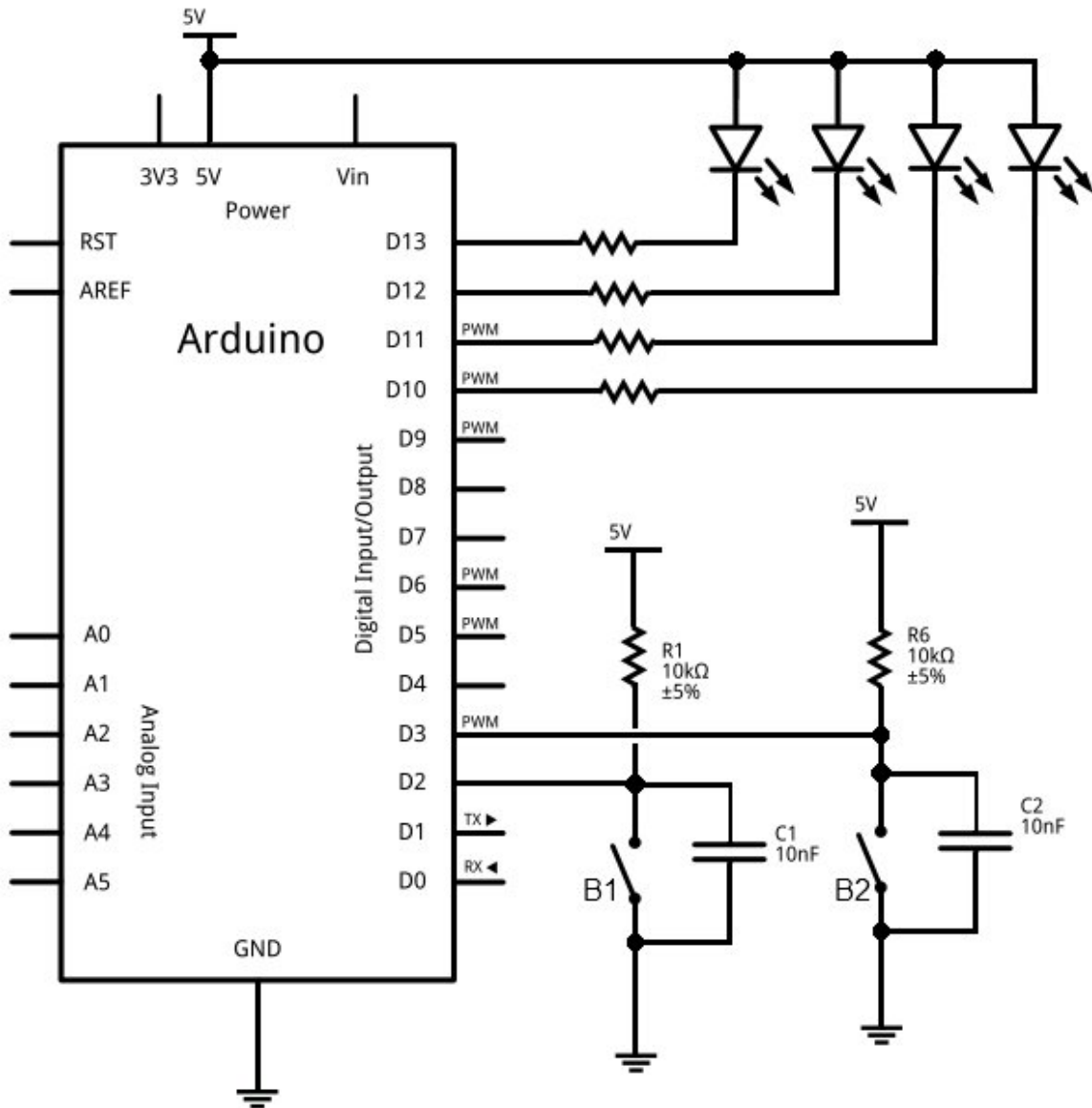


Figure 3.23 – Deux boutons et quatre LED – Schéma

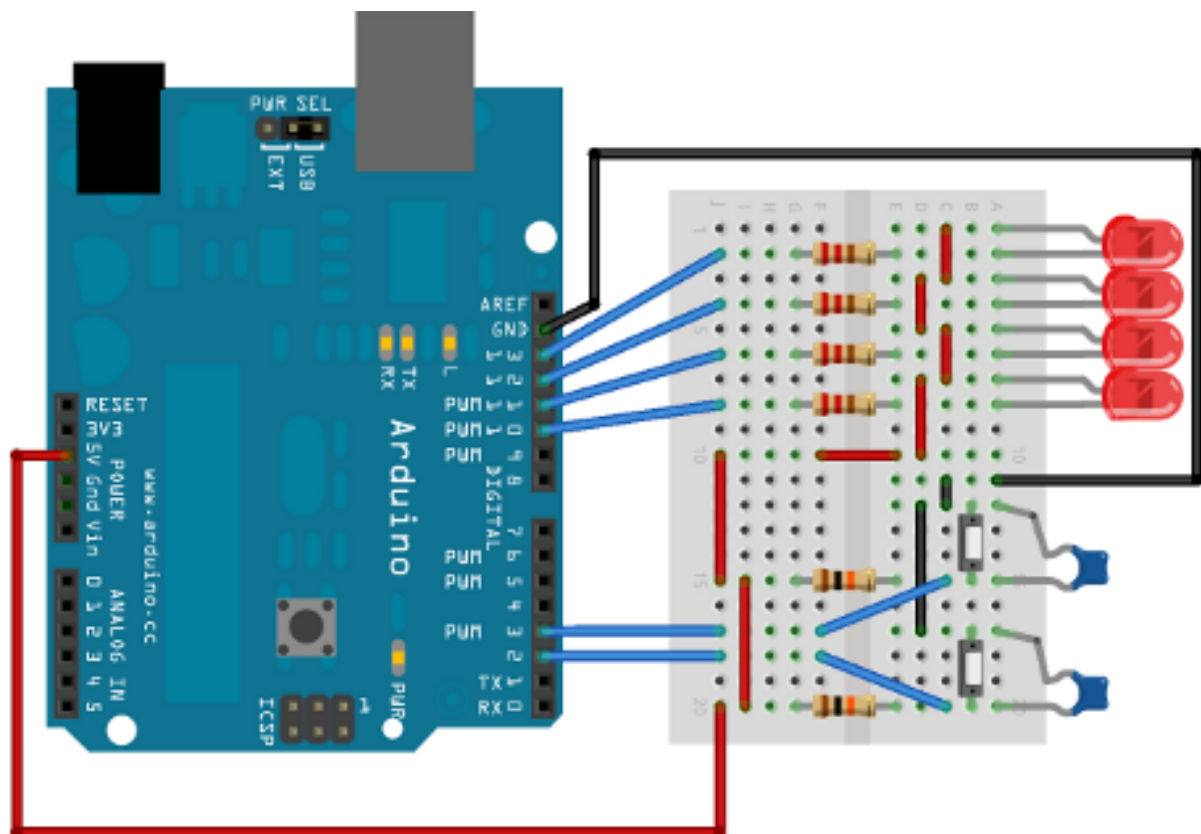


Figure 3.24 – Deux boutons et quatre LED – Montage

3.4.3.2.1 Cahier des charges

La réalisation prévue devra :

- posséder quatre LED (ou plus pour les plus téméraires)
- posséder deux boutons : un qui incrémentera le nombre de LED allumées, l'autre qui le décrémentera

Vous devrez utiliser une variable qui voit sa valeur augmenter ou diminuer entre 0 et 4 selon l'appui du bouton d'incrémentatation ou de décrémentatation.

[[i]] |Vous pouvez maintenant vous lancer dans l'aventure. |Ceux qui se sentent encore un peu mal à l'aise avec la programmation peuvent poursuivre la lecture, qui leur expliquera pas à pas comment procéder pour arriver au résultat final. ;)

3.4.3.3 Correction

3.4.3.3.1 Initialisation

Pour commencer, on crée et on initialise toutes les variables dont on a besoin dans notre programme :

```
/* déclaration des constantes pour les noms des broches ; ceci selon le schéma */
const int btn_minus = 2;
const int btn_plus = 3;
const int led_0 = 10;
const int led_1 = 11;
const int led_2 = 12;
const int led_3 = 13;
```

```

/* déclaration des variables utilisées pour le comptage et le décomptage */

// le nombre qui sera incrémenté et décrétementé
int nombre_led = 0;
// lecture de l'état des boutons (un seul à la fois donc une variable suffit)
int etat_bouton;

/* initialisation des broches en entrée/sortie */
void setup()
{
    pinMode(btn_plus, INPUT);
    pinMode(btn_minus, INPUT);
    pinMode(led_0, OUTPUT);
    pinMode(led_1, OUTPUT);
    pinMode(led_2, OUTPUT);
    pinMode(led_3, OUTPUT);
}

void loop()
{
    // les instructions de votre programme
}

```

3.4.3.3.2 Détection des différences appuyé/relâché Afin de détecter un appui sur un bouton, nous devons comparer son état **courant** avec son état **précédent**. C'est-à-dire qu'avant qu'il soit appuyé ou relâché, on lit son état et on l'inscrit dans une variable. Ensuite, on relit si son état a changé. Si c'est le cas alors on incrémente la variable `nombre_led`. Pour faire cela, on va utiliser une variable de plus par bouton :

```

int memoire_plus = HIGH; // état relâché par défaut
int memoire_minus = HIGH;

```

3.4.3.3.3 Détection du changement d'état Comme dit précédemment, nous devons détecter le changement de position du bouton, sinon on ne verra rien car tout se passera trop vite. Voilà le programme de la boucle principale :

```

void loop()
{
    // lecture de l'état du bouton d'incréméntation
    etat_bouton = digitalRead(btn_plus);

    // Si le bouton a un état différent de celui enregistré ET
    // que cet état est "appuyé"
    if((etat_bouton != memoire_plus) && (etat_bouton == LOW))
    {
        // on incrémente la variable qui indique
    }
}

```

```

        // combien de LED devons s'allumer
        nombre_led++;
    }

    // on enregistre l'état du bouton pour le tour suivant
    memoire_plus = etat_bouton;

    // et maintenant pareil pour le bouton qui décrémente
    etat_bouton = digitalRead(btn_minus); // lecture de son état

    // Si le bouton a un état différent que celui enregistré ET
    // que cet état est "appuyé"
    if((etat_bouton != memoire_minus) && (etat_bouton == LOW))
    {
        nombre_led--; // on décrémente la valeur de nombre_led
    }
    // on enregistre l'état du bouton pour le tour suivant
    memoire_minus = etat_bouton;

    // on applique des limites au nombre pour ne pas dépasser 4 ou 0
    if(nombre_led > 4)
    {
        nombre_led = 4;
    }
    if(nombre_led < 0)
    {
        nombre_led = 0;
    }

    // appel de la fonction affiche() que l'on aura créée
    // on lui passe en paramètre la valeur du nombre de LED à éclairer
    affiche(nombre_led);
}

```

Code : Programme de detection d'évènements sur un bouton

Nous avons terminé de créer le squelette du programme et la détection d'évènements, il ne reste plus qu'à afficher le résultat du nombre !

3.4.3.3.4 L'affichage Pour éviter de se compliquer la vie et d'alourdir le code, on va créer une fonction d'affichage. Celle dont je viens de vous parler : `affiche(int le_parametre)`. Cette fonction reçoit un paramètre représentant le nombre à afficher. À présent, nous devons allumer les LED selon la valeur reçue. On sait que l'on doit afficher une LED lorsque l'on reçoit le nombre 1, deux LED lorsqu'on reçoit le nombre 2, ...

```

void affiche(int valeur_recue)
{
    // on éteint toutes les LED

```

```

digitalWrite(led_0, HIGH);
digitalWrite(led_1, HIGH);
digitalWrite(led_2, HIGH);
digitalWrite(led_3, HIGH);

// Puis on les allume une à une
if(valeur_recue >= 1)
{
    digitalWrite(led_0, LOW);
}
if(valeur_recue >= 2)
{
    digitalWrite(led_1, LOW);
}
if(valeur_recue >= 3)
{
    digitalWrite(led_2, LOW);
}
if(valeur_recue >= 4)
{
    digitalWrite(led_3, LOW);
}
}

```

Code : Fonction d'affichage

Donc, si la fonction reçoit le nombre 1, on allume la LED 1. Si elle reçoit le nombre 2, elle allume la LED 1 et 2. Si elle reçoit 3, elle allume la LED 1, 2 et 3. Enfin, si elle reçoit 4, alors elle allume toutes les LED. Le code au grand complet :

```

[[secret]] | | cpp | // déclaration des constantes pour les nom des broches ;
selon le schéma | const int btn_minus = 2; | const int btn_plus = 3; |
const int led_0 = 10; | const int led_1 = 11; | const int led_2 = 12; |
const int led_3 = 13; | | // déclaration des variables utilisées pour le
comptage et le décomptage | // le nombre qui sera incrémenté et décrémente
| int nombre_led = 0; | // lecture de l'état des boutons (un seul à la
fois mais une variable suffit) | int etat_bouton; | | int memoire_plus
= HIGH; // état relâché par défaut | int memoire_minus = HIGH; | | /*
initilisation des broches en entrée/sortie */ | void setup() | { | pinMode(btn_plus,
INPUT); | pinMode(btn_minus, INPUT); | pinMode(led_0, OUTPUT); | pinMode(led_1,
OUTPUT); | pinMode(led_2, OUTPUT); | pinMode(led_3, OUTPUT); | } | | void
loop() | { | // lecture de l'état du bouton d'incrémementation | etat_bouton
= digitalRead(btn_plus); | // Si le bouton a un état différent que celui
enregistré ET | // que cet état est "appuyé" | if((etat_bouton != memoire_plus)
&& (etat_bouton == LOW)) | { | // on incrémente la variable qui indique |
// combien de LED devons s'allumer | nombre_led++; | } | | // on enregistre
l'état du bouton pour le tour suivant | memoire_plus = etat_bouton; | |
| // et maintenant pareil pour le bouton qui décrémente | etat_bouton =
digitalRead(btn_minus); // lecture de son état | // Si le bouton a un
état différent que celui enregistré ET | // que cet état est "appuyé" |

```

```

if((etat_bouton != memoire_minus) && (etat_bouton == LOW)) | { | nombre_led--;
// on décrémente la valeur de nombre_led | } | // on enregistre l'état
du bouton pour le tour suivant | memoire_minus = etat_bouton; | | // on
applique des limites au nombre pour ne pas dépasser 4 ou 0 | if(nombre_led
> 4) | { | nombre_led = 4; | } | if(nombre_led < 0) | { | nombre_led = 0;
| } | | // appel de la fonction affiche() que l'on aura créée | // on lui
passe en paramètre la valeur du nombre de LED à éclairer | affiche(nombre_led);
| } | | void affiche(int valeur_recue) | { | // on éteint toutes les led |
digitalWrite(led_0, HIGH); | digitalWrite(led_1, HIGH); | digitalWrite(led_2,
HIGH); | digitalWrite(led_3, HIGH); | | // Puis on les allume une à une |
if(valeur_recue >= 1) | { | digitalWrite(led_0, LOW); | } | if(valeur_recue
>= 2) | { | digitalWrite(led_1, LOW); | } | if(valeur_recue >= 3) | { |
digitalWrite(led_2, LOW); | } | if(valeur_recue >= 4) | { | digitalWrite(led_3,
LOW); | } | } | } | Code : Le code final

```

Une petite vidéo du résultat que vous devriez obtenir, même si votre code est différent du mien :

->!(<https://www.youtube.com/watch?v=2fK6nk5NDAU>)<-

3.4.4 Les interruptions matérielles

[[a]] |Voici maintenant un sujet plus délicat (mais pas tant que ça! :ninja :) qui demande votre attention.

Comme vous l'avez remarqué dans la partie précédente, pour récupérer l'état du bouton il faut surveiller régulièrement l'état de ce dernier. Cependant, si le programme a quelque chose de long à traiter, par exemple s'occuper de l'allumage d'une LED et faire une pause avec `delay()` (bien que l'on puisse utiliser `millis()`), l'appui sur le bouton ne sera pas très réactif et lent à la détente. Pour certaines applications, cela peut gêner.

Problème : si l'utilisateur appuie et relâche rapidement le bouton, vous pourriez ne pas détecter l'appui (si vous êtes dans un traitement long).

Solution : utiliser le mécanisme d'**interruption**.

3.4.4.1 Principe

Dans les parties précédentes de ce chapitre, la lecture d'un changement d'état se faisait en comparant régulièrement l'état du bouton à un moment avec son état précédent. Cette méthode fonctionne bien, mais pose un problème : l'appui ne peut pas être détecté s'il est trop court. Autre situation, si l'utilisateur fait un appui très long, mais que vous êtes déjà dans un traitement très long (calcul de la millième décimale de PI, soyons fous), le temps de réponse à l'appui ne sera pas du tout optimal, l'utilisateur aura une impression de lag (= pas réactif). Pour pallier ce genre de problème, les constructeurs de microcontrôleurs ont mis en place des systèmes qui permettent de détecter des événements et d'exécuter des fonctions dès la détection de ces derniers. Par exemple, lorsqu'un pilote d'avion de chasse demande au siège de s'éjecter, le siège doit réagir au moment de l'appui, pas une minute plus tard (trop tard).

[[q]] |Qu'est-ce qu'une interruption ?

Une interruption est en fait un déclenchement qui arrête l'exécution du programme pour faire une tâche demandée. Par exemple, imaginons que le programme compte jusqu'à l'infini. Moi,

programmeur, je veux que le programme arrête de compter lorsque j'appuie sur un bouton. Or, il s'avère que la fonction qui compte est une boucle `for()`, dont on ne peut sortir sans avoir atteint l'infini (autrement dit jamais, en théorie). Nous allons donc nous tourner vers les interruptions qui, dès que le bouton sera appuyé, interrompons le programme pour lui dire : “Arrête de compter, c'est l'utilisateur qui le demande !”.

Pour résumer : **une interruption du programme est générée lors d'un événement attendu. Ceci dans le but d'effectuer une tâche, puis de reprendre l'exécution du programme.** Arduino propose aussi ce genre de gestion d'évènements. On les retrouvera sur certaines broches, sur des timers, des liaisons de communication, etc.

3.4.4.2 Mise en place

Nous allons illustrer ce mécanisme avec ce qui nous concerne ici, les boutons. Dans le cas d'une carte Arduino UNO, on trouve deux broches pour gérer des interruptions externes (qui ne sont pas dues au programme lui-même), la 2 et la 3. Pour déclencher une interruption, plusieurs cas de figure sont possibles :

- **LOW** : Passage à l'état bas de la broche
- **FALLING** : Détection d'un front descendant (passage de l'état haut à l'état bas)
- **RISING** : Détection d'un front montant (pareil qu'avant, mais dans l'autre sens)
- **CHANGE** : Changement d'état de la broche

Autrement dit, s'il y a un changement d'un type énuméré au-dessus, alors le programme sera interrompu pour effectuer une action.

3.4.4.2.1 Créer une nouvelle interruption Comme d'habitude, nous allons commencer par faire des réglages dans la fonction `setup()`. La fonction importante à utiliser est `attachInterrupt(interrupt, fonction, mode)`. Elle accepte trois paramètres :

- `interrupt` : qui est le numéro de la broche utilisée pour l'interruption (0 pour la broche 2 et 1 pour la broche 3)
- `fonction` : qui est le nom de la fonction à appeler lorsque l'interruption est déclenchée
- `mode` : qui est le type de déclenchement (cf. ci-dessus)

Si l'on veut appeler une fonction nommée `Reagir()` lorsque l'utilisateur appuie sur un bouton branché sur la broche 2 on fera :

```
attachInterrupt(0, Reagir, FALLING);
```

[[i]] |Vous remarquerez l'absence des parenthèses après le nom de la fonction “Reagir”.

Ensuite, il vous suffit de coder votre fonction `Reagir()` un peu plus loin.

[[e]] |Attention, cette fonction ne peut pas prendre d'argument et ne retournera aucun résultat.

Lorsque quelque chose déclenchera l'interruption, le programme principal sera mis en pause. Ensuite, lorsque l'interruption aura été exécutée et traitée, il reprendra comme si rien ne s'était produit (avec peut-être des variables mises à jour).

3.4.4.3 Mise en garde

Si je fais une partie entière sur les interruptions, ce n'est pas que c'est difficile mais c'est surtout pour vous mettre en garde sur certains points. Tout d'abord, **les interruptions ne sont pas une solution miracle**. En effet, gardez bien en tête que leur utilisation répond à un besoin **justifié**. Elles mettent tout votre programme en pause, et une mauvaise programmation (ce qui n'arrivera pas, je vous fais confiance ;)) peut entraîner une altération de l'état de vos variables. De plus, les fonctions `delay()` et `millis()` n'auront pas un comportement correct. En effet, pendant ce temps le programme principal est complètement stoppé, donc les fonctions gérant le temps ne fonctionneront plus, elles seront aussi en pause et laisseront la priorité à la fonction d'interruption.

La fonction `delay()` est donc désactivée et la valeur retournée par `millis()` ne changera pas. Justifiez donc votre choix avant d'utiliser les interruptions. ;)

Et voilà, vous savez maintenant comment donner de l'interactivité à l'expérience utilisateur. Vous avez pu voir quelques applications, mais nul doute que votre imagination fertile va en apporter de nouvelles !

3.5 Afficheurs 7 segments

Vous connaissez les afficheurs 7 segments ? Ou alors vous ne savez pas que ça s'appelle comme ça ? Il s'agit des petites lumières qui forment le chiffre 8 et qui sont de couleur rouge ou verte, la plupart du temps, mais peuvent aussi être bleus, blancs, etc. On en trouve beaucoup dans les radio-réveils, car ils servent principalement à afficher l'heure. Autre particularité, non seulement de pouvoir afficher des chiffres (0 à 9), ils peuvent également afficher certaines lettres de l'alphabet.

3.5.0.1 Matériel

Pour ce chapitre, vous aurez besoin de :

- Un (et plus) afficheur 7 segments (évidemment)
- 8 résistances de 330Ω
- Un (ou deux) décodeurs BCD 7 segments
- Une carte Arduino ! Mais dans un premier temps on va d'abord bien saisir le truc avant de faire du code :)

Nous allons commencer par une découverte de l'afficheur, comment il fonctionne et comment le branche-t-on. Ensuite nous verrons comment l'utiliser avec la carte Arduino. Enfin, le chapitre suivant amènera un TP résumant les différentes parties vues.

3.5.1 Première approche : côté électronique

3.5.1.1 Un peu (beaucoup) d'électronique

Comme son nom l'indique, l'afficheur 7 segments possède... 7 segments. Mais un segment c'est quoi au juste ? Et bien c'est une portion de l'afficheur, qui est allumée ou éteinte pour réaliser l'affichage. Cette portion n'est en fait rien d'autre qu'une LED qui au lieu d'être ronde comme d'habitude est plate et encastré dans un boîtier. On dénombre donc 8 portions en comptant le point de l'afficheur (mais il ne compte pas en tant que segment à part entière car il n'est pas toujours présent). Regardez à quoi ça ressemble :

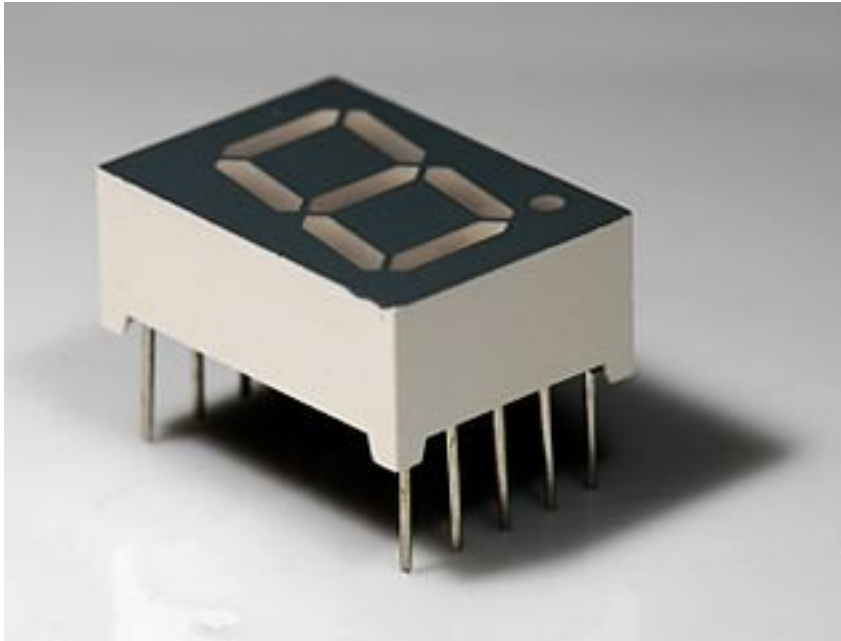


Figure : Un afficheur 7 segments - (CC-BY-SA, Pengo)

ments - (CC-BY-SA, Pengo)

3.5.1.1.1 Des LED, encore des LED Et des LED, il y en a ! Entre 7 et 8 selon les modèles (c'est ce que je viens d'expliquer), voir beaucoup plus, mais on ne s'y attardera pas dessus. Voici un schéma vous présentant un modèle d'afficheur sans le point (qui au final est juste une LED supplémentaire rappelez-vous) :

Les interrupteurs a,b,c,d,e,f,g représentent les signaux pilotant chaque segments

Comme vous le voyez sur ce schéma, toutes les LED possèdent une broche commune, reliée entre elle. Selon que cette broche est la cathode ou l'anode on parlera d'afficheur à cathode commune ou... anode commune (vous suivez ?). Dans l'absolu, ils fonctionnent de la même façon, seule la manière de les brancher diffère (actif sur état bas ou sur état haut).

3.5.1.1.2 Cathode commune ou Anode commune Dans le cas d'un afficheur à cathode commune, toutes les cathodes sont reliées entre elles en un seul point lui-même connecté à la masse. Ensuite, chaque anode de chaque segment sera reliée à une broche de signal. Pour allumer chaque segment, le signal devra être une tension positive. En effet, si le signal est à 0, il n'y a pas de différence de potentiel entre les deux broches de la LED et donc elle ne s'allumera pas ! Si nous sommes dans le cas d'une anode commune, les anodes de toutes les LED sont reliées entre elles en un seul point qui sera connecté à l'alimentation. Les cathodes elles seront reliées

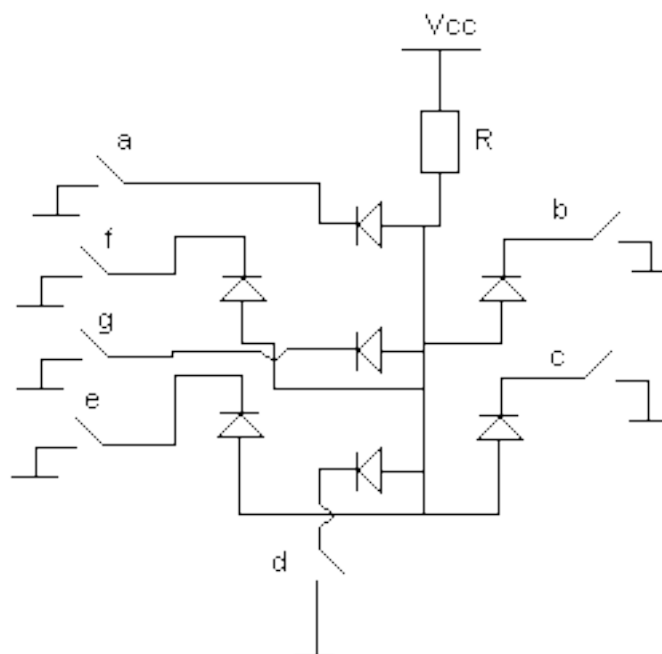


Figure 3.25 – Schéma de l'afficheur 7 segments

une par une aux broches de signal. En mettant une broche de signal à 0, le courant passera et le segment en question s'allumera. Si la broche de signal est à l'état haut, le potentiel est le même de chaque côté de la LED, donc elle est bloquée et ne s'allume pas ! Que l'afficheur soit à anode ou à cathode commune, on doit toujours prendre en compte qu'il faut ajouter une résistance de limitation de courant entre la broche isolée et la broche de signal. Traditionnellement, on prendra une résistance de 330Ω pour une tension de +5V, mais cela se calcul (cf. chapitre 1, partie 2).

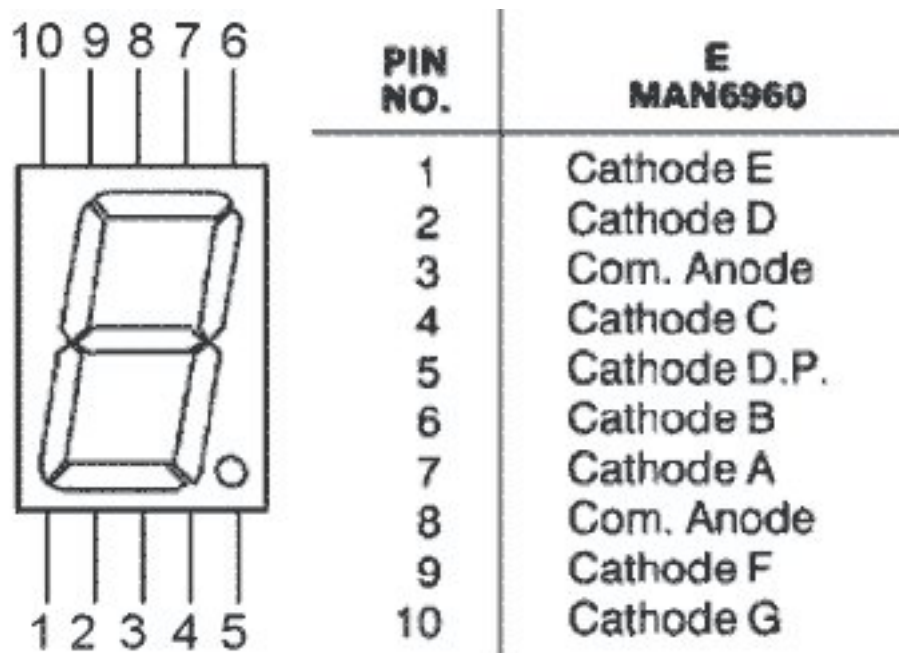
- Si vous voulez augmenter la luminosité, il suffit de diminuer cette valeur.
- Si au contraire vous voulez diminuer la luminosité, augmenter la résistance.

3.5.1.1.3 Choix de l'afficheur Pour la rédaction j'ai fait le choix d'utiliser des afficheurs à anode commune et ce n'est pas anodin. En effet et on l'a vu jusqu'à maintenant, on branche les LED du +5V vers la broche de la carte Arduino. Ainsi, dans le cas d'un afficheur à anode commune, les LED seront branchés d'un côté au +5V, et de l'autre côté aux broches de signaux. Ainsi, pour allumer un segment on mettra la broche de signal à 0 et on l'éteindra en mettant le signal à 1. On a toujours fait comme ça depuis le début, ça ne vous posera donc aucun problème. ;)

3.5.1.2 Branchement "complet" de l'afficheur

Nous allons maintenant voir comment brancher l'afficheur à anode commune.

3.5.1.2.1 Présentation du boîtier Les afficheurs 7 segments se présentent sur un boîtier de type DIP 10.* Le format DIP régie l'espacement entre les différentes broches du circuit intégré ainsi que d'autres contraintes (présence d'échangeur thermique etc...). Le chiffre 10 signifie qu'il possède 10 broches (5 de part et d'autre du boîtier). Voici une représentation de ce dernier (à gauche) :



segments - (source : datasheet)

Figure : Boîtier du 7

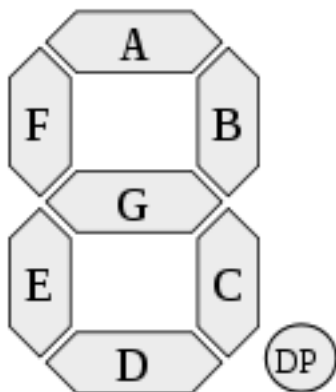


Figure : Dénomination des segments - (CC-BY-

SA, [h2g2bob](#))

Voici la signification des différentes broches :

1. LED de la cathode E
2. LED de la cathode D
3. Anode commune des LED
4. LED de la cathode C
5. (facultatif) le point décimal.
6. LED de la cathode B
7. LED de la cathode A
8. Anode commune des LED
9. LED de la cathode F
10. LED de la cathode G

Pour allumer un segment c'est très simple, il suffit de le relier à la masse !

[[e]] |Nous cherchons à allumer les LED de l'afficheur, il est donc impératif de ne pas oublier les résistances de limitations de courant !

3.5.1.2.2 Exemple Pour commencer, vous allez tout d'abord mettre l'afficheur à cheval sur la plaque d'essai (breadboard). Ensuite, trouvez la broche représentant l'anode commune et reliez la à la future colonne du +5V. Prochaine étape, mettre une résistance de 330Ω sur chaque broche de signal. Enfin, reliez quelques une de ces résistances à la masse. Si tous se passe bien, les segments reliés à la masse via leur résistance doivent s'allumer lorsque vous alimentez le circuit. Voici un exemple de branchement :

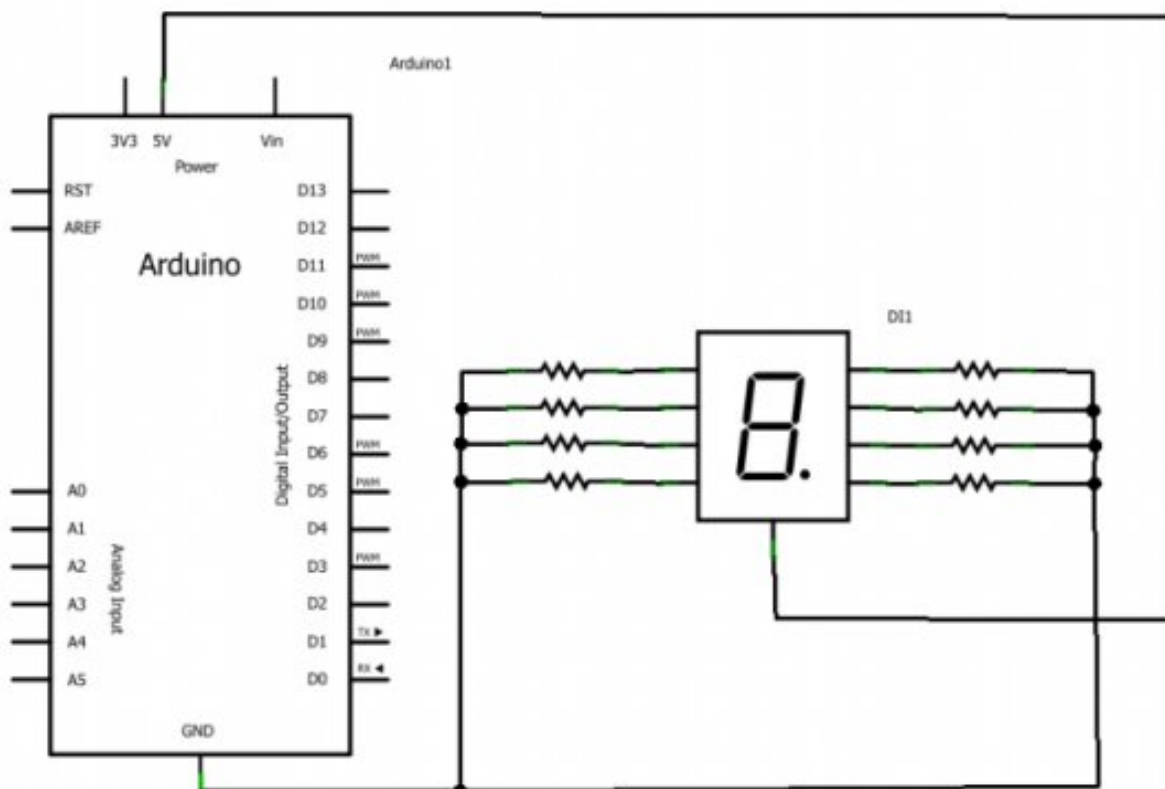


Figure 3.26 – 7 segments schéma

Dans cet exemple de montage, vous verrez que tous les segment de l'afficheur s'allument ! Vous pouvez modifier le montage en déconnectant quelques unes des résistance de la masse et afficher de nombreux caractères.

[[a]] |Pensez à couper l'alimentation lorsque vous changer des fils de place. Les composants n'aiment pas forcément être (dé)branchés lorsqu'ils sont alimentés. Vous pourriez éventuellement leur causer des dommages.

3.5.1.2.3 Seulement 7 segments mais plein de caractère(s) ! Vous l'avez peut-être remarqué avec "l'exercice" précédent, un afficheurs 7 segments ne se limite pas à afficher juste des chiffres. Voici un tableau illustrant les caractères possibles et quels segments allumés. Attention, il est possible qu'il manque certains caractères !

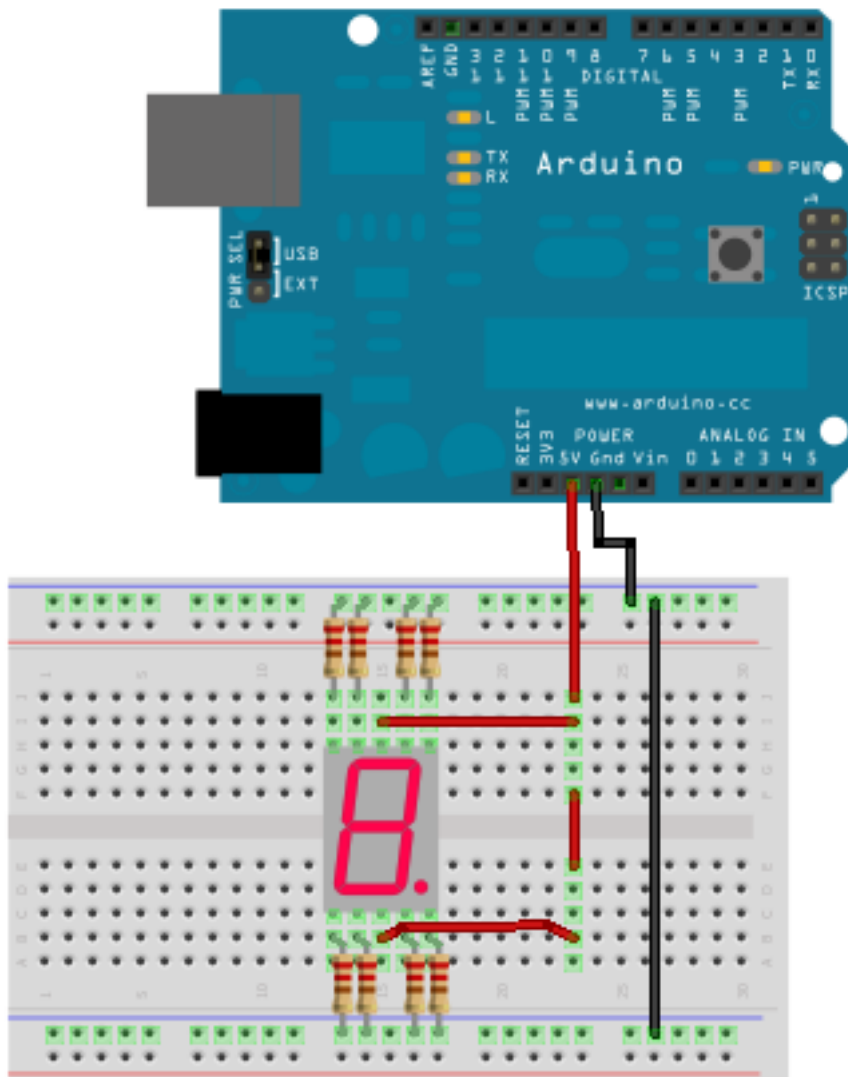


Figure 3.27 – 7 segments breadboard

->

Caractère	seg. A	seg. B	seg. C	seg. D	seg. E	seg. F	seg. G
0	->x<-	->x<-	->x<-	->x<-	->x<-	->x<-	
1		->x<-	->x<-				
2	->x<-	->x<-		->x<-	->x<-		->x<-
3	->x<-	->x<-	->x<-	->x<-			->x<-
4		->x<-	->x<-			->x<-	->x<-
5	->x<-		->x<-	->x<-		->x<-	->x<-
6	->x<-		->x<-	->x<-	->x<-	->x<-	->x<-
7	->x<-	->x<-	->x<-				
8	->x<-	->x<-	->x<-	->x<-	->x<-	->x<-	->x<-
9	->x<-	->x<-	->x<-	->x<-		->x<-	->x<-
A	->x<-	->x<-	->x<-		->x<-	->x<-	->x<-
b			->x<-	->x<-	->x<-	->x<-	->x<-
C	->x<-			->x<-	->x<-	->x<-	
d		->x<-	->x<-	->x<-	->x<-	->x<-	
E	->x<-			->x<-	->x<-	->x<-	->x<-
F	->x<-				->x<-	->x<-	->x<-
H		->x<-	->x<-		->x<-	->x<-	->x<-
I		->x<-	->x<-				
J		->x<-	->x<-	->x<-	->x<-		
L				->x<-	->x<-	->x<-	
o			->x<-	->x<-	->x<-		->x<-
P	->x<-	->x<-			->x<-	->x<-	->x<-
S	->x<-		->x<-	->x<-	->x<-		->x<-
t					->x<-	->x<-	->x<-
U		->x<-	->x<-	->x<-	->x<-	->x<-	
u		->x<-	->x<-	->x<-	->x<-		->x<-
°	->x<-	->x<-				->x<-	->x<-

Table 3.3 – Caractères affichage avec un afficheur 7 segments

<-

[[i]] |Aidez-vous de ce tableau lorsque vous aurez à coder l’affichage de caractères !;)

*[DIP] : Dual Inline Package

3.5.2 Afficher son premier chiffre !

Pour commencer, nous allons prendre en main un afficheur et lui faire s'afficher notre premier chiffre ! C'est assez simple et ne requiert qu'un programme très simple, mais un peu rébarbatif.

3.5.2.1 Schéma de connexion

Je vais reprendre le schéma précédent, mais je vais connecter chaque broche de l'afficheur à une sortie de la carte Arduino. Comme ceci :

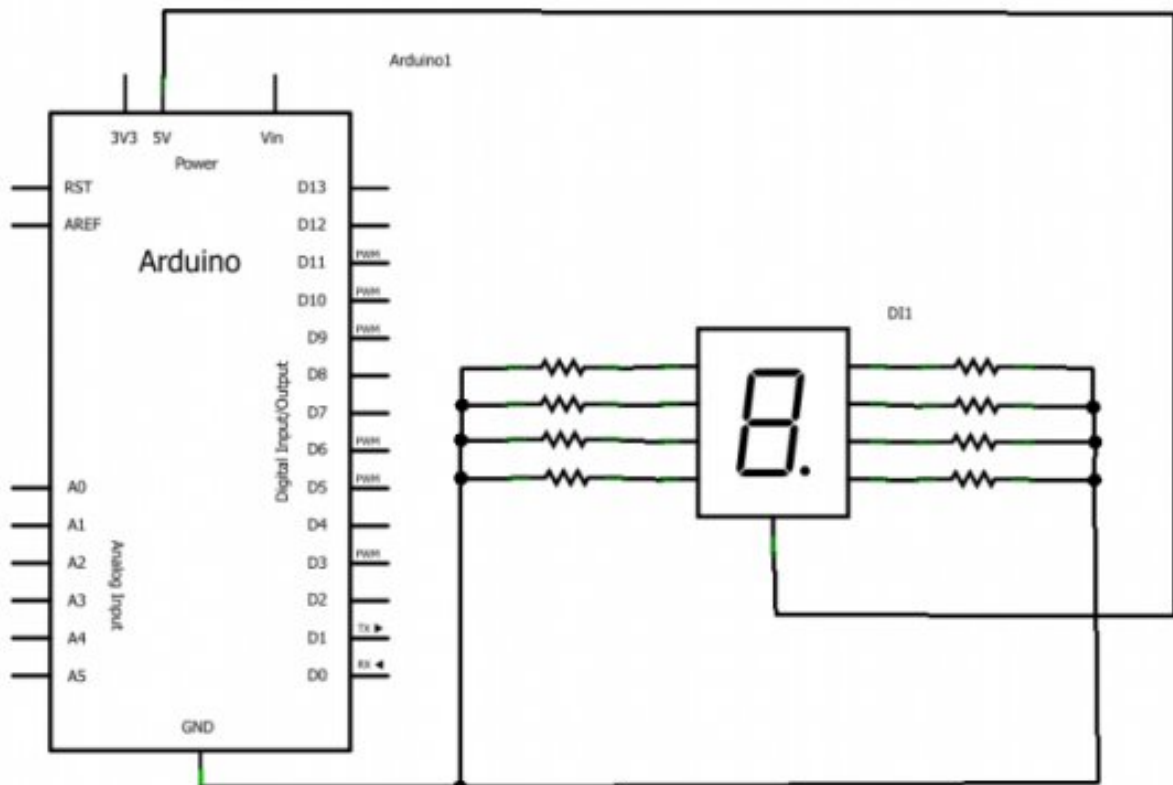


Figure 3.28 – 7 segments schéma

Vous voyez donc que chaque LED de l'afficheur va être commandée séparément les unes des autres. Il n'y a rien de plus à faire, si ce n'est qu'à programmer...

3.5.2.2 Le programme

L'objectif du programme va être d'afficher un chiffre. Eh bien... c'est parti ! Quoi ?! Vous voulez de l'aide ? o_O Ben je vous ai déjà tout dit y'a plus qu'à faire. En plus vous avez un tableau avec lequel vous pouvez vous aider pour afficher votre chiffre. Cherchez, je vous donnerais la solution ensuite.

```
[[secret]] | cpp | /* On assigne chaque LED à une broche de l'arduino */ |
const int A = 2; | const int B = 3; | const int C = 4; | const int D = 5;
| const int E = 6; | const int F = 7; | const int G = 8; | // notez que
l'on ne gère pas l'affichage du point, | // mais vous pouvez le rajouter
```

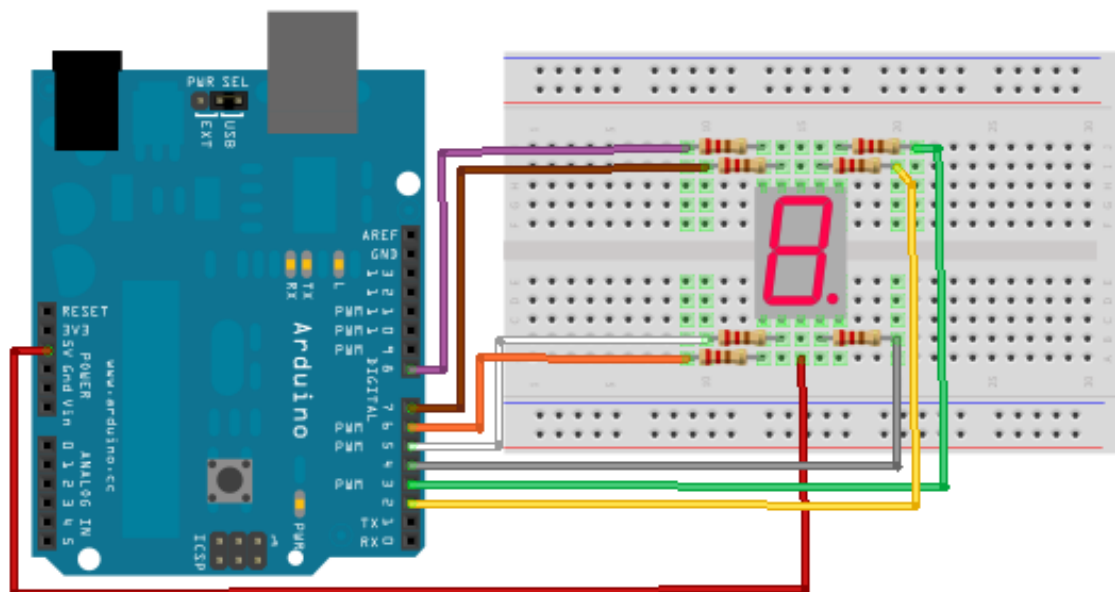


Figure 3.29 – Afficheur 7 segments montage

```

si cela vous chante | | void setup() | { | // définition des broches en
sortie | pinMode(A, OUTPUT); | pinMode(B, OUTPUT); | pinMode(C, OUTPUT); |
pinMode(D, OUTPUT); | pinMode(E, OUTPUT); | pinMode(F, OUTPUT); | pinMode(G,
OUTPUT); | | // mise à l'état HAUT de ces sorties pour éteindre les LED de
l'afficheur | digitalWrite(A, HIGH); | digitalWrite(B, HIGH); | digitalWrite(C,
HIGH); | digitalWrite(D, HIGH); | digitalWrite(E, HIGH); | digitalWrite(F,
HIGH); | digitalWrite(G, HIGH); | } | | void loop() | { | // affichage du
chiffre 5, d'après le tableau précédent | digitalWrite(A, LOW); | digitalWrite(B,
HIGH); | digitalWrite(C, LOW); | digitalWrite(D, LOW); | digitalWrite(E,
HIGH); | digitalWrite(F, LOW); | digitalWrite(G, LOW); | } ||Code:Solution

```

Vous le voyez par vous-même, c'est un code hyper simple. Essayez de le bidouiller pour afficher des messages, par exemple, en utilisant les fonctions introduisant le temps. Ou bien compléter ce code pour afficher tous les chiffres, en fonction d'une variable définie au départ (ex : var = 1, affiche le chiffre 1; etc.).

3.5.3 Techniques d'affichage

Vous vous en doutez peut-être, lorsque l'on veut utiliser plusieurs afficheur il va nous falloir beaucoup de broches. Imaginons, nous voulons afficher un nombre entre 0 et 99, il nous faudra utiliser deux afficheurs avec $2 * 7 = 14$ broches connectées sur la carte Arduino. Rappel : une carte Arduino UNO possède... 14 broches entrées/sorties classiques. Si on ne fait rien d'autre que d'utiliser les afficheurs, cela ne nous gêne pas, cependant, il est fort probable que vous serez amené à utiliser d'autres entrées avec votre carte Arduino. Mais si on ne libère pas de place vous serez embêté.

Nous allons donc voir deux techniques qui, une fois cumulées, vont nous permettre d'utiliser seulement 4 broches pour obtenir le même résultat qu'avec 14 broches !

3.5.3.1 Les décodeurs “4 bits -> 7 segments”

La première technique que nous allons utiliser met en œuvre un circuit intégré. Vous vous souvenez quand je vous ai parlé de ces bêtes là ? Oui, c’est le même type que le microcontrôleur de la carte Arduino. Cependant, le circuit que nous allons utiliser ne fait pas autant de choses que celui sur votre carte Arduino.

3.5.3.1.1 Décodeur BCD -> 7 segments C’est le nom du circuit que nous allons utiliser. Son rôle est simple. Vous vous souvenez des conversions ? Pour passer du binaire au décimal ? Et bien c’est le moment de vous en servir, donc si vous ne vous rappelez plus de ça, allez revoir un peu [le cours](#). Je disais donc que son rôle est simple. Et vous le constaterez par vous même, il va s’agir de convertir du binaire codé sur 4 bits vers un “code” utilisé pour afficher les chiffres. Ce code correspond en quelque sorte au tableau précédemment évoqué.

3.5.3.1.2 Principe du décodeur Sur un afficheur 7 segments, on peut représenter aisément les chiffres de 0 à 9. En informatique, pour représenter ces chiffres, il nous faut au maximum 4 bits. Comme vous êtes des experts et que vous avez bien lu la partie sur le binaire, vous n’avez pas de mal à le comprendre. $(0000)_2$ fera $(0)_{10}$ et $(1111)_2$ fera $(15)_{10}$ ou $(F)_{16}$. Pour faire 9 par exemple on utilisera les bits 1001. En partant de se constat, des ingénieurs ont inventé un composant au doux nom de “décodeur” ou “driver” 7 segments. Il reçoit sur 4 broches les 4 bits de la valeur à afficher, et sur 7 autres broches ils pilotent les segments pour afficher ladite valeur. Ajouter à cela une broche d’alimentation et une broche de masse on obtient 13 broches ! Et ce n’est pas fini. La plupart des circuits intégrés de type décodeur possède aussi une broche d’activation et une broche pour tester si tous les segments fonctionnent.

3.5.3.1.3 Choix du décodeur Nous allons utiliser le composant nommé MC14543B comme exemple (un équivalent utilisable et trouvable facilement est le CD4543BE). Tout d’abord, ouvrez ce lien dans un nouvel onglet, il vous mènera directement vers le pdf du décodeur :

->[Datasheet du MC14543B](#)<-

Les datasheets se composent souvent de la même manière. On trouve tout d’abord un résumé des fonctions du produit puis un schéma de son boîtier. Dans notre cas, on voit qu’il est monté sur un DIP 16 (DIP : Dual Inline Package, en gros “boîtier avec deux lignes de broches”). Si l’on continue, on voit la **table de vérité** faisant le lien entre les signaux d’entrées (INPUT) et les sorties (OUTPUT). On voit ainsi plusieurs choses :

- Si l’on met la broche BI (Blank, n°7) à un, toutes les sorties passent à zéro. En effet, comme son nom l’indique cette broche sert à effacer l’afficheur. Si vous ne voulez pas l’utiliser il faut donc la connecter à la masse pour la désactiver ;
- Les entrées A, B, C et D (broches 5,3,2 et 4 respectivement) sont actives à l’état HAUT. Les sorties elles sont actives à l’état BAS (pour piloter un afficheur à anode commune) **OU** HAUT selon l’état de la broche PH (6). C’est là un gros avantage de ce composant, il peut inverser la logique de la sortie, le rendant alors compatible avec des afficheurs à anode commune (broche PH à l’état 1) ou cathode commune (Ph = 0) ;
- La broche BI (Blank Input, n°7) sert à inhiber les entrées. On ne s’en servira pas et donc on la mettra à l’état HAUT (+5V) ;
- LD (n°1) sert à faire une mémoire de l’état des sorties, on ne s’en servira pas ici. Elle signifie “Latch Disable”. En la mettant à 1 on désactive donc le “latch” (verrou) et nos entrées sont

alors bien prises en considération ;

- Enfin, les deux broches d'alimentation sont la 8 (GND/VSS, masse) et la 16 (VCC, +5V).

[[a]] N'oubliez pas de mettre des résistances de limitations de courant entre chaque segment et la broche de signal du circuit !

3.5.3.1.4 Fonctionnement [[q]] C'est bien beau tout ça mais comment je lui dis au décodeur d'afficher le chiffre 5 par exemple ?

Il suffit de regarder le datasheet et sa table de vérité (c'est le tableau avec les entrées et les sorties). Ce que reçoit le décodeur sur ses entrées (A, B, C et D) définit les états de ses broches de sortie (a,b,c,d,e,f et g). C'est tout ! Donc, on va donner un code binaire sur 4 bits à notre décodeur et en fonction de ce code, le décodeur affichera le caractère voulu. En plus le fabricant est sympa, il met à disposition des notes d'applications à la page 6 pour bien brancher le composant :

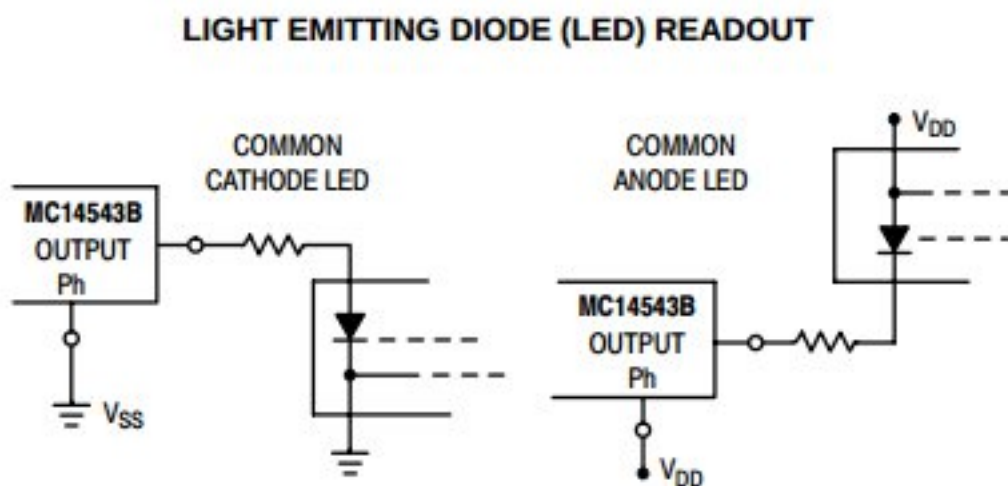
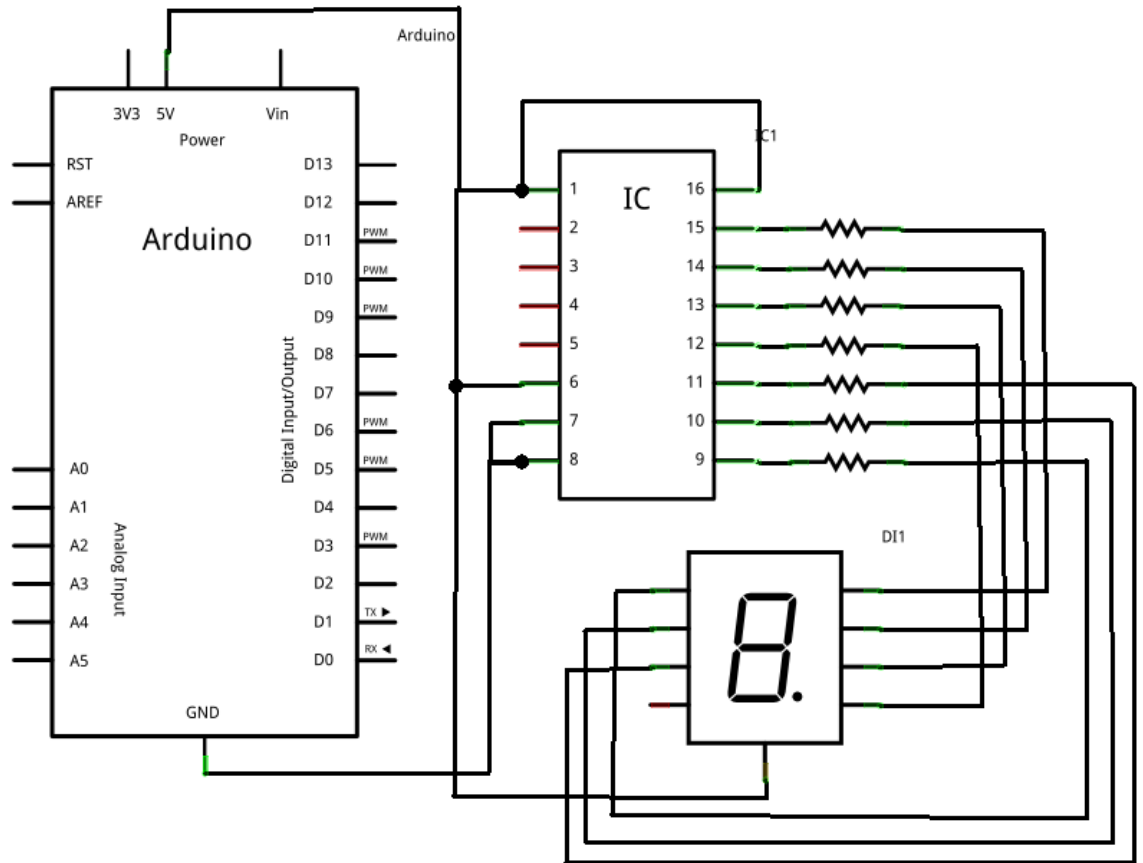


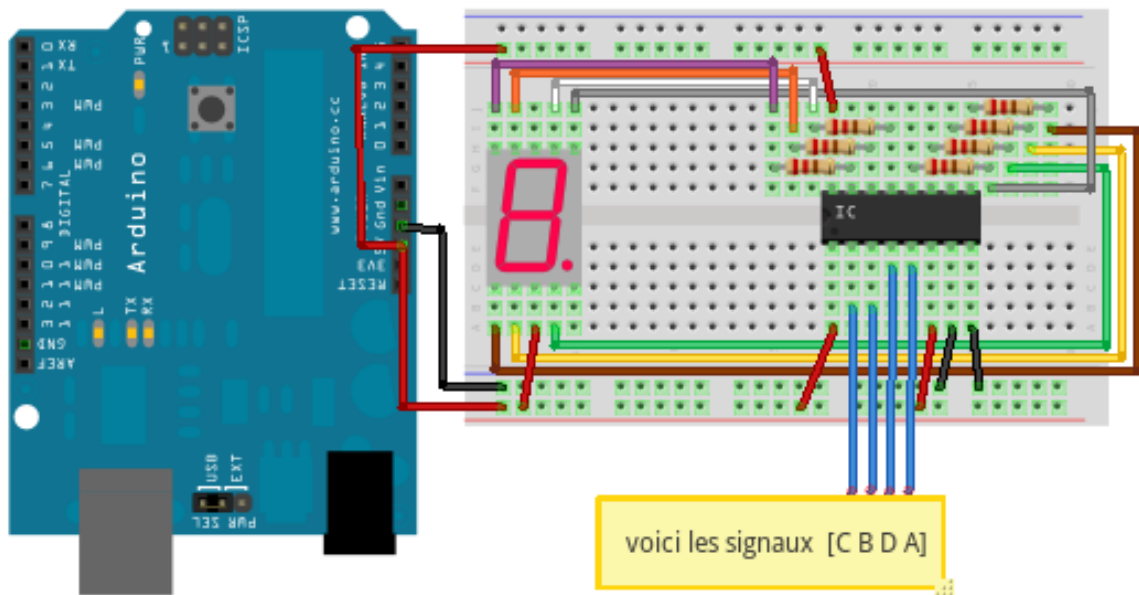
Figure :

Branchement du MC14543B - (source : datasheet)

On voit alors qu'il suffit simplement de brancher la résistance entre le CI et les segments et s'assurer que PH à la bonne valeur et c'est tout ! En titre d'exercice afin de vous permettre de mieux comprendre, je vous propose de changer les états des entrées A, B, C et D du décodeur pour observer ce qu'il affiche. Après avoir réaliser votre schéma, regarder s'il correspond avec celui présent dans cette balise secrète. Cela vous évitera peut-être un mauvais branchement, qui sait ?



[[secret]] |



||

3.5.3.2 L'affichage par alternance

La seconde technique est utilisée dans le cas où l'on veut faire un affichage avec plusieurs afficheurs. Elle utilise le phénomène de **persistance rétinienne**. Pour faire simple, c'est grâce à cela que le cinéma vous paraît fluide. On change une image toutes les 40 ms et votre œil n'a pas le temps de le voir, donc les images semblent s'enchaîner sans transition. Bref... Ici, la même straté-

gie sera utilisée. On va allumer un afficheur un certain temps, puis nous allumerons l'autre en éteignant le premier. Cette action est assez simple à réaliser, mais nécessite l'emploi de deux broche supplémentaires, de quatre autres composants et d'un peu de code. Nous l'étudierons un petit peu plus tard, lorsque nous saurons gérer un afficheur seul.

*[BCD] : Binary Coded Decimal ou Binaire Codé Décimal

3.5.4 Utilisation du décodeur BCD

Nous y sommes, nous allons (enfin) utiliser la carte Arduino pour faire un affichage plus poussé qu'un unique afficheur. Pour cela, nous allons très simplement utiliser le montage précédent composé du décodeur BCD, de l'afficheur 7 segments et bien entendu des résistances de limitations de courant pour les LED de l'afficheur. Je vais vous montrer deux techniques qui peuvent être employées pour faire le programme.

3.5.4.1 Initialisation

Vous avez l'habitude maintenant, nous allons commencer par définir les différentes broches d'entrées/sorties. Pour débiter (et conformément au schéma), nous utiliserons seulement 4 broches, en sorties, correspondantes aux entrées du décodeur 7 segments. Voici le code pouvant traduire cette explication :

```
const int bit_A = 2;
const int bit_B = 3;
const int bit_C = 4;
const int bit_D = 5;

void setup()
{
    // on met les broches en sorties
    pinMode(bit_A, OUTPUT);
    pinMode(bit_B, OUTPUT);
    pinMode(bit_C, OUTPUT);
    pinMode(bit_D, OUTPUT);

    // on commence par écrire le chiffre 0, donc toutes les sorties à l'état bas
    digitalWrite(bit_A, LOW);
    digitalWrite(bit_B, LOW);
    digitalWrite(bit_C, LOW);
    digitalWrite(bit_D, LOW);
}
```

Code : Initialisation des broches

Ce code permet juste de déclarer les quatre broches à utiliser, puis les affectes en sorties. On les met ensuite toutes les quatre à zéro. Maintenant que l'afficheur est prêt, nous allons pouvoir commencer à afficher un chiffre !

3.5.4.2 Programme principal

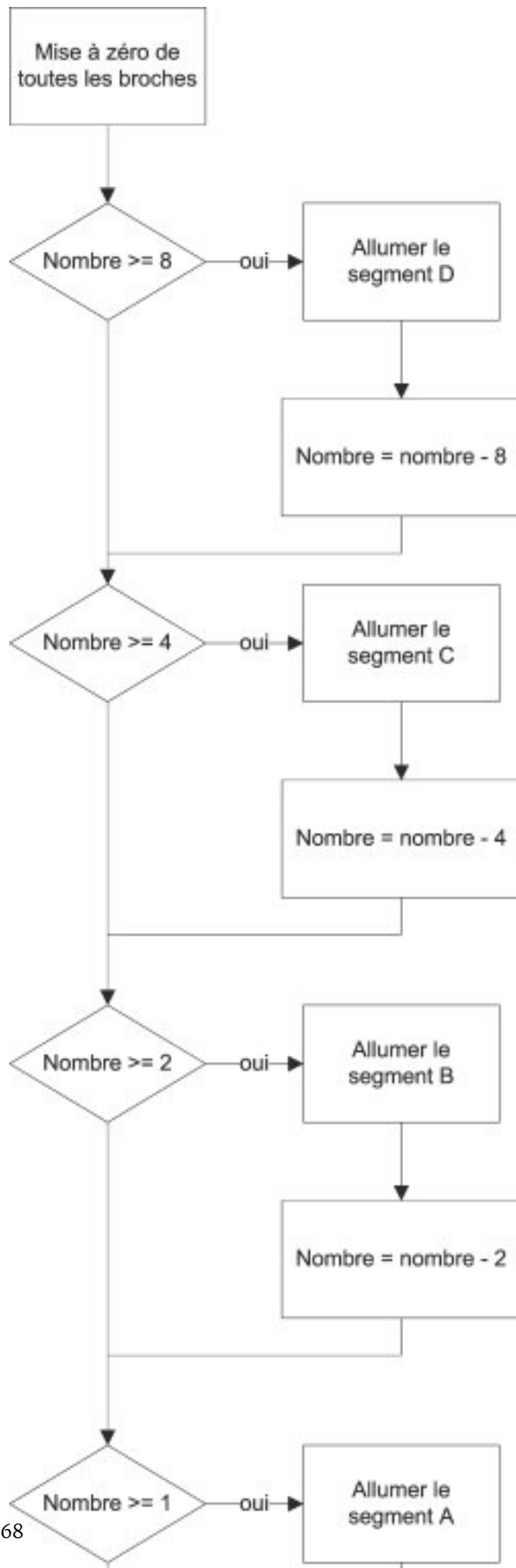
Si tout se passe bien, en ayant la boucle vide pour l'instant vous devriez voir un superbe 0 sur votre afficheur. Nous allons maintenant mettre en place un petit programme pour afficher les nombres de 0 à 9 en les incrémentant (à partir de 0) toutes les secondes. C'est donc un compteur. Pour cela, on va utiliser une boucle, qui comptera de 0 à 9. Dans cette boucle, on exécutera appellera la fonction `afficher()` qui s'occupera donc de l'affichage (belle démonstration de ce qui est une évidence :P ^^).

```
void loop()
{
  char i=0; // variable "compteur"
  for(i=0; i<10; i++)
  {
    afficher(i); // on appel la fonction d'affichage
    delay(1000); // on attend 1 seconde
  }
}
```

Code : Le compteur

3.5.4.3 Fonction d'affichage

Nous touchons maintenant au but ! Il ne nous reste plus qu'à réaliser la fonction d'affichage pour pouvoir convertir notre variable en chiffre sur l'afficheur. Pour cela, il existe différentes solutions. Nous allons en voir ici une qui est assez simple à mettre en œuvre mais qui nécessite de bien être comprise. Dans cette méthode, on va faire des opérations mathématiques (tout de suite c'est moins drôle :lol :) successives pour déterminer quels bits mettre à l'état haut. Rappelez-vous, nous avons quatre broches à notre disposition, avec chacune un poids différent (8, 4, 2 et 1). En combinant ces différentes broches on peut obtenir n'importe quel nombre de 0 à 15. Voici une démarche mathématique envisageable :



On peut coder cette méthode de manière assez simple et direct, en suivant cet organigramme :

```
// fonction écrivant sur un seul afficheur
void afficher(char chiffre)
{
    // on met à zéro tout les bits du décodeur
    digitalWrite(bit_A, LOW);
    digitalWrite(bit_B, LOW);
    digitalWrite(bit_C, LOW);
    digitalWrite(bit_D, LOW);

    // On allume les bits nécessaires
    if(chiffre >= 8)
    {
        digitalWrite(bit_D, HIGH);
        chiffre = chiffre - 8;
    }
    if(chiffre >= 4)
    {
        digitalWrite(bit_C, HIGH);
        chiffre = chiffre - 4;
    }
    if(chiffre >= 2)
    {
        digitalWrite(bit_B, HIGH);
        chiffre = chiffre - 2;
    }
    if(chiffre >= 1)
    {
        digitalWrite(bit_A, HIGH);
        chiffre = chiffre - 1;
    }
}
```

Code : Implémentation de l’affichage

Quelques explications s’imposent... Le code gérant l’affichage réside sur les valeurs binaires des chiffres. Rappelons les valeurs binaires des chiffres :

->

Chiffre	DCBA
0	(0000) ₂
1	(0001) ₂
2	(0010) ₂
3	(0011) ₂
4	(0100) ₂
5	(0101) ₂

Chiffre	DCBA
6	$(0110)_2$
7	$(0111)_2$
8	$(1000)_2$
9	$(1001)_2$

Table 3.4 – La représentation binaires des chiffres

<-

D'après ce tableau, si on veut le chiffre 8, on doit allumer le segment D, car 8 s'écrit $(1000)_2$ ayant pour segment respectif DCBA. Soit D=1, C=0, B=0 et A=0. En suivant cette logique, on arrive à déterminer les entrées du décodeur qui sont à mettre à l'état HAUT ou BAS. D'une manière plus lourde, on aurait pu écrire un code ressemblant à ça :

```
// fonction écrivant sur un seul afficheur
void afficher(char chiffre)
{
    switch(chiffre)
    {
        case 0 :
            digitalWrite(bit_A, LOW);
            digitalWrite(bit_B, LOW);
            digitalWrite(bit_C, LOW);
            digitalWrite(bit_D, LOW);
            break;
        case 1 :
            digitalWrite(bit_A, HIGH);
            digitalWrite(bit_B, LOW);
            digitalWrite(bit_C, LOW);
            digitalWrite(bit_D, LOW);
            break;
        case 2 :
            digitalWrite(bit_A, LOW);
            digitalWrite(bit_B, HIGH);
            digitalWrite(bit_C, LOW);
            digitalWrite(bit_D, LOW);
            break;
        case 3 :
            digitalWrite(bit_A, HIGH);
            digitalWrite(bit_B, HIGH);
            digitalWrite(bit_C, LOW);
            digitalWrite(bit_D, LOW);
            break;
        case 4 :
            digitalWrite(bit_A, LOW);
            digitalWrite(bit_B, LOW);
```

```

        digitalWrite(bit_C, HIGH);
        digitalWrite(bit_D, LOW);
        break;
    case 5 :
        digitalWrite(bit_A, HIGH);
        digitalWrite(bit_B, LOW);
        digitalWrite(bit_C, HIGH);
        digitalWrite(bit_D, LOW);
        break;
    case 6 :
        digitalWrite(bit_A, LOW);
        digitalWrite(bit_B, HIGH);
        digitalWrite(bit_C, HIGH);
        digitalWrite(bit_D, LOW);
        break;
    case 7 :
        digitalWrite(bit_A, HIGH);
        digitalWrite(bit_B, HIGH);
        digitalWrite(bit_C, HIGH);
        digitalWrite(bit_D, LOW);
        break;
    case 8 :
        digitalWrite(bit_A, LOW);
        digitalWrite(bit_B, LOW);
        digitalWrite(bit_C, LOW);
        digitalWrite(bit_D, HIGH);
        break;
    case 9 :
        digitalWrite(bit_A, HIGH);
        digitalWrite(bit_B, LOW);
        digitalWrite(bit_C, LOW);
        digitalWrite(bit_D, HIGH);
        break;
    }
}

```

Code : L'affichage DCBA version longue

Mais, c'est bien trop lourd à écrire. Enfin c'est vous qui voyez. ;)

3.5.5 Utiliser plusieurs afficheurs

Maintenant que nous avons affiché un chiffre sur un seul afficheur, nous allons pouvoir apprendre à en utiliser plusieurs (avec un minimum de composants en plus!). Comme expliqué précédemment, la méthode employée ici va reposer sur le principe de la persistance rétinienne, qui donnera l'impression que les deux afficheurs fonctionnent en *même temps*.

3.5.5.1 Problématique

Nous souhaiterions utiliser deux afficheurs, mais nous ne disposons que de seulement 6 broches sur notre Arduino, le reste des broches étant utilisé pour une autre application. Pour réduire le nombre de broches, on peut d'ores et déjà utiliser un décodeur BCD, ce qui nous ferait 4 broches par afficheurs, soit 8 broches au total. Bon, ce n'est toujours pas ce que l'on veut. Et si on connectait les deux afficheurs ensemble, en parallèle, sur les sorties du décodeur ? Oui mais dans ce cas, on ne pourrait pas afficher des chiffres différents sur chaque afficheur. Tout à l'heure, je vous ai parlé de *commutation*. Oui, la seule solution qui soit envisageable est d'allumer un afficheur et d'éteindre l'autre tout en les connectant ensemble sur le même décodeur. Ainsi un afficheur s'allume, il affiche le chiffre voulu, puis il s'éteint pour que l'autre puisse s'allumer à son tour. Cette opération est en fait un clignotement de chaque afficheur par alternance.

3.5.5.2 Un peu d'électronique...

Pour faire commuter nos deux afficheurs, vous allez avoir besoin d'un nouveau composant, j'ai nommé : le **transistor** !

[[q]] |Transistor ? J'ai entendu dire qu'il y en avait plusieurs milliards dans nos ordinateurs ?

Et c'est tout à fait vrai. Des transistors, il en existe de différents types et pour différentes applications : amplification de courant/tension, commutation, etc. répartis dans plusieurs familles. Bon je ne vais pas faire trop de détails, si vous voulez en savoir plus, allez lire la première partie de **ce chapitre** ([lien à rajouter, en attente de la validation du chapitre en question](#)).

3.5.5.2.1 Le transistor bipolaire : présentation Je le disais, je ne vais pas faire de détails. On va voir comment fonctionne un transistor bipolaire selon les besoins de notre application, à savoir, faire commuter les afficheurs. Un transistor, cela ressemble à ça :



Figure : Photo d'un transistor -

(CC-BY-SA, [Marvelshine](#))

Pour notre application, nous allons utiliser des **transistors bipolaires**. Je vais vous expliquer comment cela fonctionne. Déjà, vous pouvez observer qu'un transistor possède trois pattes. Cela n'est pas de la moindre importance, au contraire il s'agit là d'une chose essentielle ! En fait, le transistor bipolaire a une *broche d'entrée* (**collecteur**), une *broche de sortie* (**émetteur**) et une *broche de commande* (**base**). Son symbole est le suivant :

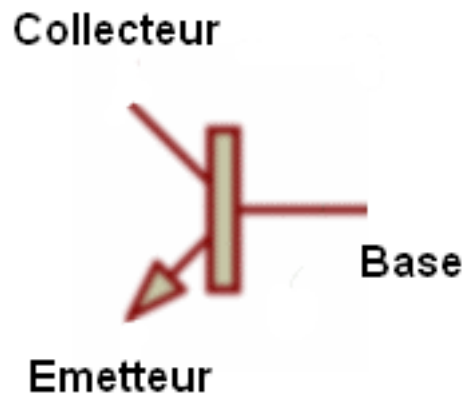


Figure 3.30 – Symbole du transistor bipolaire

[[i]] |Ce symbole est celui d'un transistor bipolaire de **type NPN**.

Il en existe qui sont de **type PNP**, mais ils sont beaucoup moins utilisés que les NPN. Quoiqu'il en soit, nous n'utiliserons que des transistors NPN dans ce chapitre.

3.5.5.2.2 Fonctionnement en commutation du transistor bipolaire Pour faire simple, le **transistor bipolaire NPN** (c'est la dernière fois que je précise ce point) est un **interrupteur commandé en courant**.

[[a]] |Ceci est une présentation très vulgarisée et simplifiée sur le transistor pour l'utilisation que nous en ferons ici. |Les usages et possibilités des transistors sont très nombreux et ils mériteraient un grand livre à eux seuls ! |Si vous voulez plus d'informations, rendez-vous sur le cours sur l'électronique ou approfondissez en cherchant des tutoriels sur le web. ;)

C'est tout ce qu'il faut savoir, pour ce qui est du fonctionnement. Après, on va voir ensemble comment l'utiliser et sans le faire griller! ^^

3.5.5.2.3 Utilisation générale On peut utiliser notre transistor de deux manières différentes (pour notre application toujours, mais on peut bien évidemment utiliser le transistor avec beaucoup plus de flexibilités). A commencer par le câblage :

Dans le cas présent, le collecteur (qui est l'entrée du transistor) se trouve être après l'ampoule, elle-même connectée à l'alimentation. L'émetteur (broche où il y a la flèche) est relié à la masse du montage. Cette disposition est "universelle", on ne peut pas inverser le sens de ces broches et mettre le collecteur à la place de l'émetteur et vice versa. Sans quoi, le montage ne fonctionnerait pas. Pour le moment, l'ampoule est éteinte car le transistor ne conduit pas. On dit qu'il est **bloqué** et empêche donc le courant I_C de circuler à travers l'ampoule. Soit $I_C = 0$ car $I_B = 0$. A présent, appuyons sur l'interrupteur :

Que se passe-t-il ? Eh bien la base du transistor, qui était jusqu'à présent "en l'air", est parcourue par un courant électrique. Cette cause à pour conséquence de rendre le transistor **passant** ou **saturé** et permet au courant de s'établir à travers l'ampoule. Soit $I_C \neq 0$ car $I_B \neq 0$.

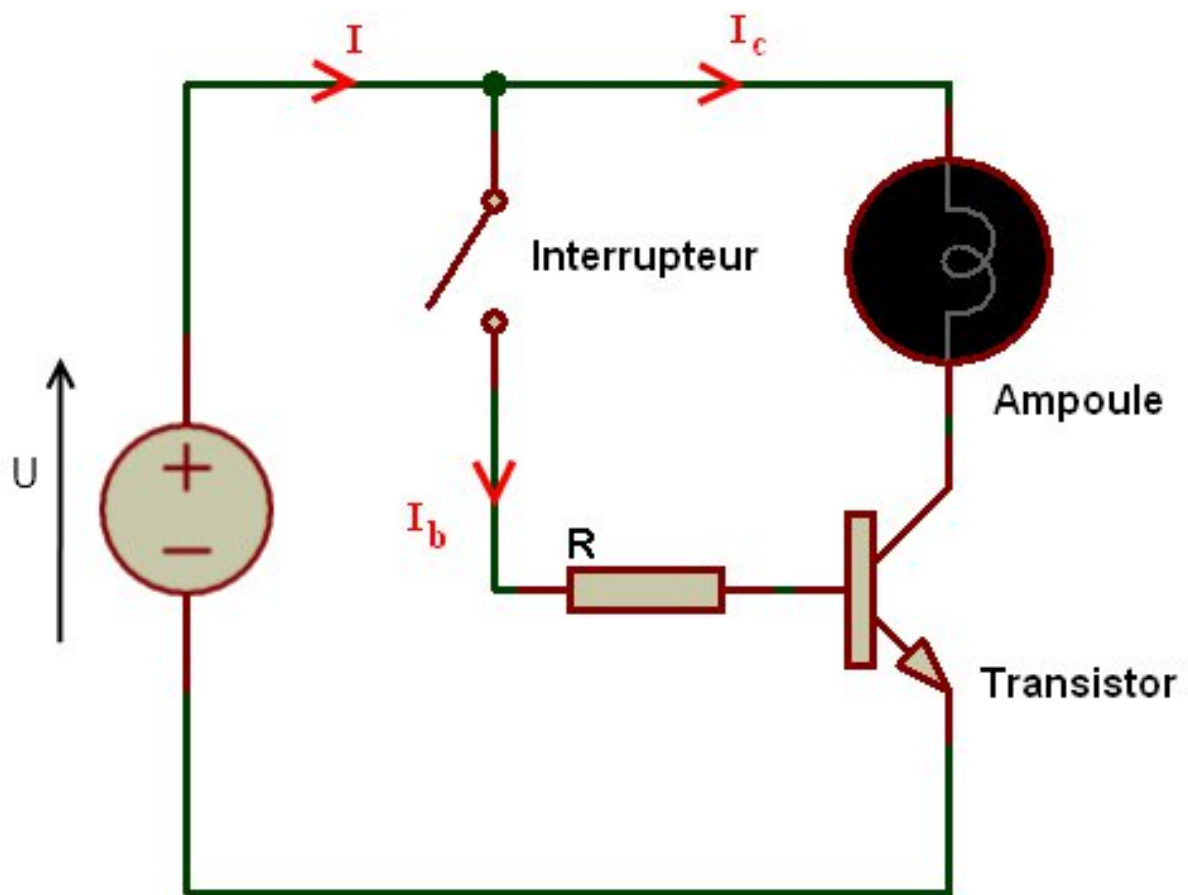


Figure 3.31 – Câblage du transistor en commutation

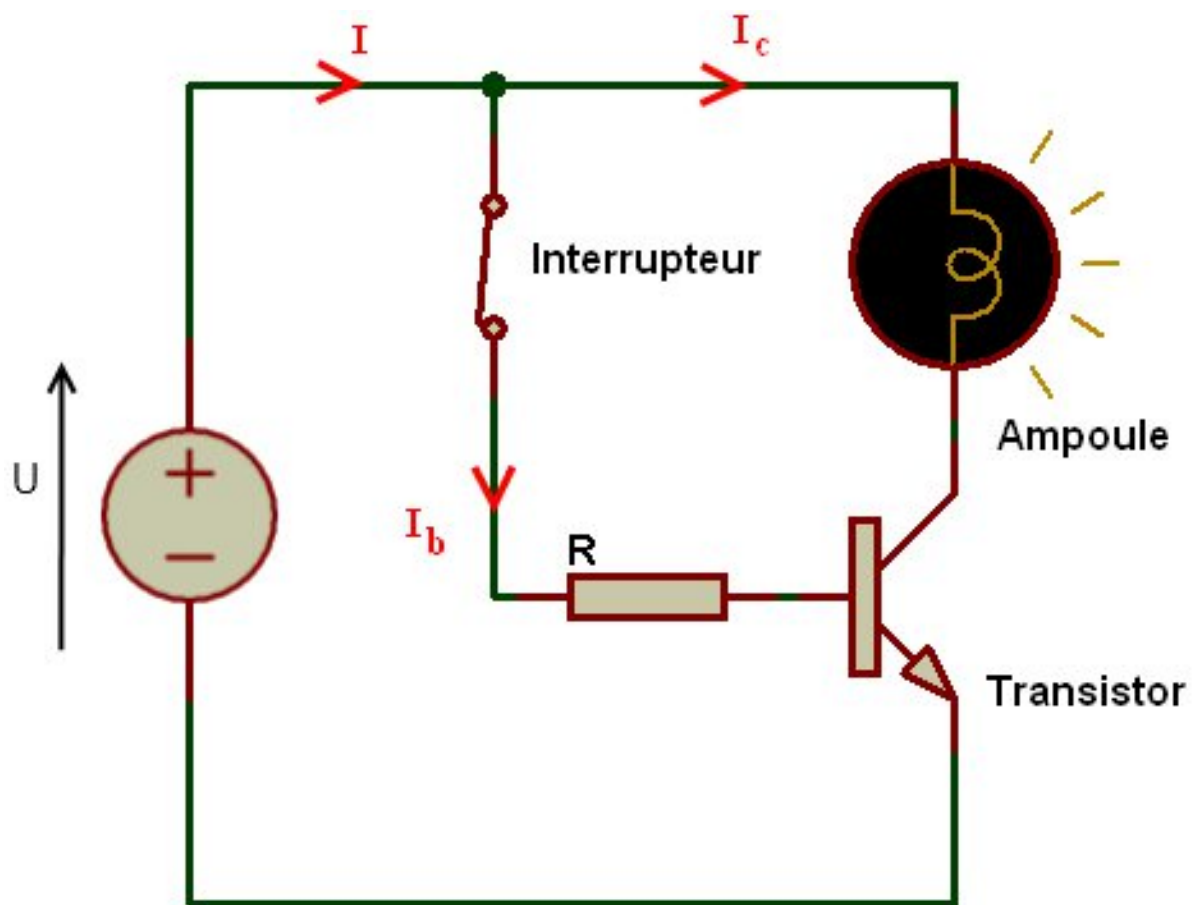


Figure 3.32 – Allumage de la lampe

[[i]] |La résistance sur la base du transistor permet de le protéger des courants trop forts. |Plus la résistance est de faible valeur, plus l'ampoule sera lumineuse. |A l'inverse, une résistance trop forte sur la base du transistor pourra l'empêcher de conduire et de faire s'allumer l'ampoule. |Rassurez-vous, je vous donnerais les valeurs de résistances à utiliser. ;)

3.5.5.2.4 Utilisation avec nos afficheurs Voyons un peu comment on va pouvoir utiliser ce transistor avec notre Arduino. La carte Arduino est en fait le générateur de tension (schéma précédent) du montage. Elle va définir si sa sortie est de 0V (transistor bloqué) ou de 5V (transistor saturé). Ainsi, on va pouvoir allumer ou éteindre les afficheurs. Voilà le modèle équivalent de la carte Arduino et de la commande de l'afficheur :

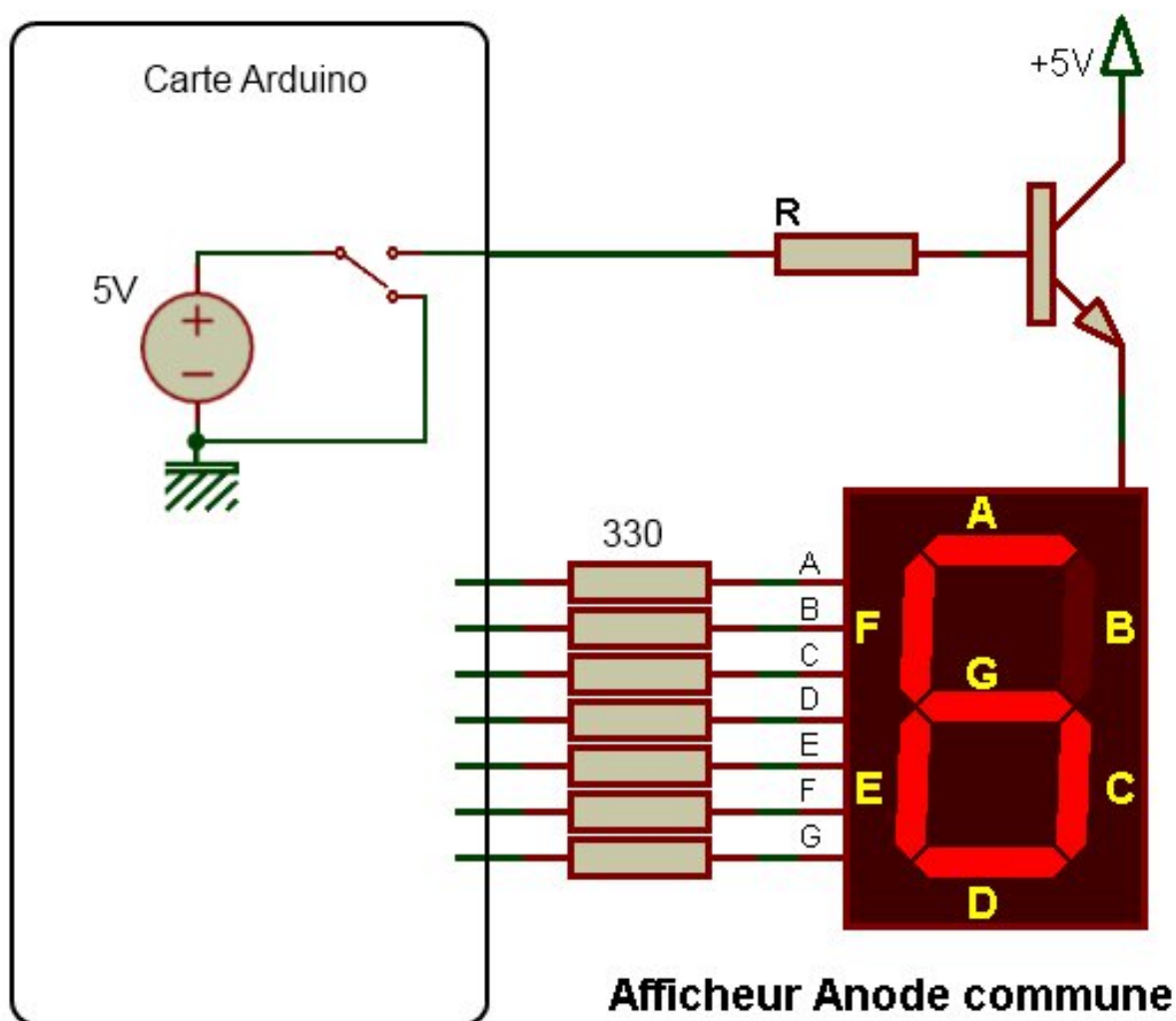


Figure 3.33 – Montage de la commande

La carte Arduino va soit mettre à la masse la base du transistor, soit la mettre à +5V. Dans le premier cas, il sera bloqué et l'afficheur sera éteint, dans le second il sera saturé et l'afficheur allumé. Il en est de même pour chaque broche de l'afficheur. Elles seront au +5V ou à la masse selon la configuration que l'on aura définie dans le programme.

3.5.5.2.5 Schéma final Et comme vous l'attendez sûrement depuis tout à l'heure, voici le schéma tant attendu (nous verrons juste après comment programmer ce nouveau montage)!

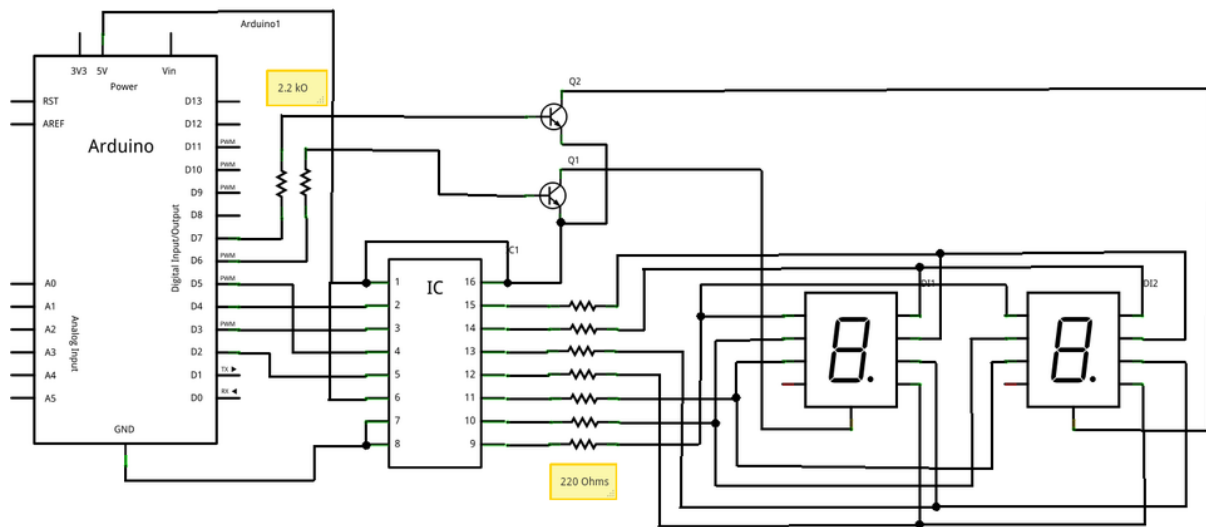


Figure 3.34 – 2*7 segments - Schéma

3.5.5.3 Quelques détails techniques

- Dans notre cas (et je vous passe les détails vraiment techniques et calculatoires), la résistance sur la base du transistor sera de $2.2k\Omega$ (si vous n'avez pas cette valeur, elle pourra être de $3.3k\Omega$, ou encore de $3.9k\Omega$, voir même de $4.7k\Omega$).
- Les transistors seront des transistors bipolaires NPN de référence 2N2222, ou bien un équivalent qui est le BC547. Il en faudra deux donc.
- Le décodeur BCD est le même que précédemment (ou équivalent).

Et avec tout ça, on est prêt pour programmer! :)

3.5.5.4 ...et de programmation

Nous utilisons deux nouvelles broches servant à piloter chacun des interrupteurs (transistors). Chacune de ces broches doivent donc être déclarées en global (pour son numéro) puis régler comme sortie. Ensuite, il ne vous restera plus qu'à alimenter chacun des transistors au bon moment pour allumer l'afficheur souhaité. En synchronisant l'allumage avec la valeur envoyé au décodeur, vous afficherez les nombres souhaités comme bon vous semble. Voici un exemple de code complet, de la fonction `setup()` jusqu'à la fonction d'affichage. Ce code est commenté et vous ne devriez donc avoir aucun mal à le comprendre! Ce programme est un compteur sur 2 segments, il compte donc de 0 à 99 et recommence au début dès qu'il a atteint 99. La vidéo se trouve juste après ce code.

```
// définition des broches du décodeur 7 segments
// (vous pouvez changer les numéros si vous voulez)
const int bit_A = 2;
const int bit_B = 3;
const int bit_C = 4;
```

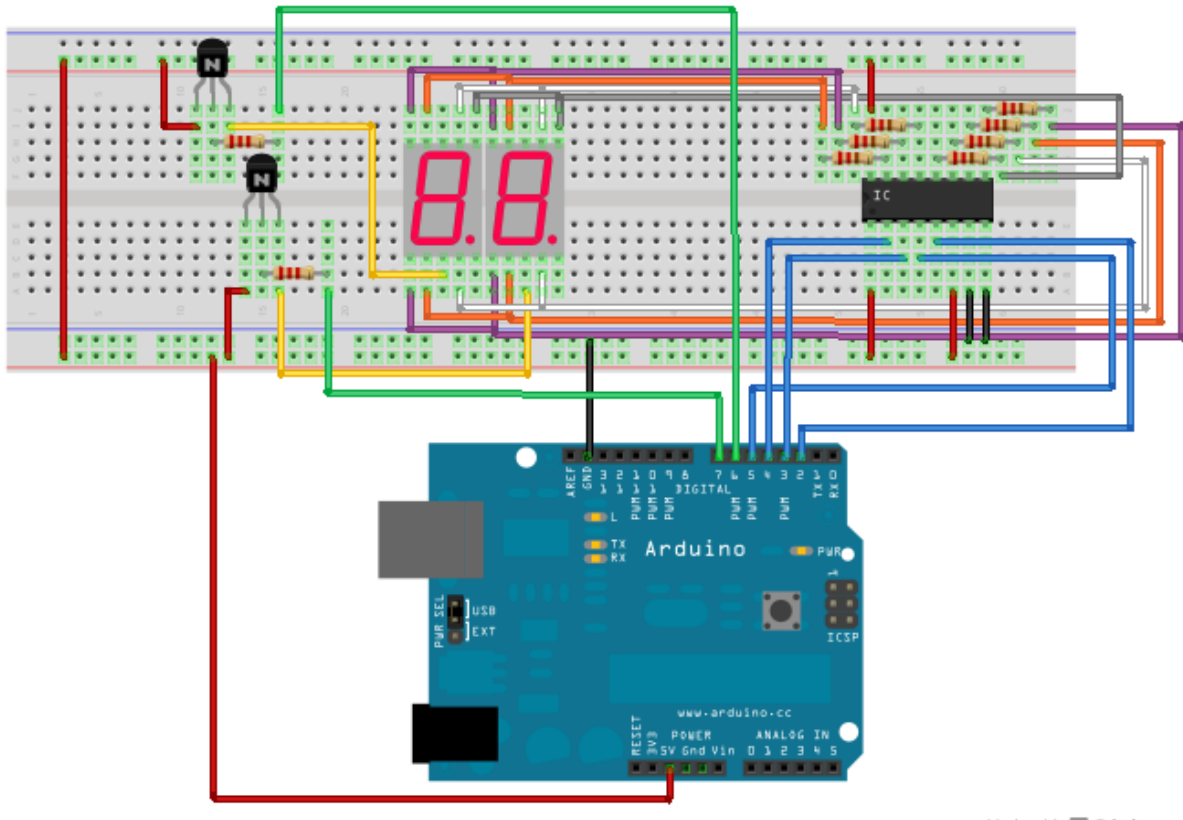


Figure 3.35 – 2*7 segments – Montage

```

const int bit_D = 5;

// définitions des broches des transistors pour chaque afficheur
const int alim_dizaine = 6; // les dizaines
const int alim_unite = 7; // les unites

void setup()
{
    // Les broches sont toutes des sorties
    pinMode(bit_A, OUTPUT);
    pinMode(bit_B, OUTPUT);
    pinMode(bit_C, OUTPUT);
    pinMode(bit_D, OUTPUT);
    pinMode(alim_dizaine, OUTPUT);
    pinMode(alim_unite, OUTPUT);

    // Les broches sont toutes mises à l'état bas
    digitalWrite(bit_A, LOW);
    digitalWrite(bit_B, LOW);
    digitalWrite(bit_C, LOW);
    digitalWrite(bit_D, LOW);
    digitalWrite(alim_dizaine, LOW);
    digitalWrite(alim_unite, LOW);
}

```

```

}

void loop() // fonction principale
{
    // boucle qui permet de compter de 0 à 99 (= 100 valeurs)
    for(char i = 0; i<100; i++)
    {
        // appel de la fonction affichage avec envoi du nombre à afficher
        afficher_nombre(i);
    }
}

// fonction permettant d'afficher un nombre sur deux afficheurs
void afficher_nombre(char nombre)
{
    long temps; // variable utilisée pour savoir le temps écoulé...
    char unite = 0, dizaine = 0; // variable pour chaque afficheur

    if(nombre > 9) // si le nombre reçu dépasse 9
    {
        dizaine = nombre / 10; // on récupère les dizaines
    }

    unite = nombre - (dizaine*10); // on récupère les unités

    temps = millis(); // on récupère le temps courant

    // tant qu'on a pas affiché ce chiffre pendant au moins 500 millisecondes
    // permet donc de pouvoir lire le nombre affiché
    while((millis()-temps) < 500)
    {
        // on affiche le nombre

        // d'abord les dizaines pendant 10 ms

        // le transistor de l'afficheur des dizaines est saturé,
        // donc l'afficheur est allumé
        digitalWrite(alim_dizaine, HIGH);
        // on appelle la fonction qui permet d'afficher le chiffre dizaine
        afficher(dizaine);
        // l'autre transistor est bloqué et l'afficheur éteint
        digitalWrite(alim_unite, LOW);
        delay(10);

        // puis les unités pendant 10 ms

        // on éteint le transistor allumé
        digitalWrite(alim_dizaine, LOW);
    }
}

```

```
        // on appelle la fonction qui permet d'afficher le chiffre unité
        afficher(unite);
        // et on allume l'autre
        digitalWrite(alim_unite, HIGH);
        delay(10);
    }
}

// fonction écrivant sur un seul afficheur
// on utilise le même principe que vu plus haut
void afficher(char chiffre)
{
    digitalWrite(bit_A, LOW);
    digitalWrite(bit_B, LOW);
    digitalWrite(bit_C, LOW);
    digitalWrite(bit_D, LOW);

    if(chiffre >= 8)
    {
        digitalWrite(bit_D, HIGH);
        chiffre = chiffre - 8;
    }
    if(chiffre >= 4)
    {
        digitalWrite(bit_C, HIGH);
        chiffre = chiffre - 4;
    }
    if(chiffre >= 2)
    {
        digitalWrite(bit_B, HIGH);
        chiffre = chiffre - 2;
    }
    if(chiffre >= 1)
    {
        digitalWrite(bit_A, HIGH);
        chiffre = chiffre - 1;
    }
}
```

Code : Le compteur de 0 à 99

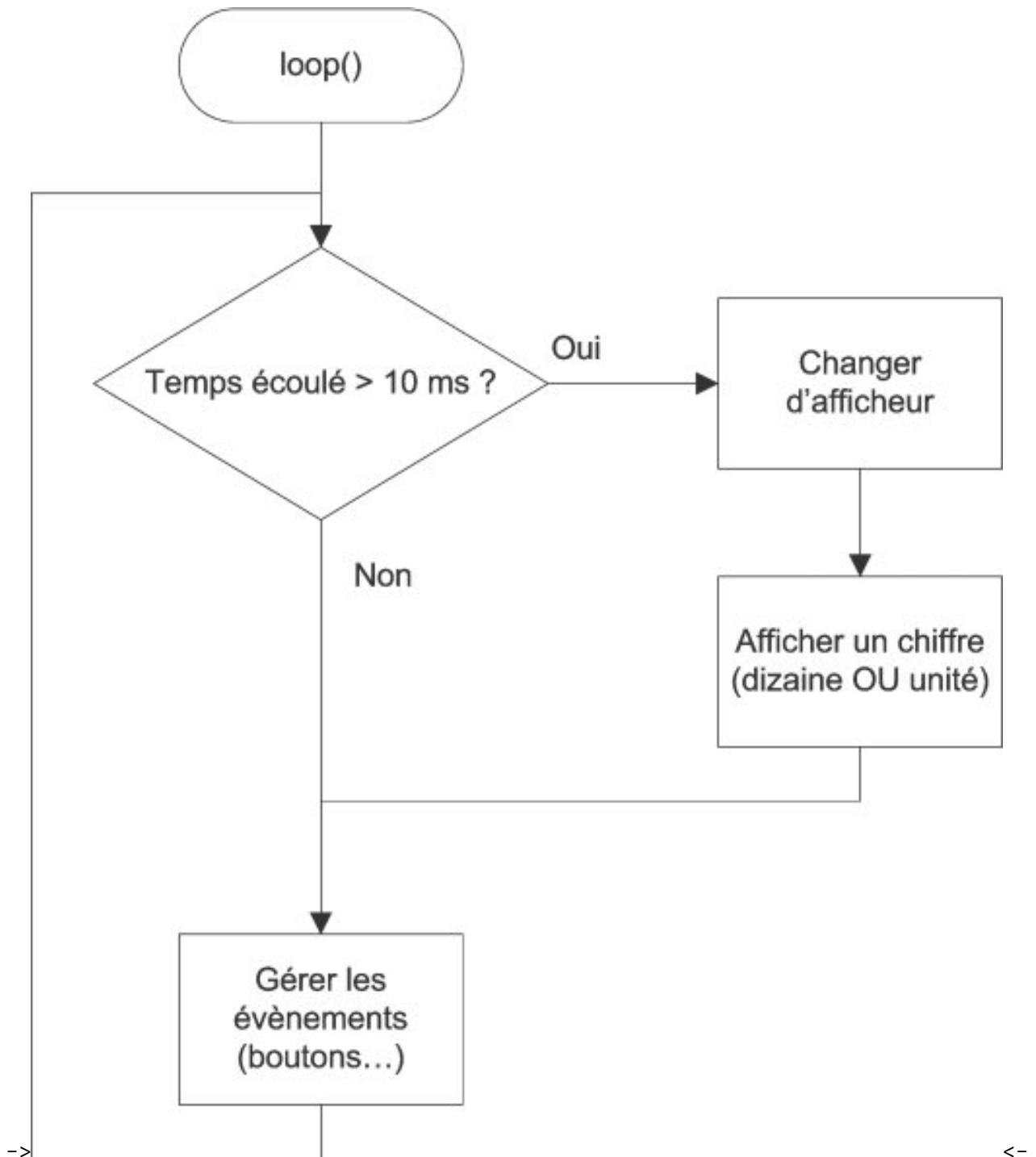
Voilà donc la vidéo présentant le résultat final :

->!(https://www.youtube.com/watch?v=zgvV25s_ilQ)<-

3.5.6 Contraintes des évènements

Comme vous l'avez vu juste avant, afficher de manière alternative n'est pas trop difficile. Cependant, vous avez sûrement remarqué, nous avons utilisé des fonctions bloquantes (delay). Si jamais

un évènement devait arriver pendant ce temps, nous aurions beaucoup de chance de le rater car il pourrait arriver “pendant” un délai d’attente pour l’affichage. Pour parer à cela, je vais maintenant vous expliquer une autre méthode, préférable, pour faire de l’affichage. Elle s’appuiera sur l’utilisation de la fonction `millis()`, qui nous permettra de générer une boucle de rafraîchissement de l’affichage. Voici un organigramme qui explique le principe :



Comme vous pouvez le voir, il n’y a plus de fonction qui “attend”. Tout se passe de manière continue, sans qu’il n’y ai jamais de pause. Ainsi, aucun évènement ne sera raté (en théorie, un évènement très très très rapide pourra toujours passer inaperçu). Voici un exemple de programmation de la boucle principal (suivi de ses fonctions annexes) :

```
bool afficheur = false; // variable pour le choix de l'afficheur

// --- setup() ---

void loop()
{
    // gestion du rafraichissement
    // si ça fait plus de 10 ms qu'on affiche,
    // on change de 7 segments (alternance unité <-> dizaine)
    if((millis() - temps) > 10)
    {
        // on inverse la valeur de "afficheur"
        // pour changer d'afficheur (unité ou dizaine)
        afficheur = !afficheur;
        // on affiche la valeur sur l'afficheur
        // afficheur : true->dizaines, false->unités
        afficher_nombre(valeur, afficheur);
        temps = millis(); // on met à jour le temps
    }

    // ici, on peut traiter les évènements (bouton...)
}

// fonction permettant d'afficher un nombre
// elle affiche soit les dizaines soit les unités
void afficher_nombre(char nombre, bool afficheur)
{
    char unite = 0, dizaine = 0;
    if(nombre > 9)
        dizaine = nombre / 10; // on recupere les dizaines
    unite = nombre - (dizaine*10); // on recupere les unités

    // si "
    if(afficheur)
    {
        // on affiche les dizaines
        digitalWrite(alim_unite, LOW);
        afficher(dizaine);
        digitalWrite(alim_dizaine, HIGH);
    }
    else // égal à : else if(!afficheur)
    {
        // on affiche les unités
        digitalWrite(alim_dizaine, LOW);
        afficher(unite);
        digitalWrite(alim_unite, HIGH);
    }
}
```

```
// fonction écrivant sur un seul afficheur
void afficher(char chiffre)
{
    digitalWrite(bit_A, LOW);
    digitalWrite(bit_B, LOW);
    digitalWrite(bit_C, LOW);
    digitalWrite(bit_D, LOW);

    if(chiffre >= 8)
    {
        digitalWrite(bit_D, HIGH);
        chiffre = chiffre - 8;
    }
    if(chiffre >= 4)
    {
        digitalWrite(bit_C, HIGH);
        chiffre = chiffre - 4;
    }
    if(chiffre >= 2)
    {
        digitalWrite(bit_B, HIGH);
        chiffre = chiffre - 2;
    }
    if(chiffre >= 1)
    {
        digitalWrite(bit_A, HIGH);
        chiffre = chiffre - 1;
    }
}
```

Code : L’affichage de deux chiffres en utilisant millis

[[i]] Si vous voulez tester le phénomène de persistance rétinienne, vous pouvez changer le temps de la boucle de rafraîchissement (ligne 9). Si vous l’augmenter, vous commencerez à voir les afficheurs clignoter. En mettant une valeur d’un peu moins de une seconde vous verrez les afficheurs s’illuminer l’un après l’autre.

Ce chapitre vous a appris à utiliser un nouveau moyen pour afficher des informations avec votre carte Arduino. L’afficheur peut sembler peu utilisé mais en fait de nombreuses applications existe! (chronomètre, réveil, horloge, compteur de passage, afficheur de score, etc.). Par exemple, il pourra vous servir pour déboguer votre code et afficher la valeur des variables souhaitées...

3.6 [TP] Parking

Ça y est, une page se tourne avec l'acquisition de nombreuses connaissances de base. C'est donc l'occasion idéale pour faire un (gros :diable :) TP qui utilisera l'ensemble de vos connaissances durement acquises. J'aime utiliser les situations de la vie réelle, je vais donc en prendre une pour ce sujet. Je vous propose de réaliser la gestion d'un parking souterrain... RDV aux consignes pour les détails.

3.6.1 Consigne

Après tant de connaissances chacune séparée dans son coin, nous allons pouvoir mettre en œuvre tout ce petit monde dans un TP traitant sur un sujet de la vie courante : les **parkings!

3.6.1.0.1 Histoire Le maire de zCity a décidé de rentabiliser le parking communal d'une capacité de 99 places (pas une de plus ni de moins). En effet, chaque jour des centaines de zTouristes viennent se promener en voiture et ont besoin de la garer quelque part. Le parking, n'étant pour le moment pas rentable, servira à financer l'entretien de la ville. Pour cela, il faut rajouter au parking existant un afficheur permettant de savoir le nombre de places disponibles en temps réel (le système de paiement du parking ne sera pas traité). Il dispose aussi dans la ville des lumières vertes et rouges signalant un parking complet ou non. Enfin, l'entrée du parking est équipée de deux barrières (une pour l'entrée et l'autre pour la sortie). Chaque entrée de voiture ou sortie génère un signal pour la gestion du nombre de places. Le maire vous a choisi pour vos compétences, votre esprit de créativité et il sait que vous aimez les défis. Vous acceptez évidemment en lui promettant de réussir dans les plus brefs délais!

3.6.1.0.2 Matériel Pour mener à bien ce TP voici la liste des courses conseillée :

- Une carte Arduino (évidemment)
- 2 LEDs avec leur résistance de limitations de courant (habituellement 330 Ohms) -> Elles symbolisent les témoins lumineux disposés dans la ville
- 2 boutons (avec 2 résistances de 10 kOhms et 2 condensateurs de 10 nF) -> Ce sont les "capteurs" d'entrée et de sortie.
- 2 afficheurs 7 segments -> pour afficher le nombre de places disponibles
- 1 décodeur 4 bits vers 7 segments
- 7 résistances de 330 Ohms (pour les 7 segments)
- Une breadboard pour assembler le tout
- Un paquet de fils
- Votre cerveau et quelques doigts...

Voici une vidéo pour vous montrer le résultat attendu par le maire :

->!(<https://www.youtube.com/watch?v=pZWYGTFN7nM>)<-

->Bon courage!<-

3.6.2 Correction !

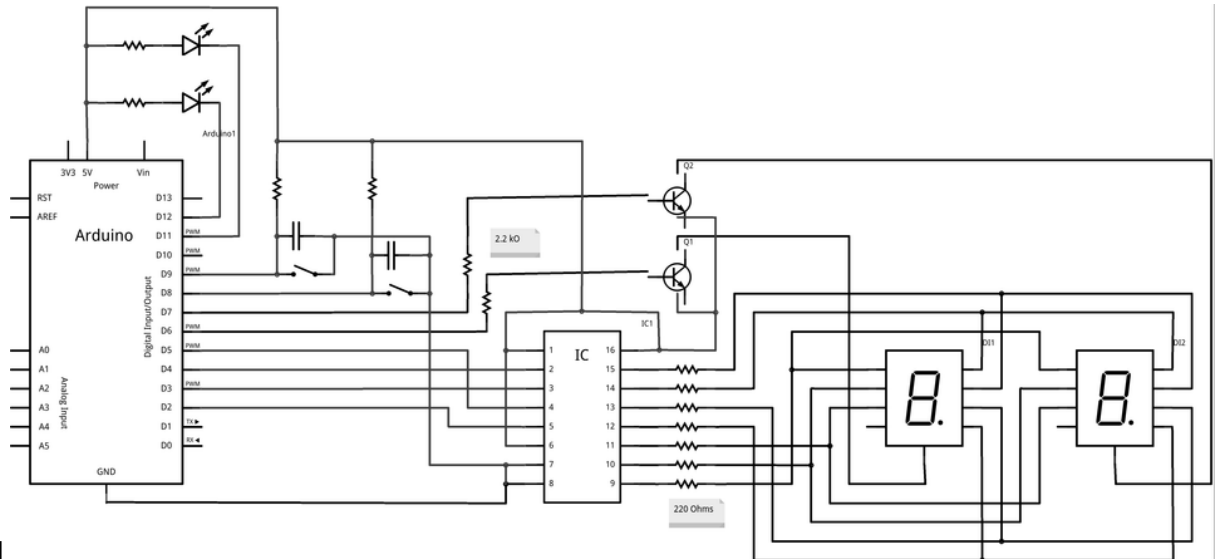
J'espère que tout s'est bien passé pour vous et que le maire sera content de votre travail. Voilà maintenant une correction (parmi tant d'autres, comme souvent en programmation et en élec-

tronique). Nous commencerons par voir le schéma électronique, puis ensuite nous rentrerons dans le code.

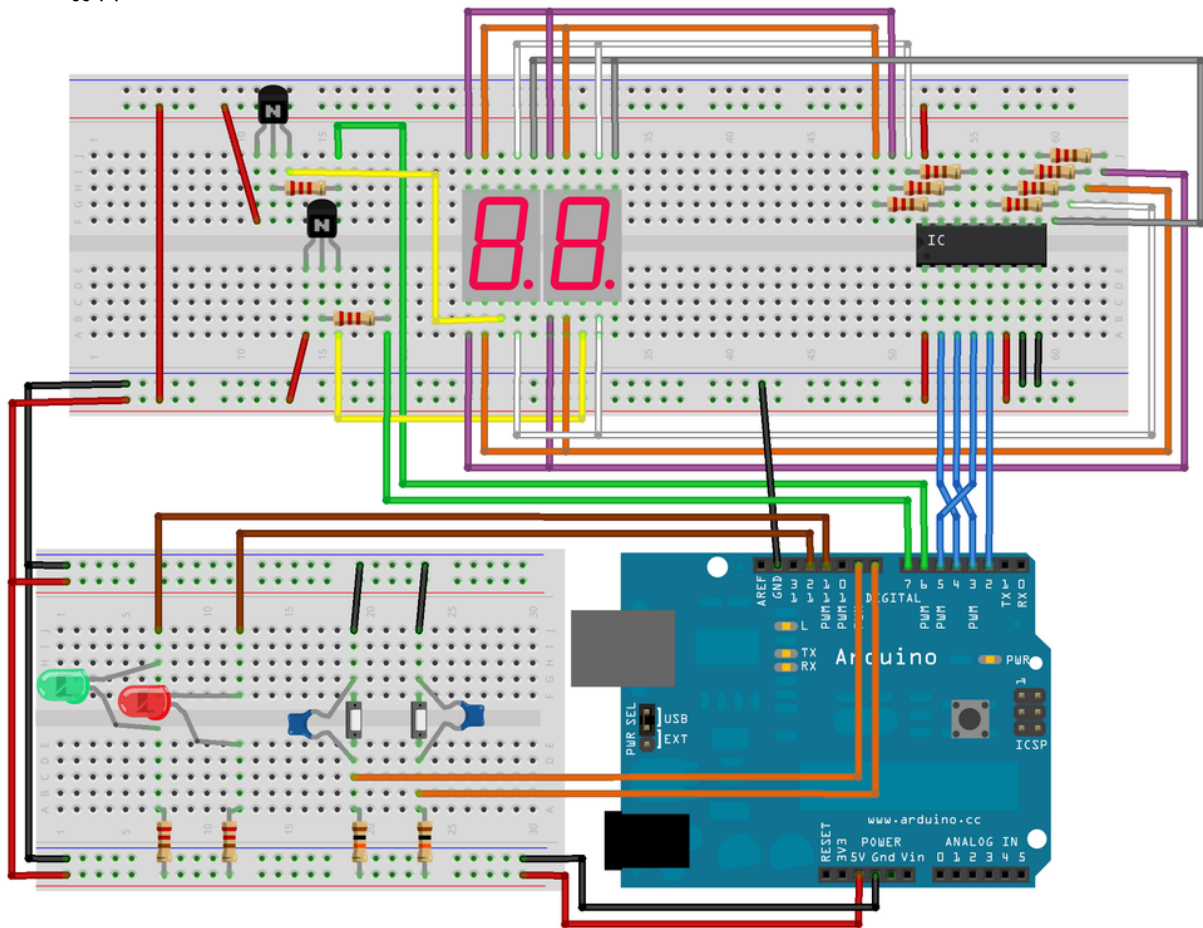
3.6.2.1 Montage

Le montage électronique est la base de ce qui va nous servir pour réaliser le système. Une fois qu'il est terminé on pourra l'utiliser grâce aux entrées/sorties de la carte Arduino et lui faire faire pleins de choses. Mais ça, vous le savez déjà. Alors ici pas de grand discours, il "suffit" de reprendre les différents blocs vus un par un dans les chapitres précédents et de faire le montage de façon simple.

3.6.2.1.1 Schéma Je vous montre le schéma que j'ai réalisé, il n'est pas absolu et peut différer selon ce que vous avez fait, mais il reprend essentiellement tous les "blocs" (ou mini montages électroniques) que l'on a vus dans les précédents chapitres, en les assemblant de façon logique et ordonnée :



[[secret]] | |



| |

3.6.2.1.2 Procédure de montage Voici l'ordre que j'ai suivi pour réaliser le montage :

- Débrancher la carte Arduino !
- Mettre les boutons
 - Mettre les résistances de pull-up
 - Puis les condensateurs de filtrage
 - Et tirez des fils de signaux jusqu'à la carte Arduino
- Enfin, vérifiez la position des alimentations (+5V et masse)

- Mettre les LEDs rouge et verte avec leur résistance de limitation de courant et un fil vers Arduino
- Mettre les décodeurs
 - Relier les fils ABCD à Arduino
 - Mettre au +5V ou à la masse les signaux de commandes du décodeur
 - Mettre les résistances de limitations de courant des 7 segments
 - Enfin, vérifier la position des alimentations (+5V et masse)
- Puis mettre les afficheurs -> les relier entre le décodeur et leurs segments) -> les connecter au +5V
- Amener du +5V et la masse sur la breadboard

Ce étant terminé, la maquette est fin prête à être utilisée ! Évidemment, cela fait un montage (un peu) plus complet que les précédents !

3.6.2.2 Programme

Nous allons maintenant voir une solution de programme pour le problème de départ. La vôtre sera peut-être (voire sûrement) différente, et ce n'est pas grave, un problème n'exige pas une solution unique. Je n'ai peut-être même pas la meilleure solution ! (mais ça m'étonnerait :P :ninja :)

3.6.2.2.1 Les variables utiles et déclarations Tout d'abord, nous allons voir les variables globales que nous allons utiliser ainsi que les déclarations utiles à faire. Pour ma part, j'utilise six variables globales. Vous reconnaîtrez la plupart d'entre elles car elles viennent des chapitres précédents.

- Celles pour stocker l'état des boutons un coup sur l'autre et une pour le stocker de manière courante
- Un char stockant le nombre de places disponibles dans le parking
- Un booléen désignant l'afficheur utilisé en dernier
- Un long stockant l'information de temps pour le rafraichissement de l'affichage

Voici ces différentes variables commentées. `[[secret]] | cpp | // les broches du décodeur 7 segments | const int bit_A = 2; | const int bit_B = 3; | const int bit_C = 4; | const int bit_D = 5; | // les broches des transistors pour l'afficheur des dizaines et des unités | const int alim_dizaine = 6; | const int alim_unite = 7; | // les broches des boutons | const int btn_entree = 8; | const int btn_sortie = 9; | // les leds de signalements | const int led_rouge = 12; | const int led_verte = 11; | // les mémoires d'état des boutons | int mem_entree = HIGH; | int mem_sortie = HIGH; | int etat = HIGH; // variable stockant l'état courant d'un bouton | char place_dispo = 99; // contenu des places dispos | bool afficheur = false; | long temps; | | Code : Les variables et constantes`

3.6.2.2.2 L'initialisation de la fonction setup() Je ne vais pas faire un long baratin sur cette partie car je pense que vous serez en mesure de tout comprendre très facilement car il n'y a vraiment rien d'original par rapport à tout ce que l'on a fait avant (réglages des entrées/sorties et de leurs niveaux).

```

[[secret]] | cpp | void setup() | { | // Les broches sont toutes des sorties
(sauf les boutons) | pinMode(bit_A, OUTPUT); | pinMode(bit_B, OUTPUT); |
pinMode(bit_C, OUTPUT); | pinMode(bit_D, OUTPUT); | pinMode(alim_dizaine,
OUTPUT); | pinMode(alim_unite, OUTPUT); | pinMode(led_rouge, OUTPUT); |
pinMode(led_verte, OUTPUT); | | pinMode(btn_entree, INPUT); | pinMode(btn_sortie,
INPUT); | | // Les broches sont toutes mise à l'état bas (sauf led rouge
éteinte) | digitalWrite(bit_A, LOW); | digitalWrite(bit_B, LOW); | digitalWrite(bit_C,
LOW); | digitalWrite(bit_D, LOW); | digitalWrite(alim_dizaine, LOW); |
digitalWrite(alim_unite, LOW); | digitalWrite(led_rouge, HIGH); | digitalWrite(led_ver
LOW); // vert par défaut | // rappel: dans cette configuration, la LED est
éteinte à l'état HIGH | | temps = millis(); // enregistre "l'heure" | } | |
Code : L'initialisation

```

3.6.2.2.3 La boucle principale (loop) Ici se trouve la partie la plus compliquée du TP. En effet, elle doit s'occuper de gérer d'une part une boucle de rafraichissement de l'allumage des afficheurs 7 segments et d'autre part gérer les évènements. Rappelons-nous de l'organigramme vu dans la dernière partie sur les 7 segments :

Dans notre application, la gestion d'évènements sera "une voiture rentre-t/sort-elle du parking?" qui sera symbolisée par un appui sur un bouton. Ensuite, il faudra aussi prendre en compte l'affichage de la disponibilité sur les LEDs selon si le parking est complet ou non... Voici une manière de coder tout cela : [[secret]]

```

| | cpp | | void loop() | {
| // si ca fait plus de 10 ms qu'on affiche, on change de 7 segments |
if((millis() - temps) > 10) | { | // on inverse la valeur de "afficheur" |
// pour changer d'afficheur (unité ou dizaine) | afficheur = !afficheur; |
// on affiche | afficher_nombre(place_dispo, afficheur); | temps = millis();
// on met à jour le temps | } | | // on test maintenant si les boutons ont
subi un appui (ou pas) | // d'abord le bouton plus puis le moins | etat
= digitalRead(btn_entree); | if((etat != mem_entree) && (etat == LOW)) |
place_dispo += 1; | mem_entree = etat; // on enregistre l'état du bouton
pour le tour suivant | | // et maintenant pareil pour le bouton qui décrémente
| etat = digitalRead(btn_sortie); | if((etat != mem_sortie) && (etat ==
LOW)) | place_dispo -= 1; | mem_sortie = etat; // on enregistre l'état
du bouton pour le tour suivant | | // on applique des limites au nombre
pour ne pas dépasser 99 ou 0 | if(place_dispo > 99) | place_dispo = 99;
| if(place_dispo < 0) | place_dispo = 0; | | // on met à jour l'état des
leds | // on commence par les éteindre | digitalWrite(led_verte, HIGH); |
digitalWrite(led_rouge, HIGH); | if(place_dispo == 0) // s'il n'y a plus
de place | digitalWrite(led_rouge, LOW); | else | digitalWrite(led_verte,
LOW); | } |

```

Dans les lignes 4 à 11, on retrouve la gestion du rafraichissement des 7 segments. Ensuite, on s'occupe de réceptionner les évènements en faisant un test par bouton pour savoir si son état a changé et s'il est à l'état bas. Enfin, on va borner le nombre de places et faire l'affichage sur les LED en conséquence. Vous voyez, ce n'était pas si difficile en fait ! Si, un peu quand même, non ? ^^ Il ne reste maintenant plus qu'à faire les fonctions d'affichages.

3.6.2.2.4 Les fonctions d'affichages Là encore, je ne vais pas faire de grand discours puisque ces fonctions sont exactement les mêmes que celles réalisées dans la partie concernant

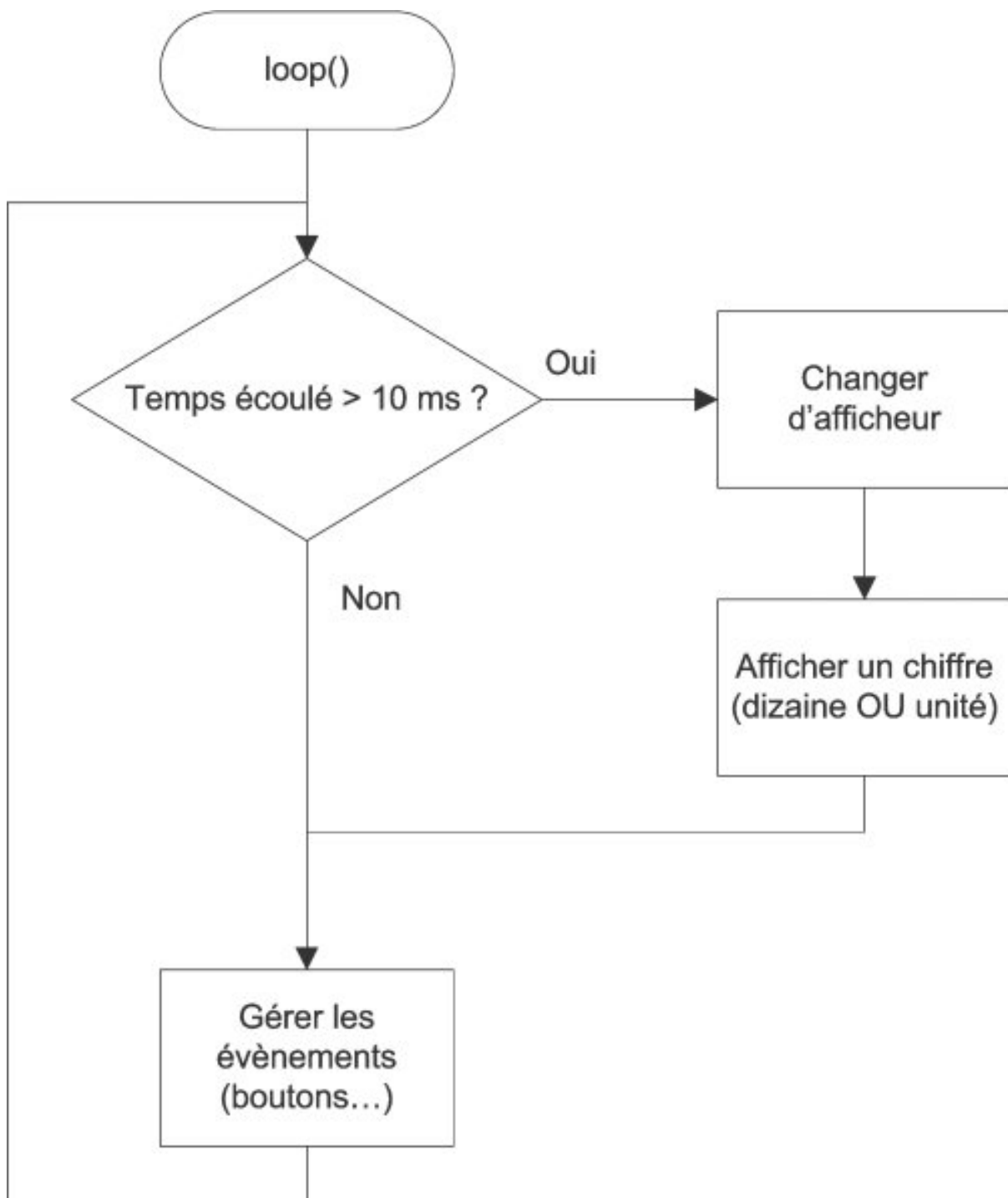


Figure 3.36 – Organigramme TP Parking

l'affichage sur plusieurs afficheurs. Si elles ne vous semblent pas claires, je vous conseille de revenir sur le chapitre concernant les 7 segments. `[[secret]]`

```

| | cpp | | // fonction
| | // fonction
| | permettant d'afficher un nombre | void afficher_nombre(char nombre, bool
| | afficheur) | { | long temps; | char unite = 0, dizaine = 0; | if(nombre >
| | 9) | dizaine = nombre / 10; // on recupere les dizaines | unite = nombre
| | - (dizaine*10); // on recupere les unités | | if(afficheur) | { | // on
| | affiche les dizaines | digitalWrite(alim_unite, LOW); | digitalWrite(alim_dizaine,
| | HIGH); | afficher(dizaine); | } | else | { | // on affiche les unités |
| | digitalWrite(alim_dizaine, LOW); | digitalWrite(alim_unite, HIGH); | afficher(unite);
| | } | } | | // fonction écrivant sur un seul afficheur | void afficher(char
| | chiffre) | { | // on commence par écrire 0, donc tout à l'état bas | digitalWrite(bit_
| | LOW); | digitalWrite(bit_B, LOW); | digitalWrite(bit_C, LOW); | digitalWrite(bit_D,
| | LOW); | | if(chiffre >= 8) | { | digitalWrite(bit_D, HIGH); | chiffre =
| | chiffre - 8; | } | if(chiffre >= 4) | { | digitalWrite(bit_C, HIGH); |
| | chiffre = chiffre - 4; | } | if(chiffre >= 2) | { | digitalWrite(bit_B,
| | HIGH); | chiffre = chiffre - 2; | } | if(chiffre >= 1) | { | digitalWrite(bit_A,
| | HIGH); | chiffre = chiffre - 1; | } | } | } | | Code: Les fonctions d'affichage

```

3.6.2.2.5 Et le code au complet Si vous voulez tester l'ensemble de l'application sans faire d'erreurs de copier/coller, voici le code complet (qui doit fonctionner si on considère que vous avez branché chaque composant au même endroit que sur le schéma fourni au départ !)

```

// les broches du décodeur 7 segments
const int bit_A = 2;
const int bit_B = 3;
const int bit_C = 4;
const int bit_D = 5;
// les broches des transistors pour l'afficheur des dizaines et celui des unités
const int alim_dizaine = 6;
const int alim_unite = 7;
// les broches des boutons
const int btn_entree = 8;
const int btn_sortie = 9;
// les leds de signalements
const int led_rouge = 12;
const int led_verte = 11;
// les mémoires d'état des boutons
int mem_entree = HIGH;
int mem_sortie = HIGH;
int etat = HIGH; // variable stockant l'état courant d'un bouton

char place_dispo = 10; // contenu des places dispos
bool afficheur = false;
long temps;

void setup()
{
    // Les broches sont toutes des sorties (sauf les boutons)

```

```

pinMode(bit_A, OUTPUT);
pinMode(bit_B, OUTPUT);
pinMode(bit_C, OUTPUT);
pinMode(bit_D, OUTPUT);
pinMode(alim_dizaine, OUTPUT);
pinMode(alim_unite, OUTPUT);
pinMode(btn_entree, INPUT);
pinMode(btn_sortie, INPUT);
pinMode(led_rouge, OUTPUT);
pinMode(led_verte, OUTPUT);

// Les broches sont toutes mises à l'état bas (sauf led rouge éteinte)
digitalWrite(bit_A, LOW);
digitalWrite(bit_B, LOW);
digitalWrite(bit_C, LOW);
digitalWrite(bit_D, LOW);
digitalWrite(alim_dizaine, LOW);
digitalWrite(alim_unite, LOW);
digitalWrite(led_rouge, HIGH);
digitalWrite(led_verte, LOW); // vert par défaut
temps = millis(); // enregistre "l'heure"
}

void loop()
{
// si ca fait plus de 10 ms qu'on affiche, on change de 7 segments
if((millis() - temps) > 10)
{
// on inverse la valeur de "afficheur"
// pour changer d'afficheur (unité ou dizaine)
afficheur = !afficheur;
// on affiche
afficher_nombre(place_dispo, afficheur);
temps = millis(); // on met à jour le temps
}

// on test maintenant si les boutons ont subi un appui (ou pas)
// d'abord le bouton plus puis le moins
etat = digitalRead(btn_entree);
if((etat != mem_entree) && (etat == LOW))
    place_dispo += 1;
mem_entree = etat; // on enregistre l'état du bouton pour le tour suivant

// et maintenant pareil pour le bouton qui décrémente
etat = digitalRead(btn_sortie);
if((etat != mem_sortie) && (etat == LOW))
    place_dispo -= 1;
mem_sortie = etat; // on enregistre l'état du bouton pour le tour suivant
}

```

```

// on applique des limites au nombre pour ne pas dépasser 99 ou 0
if(place_dispo > 99)
    place_dispo = 99;
if(place_dispo < 0)
    place_dispo = 0;

// on met à jour l'état des leds
// on commence par les éteindre
digitalWrite(led_verte, HIGH);
digitalWrite(led_rouge, HIGH);
if(place_dispo == 0) // s'il n'y a plus de place
    digitalWrite(led_rouge, LOW);
else
    digitalWrite(led_verte, LOW);
}

// fonction permettant d'afficher un nombre
void afficher_nombre(char nombre, bool afficheur)
{
    long temps;
    char unite = 0, dizaine = 0;
    if(nombre > 9)
        dizaine = nombre / 10; // on récupère les dizaines
        unite = nombre - (dizaine*10); // on récupère les unités

    if(afficheur)
    {
        // on affiche les dizaines
        digitalWrite(alim_unite, LOW);
        digitalWrite(alim_dizaine, HIGH);
        afficher(dizaine);
    }
    else
    {
        // on affiche les unités
        digitalWrite(alim_dizaine, LOW);
        digitalWrite(alim_unite, HIGH);
        afficher(unite);
    }
}

// fonction écrivant sur un seul afficheur
void afficher(char chiffre)
{
    // on commence par écrire 0, donc tout à l'état bas
    digitalWrite(bit_A, LOW);
    digitalWrite(bit_B, LOW);
}

```



```

digitalWrite(bit_C, LOW);
digitalWrite(bit_D, LOW);

if(chiffre >= 8)
{
    digitalWrite(bit_D, HIGH);
    chiffre = chiffre - 8;
}
if(chiffre >= 4)
{
    digitalWrite(bit_C, HIGH);
    chiffre = chiffre - 4;
}
if(chiffre >= 2)
{
    digitalWrite(bit_B, HIGH);
    chiffre = chiffre - 2;
}
if(chiffre >= 1)
{
    digitalWrite(bit_A, HIGH);
    chiffre = chiffre - 1;
}
}
// Fin du programme!

```

Code : Le code complet

3.6.2.3 Conclusion

Bon, si vous ne comprenez pas tout du premier coup, c'est un petit peu normal, c'est en effet difficile de reprendre un programme que l'on a pas fait soi-même et ce pour diverses raisons. Le principal est que vous ayez cherché une solution par vous-même et que vous soyez arrivé à réaliser l'objectif final. Si vous n'avez pas réussi mais que vous pensiez y être presque, alors je vous invite à chercher profondément le pourquoi du comment votre programme ne fonctionne pas ou pas entièrement, cela vous aidera à trouver vos erreurs et à ne plus en refaire !

Il est pas magnifique ce parking ? J'espère que vous avez apprécié sa réalisation. Nous allons maintenant continuer à apprendre de nouvelles choses, toujours plus sympas les unes que les autres. Un conseil, gardez votre travail quelques part au chaud, vous pourriez l'améliorer avec vos connaissances futures !

4 La communication avec Arduino

4.1 Généralités sur la voie série

La communication... que ferait-on sans ! Le téléphone, Internet, la télévision, les journaux, la publicité... rien de tout cela n'existerait s'il n'y avait pas de communication. Évidemment, ce n'est pas de ces moyens là dont nous allons faire l'objet dans la partie présente. Non, nous allons voir un moyen de communication que possède la carte Arduino. Vous pourrez ainsi faire communiquer votre carte avec un ordinateur ou bien une autre carte Arduino ! Et oui ! Elle en a sous le capot cette petite carte ! ;)

4.1.1 Communiquer, pourquoi ?

Nous avons vu dans la partie précédente où nous faisons nos premiers pas avec Arduino, comment utiliser la carte. Nous avons principalement utilisé des LED pour communiquer à l'utilisateur (donc vous, à priori) certaines informations. Cela pouvait être une LED ou un groupe de LED qui peut indiquer tout et n'importe quoi, ou bien un afficheur 7 segments qui affiche des chiffres ou certains caractères pouvant tout aussi bien indiquer quelque chose. Tout dépend de ce que vous voulez signaler avec les moyens que vous mettez à disposition. On peut très bien imaginer un ensemble de LED ayant chacune un nom, sigle ou autre marqueur pour indiquer, selon l'état d'une ou plusieurs d'entre-elles, un mode de fonctionnement ou bien une erreur ou panne d'un système. Cependant, cette solution reste tout de même précaire et demande à l'utilisateur d'être devant le système de signalisation. Aujourd'hui, avec l'avancée de la technologie et du "tout connecté", il serait fâcheux de ne pouvoir aller plus loin. Je vais donc vous présenter un nouveau moyen de **communication** grâce à la **voie série** (ou "liaison série"), qui va vous permettre de communiquer des informations à l'utilisateur par divers intermédiaires. A la fin de la partie, vous serez capable de transmettre des informations à un ordinateur ou une autre carte Arduino.

4.1.1.1 Transmettre des informations

Tel est le principal objectif de la communication. Mais comment transmettre des informations... et puis quelles informations ? Avec votre carte Arduino, vous aurez certainement besoin de transmettre des mesures de températures ou autres grandeurs (tension, luminosité, etc.). Ces informations pourront alimenter une base de donnée, servir dans un calcul, ou à autre chose. Tout dépendra de ce que vous en ferez.

4.1.1.1.1 Émetteur et récepteur Lorsque l'on communique des informations, il faut nécessairement un **émetteur**, qui va transmettre les informations à communiquer, et un **récepteur**, qui va recevoir les informations pour les traiter.

Dans le cas présent, deux carte Arduino communiquent. L'une communique à l'autre tandis que l'autre réceptionne le message envoyé par la première.

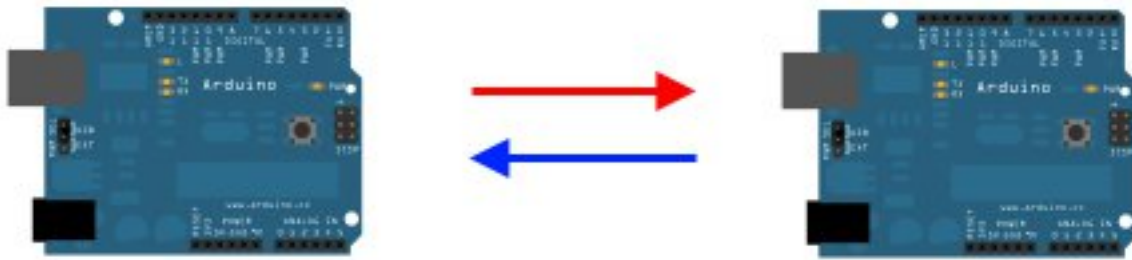


Figure 4.1 – Communication entre deux cartes

[[question]] | Pourtant, il y a deux flèches sur ton dessin. L'autre aussi, qui réceptionne le message, peut envoyer des données ?

Absolument ! Cependant, tout dépend du type de communication.

4.1.1.1.2 La communication en trois cas Pour parler, on peut par exemple différencier trois types de conversations. A chaque conversation, il n'y a que deux interlocuteurs. On ne peut effectivement pas en faire communiquer plus dans notre cas ! On dit que c'est une communication **point-à-point**.

- Le premier type serait lorsqu'un interlocuteur parle à son compère sans que celui-ci dise quoi que ce soit puisqu'il ne peut pas répondre. Il est muet et se contente d'écouter. C'est une communication à sens unilatérale, ou techniquement appelée communication **simplex**. L'un parle et l'autre écoute.
- Le deuxième type serait une conversation normale où chacun des interlocuteurs est poli et attend que l'autre est finie de parler pour parler à son tour. Il s'agit d'une communication **half-duplex**. Chaque interlocuteur parle à **tour de rôle**.
- Enfin, il y a la conversation du type "débat politique" (ce n'est évidemment pas son vrai nom ^^) où chaque interlocuteur parle en même temps que l'autre. Bon, cela dit, ce type de communication marche très bien (pas au sens politique, je parle au niveau technique !) et est très utilisé ! C'est une communication dite **full-duplex**.

A notre échelle, Arduino est capable de faire des communications de type full-duplex, puisqu'elle est capable de comprendre son interlocuteur tout en lui parlant en même temps.

4.1.1.2 Le récepteur

Qu'en est-il ? Eh bien il peut s'agir, comme je le sous-entendais plus tôt, d'une autre carte Arduino. Cela étant, n'importe quel autre appareil utilisant la voie série et son **protocole de communication** pourrait communiquer avec. Cela peut être notamment un ordinateur, c'est d'ailleurs le principal interlocuteur que nous mettrons en relation avec Arduino.

[[question]] | C'est quoi ça, un protocole de communication ?

C'est un ensemble de règles qui régissent la façon dont communiquent deux dispositifs entre eux. Cela définit par exemple le rythme de la conversation (le débit de parole des acteurs si vous préférez), l'ordre des informations envoyées (la grammaire en quelque sorte), le nombre d'informations, etc... On peut analogiquement comparer à une phrase en français, qui place le sujet, le verbe puis le complément. C'est une forme de protocole. Si je mélange tout ça, en plaçant par exemple le sujet, le complément et le verbe, cela donnerait un style parlé de maître Yoda... bon

c'est moins facilement compréhensible, mais ça le reste. En revanche, deux dispositifs qui communiquent avec un protocole différent ne se comprendront pas correctement et pourraient même interpréter des actions à effectuer qui seraient à l'opposé de ce qui est demandé. Ce serait en effet dommage que votre interlocuteur "donne le chat à manger" alors que vous lui avez demandé "donne à manger au chat". ^^ Bref, si les dispositifs communiquant n'utilisent pas le bon protocole, cela risque de devenir un véritable capharnaüm !

4.1.2 La norme RS232

Des liaisons séries, il en existe un paquet ! Je peux en citer quelques unes : RS-232, Universal Serial Bus (USB), Serial ATA, SPI, ... Et pour dire, vous pouvez très bien inventer votre propre norme de communication pour la voie série que vous décidez de créer. L'inconvénient, bien que cela puisse être également un avantage, il n'y a que vous seul qui puissiez alors utiliser une telle communication.

[[question]] | Et nous, laquelle allons-nous voir parmi celles-là ? Il y en a des meilleurs que d'autres ? oO

D'abord, nous allons voir la voie série utilisant la norme RS-232. Ensuite, oui, il y en a qui ont des avantages par rapport à d'autres. On peut essentiellement noter le type d'utilisation que l'on veut en faire et la vitesse à laquelle les dispositifs peuvent communiquer avec.

4.1.2.0.1 Applications de la norme La norme RS-232 s'applique sur trois champs d'une communication de type série. Elle définit le signal électrique, le protocole utilisé et tout ce qui est lié à la mécanique (la connectique, le câblage, etc...).

4.1.2.1 La mécanique

Pour communiquer via la voie série, deux dispositifs doivent avoir 3 câbles minimum.

- Le premier câble est la **référence électrique**, communément appelée **masse électrique**. Cela permet de prendre les mesures de tension en se fixant un même référentiel. Un peu lorsque vous vous mesurez : vous mesurez 1,7 mètre du sol au sommet de votre tête et non pas 4,4 mètre parce que vous êtes au premier étage et que vous vous basez par rapport au sol du rez-de-chaussé. Dans notre cas, on considérera que le 0V sera notre référentiel électrique commun.
- Les deux autres câbles permettent la transmission des données. L'un sert à l'envoi des données pour un émetteur, mais sert aussi pour la réception des données venant de l'autre émetteur. Idem pour l'autre câble. Il permet l'émission de l'un et la réception de l'autre.

Deux cartes Arduino reliées par 3 câbles :

- Le **noir** est la masse électrique commune
- Le **vert** est celui utilisé pour l'envoi des données de la première carte (à gauche), mais sert également à la réception des données envoyées pour la deuxième carte (à droite)
- Le **orange** est celui utilisé pour l'envoi des données de la deuxième carte (à droite), mais sert également à la réception des données envoyées pour la première carte (à gauche)

Cela, il s'agit du strict minimum utilisé. La norme n'interdit pas l'utilisation d'autres câbles qui servent à faire du contrôle de flux et de la gestion des erreurs.

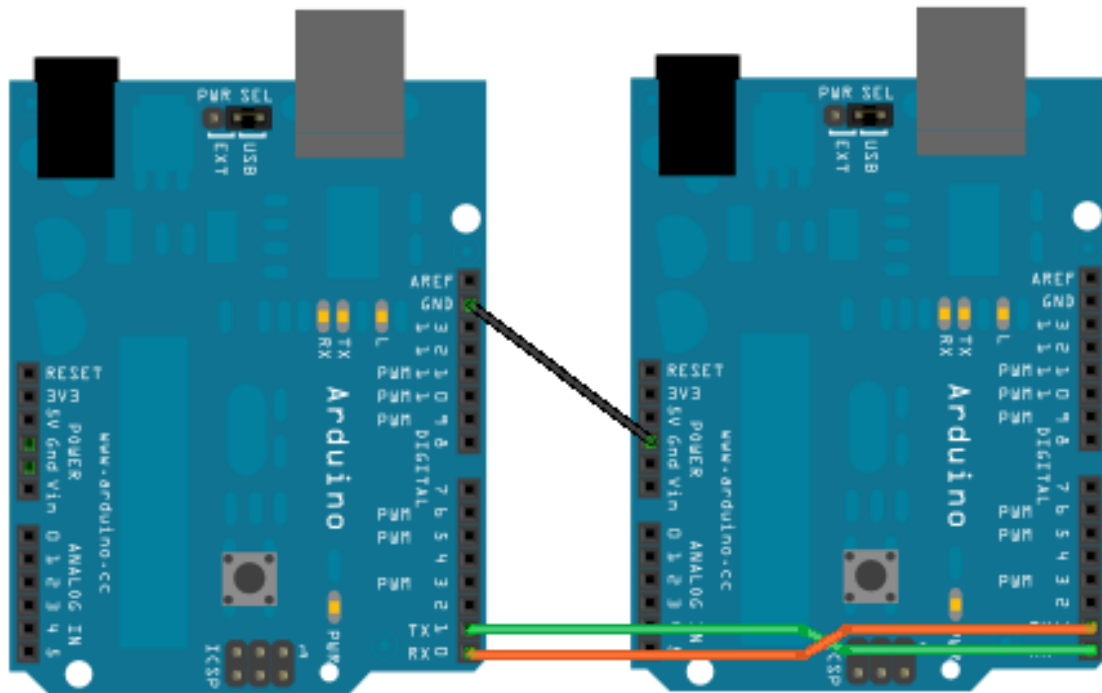


Figure 4.2 – Deux Arduino reliées entre elles

4.1.2.2 Le signal électrique et le protocole

Avant tout, il faut savoir que pour communiquer, deux dispositifs électronique ou informatique utilisent des données sous forme de **bits**. Ces bits, je le rappel, sont des états logiques (vrai ou faux) qui peuvent être regroupés pour faire des ensembles de bits. Généralement, ces ensembles sont constitués de 8 bits qui forment alors un **octet**.

4.1.2.2.1 Les tensions utilisées Ces bits sont en fait des niveaux de tension électrique. Et la norme RS-232 définit quelles tensions doivent être utilisées. On peut spécifier les niveaux de tension imposés par la norme dans un tableau, que voici :

->

	Niveau logique 0	Niveau logique 1
Tension électrique minimale	+3V	-3V
Tension électrique maximale	+25V	-25V

Table 4.1 – Niveau électrique de la norme RS-232

<-

Ainsi, toutes les tensions au delà des valeurs imposées, donc entre -3V et +3V, au dessous de -25V et au dessus de +25V, sont hors normes. Pour les tensions trop élevées (aux extrêmes de + et -25V) elles pourraient endommager le matériel. Quand aux tensions comprises entre + et -3V, eh bien elles sont ignorées car c'est dans ces zones là que se trouvent la plupart et même la quasi totalité des parasites. C'est un moyen permettant d'éviter un certain nombre d'erreurs de transmissions.

[[information]] | Les parasites dont je parle sont simplement des pics de tensions qui peuvent survenir à cause de différentes sources (interrupteur, téléviseur, micro-ondes, ...) et qui risquent alors de modifier des données lors d'une transmission effectuée grâce à la voie série.

Lorsqu'il n'y a pas de communication sur la voie série, il me ce qu'on appelle un état de repos

0 (LOW) et 1 (HIGH). En observant la table, on tombe sur la lettre “P” majuscule et l’on voit sa correspondance en binaire : 01010000.

[[question]] | Je crois ne pas bien comprendre pourquoi on envoie une lettre... qui va la recevoir et pour quoi faire? o_o

Il faut vous imaginer qu’il y a un destinataire. Dans notre cas, il s’agira avant tout de l’ordinateur avec lequel vous programmez votre carte. On va lui envoyer la lettre “P” mais cela pourrait être une autre lettre, une suite de lettres ou autres caractères, voire même des phrases complètes. Pour ne pas aller trop vite, nous resterons avec cette unique lettre. Lorsque l’on enverra la lettre à l’ordinateur, nous utiliserons un petit module intégré dans le logiciel Arduino pour visualiser le message réceptionné. C’est donc nous qui allons voir ce que l’on transmet via la voie série.

4.1.2.2.3 L’ordre et les délimiteurs On va à présent voir comment est transmis un octet sur la voie série en envoyant notre exemple, la lettre “P”. Analogiquement, je vais vous montrer que cette communication par la voie série se présente un peu comme un appel téléphonique :

1. Lorsque l’on passe un coup de fil, bien généralement on commence par dire “Bonjour” ou “Allo”. Ce début de message permet de faire l’ouverture de la conversation. En effet, si l’on reçoit un appel et que personne ne répond après avoir décroché, la conversation ne peut avoir lieu. Dans la norme RS-232, on va avoir une ouverture de la communication grâce à un **bit de départ**. C’est lui qui va engager la conversation avec son interlocuteur. Dans la norme RS-232, ce dernier est un état 0.
2. Ensuite, vous allez commencer à parler et donner les informations que vous souhaitez transmettre. Ce sera les **données**. L’élément principal de la conversation (ici notre lettre ‘P’).
3. Enfin, après avoir renseigné tout ce que vous aviez à dire, vous terminez la conversation par un “Au revoir” ou “Salut!”, “A plus!” etc. Cela termine la conversation. Il y aura donc un **bit de fin** ou **bit de stop** qui fera de même sur la voie série. Dans la norme RS-232, c’est un état 1.

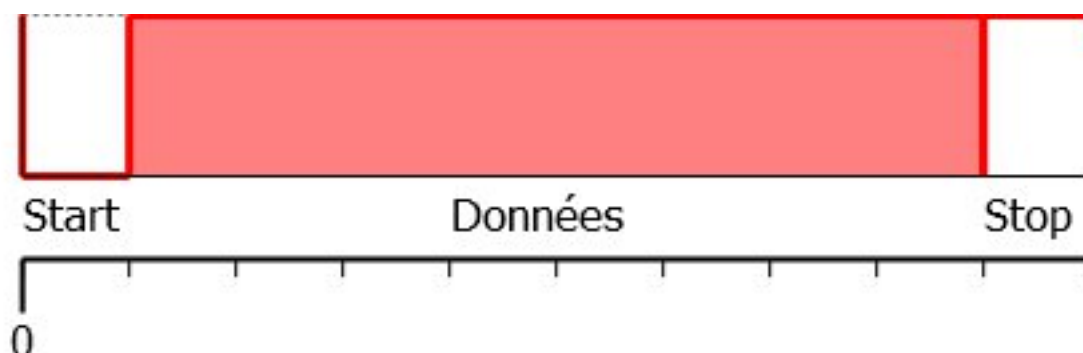


Figure 4.3 – Chronogramme d’un échange sur la voie série

C’est de cette manière là que la communication série fonctionne. D’ailleurs, savez-vous pourquoi la voie série s’appelle ainsi? En fait, c’est parce que les données à transmettre sont envoyées une par une. Si l’on veut, elles sont à la queue leu-leu. Exactement comme une conversation entre deux personnes : la personne qui parle ne peut pas dire plusieurs phrases en même temps, ni plusieurs mots ou sons. Chaque élément se suit selon un ordre logique. L’image précédente

résume la communication que l'on vient d'avoir, il n'y a plus qu'à la compléter pour envoyer la lettre "P".

[[question]] | Ha, je vois. Donc il y a le bit de start, notre lettre P et le bit de stop. D'après ce qu'on a dit, cela donnerait, dans l'ordre, ceci : 0 (Start) 01010000 (Données) et 1 (Stop). :D

Eh bien... c'est presque ça. Sauf que les ~~petits-malins~~ ingénieurs qui ont inventé ce protocole ont eu la bonne idée de transmettre les données à l'envers... Par conséquent, la bonne réponse était : 000010101. Avec un chronogramme, on observerait ceci :

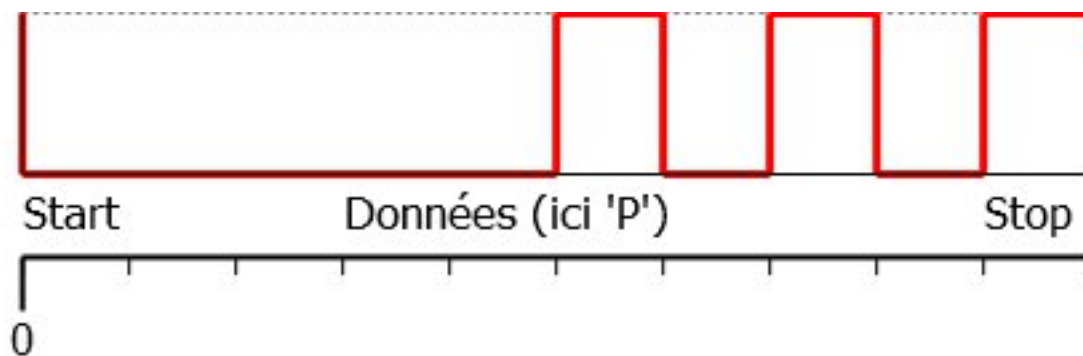


Figure 4.4 – Chronogramme de la transmission de la lettre 'P'

4.1.2.2.4 Un peu de vocabulaire Avant de continuer à voir ce que compose le protocole RS-232, voyons un peu de vocabulaire, mais sans trop en abuser bien sûr ! :P Les données sont envoyées à l'envers, je le disais. Ce qu'il faut savoir c'est que le bit de donnée qui vient après le bit de start s'appelle le **bit de poids faible** ou **LSB** en anglais pour Less Significant Bit. C'est un peu comme un nombre qui a des unités (tout à droite), des dizaines, des centaines, des milliers (à gauche), etc.

Par exemple le nombre 6395 possède 5 unités (à droite), 9 dizaines, 3 centaines et 6 milliers (à gauche). On peut faire référence au bit de poids faible en binaire qui est donc à droite. Plus on s'éloigne et plus on monte vers... le bit de **poids fort** ou **MSB** en anglais pour Most Significant Bit. Et comme les données sont envoyées à l'envers sur la liaison série, on aura le bit de poids faible juste après le start, donc à gauche et le bit de poids fort à droite.

Avec le nombre précédent, si l'on devait le lire à l'envers cela donnerait : 5936. Bit de poids faible à gauche et à droite le bit de poids fort.

[[erreur]] | Il est donc essentiel de savoir où est le bit de poids faible pour pouvoir lire les données à l'endroit. Sinon on se retrouve avec une donnée erronée !

Pour regrouper un peu tout ce que l'on a vu sur le protocole de la norme RS-232, voici une image le résumant :

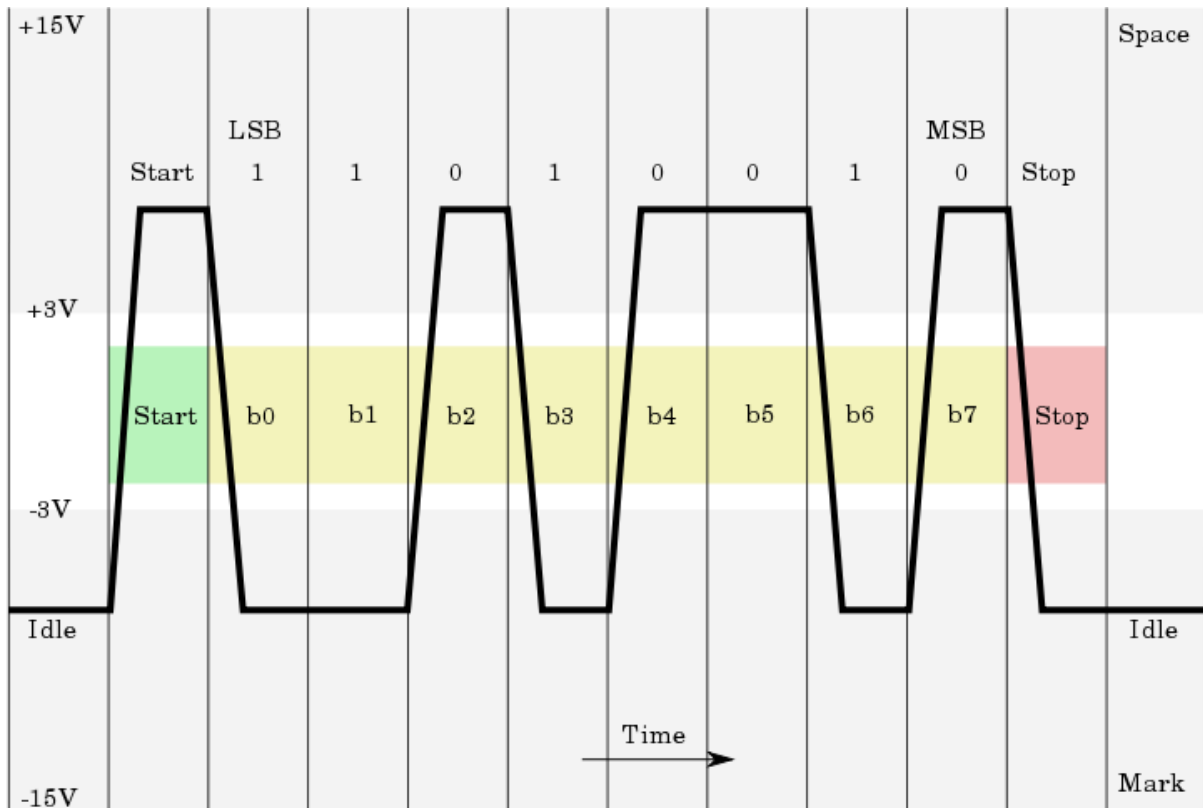


Figure : Protocole de la norme RS-232 - (CC-SA, Ktnbn)

Vous devrez être capable de trouver quel est le caractère envoyé sur cette trame... alors ? :D Indice : c'est une lettre... On lit les niveaux logiques de gauche à droite, soit 11010010 ; puis on les retourne soit 01001011 ; enfin on compare à la table ASCII et on trouve la lettre "K" majuscule. Attention aux tensions négatives qui correspondent à l'état logique 1 et les tensions positives à l'état logique 0.

4.1.2.2.5 La vitesse La norme RS-232 définit la vitesse à laquelle sont envoyées les données. Elles sont exprimées en bit par seconde (bit/s). Elle préconise des vitesses inférieures à 20 000 bits/s. Sauf qu'en pratique, il est très courant d'utiliser des débits supérieurs pouvant atteindre les 115 200 bits/s. Quand on va utiliser la voie série, on va définir la vitesse à laquelle sont transférées les données.

Cette vitesse dépend de plusieurs contraintes que sont : la longueur du câble utilisé reliant les deux interlocuteurs et la vitesse à laquelle les deux interlocuteurs peuvent se comprendre. Pour vous donner un ordre d'idée, je reprend le tableau fourni sur la page Wikipédia sur la norme RS-232 :

->

Débit en bit/s	Longueur du câble en mètres (m)
2 400	900
4 800	300
9 600	150
19 200	15

Débit en bit/s	Longueur du câble en mètres (m)
----------------	---------------------------------

Table 4.2 – Vitesses théoriques de la norme RS-232

<-

Plus le câble est court, plus le débit pourra être élevé car moins il y a d'affaiblissement des tensions et de risque de parasites. Tandis que si la distance séparant les deux interlocuteurs grandie, la vitesse de communication diminuera de façon effective.

4.1.2.2.6 La gestion des erreurs Malgré les tensions imposées par la norme, il arrive qu'il y ai d'autres parasites et que des erreurs de transmission surviennent. Pour limiter ce risque, il existe une solution. Elle consiste à ajouter un **bit de parité**. Vous allez voir, c'est hyper simple ! ;)

Juste avant le bit de stop, on va ajouter un bit qui sera pair ou impair. Donc, respectivement, soit un 0 soit un 1. Lorsque l'on utilisera la voie série, si l'on choisi une parité paire, alors le nombre de niveaux logiques 1 dans les données plus le bit de parité doit donner un nombre paire. Donc, dans le cas ou il y a 5 niveaux logiques 1 sans le bit de parité, ce dernier devra prendre un niveau logique 1 pour que le nombre de 1 dans le signal soit paire. Soit 6 au total :

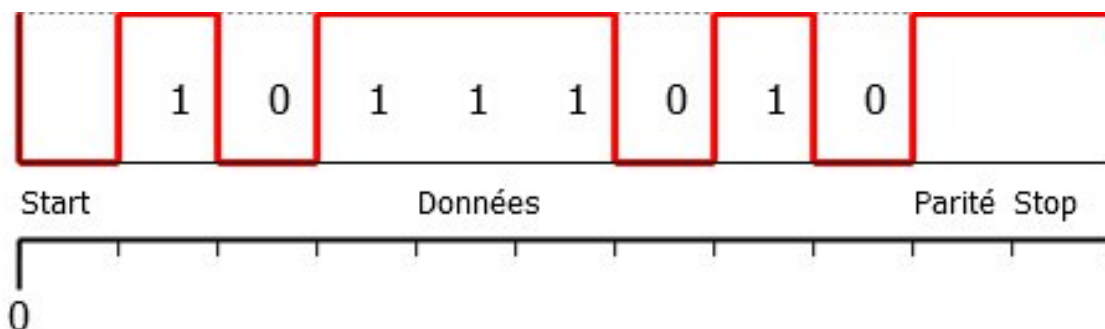


Figure 4.5 – Chronogramme pair

Dans le cas où l'on choisirait une parité impaire, alors dans le même signal où il y a 5 niveaux logiques 1, eh bien le bit de parité devra prendre la valeur qui garde un nombre impaire de 1 dans le signal. Soit un bit de parité égal à 0 dans notre cas :

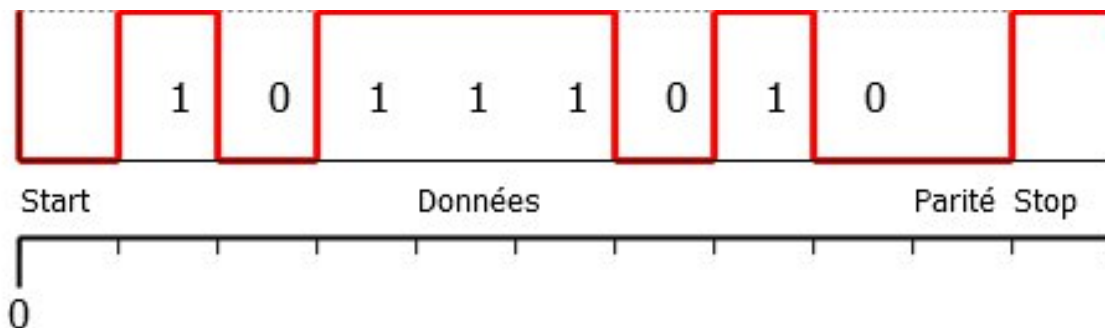


Figure 4.6 – Chronogramme impair

Après, c'est le récepteur qui va vérifier si le nombre de niveaux logiques 1 est bien égale à ce que indique le bit de parité. Dans le cas où une erreur de transmissions serait survenu, ce sera au

récepteur de traiter le problème et de demander à son interlocuteur de répéter. Au fait, ne vous inquiétez pas, on aura l'occasion de voir tout ça plus tard dans les prochains chapitres. De quoi s'occuper en somme... :diable :

4.1.3 Connexion série entre Arduino et ...

[[question]] | Et on connecte quoi à où pour utiliser la voie série avec la carte Arduino et le PC ? C'est le même câblage ? Et on connecte où sur le PC ?

Là, on va avoir le choix...

4.1.3.1 Émulation du port série

Le premier objectif et le seul que nous mettrons en place dans le cours, va être de connecter et d'utiliser la voie série avec l'ordinateur. Pour cela, rien de plus simple, il n'y a que le câble USB à brancher entre la carte Arduino et le PC. En fait, la voie série va être **émulée** à travers l'USB. C'est une forme virtuelle de cette liaison. Elle n'existe pas réellement, mais elle fonctionne comme si c'était bien une vraie voie série. Tout ça va être géré par un petit composant présent sur votre carte Arduino et le gestionnaire de port USB et périphérique de votre ordinateur.

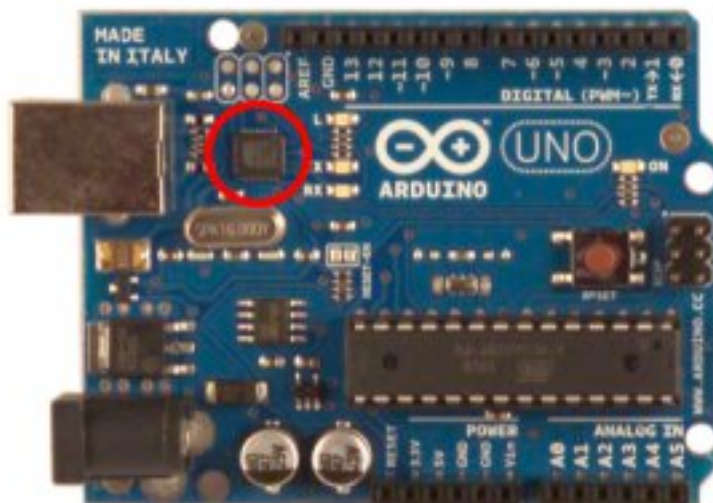


Figure 4.7 – Le composant entouré en rouge gère l'émulation de la voie série

C'est la solution la plus simple et celle que nous allons utiliser pour vos débuts.

4.1.3.2 Arduino et un autre microcontrôleur

On a un peu abordé ce sujet, au début de la présentation sur la voie série. Mais, on va voir un peu plus de choses. Le but de connecter deux microcontrôleurs ensemble est de pouvoir les faire communiquer entre eux pour qu'ils puissent s'échanger des données.

4.1.3.2.1 La tension des microcontrôleurs ->

	Tension
NL0	0V
NL1	+5V

<-

Contrairement à ce qu'impose la norme RS-232, les microcontrôleur ne peuvent pas utiliser des tensions négatives. Du coup, ils utilisent les seuls et uniques tensions qu'ils peuvent utiliser, à savoir le 0V et le +5V. Il y a donc quelques petits changement au niveau de la transmission série.

Un niveau logique 0 correspond à une tension de 0V et un niveau logique 1 correspond à une tension de +5V (cf. tableau ci-contre). Fort heureusement, comme les microcontrôleurs utilisent quasiment tous cette norme, il n'y a aucun problème à connecter deux microcontrôleurs entre-eux. Cette norme s'appelle alors UART pour **U**niversal **A**synchronous **R**eceiver **T**ransmitter plutôt que RS232. Hormis les tensions électriques et le connecteur, c'est la même chose !

4.1.3.2.2 Croisement de données Il va simplement falloir faire attention à bien croiser les fils. On connecte le Tx (broche de transmission) d'un microcontrôleur au Rx (broche de réception) de l'autre microcontrôleur. Et inversement, le Tx de l'autre au Rx du premier. Et bien sûr, la masse à la masse pour faire une référence commune. Exactement comme le premier schéma que je vous ai montré :

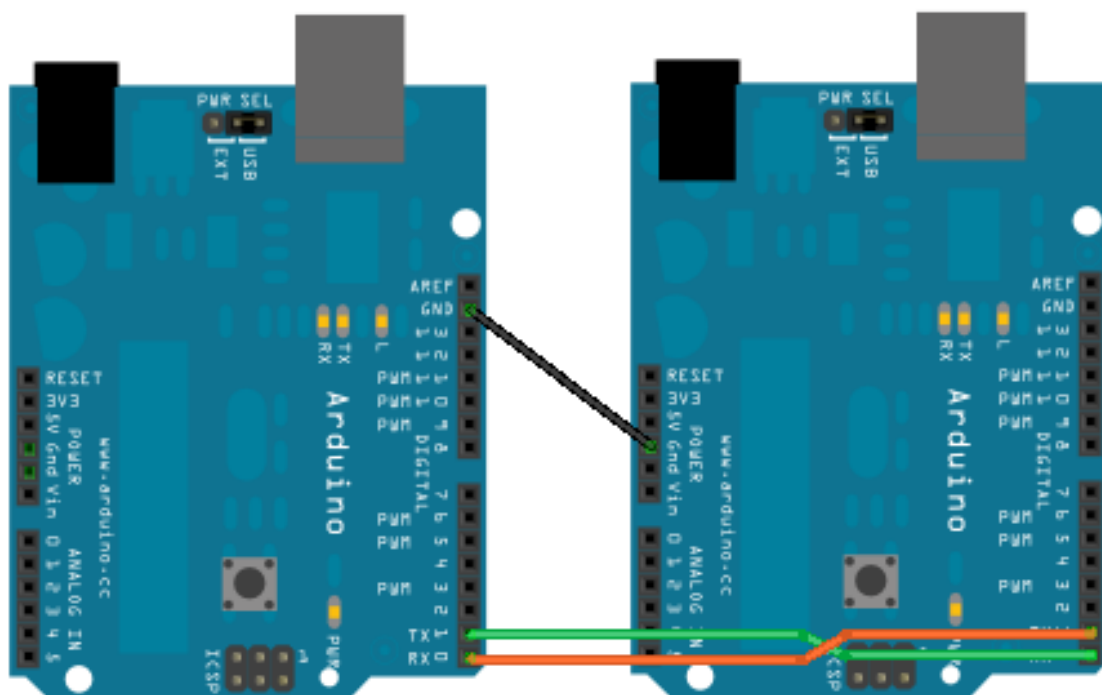


Figure 4.8 – Deux Arduino reliées entre elles

Tx → Rx, fil vert || Rx → Tx, fil orange Masse → Masse, fil noir

La couleur des fils importe peu, évidemment ! :P

4.1.3.3 Arduino au PC

4.1.3.3.1 Le connecteur série (ou sortie DB9) Alors là, les enfants, je vous parle d'un temps que les moins de vingt ans ne peuvent pas connaître... Bon on reprend ! Comme énoncé, je vous parle de quelque chose qui n'existe presque plus. Ou du moins, vous ne trouverez certainement plus cette "chose" sur la connectique de votre ordinateur. En effet, je vais vous parler du connecteur DB9 (ou DE9). Il y a quelques années, l'USB n'était pas si véloce et surtout pas tant répandu. Beaucoup de matériels (surtout d'un point de vue industriel) utilisaient la voie série (et le font encore).

À l'époque, les équipements se branchaient sur ce qu'on appelle une prise DB9 (9 car 9 broches). Sachez simplement que ce nom est attribué à un connecteur qui permet de relier divers matériels informatiques entre eux.



Figure 4.9 – Connecteur DB9 Mâle



Figure : Connecteur DB9 Fe-

melle - (CC-BY-SA, [Faxe](#))

Photos extraites du site Wikipédia - Connecteur DB9 Mâle à gauche ; Femelle à droite

[[question]] | A quoi ça sert ?

Si je vous parle de ça dans le chapitre sur la voie série, c'est qu'il doit y avoir un lien, non ? o_O Juste, car la voie série (je parle là de la transmission des données) est véhiculée par ce connecteur dans la norme RS-232. Donc, notre ordinateur dispose d'un connecteur DB9, qui permet de relier, via un câble adapté, sa connexion série à un autre matériel. Avant, donc, lorsqu'il était très répandu, on utilisait beaucoup ce connecteur. D'ailleurs, la première version de la carte Arduino disposait d'un tel connecteur !

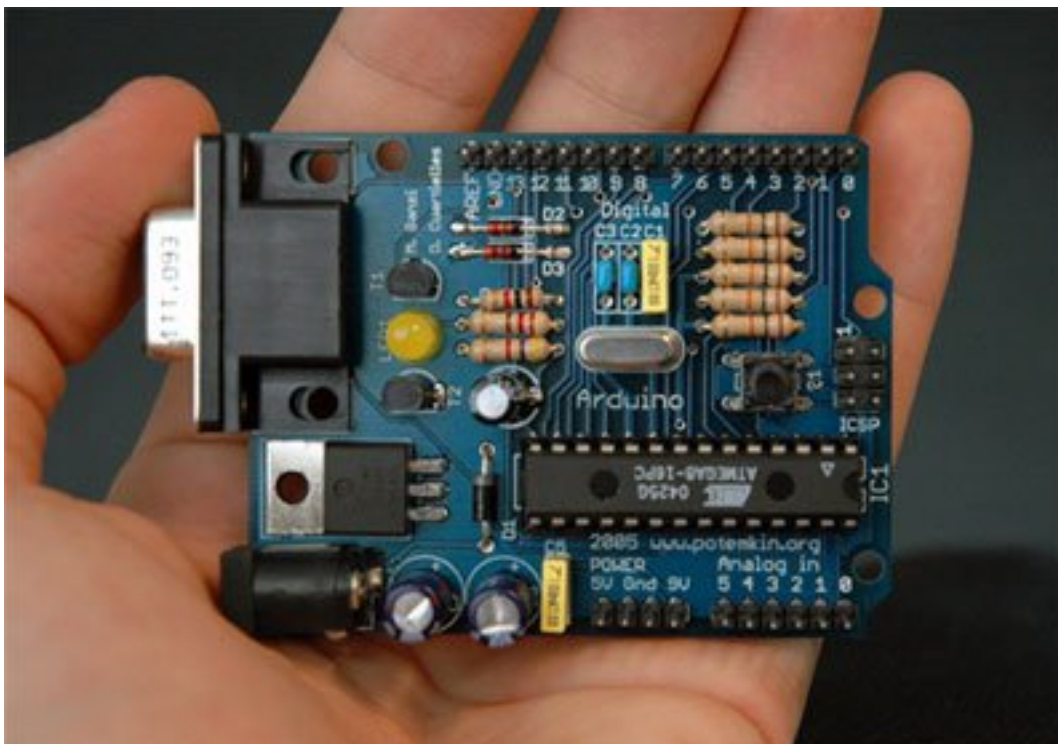


Figure : La

première version de la carte Arduino, avec un connecteur DB9 - (CC-BY-SA, [arduino.cc](#))

Aujourd'hui, le connecteur DB9 a déjà bien disparu mais reste présent sur les "vieux" ordinateurs ou sur d'autres appareils utilisant la voie série. C'est pourquoi, le jour où vous aurez besoin de communiquer avec un tel dispositif, il vous faudra faire un peu d'électronique...

4.1.3.3.2 Une petite histoire d'adaptation Si vous avez donc l'occasion de connecter votre carte Arduino à un quelconque dispositif utilisant la voie série, il va falloir faire attention aux tensions...oui, encore elles ! Je l'ai déjà dit, un microcontrôleur utilise des tensions de 0V et 5V, qu'on appelle TTL. Hors, la norme RS-232 impose des tensions positives et négatives comprise en +/-3V et +/-25V. Il va donc falloir adapter ces tensions. Pour cela, il existe un composant très courant et très utilisé dans ce type de cas, qu'est le MAX232.

-> [Datasheet du MAX232](#) <-

Je vous laisse regarder la datasheet et comprendre un peu le fonctionnement. Aussi, je vous met un schéma, extrait du site internet sonelec-musique.com :

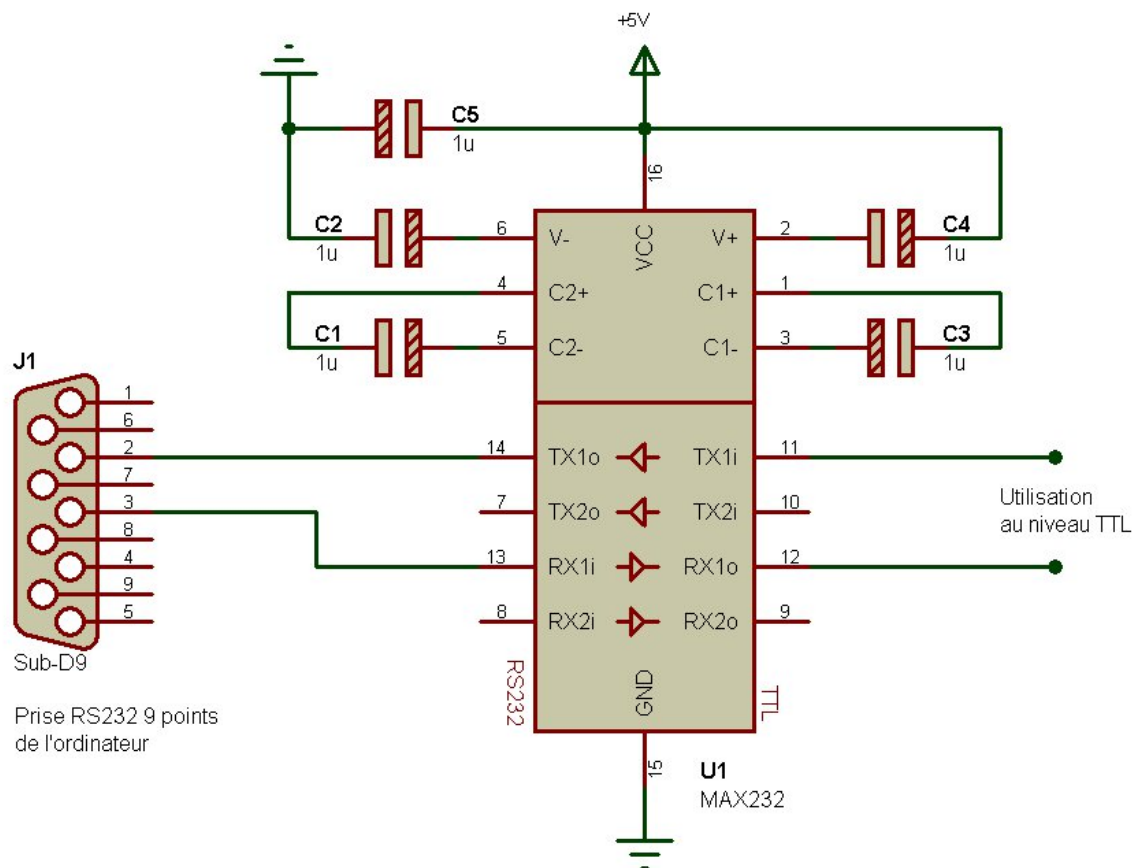


Figure : Câblage du MAX232 - (Autorisation de reproduction par l'auteur, Rémy, [source](#))

Le principe de ce composant, utilisé avec quelques condensateurs, est d'adapter les signaux de la voie série d'un microcontrôleur vers des tensions aux standards de la norme RS-232 et inversement. Ainsi, une fois le montage installé, vous n'avez plus à vous soucier de savoir quelle tension il faut, etc...

[[erreur]] | En revanche, n'utilisez jamais ce composant pour relier deux microcontrôleurs entre eux ! Vous risqueriez d'en griller un. Ou alors il faut utiliser deux fois ce composant (un pour TTL → RS232 et l'autre pour RS232 → TTL >_<), mais cela deviendrait alors peu utile.

Donc en sortie du MAX232, vous aurez les signaux Rx et Tx au standard RS-232. Elles dépendent

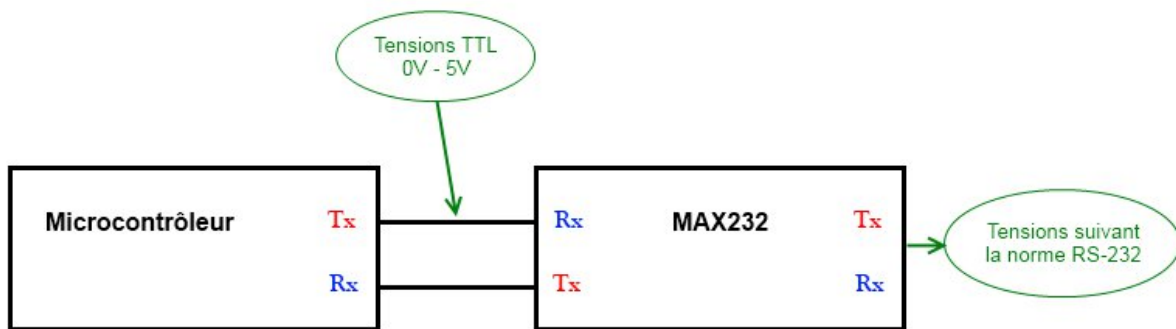


Figure 4.10 – Lien entre microcontrôleur et MAX232

de son alimentation et sont en générale centrées autour de +/-12V. Vous pourrez par exemple connecter un connecteur DB9 à la sortie du MAX232 et relier la carte Arduino à un dispositif utilisant lui aussi la voie série et un connecteur DB9. Ou même à un dispositif n'utilisant pas de connecteur DB9 mais un autre (dont il faudra connaître le brochage) et qui utilise la voie série.

*[TTL] : Transistor-Transistor Logic

4.1.4 Au delà d'Arduino avec la connexion série

Voici une petite annexe qui va vous présenter un peu l'utilisation du vrai port série. Je ne vous oblige pas à la lire, elle n'est pas indispensable et peu seulement servir si vous avez un jour besoin de communiquer avec un dispositif qui exploite cette voie série.

4.1.4.1 Le connecteur série (ou sortie DB9)

4.1.4.1.1 Le brochage au complet ! [[question]] | Oui, je veux savoir pourquoi il possède tant de broches puisque tu nous as dit que la voie série n'utilisait que 3 fils.

Eh bien, toutes ces broches ont une fonction bien précise. Je vais vous les décrire, ensuite on verra plus en détail ce que l'on peut faire avec :

1. **DCD** : Détection d'un signal sur la ligne. Utilisée uniquement pour la connexion de l'ordinateur à un modem ; détecte la porteuse
2. **RXD** : Broche de réception des données
3. **TXD** : Broche de transmission des données
4. **DTR** : Le support qui veut recevoir des données se déclare prêt à "écouter" l'autre
5. **GND** : Le référentiel électrique commun ; la masse
6. **DSR** : Le support voulant transmettre déclare avoir des choses à dire
7. **RTS** : Le support voulant transmettre des données indique qu'il voudrait communiquer
8. **CTS** : Invitation à émettre. Le support de réception attend des données
9. **RI** : Très peu utilisé, indiquait la sonnerie dans le cas des modems RS232

Vous voyez déjà un aperçu de ce que vous pouvez faire avec toutes ces broches. Mais parlons-en plus amplement.

4.1.4.2 Désolé, je suis occupé...

Dans certains cas, et il n'est pas rare, les dispositifs communicant entre eux par l'intermédiaire de la voie série ne traitent pas les données à la même vitesse. Tout comme lorsque l'on communique avec quelqu'un, il arrive parfois qu'il n'arrive plus à suivre ce que l'on dit car il en prend des notes. Il s'annonce alors indisponible à recevoir plus d'informations. Dès qu'il est à nouveau prêt, il nous le fait savoir. Il y a un moyen, mis en place grâce à certaines broches du connecteur pour effectuer ce genre d'opération que l'on appelle le **contrôle de flux**. Il y a deux manières d'utiliser un contrôle de flux, nous allons les voir tout de suite.

4.1.4.2.1 Contrôle de flux logiciel Commençons par le contrôle de flux logiciel, plus simple à utiliser que le contrôle de flux matériel. En effet, il ne nécessite que trois fils : la masse, le Rx et le TX. Eh oui, ni plus ni moins, tout se passe logiquement. Le fonctionnement très simple de ce contrôle de flux utilise des caractères de la table ASCII, le caractère 17 et 19, respectivement nommés **XON** et **XOFF**. Ceci se passe entre un équipement E, qui est l'émetteur, et un équipement R, qui est récepteur. Le récepteur reçoit des informations, il les traite et stocke celles qui continuent d'arriver en attendant de les traiter. Mais lorsqu'il ne peut plus stocker d'informations, le récepteur envoie le caractère XOFF pour indiquer à l'émetteur qu'il sature et qu'il n'est plus en mesure de recevoir d'autres informations. Lorsqu'il est à nouveau apte à traiter les informations, il envoie le caractère XON pour dire à l'émetteur qu'il est à nouveau prêt à écouter ce que l'émetteur a à lui dire.

4.1.4.2.2 Contrôle de flux matériel On n'utilisera pas le contrôle de flux matériel avec Arduino car la carte n'en est pas équipée, mais il est bon pour vous que vous sachiez ce que c'est. Je ne parlerai en revanche que du contrôle matériel à 5 fils. Il en existe un autre qui utilise 9 fils. Le principe est le même que pour le contrôle logiciel. Cependant, on utilise certaines broches du connecteur DB9 dont je parlais plus haut. Ces broches sont **RTS** et **CTS**.

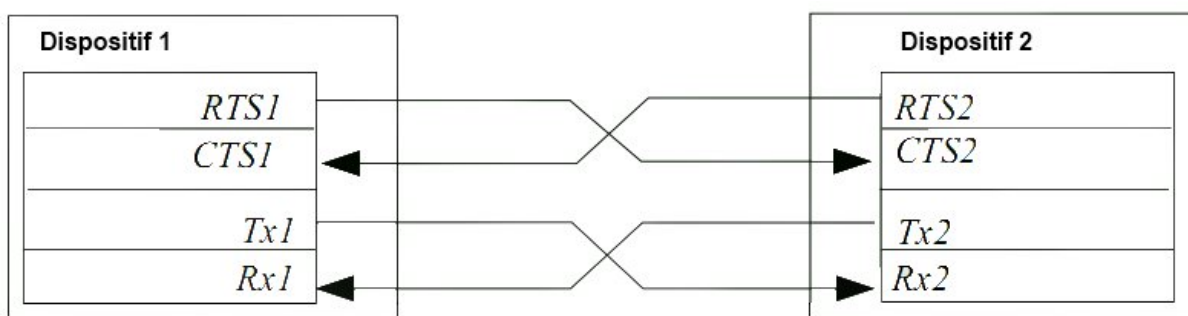


Figure 4.11 – Connexion avec contrôle de flux

Voilà le branchement adéquat pour utiliser ce contrôle de flux matériel à 5 fils. Une transmission s'effectue de la manière suivante :

- Le dispositif 1, que je nommerais maintenant *l'émetteur*, met un état logique 0 sur sa broche RTS1. Il demande donc au dispositif 2, *le récepteur*, pour émettre des données.
- Si le récepteur est prêt à recevoir des données, alors il met un niveau logique 0 sur sa broche RTS2.
- Les deux dispositifs sont prêts, l'émetteur peut donc envoyer les données qu'il a à transmettre.

- Une fois les données envoyées, l'émetteur passe à 1 l'état logique présent sur sa broche RTS₁.
- Le récepteur voit ce changement d'état et sait donc que c'est la fin de la communication des données, il passe alors l'état logique de sa broche RTS₂ à 1.

Ce contrôle n'est pas très compliqué et est utilisé lorsque le contrôle de flux logiciel ne l'est pas.

4.1.4.3 Avec ou sans horloge ?

Pour terminer, faisons une petite ouverture sur d'autres liaisons séries célèbres...

4.1.4.3.1 L'USB On la côtoie tout les jours sans s'en soucier et pourtant elle nous entoure : C'est la liaison USB! Comme son nom l'indique, Universal Serial Bus, il s'agit bien d'une voie série. Cette dernière existe en trois versions. La dernière, la 3.1, vient juste de sortir. Une des particularités de cette voie série est qu'elle se propose de livrer l'alimentation de l'équipement avec lequel elle communique. Par exemple votre ordinateur peut alimenter votre disque dur portable et en même temps lui demander des fichiers. Dans le cas de l'USB, la communication se fait de manière "maître-esclave". C'est l'hôte (l'ordinateur) qui demande des informations à l'esclave (le disque dur). Tant qu'il n'a pas été interrogé, ce dernier n'est pas censé parler. Afin de s'y retrouver, chaque périphérique se voit attribuer une adresse. La transmission électrique se fait grâce à un procédé "différentiel" entre deux fils, D+ et D-, afin de limiter les parasites.

4.1.4.3.2 L'I²C L'I²C est un autre protocole de communication qui fut tout d'abord propriétaire (inventé par Philips) et né de la nécessité d'interfacer de plus en plus de microcontrôleurs. En effet, à ce moment là une voie série "classique" ne suffisait plus car elle ne pouvait relier que deux à deux les microcontrôleurs. La particularité de cette liaison est qu'elle transporte son propre signal d'horloge. Ainsi, la vitesse n'a pas besoin d'être connu d'avance. Les données sont transportées en même temps que l'horloge grâce à deux fils : SDA (Data) et SCL (Clock). Comme pour l'USB, la communication se fait sur un système de maître/esclave.

[DCD] : Carrier Detect [RXD] : Receive Data [TXD] : Transmit Data [DTR] : Data Terminal Ready [GDN] : Provient de l'anglais GROUND [DST] : Date Set Read [RTS] : Request To Send [CTS] : Clear To Send *[RI] : Ring Indicator

4.2 Envoyer et recevoir des données sur la voie série


Dans ce chapitre, nous allons apprendre à utiliser la voie série avec Arduino. Nous allons voir comment envoyer puis recevoir des informations avec l'ordinateur, enfin nous ferons quelques exercices pour vérifier que vous avez tout compris. :) Vous allez le découvrir bientôt, l'utilisation de la voie série avec Arduino est quasiment un jeu d'enfant, puisque tout est opaque aux yeux de l'utilisateur...

4.2.1 Préparer la voie série

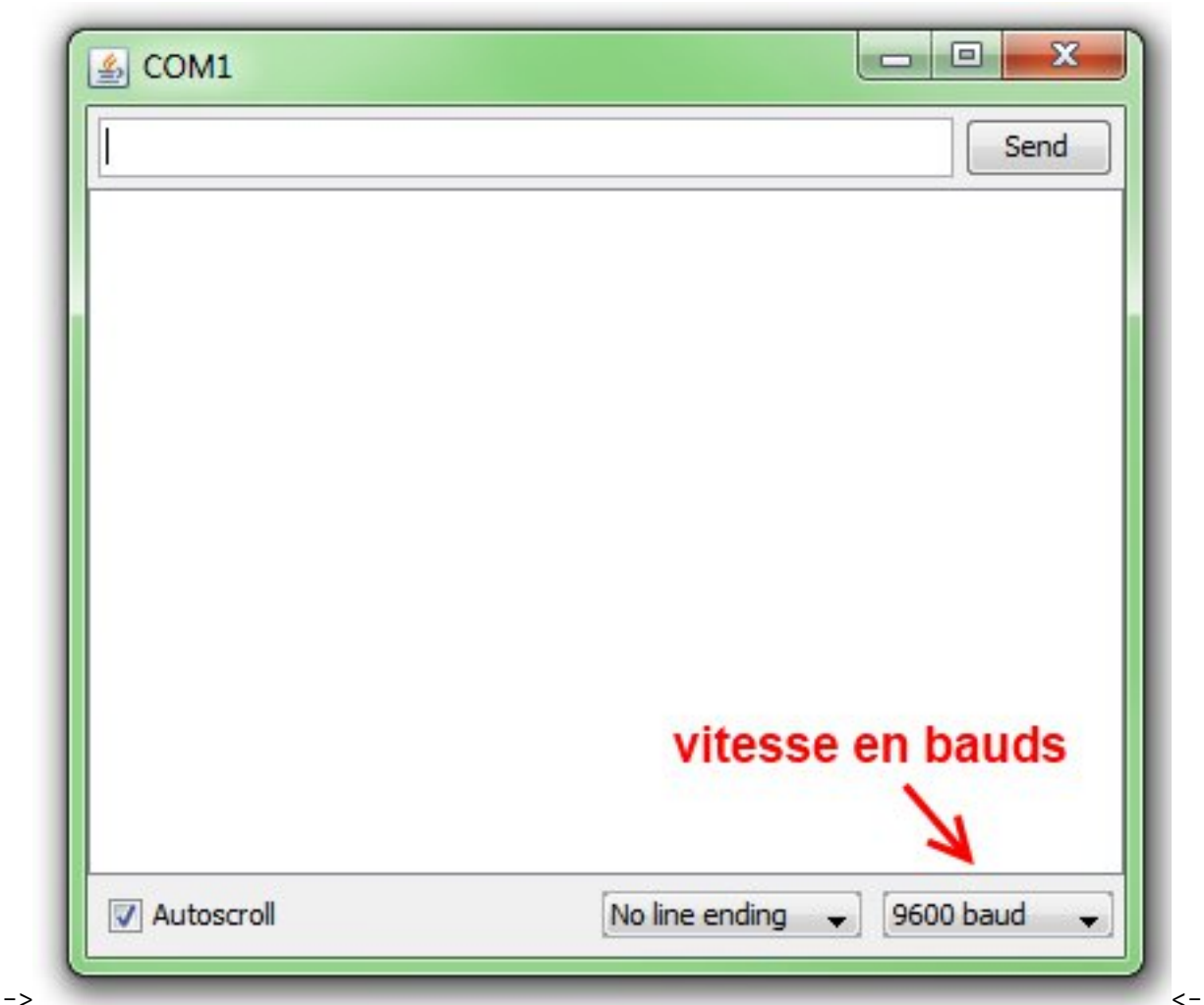
Notre objectif, pour le moment, est de communiquer des informations de la carte Arduino vers l'ordinateur et inversement. Pour ce faire, on va d'abord devoir préparer le terrain.

4.2.1.1 Du côté de l'ordinateur

Pour pouvoir utiliser la communication de l'ordinateur, rien de plus simple. En effet, L'environnement de développement Arduino propose de base un outil pour communiquer. Pour cela, il

suffit de cliquer sur le bouton  (pour les versions antérieures à la version 1.0) dans la barre de menu pour démarrer l'outil. Pour la version 1.0, l'icône a changé et de place et de visuel :

Une nouvelle fenêtre s'ouvre : c'est le **terminal série** :



Dans cette fenêtre, vous allez pouvoir envoyer des messages sur la voie série de votre ordinateur (qui est émulée¹ par l'Arduino); recevoir les messages que votre Arduino vous envoie; et régler deux trois paramètres tels que la vitesse de communication avec l'Arduino et l'autoscroll qui fait défiler le texte automatiquement. On verra plus loin à quoi sert le dernier réglage.

4.2.1.2 Du côté du programme

4.2.1.2.1 L'objet *Serial* Pour utiliser la voie série et communiquer avec notre ordinateur (par exemple), nous allons utiliser un *objet* (une sorte de variable mais plus évoluée) qui est intégré nativement dans l'ensemble Arduino : l'objet **Serial**.

1. créée de façon fictive

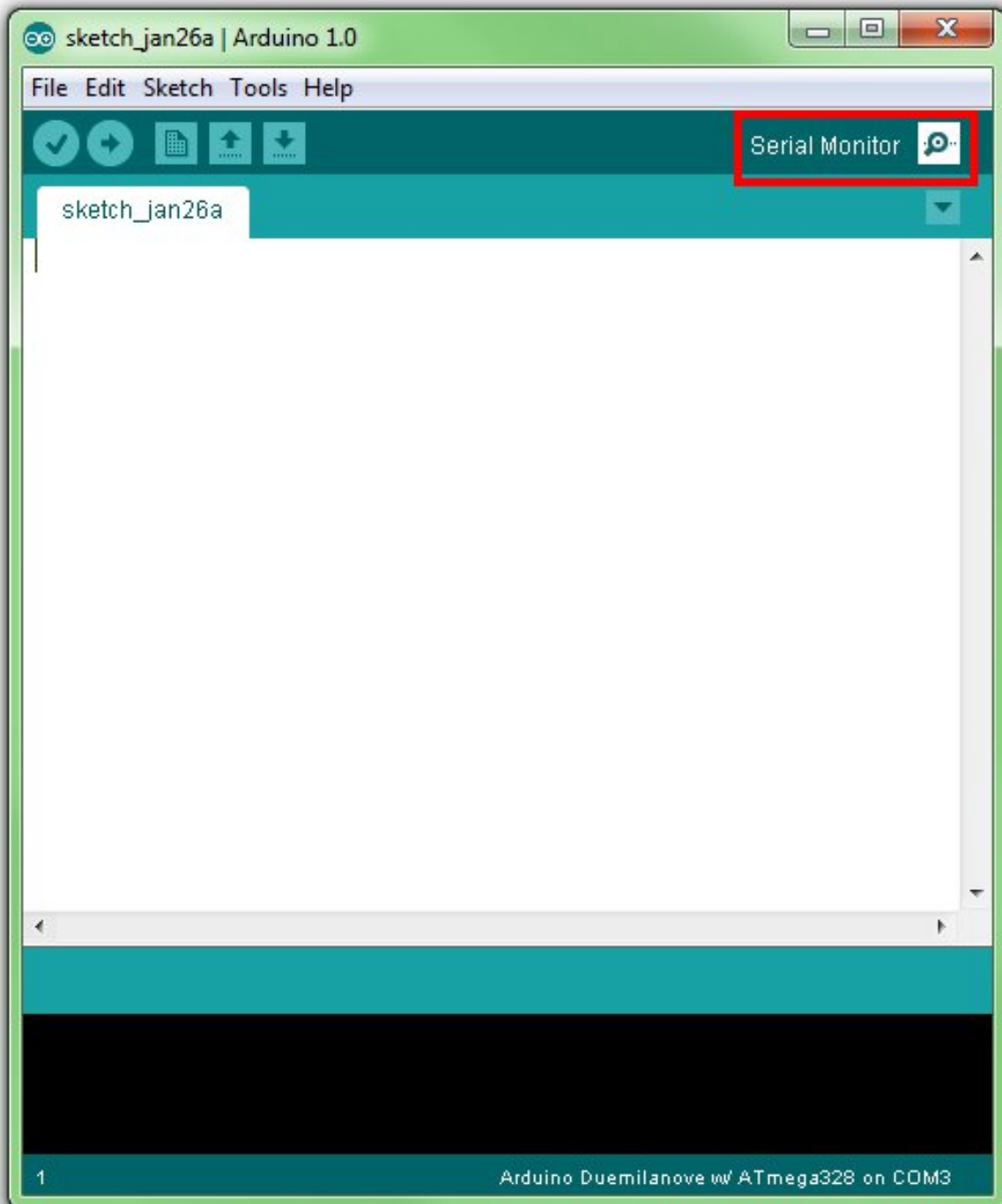


Figure 4.12 – Sélection moniteur série

[[information]] | Pour le moment, *considérez qu'un objet est une variable évoluée qui peut exécuter plusieurs fonctions*. On verra (beaucoup) plus loin ce que sont réellement des objets. On apprendra à en créer et à les utiliser lorsque l'on abordera le logiciel [Processing](#).

Cet objet rassemble des informations (vitesse, bits de données, etc.) et des fonctions (envoi, lecture de réception,...) sur ce qu'est une voie série pour Arduino. Ainsi, pas besoin pour le programmeur de recréer tout le protocole (sinon on aurait du écrire nous même TOUT le protocole, tel que "Ecrire un bit haut pendant 1 ms, puis 1 bit bas pendant 1 ms, puis le caractère 'a' en 8 ms...), bref, on gagne un temps fou et on évite les bugs !

4.2.1.2.2 Le setup Pour commencer, nous allons donc initialiser l'objet Serial. Ce code sera à copier à chaque fois que vous allez créer un programme qui utilise la voie série. Le logiciel Arduino a prévu, dans sa *bibliothèque Serial*, tout un tas de fonctions qui vont nous être très utiles, voir même indispensables afin de bien utiliser la voie série. Ces fonctions, je vous les laisse découvrir par vous même si vous le souhaitez, elles se trouvent sur [cette page](#). Dans le but de créer une communication entre votre ordinateur et votre carte Arduino, il faut déclarer cette nouvelle communication et définir la vitesse à laquelle ces deux dispositifs vont communiquer. Et oui, si la vitesse est différente, l'Arduino ne comprendra pas ce que veut lui transmettre l'ordinateur et vice versa ! Ce réglage va donc se faire dans la fonction setup, en utilisant la fonction `begin()` de l'objet Serial.

[[information]] | Lors d'une communication informatique, une vitesse s'exprime en bits par seconde ou **bauds**. Ainsi, pour une vitesse de 9600 bauds on enverra jusqu'à 9600 '0' ou '1' en une seule seconde. Les vitesses les plus courantes sont 9600, 19200 et 115200 bits par seconde.

```
void setup()
{
  // on démarre la liaison
  // en la réglant à une vitesse de 9600 bits par seconde.
  Serial.begin(9600);
}
```

Code : Démarrage de la liaison série

À présent, votre carte Arduino a ouvert une nouvelle communication vers l'ordinateur. Ils vont pouvoir communiquer ensemble.

4.2.2 Envoyer des données

Le titre est piégeur, en effet, cela peut être l'Arduino qui envoie des données ou l'ordinateur. Bon, on est pas non plus dénué d'une certaine logique puisque pour envoyer des données à partir de l'ordinateur vers la carte Arduino il suffit d'ouvrir le terminal série et de taper le texte dedans ! :P Donc, on va bien programmer et voir comment faire pour que votre carte Arduino envoie des données à l'ordinateur.

[[question]] | Et ces données, elles proviennent d'où ?

Eh bien de la carte Arduino... En fait, lorsque l'on utilise la voie série pour transmettre de l'information, c'est qu'on en a de l'information à envoyer, sinon cela ne sert à rien. Ces informations proviennent généralement de capteurs connectés à la carte ou de son programme (par exemple la valeur d'une variable). La carte Arduino traite les informations provenant de ces capteurs, s'il

faut elle adapte ces informations, puis elle les transmet. On aura l'occasion de faire ça dans la partie dédiée aux capteurs, comme afficher la température sur son écran, l'heure, le passage d'une personne, etc.

4.2.2.1 Appréhender l'objet Serial

Dans un premier temps, nous allons utiliser l'objet Serial pour tester quelques envois de données. Puis nous nous attèlerons à un petit exercice que vous ferez seul ou presque, du moins vous aurez eu auparavant assez d'informations pour pouvoir le réaliser (ben oui, sinon c'est plus un exercice!).

4.2.2.1.1 Phrase ? Caractère ? On va commencer par envoyer un caractère et une phrase. À ce propos, savez-vous quelle est la correspondance entre un caractère et une phrase ? Une phrase est constituée de caractères les uns à la suite des autres. En programmation, on parle plutôt de **chaîne caractères** pour désigner une phrase.

- Un caractère seul s'écrit entre guillemets simples : 'A', 'a', '2', '!', ...
- Une phrase est une suite de caractère et s'écrit entre guillemets doubles : "Salut tout le monde", "J'ai 42 ans", "Vive Clem' !"

[[information]] | Pour vous garantir un succès dans le monde de l'informatique, essayez d'y penser et de respecter cette convention, écrire 'A' ce n'est pas pareil qu'écrire "A"!

4.2.2.1.2 print() et println() La fonction que l'on va utiliser pour débiter, s'agit de `print()`. Ces deux fonctions sont quasiment identiques, mais à quoi servent-elles ?

- `print()` : cette fonction permet d'envoyer des données sur la voie série. On peut par exemple envoyer un caractère, une chaîne de caractère ou d'autres données dont je ne vous ai pas encore parlé.
- `println()` : c'est la même fonction que la précédente, elle permet simplement un retour à la ligne à la fin du message envoyé.

Pour utiliser ces fonctions, rien de plus simple :

```
Serial.print("Salut ca zeste?!");
```

Bien sûr, au préalable, vous devrez avoir "déclaré/créé" votre objet Serial et définis une valeur de vitesse de communication :

```
void setup()
{
  // création de l'objet Serial
  // (=établissement d'une nouvelle communication série)
  Serial.begin(9600);
  // envoie de la chaîne "Salut ca zeste?!" sur la voie série
  Serial.print("Salut ca zeste?!");
}
```

Code : Envoi d'un message simple via la liaison série

Cet objet, parlons-en. Pour vous aider à représenter de façon plus concise ce qu'est l'objet Serial, je vous propose cette petite illustration de mon cru :

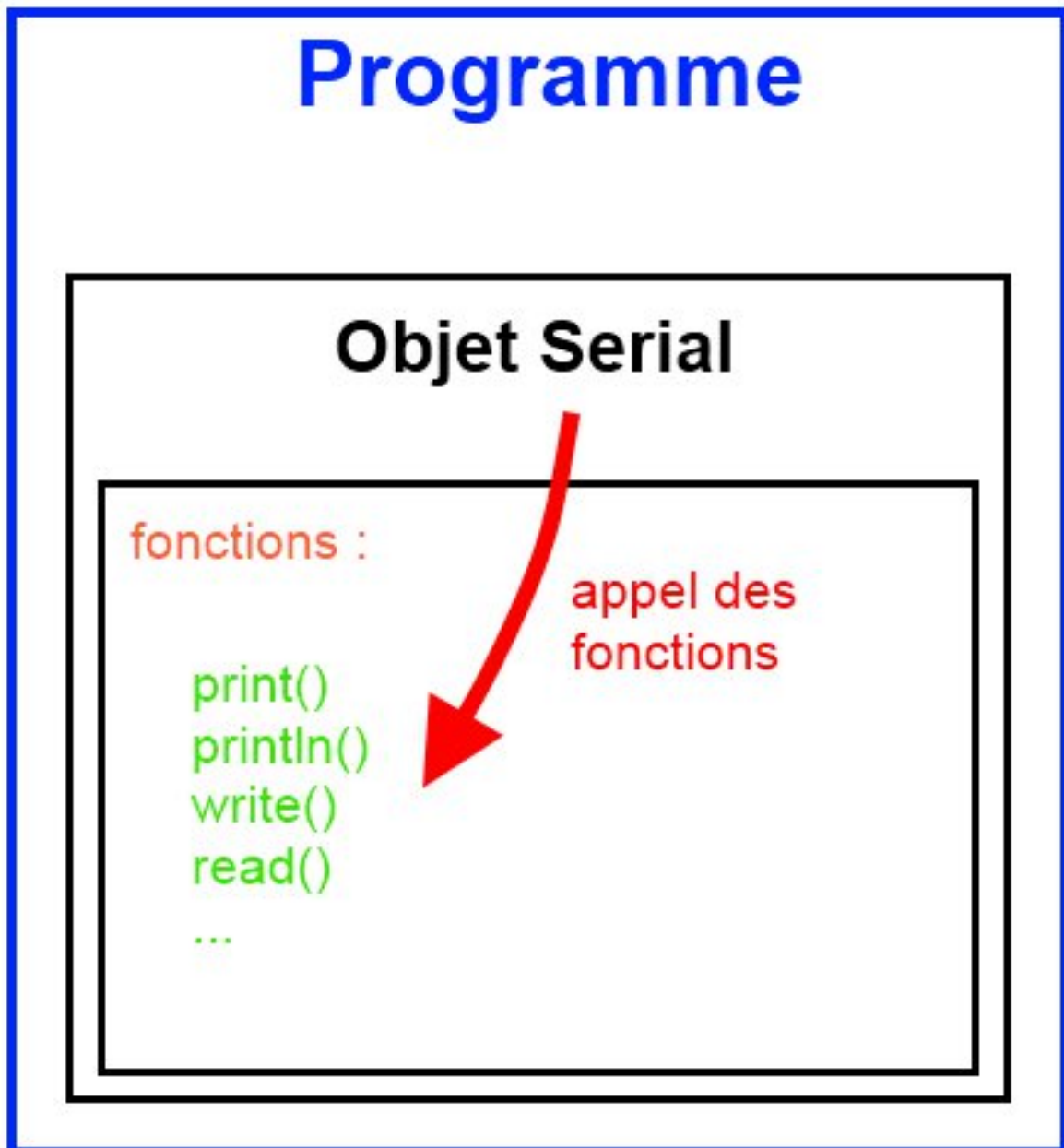


Figure 4.13 – L'objet Serial

Comme je vous le présente, l'objet Serial est muni d'un panel de fonctions qui lui sont propres. Cet objet est capable de réaliser ces fonctions selon ce que le programme lui ordonne de faire. Donc, par exemple, quand j'écris : `print()` en lui passant pour paramètre la chaîne de caractère : "Salut ca zeste?!". On peut compléter le code précédent comme ceci :

```
void setup()
{
    Serial.begin(9600);

    // l'objet exécute une première fonction
    Serial.print("Salut ca zeste?!");
    // puis une deuxième fonction, différente cette fois-ci
    Serial.println("Vive Clem'!");
    // et exécute à nouveau la même
    Serial.println("Cette phrase passe en dessous des deux précédentes");
}
```

Sur le terminal série, on verra ceci :

```
Salut ca zeste?! Vive Clem' !
Cette phrase passe en dessous des deux précédentes
```

4.2.2.2 La fonction `print()` en détail

Après cette courte prise en main de l'objet Serial, je vous propose de découvrir plus en profondeur les surprises que nous réserve la fonction `print()`.

[[information]] | Petite précision, je vais utiliser de préférence `print()`.

Résumons un peu ce que nous venons d'apprendre : on sait maintenant envoyer des caractères sur la voie série et des phrases. C'est déjà bien, mais ce n'est qu'un très bref aperçu de ce que l'on peut faire avec cette fonction.

4.2.2.2.1 Envoyer des nombres Avec la fonction `print()`, il est aussi possible d'envoyer des chiffres ou des nombres car ce sont des caractères :

```
void setup()
{
    Serial.begin(9600);

    Serial.println(9);           // chiffre
    Serial.println(42);         // nombre
    Serial.println(32768);      // nombre
    Serial.print(3.1415926535); // nombre décimale
}
```

```
9
42
32768
3.14
```


[[question]] | Tiens, le nombre pi n'est pas affiché complètement ! C'est quoi le bug ? o_O

Rassurez-vous, ce n'est ni un bug, ni un oubli inopiné de ma part. ^^ En fait, pour les nombres décimaux, la fonction `print()` affiche par défaut seulement deux chiffres après la virgule. C'est la valeur par défaut et heureusement elle est modifiable. Il suffit de rajouter le nombre de décimales que l'on veut afficher :

```
void setup()
{
  Serial.begin(9600);

  Serial.println(3.1415926535, 0);
  Serial.println(3.1415926535, 2); // valeur par défaut
  Serial.println(3.1415926535, 4);
  Serial.println(3.1415926535, 10);
}

3
3.14
3.1415
3.1415926535
```

4.2.2.2 Envoyer la valeur d'une variable Là encore, on utilise toujours la même fonction (qu'est-ce qu'elle polyvalente !). Ici aucune surprise. Au lieu de mettre un caractère ou un nombre, il suffit de passer la variable en paramètre pour qu'elle soit ensuite affichée à l'écran :

```
int variable = 512;
char lettre = 'a';

void setup()
{
  Serial.begin(9600);

  Serial.println(variable);
  Serial.print(lettre);
}

512
a
```

Trop facile n'est-ce pas ?

4.2.2.2.3 Envoyer d'autres données Ce n'est pas fini, on va terminer notre petit tour avec les types de variables que l'on peut transmettre grâce à cette fonction `print()` sur la voie série. Prenons l'exemple d'un nombre choisi judicieusement : 65.

[[question]] | Pourquoi ce nombre en particulier ? Et pourquoi pas 12 ou 900 ?

Eh bien, c'est relatif à la **table ASCII** que nous allons utiliser dans un instant.

[[information]] | Tout d'abord, petit cours de prononciation, ASCII se prononce comme si on disait "A ski", on a donc : "la table à ski" en prononciation phonétique.

La table ASCII, de l'américain "American Standard Code for Information Interchange", soit en bon français : "Code américain normalisé pour l'échange d'information" est, selon Wikipédia :

"la norme de codage de caractères en informatique la plus connue, la plus ancienne et la plus largement compatible" Source :Wikipédia

En somme, c'est un tableau de valeurs codées sur 8bits qui à chaque valeur associe un caractère. Ces caractères sont les lettres de l'alphabet en minuscule et majuscule, les chiffres, des caractères spéciaux et des symboles bizarres. Dans cette table, il y a plusieurs colonnes avec la valeur décimale, la valeur hexadécimale, la valeur binaire et la valeur octale parfois. Nous n'aurons pas besoin de tout ça, mais pour votre culture voici une table ASCII étendu (0 à 255).

The ASCII code

American Standard Code for Information Interchange

ASCII control characters				ASCII printable characters												Extended ASCII characters											
DEC	HEX	Symbole ASCII		DEC	HEX	Symbole	DEC	HEX	Symbole	DEC	HEX	Symbole	DEC	HEX	Symbole	DEC	HEX	Symbole	DEC	HEX	Symbole						
00	00h	NULL	(caractère nulo)	32	20h	espacio	64	40h	@	96	60h	`	128	80h	Ç	160	A0h	à	192	C0h	À	224	E0h	Ó			
01	01h	SOH	(inicio encabezado)	33	21h	!	65	41h	A	97	61h	a	129	81h	ü	161	A1h	á	193	C1h	Á	225	E1h	Ô			
02	02h	STX	(inicio texto)	34	22h	"	66	42h	B	98	62h	b	130	82h	é	162	A2h	â	194	C2h	Â	226	E2h	Û			
03	03h	ETX	(fin de texto)	35	23h	#	67	43h	C	99	63h	c	131	83h	à	163	A3h	ã	195	C3h	Ã	227	E3h	Ô			
04	04h	EOT	(fin transmisión)	36	24h	\$	68	44h	D	100	64h	d	132	84h	ä	164	A4h	ä	196	C4h	Ä	228	E4h	Ö			
05	05h	ENQ	(enquiry)	37	25h	%	69	45h	E	101	65h	e	133	85h	å	165	A5h	Å	197	C5h	Å	229	E5h	Ø			
06	06h	ACK	(acknowledgement)	38	26h	&	70	46h	F	102	66h	f	134	86h	ä	166	A6h	Ä	198	C6h	Ä	230	E6h	Ù			
07	07h	BEL	(bimbre)	39	27h	'	71	47h	G	103	67h	g	135	87h	å	167	A7h	Å	199	C7h	Å	231	E7h	Ú			
08	08h	BS	(retroceso)	40	28h	(72	48h	H	104	68h	h	136	88h	æ	168	A8h	Æ	200	C8h	Æ	232	E8h	Û			
09	09h	HT	(tab horizontal)	41	29h)	73	49h	I	105	69h	i	137	89h	æ	169	A9h	Æ	201	C9h	Æ	233	E9h	Ü			
10	0Ah	LF	(salto de línea)	42	2Ah	*	74	4Ah	J	106	6Ah	j	138	8Ah	è	170	AAh	È	202	CAh	È	234	EAh	Ü			
11	0Bh	VT	(tab vertical)	43	2Bh	+	75	4Bh	K	107	6Bh	k	139	8Bh	é	171	ABh	É	203	CBh	É	235	EBh	Ý			
12	0Ch	FF	(form feed)	44	2Ch	,	76	4Ch	L	108	6Ch	l	140	8Ch	í	172	ACH	Í	204	CDh	Í	236	ECh	ÿ			
13	0Dh	CR	(retorno de carro)	45	2Dh	-	77	4Dh	M	109	6Dh	m	141	8Dh	î	173	ADh	Î	205	CDh	Î	237	EDh	ÿ			
14	0Eh	SO	(shift out)	46	2Eh	.	78	4Eh	N	110	6Eh	n	142	8Eh	ï	174	Aeh	Ï	206	CEh	Ï	238	Eeh	ÿ			
15	0Fh	SI	(shift in)	47	2Fh	/	79	4Fh	O	111	6Fh	o	143	8Fh	ä	175	Afh	Ä	207	CFh	Ä	239	Efh	ÿ			
16	10h	DLE	(data link escape)	48	30h	0	80	50h	P	112	70h	p	144	90h	É	176	B0h	È	208	D0h	È	240	F0h	ÿ			
17	11h	DC1	(device control 1)	49	31h	1	81	51h	Q	113	71h	q	145	91h	æ	177	B1h	È	209	D1h	È	241	F1h	ÿ			
18	12h	DC2	(device control 2)	50	32h	2	82	52h	R	114	72h	r	146	92h	Æ	178	B2h	È	210	D2h	È	242	F2h	ÿ			
19	13h	DC3	(device control 3)	51	33h	3	83	53h	S	115	73h	s	147	93h	ø	179	B3h	È	211	D3h	È	243	F3h	ÿ			
20	14h	DC4	(device control 4)	52	34h	4	84	54h	T	116	74h	t	148	94h	ö	180	B4h	È	212	D4h	È	244	F4h	ÿ			
21	15h	NAK	(negative acknowle.)	53	35h	5	85	55h	U	117	75h	u	149	95h	ö	181	B5h	È	213	D5h	È	245	F5h	ÿ			
22	16h	SYN	(synchronous idle)	54	36h	6	86	56h	V	118	76h	v	150	96h	ü	182	B6h	È	214	D6h	È	246	F6h	ÿ			
23	17h	ETB	(end of trans. block)	55	37h	7	87	57h	W	119	77h	w	151	97h	ü	183	B7h	È	215	D7h	È	247	F7h	ÿ			
24	18h	CAN	(cancel)	56	38h	8	88	58h	X	120	78h	x	152	98h	ÿ	184	B8h	È	216	D8h	È	248	F8h	ÿ			
25	19h	EM	(end of medium)	57	39h	9	89	59h	Y	121	79h	y	153	99h	ÿ	185	B9h	È	217	D9h	È	249	F9h	ÿ			
26	1Ah	SUB	(substitute)	58	3Ah	:	90	5Ah	Z	122	7Ah	z	154	9Ah	ÿ	186	BAh	È	218	DAh	È	250	FAh	ÿ			
27	1Bh	ESC	(escape)	59	3Bh	;	91	5Bh	[123	7Bh	{	155	9Bh	ÿ	187	BBh	È	219	DBh	È	251	FBh	ÿ			
28	1Ch	FS	(file separator)	60	3Ch	<	92	5Ch	\	124	7Ch		156	9Ch	ÿ	188	BCh	È	220	DBh	È	252	FCh	ÿ			
29	1Dh	GS	(group separator)	61	3Dh	=	93	5Dh]	125	7Dh	}	157	9Dh	ÿ	189	BDh	È	221	DBh	È	253	FDh	ÿ			
30	1Eh	RS	(record separator)	62	3Eh	>	94	5Eh	^	126	7Eh	~	158	9Eh	x	190	BEh	È	222	DEh	È	254	FEh	ÿ			
31	1Fh	US	(unit separator)	63	3Fh	?	95	5Fh	_				159	9Fh	f	191	BFh	È	223	DFh	È	255	FFh	ÿ			

Figure : Table ASCII étendu - (CC-BY-SA, Yuriy Arabskyy)

Revenons à notre exemple, le nombre 65. C'est en effet grâce à la table ASCII que l'on sait passer d'un nombre à un caractère, car rappelons-le, dans l'ordinateur tout est traité sous forme de nombre en base 2 (binaire). Donc lorsque l'on code :

```
maVariable = 'A';
// l'ordinateur stocke la valeur 65 dans sa mémoire (cf. table ASCII)
```

Si vous faites ensuite :

```
maVariable = maVariable + 1;
// la valeur stockée passe à 66 (= 65 + 1)
```

// à l'écran, on verra s'afficher la lettre "B"

[[information]] | Au début, on trouvait une seule table ASCII, qui allait de 0 à 127 (codée sur 7bits) et représentait l'alphabet, les chiffres arabes et quelques signes de ponctuation. Depuis, de nombreuses tables dites "étendues" sont apparues et vont de 0 à 255 caractères (valeurs maximales codables sur un type char qui fait 8 bits).

[[question]] | Et que fait-on avec la fonction print() et cette table ?

Là est tout l'intérêt de la table, on peut envoyer des données, avec la fonction `print()`, de tous types ! En binaire, en hexadécimal, en octal et en décimal.

```
void setup()
{
  Serial.begin(9600);

  Serial.println(65, BIN); // envoie la valeur 1000001
  Serial.println(65, DEC); // envoie la valeur 65
  Serial.println(65, OCT); // envoie la valeur 101 (ce n'est pas du binaire!)
  Serial.println(65, HEX); // envoie la valeur 41
}
```

Code : Différents moyens d'afficher la même information

Vous pouvez donc manipuler les données que vous envoyez à travers la voie série ! C'est là qu'est l'avantage de cette fonction.

4.2.2.3 Exercice : Envoyer l'alphabet

4.2.2.3.1 Objectif Nous allons maintenant faire un petit exercice, histoire de s'entraîner à envoyer des données. Le but, tout simple, est d'envoyer l'ensemble des lettres de l'alphabet de manière *la plus intelligente* possible, autrement dit, sans écrire 26 fois "print();" ... La fonction `setup` restera la même que celle vue précédemment. Un délai de 250 ms est attendu entre chaque envoi de lettre et un delay de 5 secondes est attendu entre l'envoi de deux alphabets.

[[information]] | Bon courage !

4.2.2.3.2 Correction Bon j'espère que tout c'est bien passé et que vous n'avez pas joué au roi du copier/coller en me mettant 26 print...

```
[[secret]] | cpp | void loop() | { | char i = 0; | char lettre = 'a'; // ou
'A' pour envoyer en majuscule | | // petit message d'accueil | Serial.println("-----
L'alphabet des Zesteurs -----"); | | // on commence les envois | for(i=0;
i<26; i++) | { | Serial.print(lettre); // on envoie la lettre | lettre =
lettre + 1; // on passe à la lettre suivante | delay(250); // on attend
250ms avant de réenvoyer | } | Serial.println(""); // on fait un retour à
la ligne | | delay(5000); // on attend 5 secondes avant de renvoyer l'alphabet
| } | | Code : Exercice d'écriture de l'alphabet
```

Si l'exercice vous a paru trop simple, vous pouvez essayer d'envoyer l'alphabet à l'envers, ou l'alphabet minuscule ET majuscule ET les chiffres de 0 à 9... Amusez-vous bien ! ;)

4.2.3 Recevoir des données

Cette fois, il s'agit de l'Arduino qui reçoit les données que nous, utilisateur, allons transmettre à travers le terminal série. Je vais prendre un exemple courant : une communication téléphonique. En règle générale, on dit "Hallo" pour dire à l'interlocuteur que l'on est prêt à écouter le message. Tant que la personne qui appelle n'a pas cette confirmation, elle ne dit rien (ou dans ce cas elle fait un monologue ^^). Pareillement à cette conversation, l'objet `Serial` dispose d'une fonction pour "écouter" la voie série afin de savoir si oui ou non il y a une communication de données.

4.2.3.1 Réception de données

4.2.3.1.1 On m’a parlé ? Pour vérifier si on a reçu des données, on va régulièrement interroger la carte pour lui demander si des données sont disponibles dans son **buffer de réception**. Un buffer est une zone mémoire permettant de stocker des données sur un cours instant. Dans notre situation, cette mémoire est dédiée à la réception sur la voie série. Il en existe un aussi pour l’envoi de donnée, qui met à la queue leu leu les données à envoyer et les envoie dès que possible. En résumé, un buffer est une sorte de salle d’attente pour les données. Je disais donc, nous allons régulièrement vérifier si des données sont arrivées. Pour cela, on utilise la fonction `available()` (de l’anglais “disponible”) de l’objet `Serial`. Cette fonction renvoie le nombre de caractères dans le buffer de réception de la voie série. Voici un exemple de traitement :

```
void loop()
{
    // lecture du nombre de caractères disponibles dans le buffer
    int donneesALire = Serial.available();
    if(donneesALire > 0) // si le buffer n'est pas vide
    {
        // Il y a des données, on les lit et on fait du traitement
    }
    // on a fini de traiter la réception ou il n'y a rien à lire
}
```

Code : Lecture simple de données sur la voie série

[[information]] | Cette fonction de l’objet `Serial`, `available()`, renvoie la valeur `-1` quand il n’y a rien à lire sur le buffer de réception.

4.2.3.1.2 Lire les données reçues Une fois que l’on sait qu’il y a des données, il faut aller les lire pour éventuellement en faire quelque chose. La lecture se fera tout simplement avec la fonction... `read()`! Cette fonction renverra le premier caractère arrivé non traité (comme un supermarché traite la première personne arrivée dans la file d’attente de la caisse avant de passer au suivant). On accède donc *caractère par caractère* aux données reçues. Ce type de fonctionnement est appelé FIFO (First In First Out, premier arrivé, premier traité). Si jamais rien n’est à lire (personne dans la file d’attente), je le disais, la fonction renverra `-1` pour le signaler.

```
void loop()
{
    // on lit le premier caractère non traité du buffer
    char choseLue = Serial.read();

    if(choseLue == -1) // si le buffer est vide
    {
        // Rien à lire, rien lu
    }
    else // le buffer n'est pas vide
    {
        // On a lu un caractère
    }
}
```

Code : Lecture simple de données sur la voie série sans available

Ce code est une façon simple de se passer de la fonction available().

4.2.3.1.3 Le serialEvent Si vous voulez éviter de mettre le test de présence de données sur la voie série dans votre code, Arduino a rajouter une fonction qui s'exécute de manière régulière. Cette dernière se lance régulièrement avant chaque redémarrage de la loop. Ainsi, si vous n'avez pas besoin de traiter les données de la voie série à un moment précis, il vous suffit de rajouter cette fonction. Pour l'implémenter c'est très simple, il suffit de mettre du code dans une fonction nommé serialEvent() (attention à la casse) qui sera rajouté en dehors du setup et du loop. Le reste du traitement de texte se fait normalement, avec Serial.read() par exemple. Voici un exemple de squelette possible :

```
const int maLed = 11; // on met une LED sur la broche 11

void setup()
{
  pinMode(maLed, OUTPUT); // la LED est une sortie
  digitalWrite(maLed, HIGH); // on éteint la LED
  Serial.begin(9600); // on démarre la voie série
}

void loop()
{
  delay(500); // fait une petite pause
  // on ne fait rien dans la loop
  digitalWrite(maLed, HIGH); // on éteint la LED
}

void serialEvent() // déclaration de la fonction d'interruption sur la voie série
{
  // lit toutes les données (vide le buffer de réception)
  while(Serial.read() != -1);

  // puis on allume la LED
  digitalWrite(maLed, LOW);
}
```

Code : Utilisation de serialEvent pour tester la présence de données

4.2.3.2 Exemple de code complet

Voici maintenant un exemple de code complet qui va aller lire les caractères présents dans le buffer de réception s'il y en a et les renvoyer tels quels à l'expéditeur (mécanisme d'écho).

```
void setup()
{
  Serial.begin(9600);
```

```
}  
  
void loop()  
{  
  // variable contenant le caractère à lire  
  char carlu = 0;  
  // variable contenant le nombre de caractère disponibles dans le buffer  
  int cardispo = 0;  
  
  cardispo = Serial.available();  
  
  while(cardispo > 0) // tant qu'il y a des caractères à lire  
  {  
    carlu = Serial.read(); // on lit le caractère  
    Serial.print(carlu); // puis on le renvoi à l'expéditeur tel quel  
    cardispo = Serial.available(); // on relit le nombre de caractères dispo  
  }  
  // fin du programme  
}
```

Code : Code complet pour faire un *echo* avec la liaison série

Avouez que tout cela n'était pas bien difficile. Je vais donc en profiter pour prendre des vacances et vous laisser faire un exercice qui demande un peu de réflexion. :diable :

4.2.4 [Exercice] Attention à la casse !

4.2.4.1 Consigne

Le but de cet exercice est très simple. L'utilisateur saisit un caractère à partir de l'ordinateur et si ce caractère est minuscule, il est renvoyé en majuscule ; s'il est majuscule il est renvoyé en minuscule. Enfin, si le caractère n'est pas une lettre on se contente de le renvoyer normalement, tel qu'il est. Voilà le résultat de mon programme :

->!(<https://www.youtube.com/watch?v=9i-gfmQu2Cc>)<-

4.2.4.2 Correction

Je suppose que grâce au superbe tutoriel qui précède vous avez déjà fini sans problème, n'est-ce pas? :P

4.2.4.2.1 La fonction `setup()` et les variables utiles Une fois n'est pas coutume, on va commencer par énumérer les variables utiles et le contenu de la fonction `setup()`. Pour ce qui est des variables globales, on n'en retrouve qu'une seule, "carlu". Cette variable de type `int` sert à stocker le caractère lu sur le buffer de la carte Arduino. Puis on démarre une nouvelle voie série à 9600bauds :

```
[[secret]] | ccpp | int carlu; // stock le caractère lu sur la voie série | | void  
setup() | { | Serial.begin(9600); | } | | Code : Exercice, le setup
```

4.2.4.2.2 Le programme Le programme principal n'est pas très difficile non plus. Il va se faire en trois temps.

- Tout d'abord, on boucle jusqu'à recevoir un caractère sur la voie série
- Lorsqu'on a reçu un caractère, on regarde si c'est une lettre
- Si c'est une lettre, on renvoie son acolyte majuscule ; sinon on renvoie simplement le caractère lu

Voici le programme décrivant ce comportement :

```
[[secret]] |cpp |void loop() |{ | // on commence par vérifier si un caractère
est disponible dans le buffer | if(Serial.available() > 0) | { | carlu =
Serial.read(); // lecture du premier caractère disponible | | // Est-ce
que c'est un caractère minuscule? | if(carlu >= 'a' && carlu <= 'z') | {
| carlu = carlu - 'a'; // on garde juste le "numéro de lettre" | carlu =
carlu + 'A'; // on passe en majuscule | } | // Est-ce que c'est un caractère
MAJUSCULE? | else if(carlu >= 'A' && carlu <= 'Z') | { | carlu = carlu -
'A'; // on garde juste le "numéro de lettre" | carlu = carlu + 'a'; // on
passe en minuscule | } | // ni l'un ni l'autre on renvoie en tant que BYTE
| // ou alors on renvoie le caractère modifié | Serial.write(carlu); | }
|} | |Code : Exercice, la loop
```

Je vais maintenant vous expliquer les parties importantes de ce code. Comme vu dans le cours, la ligne 4 va nous servir à attendre un caractère sur la voie série. Tant qu'on ne reçoit rien, on ne fait rien ! Sitôt que l'on reçoit un caractère, on va chercher à savoir si c'est une lettre. Pour cela, on va faire deux tests. L'un est à la ligne 8 et l'autre à la ligne 13. Ils se présentent de la même façon : **SI** le caractère lu à une valeur supérieure ou égale à la lettre 'a' (ou 'A') **ET** inférieure ou égale à la lettre 'z' ('Z'), alors on est en présence d'une lettre. Sinon, c'est autre chose, donc on se contente de passer au renvoi du caractère lu ligne 21. Une fois que l'on a détecté une lettre, on effectue quelques transformations afin de changer sa casse. Voici les explications à travers un exemple :

->

Description	Opération (lettre)	Opération (nombre)	Valeur
On récupère la lettre 'e'	e	101	'e'
On isole son numéro de lettre en lui enlevant la valeur de 'a'	e-a	101-97	4
On ajoute ce nombre à la lettre 'A'	A + (e-a)	65 + (101-97) = 69	'E'
Il ne suffit plus qu'à retourner cette lettre	E	69	E

<-

On effectuera sensiblement les mêmes opérations lors du passage de majuscule à minuscule.

[[information]] | A la ligne 22, j'utilise la fonction `w r i t e ()` qui envoie le caractère en tant que variable de type *byte*, signifiant que l'on renvoie l'information sous la forme d'un seul octet. Sinon Arduino enverrait le caractère en tant que 'int', ce qui donnerait des problèmes lors de l'affichage.

Vous savez maintenant lire et écrire sur la voie série de l'Arduino ! Grâce à cette nouvelle corde à votre arc, vous allez pouvoir ajouter une touche d'interactivité supplémentaire à vos programmes.

4.3 [TP] Baignade interdite !

Afin d'appliquer vos connaissances acquises durant la lecture de ce tutoriel, nous allons maintenant faire un **gros TP**. Il regroupera tout ce que vous êtes censé savoir en terme de matériel (LED, boutons, voie série et bien entendu Arduino) et je vous fais aussi confiance pour utiliser au mieux vos connaissances en terme de "savoir coder" (variables, fonctions, tableaux...). Bon courage et, le plus important : Amusez-vous bien !

4.3.1 Sujet du TP sur la voie série

4.3.1.1 Contexte

Imaginez-vous au bord de la plage. Le ciel est bleu, la mer aussi... Ahhh le rêve. Puis, tout un coup le drapeau rouge se lève ! "Requiiiiinn" crie un nageur... L'application que je vous propose de développer ici correspond à ce genre de situation. Vous êtes au QG de la zPlage, le nouvel endroit branché pour les vacances. Votre mission si vous l'acceptez est d'afficher en temps réel un indicateur de qualité de la plage et de ses flots. Pour cela, vous devez informer les zTouristes par l'affichage d'un code de 3 couleurs. Des zSurveillants sont là pour vous prévenir que tout est rentré dans l'ordre si un incident survient.

4.3.1.2 Objectif

Comme expliqué ci-dessus, l'affichage de qualité se fera au travers de 3 couleurs qui seront représentées par des LEDs :

- **Rouge** : Danger, ne pas se baigner
- **Orange** : Baignade risquée pour les novices
- **Vert** : Tout baigne !

La zPlage est équipée de deux boutons. L'un servira à déclencher un SOS (si quelqu'un voit un nageur en difficulté par exemple). La lumière passe alors au rouge clignotant jusqu'à ce qu'un sauveteur ait appuyé sur l'autre bouton signalant "**Problème réglé, tout revient à la situation précédente**". Enfin, dernier point mais pas des moindres, le QG (vous) reçoit des informations météorologiques et provenant des marins au large. Ces messages sont retransmis sous forme de textos (symbolisés par la voie série) aux sauveteurs sur la plage pour qu'ils changent les couleurs en temps réel. Voici les mots-clés et leurs impacts :

- **Rouge** : meduse, tempete, requin : Des animaux dangereux ou la météo rendent la zPlage dangereuse. Baignade interdite
- **Orange** : vague : La natation est réservée aux bons nageurs
- **Vert** : surveillant, calme : Tout baigne, les zSauveteurs sont là et la mer est cool

4.3.1.3 Conseil

Voici quelques conseils pour mener à bien votre objectif.

4.3.1.3.1 Réalisation Une fois n'est pas coutume, **nommez bien vos variables!** Vous verrez que dès qu'une application prend du volume il est agréable de ne pas avoir à chercher qui sert à quoi. – N'hésitez pas à **décomposer votre code en fonction**. Par exemple les fonctions `changerDeCouleur()` peuvent-être les bienvenues. :euh :

4.3.1.3.2 Précision sur les chaînes de caractères Lorsque l'on écrit une phrase, on a l'habitude de la finir par un point. En informatique c'est pareil mais à l'échelle du mot! Je m'explique. Une chaîne de caractères (un mot) est, comme l'indique son nom, une suite de caractères. Généralement on la déclare de la façon suivante :

```
char mot[20] = "coucou"
```

Lorsque vous faites ça, vous ne le voyez pas, l'ordinateur rajoute juste après le dernier caractère (ici 'u') un caractère invisible qui s'écrit `\0` (antislash-zéro). Ce caractère signifie "fin de la chaîne". En mémoire, on a donc :

->

case	car.
mot[0]	'c'
mot[1]	'o'
mot[2]	'u'
mot[3]	'c'
mot[4]	'o'
mot[5]	'u'
mot[6]	'\0'

Table 4.5 – Correspondance des lettres vers le tableau

<-

[[information]] | Ce caractère est **très important** pour la suite car je vais vous donner un petit coup de pouce pour le traitement des mots reçus.

Une bibliothèque, nommée "string" (chaîne en anglais) et présente nativement dans votre logiciel Arduino, permet de traiter des chaînes de caractères. Vous pourrez ainsi plus facilement comparer deux chaînes avec la fonction `strcmp(chaine1, chaine2)`. Cette fonction vous renverra 0 si les deux chaînes sont identiques. Vous pouvez par exemple l'utiliser de la manière suivante :

```
// utilisation de la fonction strcmp(chaine1, chaine2) pour comparer des mots
int resultat = strcmp(motRecu, "requin");

if(resultat == 0)
    Serial.print("Les chaines sont identiques");
else
    Serial.print("Les chaines sont différentes");
```

Code : Utilisation de `strcmp`

Le truc, c'est que cette fonction compare *caractère par caractère* les chaînes, or celle de droite : “requin” possède ce fameux '\0' après le 'n'. Pour que le résultat soit identique, il faut donc que les deux chaînes soient parfaitement identiques ! Donc, avant d'envoyer la chaîne tapée sur la voie série, il faut lui rajouter ce fameux '\0'.

[[information]] | Je comprends que ce point soit délicat à comprendre, je ne vous taperais donc pas sur les doigts si vous avez des difficultés lors de la comparaison des chaînes et que vous allez vous balader sur la solution... Mais essayez tout de même, c'est tellement plus sympa de réussir en réfléchissant et en essayant ! :)

4.3.1.4 Résultat

Prenez votre temps, faites-moi quelque chose de beau et amusez-vous bien ! Je vous laisse aussi choisir comment et où brancher les composants sur votre carte Arduino. :ninja : Voici une photo d'illustration du montage ainsi qu'une vidéo du montage en action.

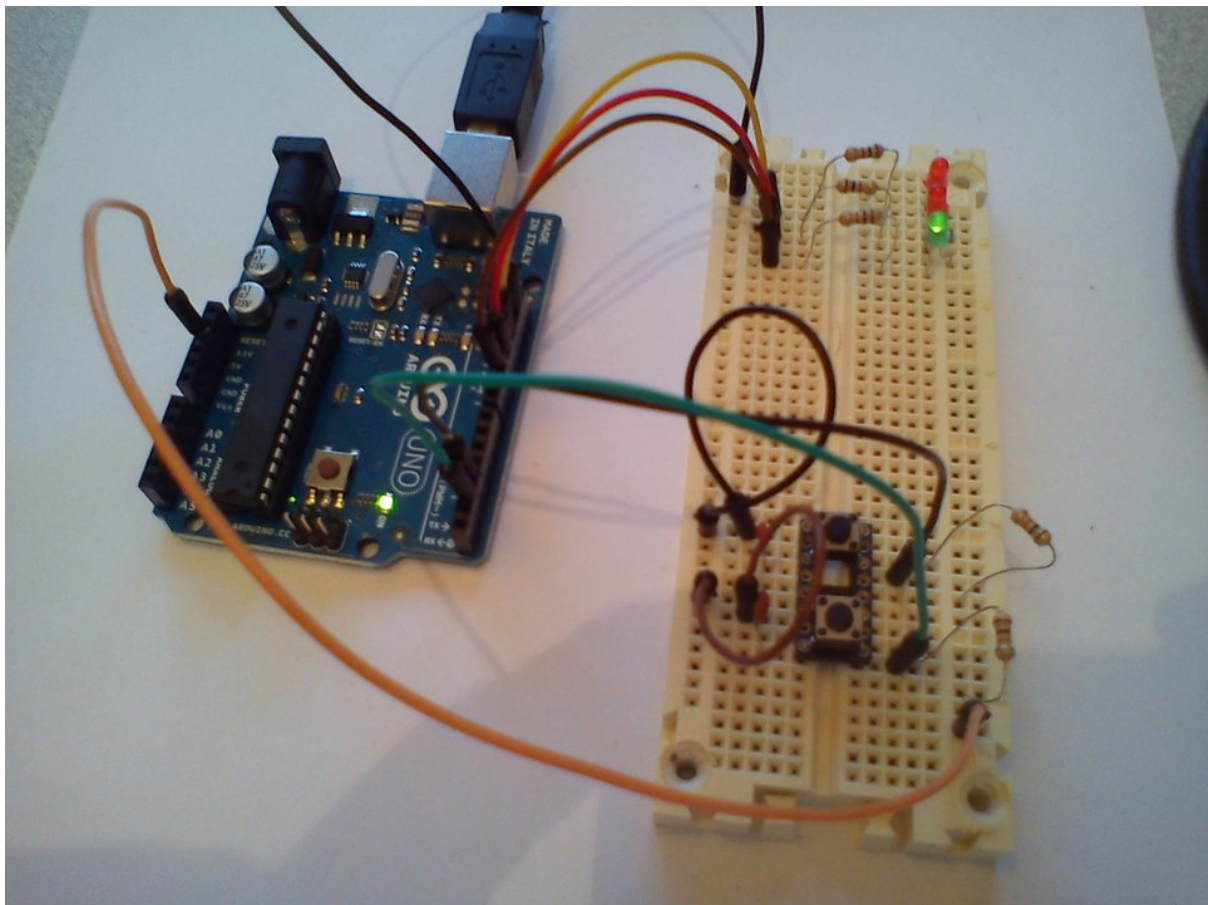


Figure 4.14 – Photo du TP : Baignade Intedite

->!(<https://www.youtube.com/watch?v=KncBc25Skxo>)<-

[[information]] | Bon Courage!

4.3.2 Correction !

J'espère que vous avez réussi à avoir un bout de solution ou une solution complète et que vous vous êtes amusé. Si vous êtes énervé sans avoir trouvé de solutions mais que vous avez cherché, ce n'est pas grave, regardez la correction et essayez de comprendre où et pourquoi vous avez fait une erreur. :)

4.3.2.1 Le schéma électronique

Commençons par le schéma électronique, voici le mien, entre vous et moi, seules les entrées/sorties ne sont probablement pas les mêmes. En effet, il est difficile de faire autrement que comme ceci :

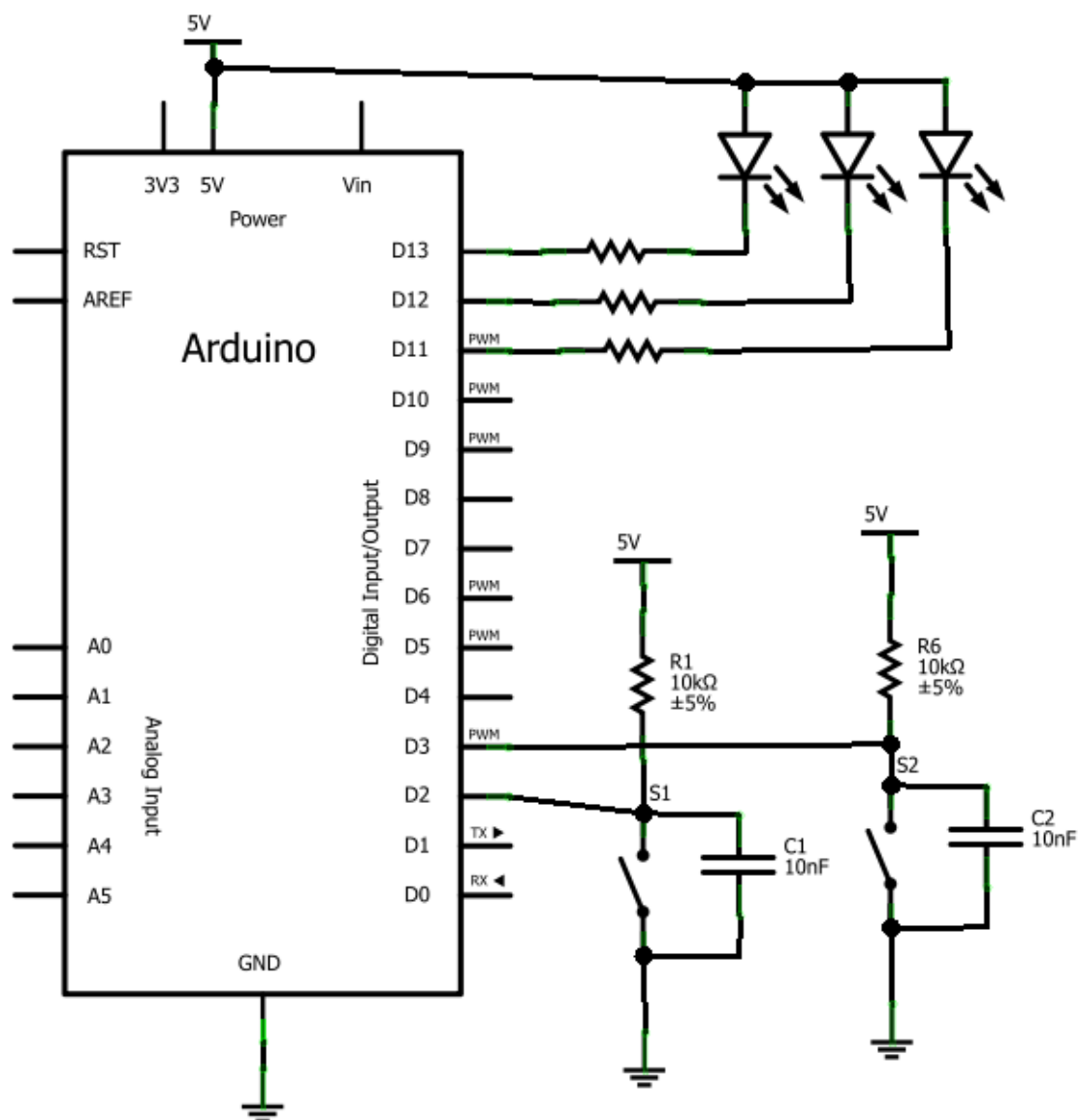


Figure 4.15 – [TP] Baignade Interdite – Schéma

Quelles raisons nous ont poussés à faire ces branchements ? Eh bien :

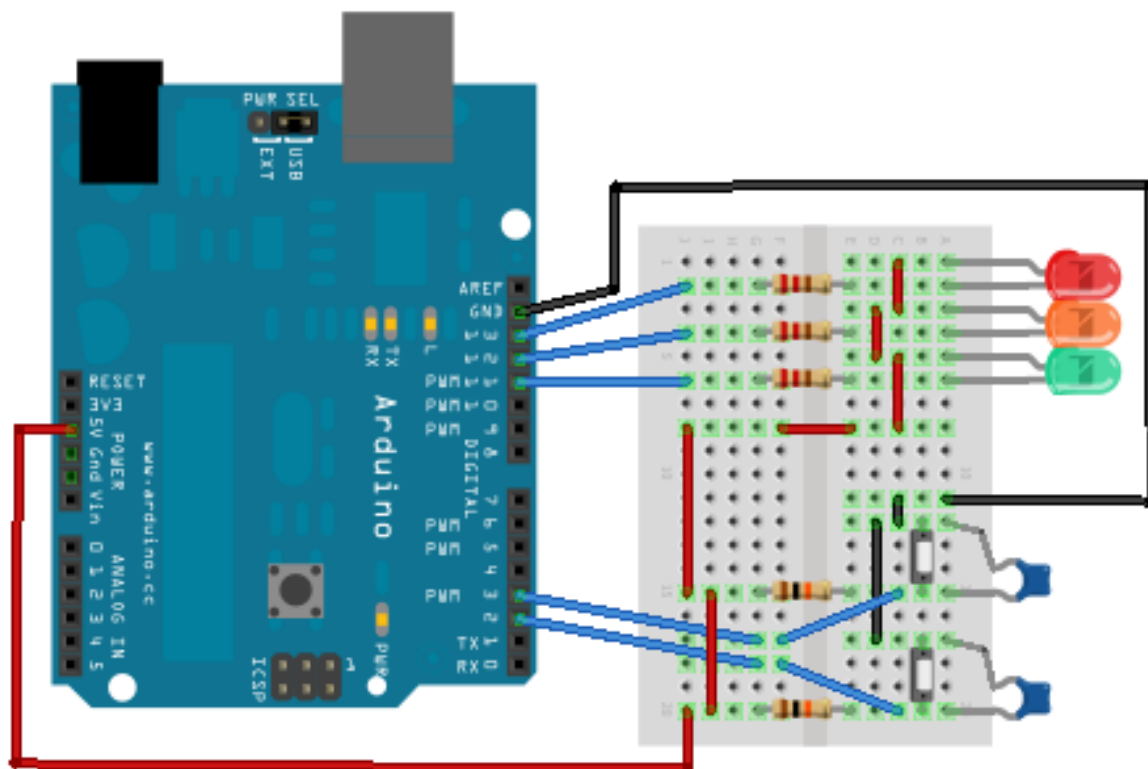


Figure 4.16 - [TP] Baignade Interdite - Montage

- On utilise la voie série, donc il ne faut pas brancher de boutons ou de LED sur les broches 0 ou 1 (broche de transmission/réception)
- On utilisera les LED à l'état bas, pour éviter que la carte Arduino délivre du courant
- Les rebonds des boutons sont filtrés par des condensateurs (au passage, les boutons sont actifs à l'état bas)

4.3.2.2 Les variables globales et la fonction setup()

Poursuivons notre explication avec les variables que nous allons utiliser dans le programme et les paramètres à déclarer dans la fonction setup().

4.3.2.2.1 Les variables globales

```
#####define VERT 0
#define ORANGE 1
#####define ROUGE 2

int etat = 0; // stock l'état de la situation (vert = 0, orange = 1, rouge = 2)
char mot[20]; // le mot lu sur la voie série

// numéro des broches utilisées
const int btn_SOS = 2;
const int btn_OK = 3;
const int leds[3] = {11,12,13}; // tableau de 3 éléments contenant les numéros de broches
```

Code : TP, initialisation des variables

Afin d'appliquer le cours, on se servira ici d'un tableau pour contenir les numéros des broches des LED. Cela nous évite de mettre trois fois `int leds_xxx` (vert, orange ou rouge). Bien entendu, dans notre cas, l'intérêt est faible, mais ça suffira pour l'exercice.

[[question]] | Et c'est quoi ça "#define" ?

Le "#define" est ce que l'on appelle **une directive de préprocesseur**. Lorsque le logiciel Arduino va compiler votre programme, il va remplacer le terme défini par la valeur qui le suit. Par exemple, chaque fois que le compilateur verra le terme VERT (en majuscule), il mettra la valeur 0 à la place. Tout simplement ! C'est exactement la même chose que d'écrire : `const int btn_SOS = 2;`

4.3.2.2.2 La fonction setup() Rien de particulier dans la fonction setup() par rapport à ce que vous avez vu précédemment, on initialise les variables

```
void setup()
{
    // On démarre la voie série avec une vitesse de 9600 bits/seconde
    Serial.begin(9600);

    // réglage des entrées/sorties
    // les entrées (2 boutons)
    pinMode(btn_SOS, INPUT);
    pinMode(btn_OK, INPUT);
}
```

```
// les sorties (3 LED) éteintes
for(int i=0; i<3; i++)
{
    pinMode(leds[i], OUTPUT);
    digitalWrite(leds[i], HIGH);
}
}
```

Code : TP, le setup

[[information]] | Dans le code précédent, l’astuce mise en œuvre est celle d’utiliser une boucle for pour initialiser les broches en tant que sorties et les mettre à l’état haut en même temps ! Sans cette astuce, le code d’initialisation (lignes 11 à 15) aurait été comme ceci :

```
// on définit les broches, où les LED sont connectées, en sortie
pinMode(led_vert, OUTPUT);
pinMode(led_rouge, OUTPUT);
pinMode(led_orange, OUTPUT);

// On éteint les LED
digitalWrite(led_vert, HIGH);
digitalWrite(led_orange, HIGH);
digitalWrite(led_rouge, HIGH);
```

Si vous n’utilisez pas cette astuce dans notre cas, ce n’est pas dramatique. En fait, cela est utilisé lorsque vous avez 20 ou même 100 LED et broches à initialiser ! C’est moins fatigant comme ça... Qui a dit programmeur ? o_O

4.3.2.3 La fonction principale et les autres

4.3.2.3.1 Algorithme Prenez l’habitude de *toujours rédiger un brouillon* de type algorithme ou quelque chose qui y ressemble avant de commencer à coder, cela vous permettra de mieux vous repérer dans l’endroit où vous en êtes sur l’avancement de votre programme. Voilà l’organigramme que j’ai fait lorsque j’ai commencé ce TP :

Et voilà en quelques mots la lecture de cet organigramme :

- On démarre la fonction loop
- Si on a un appui sur le bouton SOS :
 - On commence par faire clignoter la led rouge pour signaler l’alarme
 - Et on clignote tant que le sauveteur n’a pas appuyé sur le second bouton
- Sinon (ou si l’évènement est fini) on vérifie la présence d’un mot sur la voie série
 - S’il y a quelque chose à lire on va le récupérer
 - Sinon on continue dans le programme
- Enfin, on met à jour les drapeaux
- Puis on repart au début et refaisons le même traitement

Fort de cet outil, nous allons pouvoir coder proprement notre fonction loop() puis tout un tas de fonctions utiles tout autour.

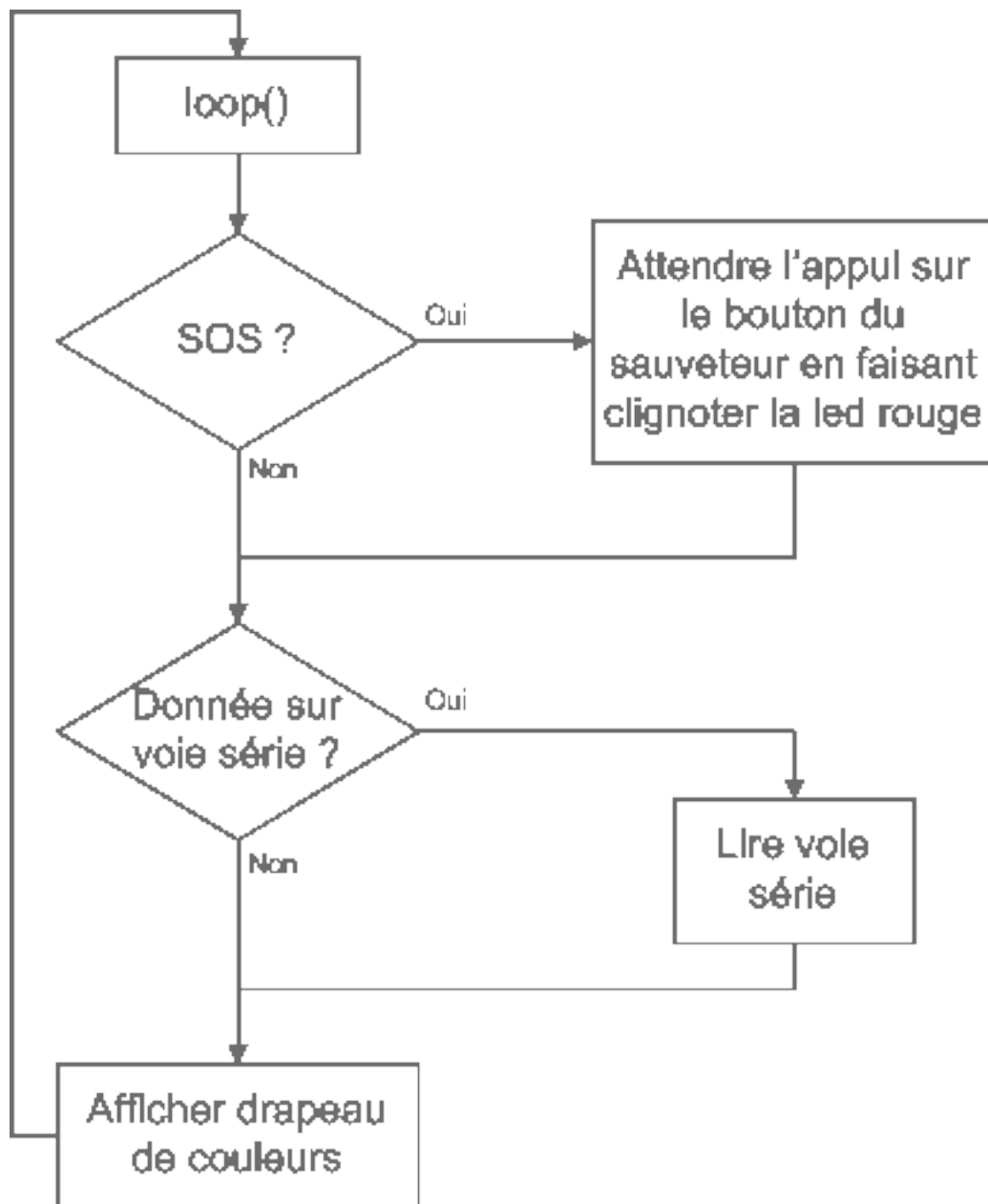


Figure 4.17 – Organigramme baignade interdite

4.3.2.3.2 Fonction loop() Voici dès maintenant la fonction `loop()`, qui va exécuter l'algorithme présenté ci-dessus. Vous voyez qu'il est assez "léger" car je fais appel à de nombreuses fonctions que j'ai créées. Nous verrons ensuite le rôle de ces différentes fonctions. Cependant, j'ai fait en sorte qu'elles aient toutes un nom explicite pour que le programme soit facilement compréhensible sans même connaître le code qu'elles contiennent.

```
void loop()
{
    // on regarde si le bouton SOS est appuyé
    if(digitalRead(btn_SOS) == LOW)
    {
        // si oui, on émet l'alerte en appelant la fonction prévue à cet effet
        alerte();
    }

    // puis on continue en vérifiant la présence de caractère sur la voie série
    // s'il y a des données disponibles sur la voie série
    // (Serial.available() renvoi un nombre supérieur à 0)
    if(Serial.available())
    {
        // alors on va lire le contenu de la réception
        lireVoieSerie();
        // on entre dans une variable la valeur retournée par
        // la fonction comparerMot()
        etat = comparerMot(mot);
    }
    // Puis on met à jour l'état des LED
    allumerDrapeau(etat);
}
```

Code : TP, la boucle principale

4.3.2.3.3 Lecture des données sur la voie série Afin de garder la fonction `loop` "légère", nous avons rajouté quelques fonctions annexes. La première sera celle de lecture de la voie série. Son job consiste à aller lire les informations contenues dans le buffer de réception du micro-contrôleur. On va lire les caractères en les stockant dans le tableau global `mot []` déclaré plus tôt. La lecture s'arrête sous deux conditions :

- Soit on a trop de caractère et donc on risque d'inscrire des caractères dans des variables n'existant pas (ici tableau limité à 20 caractères)
- Soit on a rencontré le caractère symbolisant la fin de ligne. Ce caractère est `'\n'`.

Voici maintenant le code de cette fonction :

```
// lit un mot sur la voie série (lit jusqu'à rencontrer le caractère '\n')
void lireVoieSerie(void)
{
    // variable locale pour l'incrémentatation des données du tableau
    int i = 0;
```



```

// on lit les caractères tant qu'il y en a
// OU si jamais le nombre de caractères lus atteint 19
// (limite du tableau stockant le mot - 1 caractère)
while(Serial.available() > 0 && i <= 19)
{
    // on enregistre le caractère lu
    mot[i] = Serial.read();
    // laisse un peu de temps entre chaque accès a la mémoire
    delay(10);
    // on passe à l'indice suivant
    i++;
}
// on supprime le caractère '\n'
// et on le remplace par celui de fin de chaine '\0'
mot[i] = '\0';
}

```

Code : TP, la fonction lireVoieSerie

4.3.2.3.4 Allumer les drapeaux Voilà un titre à en rendre fou plus d'un! Vous pouvez ranger vos briquets, on en aura pas besoin. ^^ Une deuxième fonction est celle permettant d'allumer et d'éteindre les LED. Elle est assez simple et prend un paramètre : le numéro de la LED à allumer. Dans notre cas : 0, 1 ou 2 correspondant respectivement à vert, orange, rouge. En passant le paramètre -1, on éteint toutes les LED.

```

/*
Rappel du fonctionnement du code qui précède celui-ci :
> lit un mot sur la voie série (lit jusqu'à rencontrer le caractère '\n')
Fonction allumerDrapeau() :
> Allume un des trois drapeaux
> paramètre : le numéro du drapeau à allumer
> (note : si le paramètre est -1, on éteint toutes les LED)
*/

```

```

void allumerDrapeau(int numLed)
{
    // On commence par éteindre les trois LED
    for(int j=0; j<3; j++)
    {
        digitalWrite(leds[j], HIGH);
    }
    // puis on allume une seule LED si besoin
    if(numLed != -1)
    {
        digitalWrite(leds[numLed], LOW);
    }
}

```

```

/* Note : vous pourrez améliorer cette fonction en
vérifiant par exemple que le paramètre ne

```

```
    dépasse pas le nombre présent de LED
    */
}
```

Code : TP, la fonction `allumerDrapeau`

Vous pouvez voir ici un autre intérêt du tableau utilisé dans la fonction `setup()` pour initialiser les LED. Une seule ligne permet de faire l'allumage de la LED concernée !

4.3.2.3.5 Faire clignoter la LED rouge Lorsque quelqu'un appui sur le bouton d'alerte, il faut immédiatement avertir les sauveteurs sur la zPlage. Dans le programme principal, on va détecter l'appui sur le bouton SOS. Ensuite, on passera dans la fonction `alerte()` codée ci-dessous. Cette fonction est assez simple. Elle va tout d'abord relever le temps à laquelle elle est au moment même (nombre de millisecondes écoulées depuis le démarrage). Ensuite, on va éteindre toutes les LED. Enfin, et c'est là le plus important, on va attendre du sauveteur un appui sur le bouton. TANT QUE cet appui n'est pas fait, on change l'état de la LED rouge toute les 250 millisecondes (choix arbitraire modifiable selon votre humeur). Une fois que l'appui du Sauveteur a été réalisé, on va repartir dans la boucle principale et continuer l'exécution du programme.

```
// Éteint les LED et fais clignoter la LED rouge
// en attendant l'appui du bouton "sauveteur"

void alerte(void)
{
    long temps = millis();
    boolean clignotant = false;
    allumerDrapeau(-1); // on éteint toutes les LED

    // tant que le bouton de sauveteur n'est pas appuyé
    // on fait clignoté la LED rouge
    while(digitalRead(btn_OK) != LOW)
    {
        // S'il s'est écoulé 250 ms ou plus depuis la dernière vérification
        if(millis() - temps > 250)
        {
            // on change l'état de la LED rouge
            // si clignotant était FALSE, il devient TRUE et inversement
            clignotant = !clignotant;
            // la LED est allumée au gré de la variable clignotant
            digitalWrite(leds[ROUGE], clignotant);
            // on se rappelle de la date de dernier passage
            temps = millis();
        }
    }
}
```

Code : TP, la fonction `alerte`

4.3.2.3.6 Comparer les mots Et voici maintenant le plus dur pour la fin, enfin j'exagère un peu. En effet, il ne vous reste plus qu'à comparer le mot reçu sur la voie série avec la banque

de données de mots possible. Nous allons donc effectuer cette vérification dans la fonction `comparerMot()`. Cette fonction recevra en paramètre la chaîne de caractères représentant le mot qui doit être vérifié et comparé. Elle renverra ensuite "l'état" (vert (0), orange (1) ou rouge (2)) qui en résulte. Si aucun mot n'a été reconnu, on renvoie "ORANGE" car incertitude.

```
int comparerMot(char mot[])
{
    // on compare les mots "VERT" (surveillant, calme)
    if(strcmp(mot, "surveillant") == 0)
    {
        return VERT;
    }
    if(strcmp(mot, "calme") == 0)
    {
        return VERT;
    }
    // on compare les mots "ORANGE" (vague)
    if(strcmp(mot, "vague") == 0)
    {
        return ORANGE;
    }
    // on compare les mots "ROUGE" (meduse, tempete, requin)
    if(strcmp(mot, "meduse") == 0)
    {
        return ROUGE;
    }
    if(strcmp(mot, "tempete") == 0)
    {
        return ROUGE;
    }
    if(strcmp(mot, "requin") == 0)
    {
        return ROUGE;
    }

    // si on a rien reconnu on renvoi ORANGE
    return ORANGE;
}
```

Code : TP, la fonction `comparerMot`

4.3.2.4 Code complet

Comme vous avez été sage jusqu'à présent, j'ai rassemblé pour vous le code complet de ce TP. Bien entendu, il va de pair avec le bon câblage des LED, placées sur les bonnes broches, ainsi que les boutons et le reste... Je vous fais cependant confiance pour changer les valeurs des variables si les broches utilisées sont différentes.

```
#####define VERT 0
#define ORANGE 1
```

```
#####define ROUGE 2

int etat = 0; // stock l'état de la situation (vert = 0, orange = 1, rouge = 2)
char mot[20]; // le mot lu sur la voie série

// numéro des broches utilisées
const int btn_SOS = 2;
const int btn_OK = 3;

// tableau de 3 éléments contenant les numéros de broches des LED
const int leds[3] = {11,12,13};

void setup()
{
    // On démarre la voie série avec une vitesse de 9600 bits/seconde
    Serial.begin(9600);

    // réglage des entrées/sorties
    // les entrées (2 boutons)
    pinMode(btn_SOS, INPUT);
    pinMode(btn_OK, INPUT);

    // les sorties (3 LED) éteintes
    for(int i=0; i<3; i++)
    {
        pinMode(leds[i], OUTPUT);
        digitalWrite(leds[i], HIGH);
    }
}

void loop()
{
    // on regarde si le bouton SOS est appuyé
    if(digitalRead(btn_SOS) == LOW)
    {
        // si oui, on émet l'alerte en appelant la fonction prévue à cet effet
        alerte();
    }

    // puis on continue en vérifiant la présence de caractère sur la voie série
    // s'il y a des données disponibles sur la voie série
    // (Serial.available() renvoi un nombre supérieur à 0)
    if(Serial.available())
    {
        // alors on va lire le contenu de la réception
        lireVoieSerie();
        // on entre dans une variable la valeur retournée
    }
}
```

```

        // par la fonction comparerMot()
        etat = comparerMot(mot);
    }
    // Puis on met à jour l'état des LED
    allumerDrapeau(etat);
}

// lit un mot sur la voie série (lit jusqu'à rencontrer le caractère '\n')
void lireVoieSerie(void)
{
    int i = 0; // variable locale pour l'incrémentation des données du tableau

    // on lit les caractères tant qu'il y en a
    // OU si jamais le nombre de caractères lus atteint 19
    // (limite du tableau stockant le mot - 1 caractère)
    while(Serial.available() > 0 && i <= 19)
    {
        mot[i] = Serial.read(); // on enregistre le caractère lu
        delay(10); // laisse un peu de temps entre chaque accès a la mémoire
        i++; // on passe à l'indice suivant
    }
    // on supprime le caractère '\n'
    // et on le remplace par celui de fin de chaine '\0'
    mot[i] = '\0';
}

/*
Rappel du fonctionnement du code qui précède celui-ci :
> lit un mot sur la voie série (lit jusqu'à rencontrer le caractère '\n')
Fonction allumerDrapeau() :
> Allume un des trois drapeaux
> paramètre : le numéro du drapeau à allumer
> (note : si le paramètre est -1, on éteint toutes les LED)
*/

void allumerDrapeau(int numLed)
{
    // On commence par éteindre les trois LED
    for(int j=0; j<3; j++)
    {
        digitalWrite(leds[j], HIGH);
    }
    // puis on allume une seule LED si besoin
    if(numLed != -1)
    {
        digitalWrite(leds[numLed], LOW);
    }
}

```

```
    }

    /* Note : vous pourrez améliorer cette fonction en
    vérifiant par exemple que le paramètre ne
    dépasse pas le nombre présent de LED
    */
}

// Éteint les LED et fais clignoter la LED rouge
// en attendant l'appui du bouton "sauveteur"

void alerte(void)
{
    long temps = millis();
    boolean clignotant = false;
    allumerDrapeau(-1); // on éteint toutes les LED

    // tant que le bouton de sauveteur n'est pas appuyé
    // on fait clignoté la LED rouge
    while(digitalRead(btn_OK) != LOW)
    {
        // S'il s'est écoulé 250 ms ou plus depuis la dernière vérification
        if(millis() - temps > 250)
        {
            // on change l'état de la LED rouge
            // si clignotant était FALSE, il devient TRUE et inversement
            clignotant = !clignotant;
            // la LED est allumée au gré de la variable clignotant
            digitalWrite(leds[ROUGE], clignotant);
            // on se rappelle de la date de dernier passage
            temps = millis();
        }
    }
}

int comparerMot(char mot[])
{
    // on compare les mots "VERT" (surveillant, calme)
    if(strcmp(mot, "surveillant") == 0)
    {
        return VERT;
    }
    if(strcmp(mot, "calme") == 0)
    {
        return VERT;
    }
}
```

```

// on compare les mots "ORANGE" (vague)
if(strcmp(mot, "vague") == 0)
{
    return ORANGE;
}
// on compare les mots "ROUGE" (meduse, tempete, requin)
if(strcmp(mot, "meduse") == 0)
{
    return ROUGE;
}
if(strcmp(mot, "tempete") == 0)
{
    return ROUGE;
}
if(strcmp(mot, "requin") == 0)
{
    return ROUGE;
}

// si on a rien reconnu on renvoi ORANGE
return ORANGE;
}

```

Code : TP, code complet

[[attention]] | Je rappelle que si vous n'avez pas réussi à faire fonctionner complètement votre programme, aidez vous de celui-ci pour comprendre le pourquoi du comment qui empêche votre programme de fonctionner correctement ! A bons entendeurs. ;)

4.3.3 Améliorations

Je peux vous proposer quelques idées d'améliorations que je n'ai pas mises en oeuvre, mais qui me sont passées par la tête au moment où j'écrivais ces lignes :

4.3.3.0.1 Améliorations logicielles Avec la nouvelle version d'Arduino, la version 1.0, ; il existe une fonction `SerialEvent()` qui est exécutée dès qu'il y a un évènement sur la voie série du micro-contrôleur. Je vous laisse le soin de chercher à comprendre comment elle fonctionne et s'utilise, sur [cette page](#).

4.3.3.0.2 Améliorations matérielles

- On peut par exemple automatiser le changement d'un drapeau en utilisant un système mécanique avec un ou plusieurs moteurs électriques. Ce serait dans le cas d'utilisation réelle de ce montage, c'est-à-dire sur une plage...
- Une liaison filaire entre un PC et une carte Arduino, ce n'est pas toujours la joie. Et puis bon, ce n'est pas toujours facile d'avoir un PC sous la main pour commander ce genre de montage. Alors pourquoi ne pas rendre la connexion sans-fil en utilisant par exemple des modules XBee ? Ces petits modules permettent une connexion sans-fil utilisant la voie série pour communiquer. Ainsi, d'un côté vous avez la télécommande (à base d'Arduino et

d'un module XBee) de l'autre vous avez le récepteur, toujours avec un module XBee et une Arduino, puis le montage de ce TP avec l'amélioration précédente. Sérieusement si ce montage venait à être réalité avec les améliorations que je vous ai données, prévenez-moi par MP et faites en une vidéo pour que l'on puisse l'ajouter en lien ici même! ^^

Voilà une grosse tâche de terminée! J'espère qu'elle vous a plu même si vous avez pu rencontrer des difficultés. Souvenez-vous, "à vaincre sans difficulté on triomphe sans gloire", donc tant mieux si vous avez passé quelques heures dessus et, surtout, j'espère que vous avez appris des choses et pris du plaisir à faire votre montage, le dompter et le faire fonctionner comme vous le souhaitiez!

4.4 [Annexe] Ordinateur et voie série dans un autre langage de programmation

Maintenant que vous savez comment utiliser la voie série avec Arduino, il peut être bon de savoir comment visualiser les données envoyées avec vos propres programmes (l'émulateur terminal Windows ou le moniteur série Arduino ne comptent pas :P). Cette annexe a donc pour but de vous montrer comment utiliser la voie série avec quelques langages de programmation. Les langages utilisés ci-dessous ont été choisis arbitrairement en fonction de mes connaissances, car je ne connais pas tous les langages possibles et une fois vu quelques exemples, il ne devrait pas être trop dur de l'utiliser avec un autre langage. Nous allons donc travailler avec :

Afin de se concentrer sur la partie "Informatique", nous allons reprendre un programme travaillé précédemment dans le cours. Ce sera celui de l'exercice : **Attention à la casse**. Pensez donc à le charger dans votre carte Arduino avant de faire les tests. :P

4.4.1 En C++ avec Qt

[[attention]] | Avant de commencer cette sous-partie, il est indispensable de connaître la programmation en C++ et savoir utiliser le framework Qt. Si vous ne connaissez pas tout cela, vous pouvez toujours aller vous renseigner avec un **tutoriel C++**!

[[question]] | Le C++, OK, mais pourquoi Qt?

J'ai choisi de vous faire travailler avec Qt pour plusieurs raisons d'ordres pratiques.

- Qt est multiplateforme, donc les réfractaires à Linux (ou à Windows) pourront quand même travailler.
- Dans le même ordre d'idée, nous allons utiliser une librairie tierce pour nous occuper de la voie série. Ainsi, aucun problème pour interfacer notre matériel que l'on soit sur un système ou un autre!
- Enfin, j'aime beaucoup Qt et donc je vais vous en faire profiter :)

En fait, sachez que chaque système d'exploitation à sa manière de communiquer avec les périphériques matériels. L'utilisation d'une librairie tierce nous permet donc de faire abstraction de tout cela. Sinon il m'aurait fallu faire un tutoriel par OS, ce qui, on l'imagine facilement, serait une perte de temps (écrire trois fois *environ* les mêmes choses) et vraiment galère à maintenir.

4.4.1.1 Installer QextSerialPort

QextSerialPort est une librairie tierce réalisée par un membre de la communauté Qt. Pour utiliser cette librairie, il faut soit la compiler, soit utiliser les sources directement dans votre projet.

4.4.1.1.1 1ère étape : télécharger les sources Le début de tout cela commence donc par récupérer les sources de la librairie. Pour cela, rendez-vous sur [la page google code](#) du projet. A partir d'ici vous avez plusieurs choix. Soit vous récupérez les sources en utilisant le gestionnaire de source mercurial (Hg). Il suffit de faire un clone du dépôt avec la commande suivante :

```
hg clone https:// code.google.com/p/qextserialport/
```

Sinon, vous pouvez [récupérer les fichiers un par un](#) (une dizaine). C'est plus contraignant mais ça marche aussi si vous n'avez jamais utilisé de gestionnaire de sources (mais c'est vraiment plus contraignant !)

[[attention]] | Cette dernière méthode est vraiment **déconseillée**. En effet, vous vous retrouverez avec le strict minimum (fichiers sources sans exemples ou docs).

La manipulation est la même sous Windows ou Linux !

4.4.1.1.2 Compiler la librairie Maintenant que nous avons tous nos fichiers, nous allons pouvoir compiler la librairie. Pour cela, nous allons laisser Qt travailler à notre place.

- Démarrez QtCreator et ouvrez le fichier .pro de QextSerialPort
- Compilez...
- C'est fini !

Normalement vous avez un nouveau dossier à côté de celui des sources qui contient des exemples, ainsi que les **librairies** QExtSerialPort.

4.4.1.1.3 Installer la librairie : Sous Linux Une fois que vous avez compilé votre nouvelle librairie, vous allez devoir placer les fichiers aux bons endroits pour les utiliser. Les librairies, qui sont apparues dans le dossier "build" qui vient d'être créé, vont être déplacées vers le dossier /usr/lib. Les fichiers sources qui étaient avec le fichier ".pro" pour la compilation sont à copier dans un sous-dossier "QextSerialPort" dans le répertoire de travail de votre projet courant.

[[attention]] | A priori il y aurait un bug avec la compilation en mode release (la librairie générée ne fonctionnerait pas correctement). Je vous invite donc à compiler aussi la debug et travailler avec.

4.4.1.1.4 Installer la librairie : Sous Windows [[erreur]] | Ce point est en cours de rédaction, merci de patienter avant sa mise en ligne. :)

4.4.1.1.5 Infos à rajouter dans le .pro Dans votre nouveau projet Qt pour traiter avec la voie série, vous aller rajouter les lignes suivantes à votre .pro :

```
INCLUDEPATH += QextSerialPort
```

```
CONFIG(debug, debug|release):LIBS += -lqextserialportd  
else:LIBS += -lqextserialport
```

Code : Ajout au .pro

La ligne “INCLUDEPATH” représente le dossier où vous avez mis les fichiers sources de QextSerialPort. Les deux autres lignes font le lien vers les bibliothèques copiées plus tôt (les .so ou les .dll selon votre OS).

4.4.1.2 Les trucs utiles

4.4.1.2.1 L’interface utilisée Comme expliqué dans l’introduction, nous allons toujours travailler sur le même exercice et juste changer le langage étudié. Voici donc l’interface sur laquelle nous allons travailler, et quels sont les noms et les types d’objets instanciés :



Figure 4.18 – interface Qt

Cette interface possède deux parties importantes : La **gestion de la connexion** (en haut) et **l’échange de résultat** (milieu -> émission, bas -> réception). Dans la partie supérieure, nous allons choisir le port de l’ordinateur sur lequel communiquer ainsi que la vitesse de cette communication. Ensuite, deux boîtes de texte sont présentes. L’une pour écrire du texte à émettre, et l’autre affichant le texte reçu. Voici les noms que j’utiliserai dans mon code :

->

Widget	Nom	Rôle
QComboBox	comboPort	Permet de choisir le port série
QComboBox	comboVitesse	Permet de choisir la vitesse de communication
QPushButton	btnconnexion	(Dé)Connecte la voie série (bouton “checkable”)
QTextEdit	boxEmission	Nous écrivons ici le texte à envoyer

Widget	Nom	Rôle
QTextEdit	boxReception	Ici apparaîtra le texte à recevoir

Table 4.6 – Liste des widgets utilisé

<-

4.4.1.2.2 Lister les liaisons séries Avant de créer et d'utiliser l'objet pour gérer la voie série, nous allons en voir quelques-uns pouvant être utiles. Tout d'abord, nous allons apprendre à obtenir la liste des ports série présents sur notre machine. Pour cela, un objet a été créé spécialement, il s'agit de `QextPortInfo`. Voici un exemple de code leur permettant de fonctionner ensemble :

```
// L'objet mentionnant les infos
QextSerialEnumerator enumerateur;
// on met ces infos dans une liste
QList<QextPortInfo> ports = enumerateur.getPorts();
```

Code : Récupération de la liste des ports séries

Une fois que nous avons récupéré une énumération de tous les ports, nous allons pouvoir les ajouter au combobox qui est censé les afficher (`comboPort`). Pour cela on va parcourir la liste construite précédemment et ajouter à chaque fois une item dans le menu déroulant :

```
// on parcourt la liste des ports
for(int i=0; i<ports.size(); i++)
    ui->ComboPort->addItem(ports.at(i).physName);
```

Code : Remplissage du combobox des ports séries

[[attention]] | Les ports sont nommés différemment sous Windows et Linux, ne soyez donc pas surpris avec mes captures d'écrans, elles viennent toutes de Linux.

Une fois que la liste des ports est faite (attention, certains ports ne sont connectés à rien), on va construire la liste des vitesses, pour se laisser le choix le jour où l'on voudra faire une application à une vitesse différente. Cette opération n'est pas très compliquée puisqu'elle consiste simplement à ajouter des items dans la liste déroulante "comboVitesse".

```
ui->comboVitesse->addItem("300");
ui->comboVitesse->addItem("1200");
ui->comboVitesse->addItem("2400");
ui->comboVitesse->addItem("4800");
ui->comboVitesse->addItem("9600");
ui->comboVitesse->addItem("14400");
ui->comboVitesse->addItem("19200");
ui->comboVitesse->addItem("38400");
ui->comboVitesse->addItem("57600");
ui->comboVitesse->addItem("115200");
```

Code : Remplissage du combobox des vitesses

Votre interface est maintenant prête. En la démarrant maintenant vous devriez être en mesure de voir s'afficher les noms des ports séries existant sur l'ordinateur ainsi que les vitesses. Un clic sur le bouton ne fera évidemment rien puisque son comportement n'est pas encore implémenté.

4.4.1.2.3 Gérer une connexion Lorsque tous les détails concernant l'interface sont terminés, nous pouvons passer au cœur de l'application : la *communication série*. La première étape pour pouvoir faire une communication est de se connecter (tout comme vous vous connectez sur une borne WiFi avant de communiquer et d'échanger des données avec cette dernière). C'est le rôle de notre bouton de connexion. A partir du système de slot automatique, nous allons créer une fonction qui va recevoir le clic de l'utilisateur. Cette fonctioninstanciera un objet QextSerialPort pour créer la communication, règlera cet objet et enfin ouvrira le canal. Dans le cas où le bouton était déjà coché (puisque'il sera "checkable" rappelons-le) nous ferons la déconnexion, puis la destruction de l'objet QextSerialPort créé auparavant. Pour commencer nous allons donc déclarer les objets et méthodes utiles dans le .h de la classe avec laquelle nous travaillons :

```
private :  
// l'objet représentant le port  
QextSerialPort * port;  
  
// une fonction utile que j'expliquerais après  
BaudRateType getBaudRateFromString(QString baudRate);  
  
private slots :  
// le slot automatique du bouton de connexion  
void on_btnconnexion_clicked();
```

Ensuite, il nous faudra instancier le slot du bouton afin de traduire un comportement. Pour rappel, il devra :

- Créer l'objet "port" de type QextSerialPort
- Le régler avec les bons paramètres
- Ouvrir la voie série

Dans le cas où la voie série est déjà ouverte (le bouton est déjà appuyé) on devra la fermer et détruire l'objet. Voici le code commenté permettant l'ouverture de la voie série (quelques précisions viennent ensuite) :

```
// Slot pour le click sur le bouton de connexion  
void Fenetre::on_btnconnexion_clicked() {  
    // deux cas de figures à gérer,  
    // soit on coche (connecte), soit on décoche (déconnecte)  
  
    // on coche -> connexion  
    if(ui->btnconnexion->isChecked()) {  
        // on essaie de faire la connexion avec la carte Arduino  
        // on commence par créer l'objet port série  
        port = new QextSerialPort();  
        // on règle le port utilisé (sélectionné dans la liste déroulante)  
        port->setPortName(ui->ComboPort->currentText());  
        // on règle la vitesse utilisée
```

```

port->setBaudRate(
    getBaudRateFromString(ui->comboVitesse->currentText()));
// quelques réglages pour que tout marche bien
port->setParity(PAR_NONE); // parité
port->setStopBits(STOP_1); // nombre de bits de stop
port->setDataBits(DATA_8); // nombre de bits de données
port->setFlowControl(FLOW_OFF); // pas de contrôle de flux
// on démarre!
port->open(QextSerialPort::ReadWrite);
// change le message du bouton
ui->btnconnexion->setText("Deconnecter");

// on fait la connexion pour pouvoir obtenir les évènements
connect(port,SIGNAL(readyRead()), this, SLOT(readData()));
connect(ui->boxEmission,SIGNAL(textChanged()),this,SLOT(sendData()));
}
else {
    // on se déconnecte de la carte Arduino
port->close();
// puis on détruit l'objet port série devenu inutile
delete port;
ui->btnconnexion->setText("Connecter");
}
}
}

```

Code : Gestion du bouton de connexion

Ce code n'est pas très compliqué à comprendre. Cependant quelques points méritent votre attention. Pour commencer, pour régler la vitesse du port série on fait appel à la fonction "setBaudRate". Cette fonction prend un paramètre de type BaudRateType qui fait partie d'une énumération de QextSerialPort. Afin de faire le lien entre le comboBox qui possède des chaînes et le type particulier attendu, on crée et utilise la fonction "getBaudRateFromString". A partir d'un simple BaudRateType.

```

BaudRateType Fenetre::getBaudRateFromString(QString baudRate) {
    int vitesse = baudRate.toInt();
    switch(vitesse) {
        case(300):return BAUD300;
        case(1200):return BAUD1200;
        case(2400):return BAUD2400;
        case(4800):return BAUD4800;
        case(9600):return BAUD9600;
        case(14400):return BAUD14400;
        case(19200):return BAUD19200;
        case(38400):return BAUD38400;
        case(57600):return BAUD57600;
        case(115200):return BAUD115200;
        default :return BAUD9600;
    }
}
}

```

Code : Gestion du changement de vitesse

Un autre point important à regarder est l'utilisation de la fonction `open()` de l'objet `QextSerialPort`. En effet, il existe plusieurs façons d'ouvrir un port série :

- En lecture seule → `QextSerialPort::ReadOnly`
- En écriture seule → `QextSerialPort::WriteOnly`
- En lecture/écriture → `QextSerialPort::ReadWrite`

Ensuite, on connecte simplement les signaux émis par la voie série et par la boîte de texte servant à l'émission (que l'on verra juste après). Enfin, lorsque l'utilisateur re-clic sur le bouton, on passe dans le `NULL`.

[[attention]] | Ce code présente le principe et n'est pas parfait ! Il faudrait par exemple s'assurer que le port est bien ouvert avant d'envoyer des données (faire un test `if (port->isOpen())` par exemple).

4.4.1.3 Émettre et recevoir des données

Maintenant que la connexion est établie, nous allons pouvoir envoyer et recevoir des données. Ce sera le rôle de deux slots qui ont été brièvement évoqués dans la fonction `connect()` du code de connexion précédent.

4.4.1.3.1 Émettre des données L'émission des données se fera dans le slot "sendData". Ce slot sera appelé à chaque fois qu'il y aura une modification du contenu de la boîte de texte "boxEmission". Pour l'application concernée (l'envoi d'un seul caractère), il nous suffit de chercher le dernier caractère tapé. On récupère donc le dernier caractère du texte contenu dans la boîte avant de l'envoyer sur la voie série. L'envoi de texte se fait à partir de la fonction `toAscii()` et on peut donc les utiliser directement. Voici le code qui illustre toutes ces explications (ne pas oublier de mettre les déclarations des slots dans le `.h`) :

```
void Fenetre::sendData() {  
    // On récupère le dernier caractère tapé  
    QString caractere = ui->boxEmission->toPlainText().right(1);  
    // si le port est instancié (donc ouvert a priori)  
    if(port != NULL)  
        port->write(caractere.toAscii());  
}
```

Code : Envoi de données

4.4.1.3.2 Recevoir des données Le programme étudié est censé nous répondre en renvoyant le caractère émis mais dans une *casse opposée* (majuscule contre minuscule et vice versa). En temps normal, deux politiques différentes s'appliquent pour savoir si des données sont arrivées. La première est d'aller voir de manière régulière (ou pas) si des caractères sont présents dans le tampon de réception de la voie série. Cette méthode dite de *Polling n'est pas très fréquemment utilisée. La seconde est de déclencher un évènement lorsque des données arrivent sur la voie série. C'est la forme qui est utilisée par défaut par l'objet `readyRead()` est émis par l'objet et peut donc être connecté à un slot. Pour changer le mode de fonctionnement, il faut utiliser la méthode

`QextSerialPort` : `EventDriven` pour la seconde (par défaut). Comme la connexion entre le signal et le slot est créée dans la fonction de connexion, il ne nous reste qu'à écrire le comportement du slot de réception lorsqu'une donnée arrive. Le travail est simple et se résume en deux étapes :

- Lire le caractère reçu grâce à la fonction `QextSerialPort`
- Le copier dans la boîte de texte "réception"

```
void Fenetre::readData() {
    QByteArray array = port->readAll();
    ui->boxReception->insertPlainText(array);
}
```

Code : Réception de données

Et voilà, vous êtes maintenant capable de travailler avec la voie série dans vos programmes Qt en C++. Au risque de me répéter, je suis conscient qu'il y a des lacunes en terme de "sécurité" et d'efficacité. Ce code a pour but de vous montrer les bases de la classe pour que vous puissiez continuer ensuite votre apprentissage. En effet, la programmation C++/Qt n'est pas le sujet de ce tutoriel. :ninja : Nous vous serons donc reconnaissants de ne pas nous harceler de commentaires relatifs au tuto pour nous dire "bwaaaa c'est mal codéééééé". Merci! :)

4.4.2 En C# (.Net)

[[attention]] | Dans cette partie (comme dans les précédentes) je pars du principe que vous connaissez le langage et avez déjà dessiné des interfaces et créé des actions sur des boutons par exemple. Cette sous-partie n'est pas là pour vous apprendre le C#!

Là encore je vais reprendre la même structure que les précédentes sous-parties.

4.4.2.1 Les trucs utiles

4.4.2.1.1 L'interface et les imports Voici tout de suite l'interface utilisée! Je vous donnerai juste après le nom que j'utilise pour chacun des composants (et tant qu'à faire je vous donnerai aussi leurs types).

Comme cette interface est la même pour tout ce chapitre, nous retrouvons comme d'habitude le bandeau pour gérer la connexion ainsi que les deux boîtes de texte pour l'émission et la réception des données. Voici les types d'objets et leurs noms pour le bandeau de connexion :

->

Composant	Nom	Rôle
System.Windows.Forms.ComboBox	comboPort	Permet de choisir le port série
System.Windows.Forms.ComboBox	comboVitesse	Permet de choisir la vitesse de communication
System.Windows.Forms.Button	btnConnexion	(Dé)Connecte la voie série (bouton "checkable")
System.Windows.Forms.TextBox	boxEmission	Nous écrirons ici le texte à envoyer
System.Windows.Forms.TextBox	boxReception	Ici apparaîtra le texte à recevoir

Table 4.7 – Liste des widgets utilisé

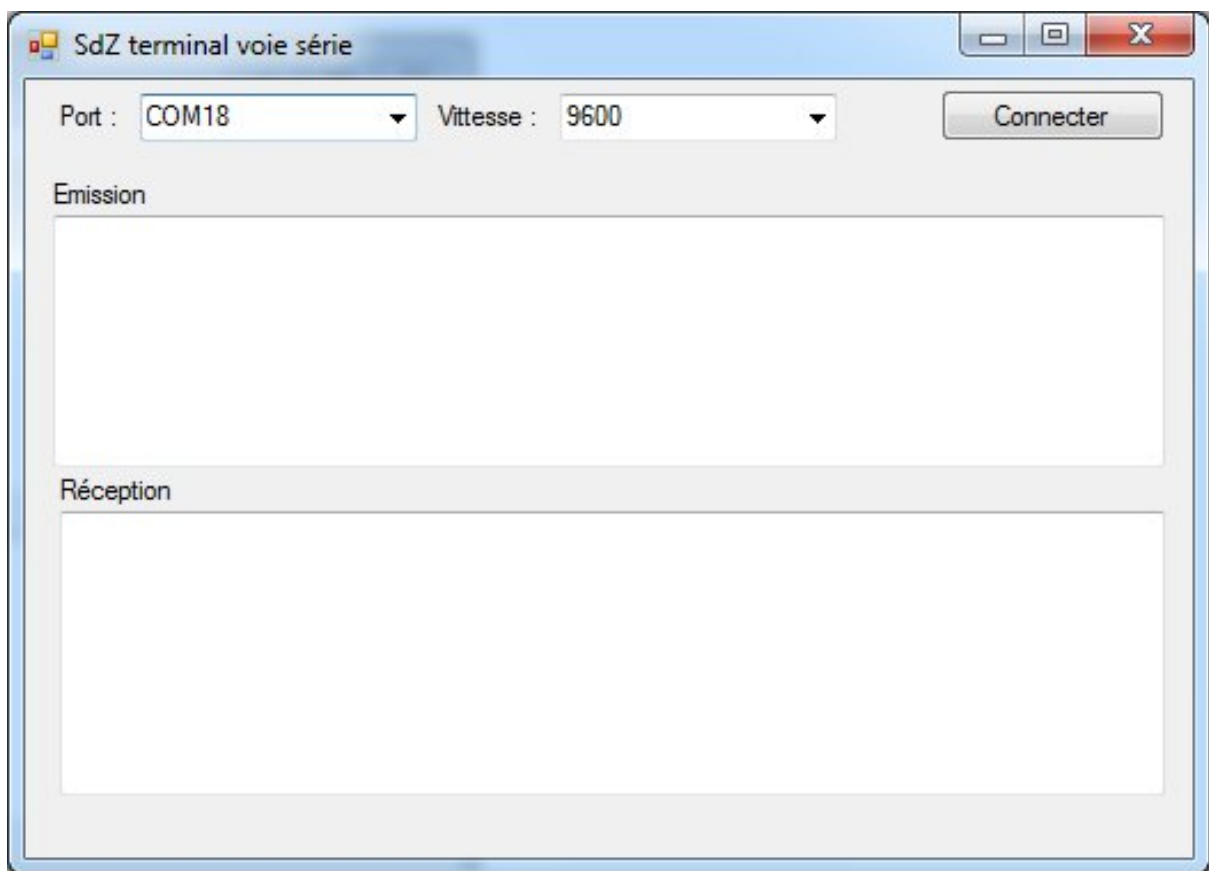


Figure 4.19 – L'interface en C#

<-

Avant de commencer les choses marrantes, nous allons d'abord devoir ajouter une librairie : celle des liaisons séries. Elle se nomme simplement `using System.IO.Ports`; . Nous allons en profiter pour rajouter une variable membre de la classe de type `SerialPort` que j'appellerai "port". Cette variable représentera, vous l'avez deviné, notre port série !

```
SerialPort port
```

Maintenant que tous les outils sont prêts, nous pouvons commencer !

4.4.2.1.2 Lister les liaisons séries La première étape sera de lister l'ensemble des liaisons séries sur l'ordinateur. Pour cela nous allons nous servir d'une fonction statique de la classe `String`. Chaque case du tableau sera une chaîne de caractère comportant le nom d'une voie série. Une fois que nous avons ce tableau, nous allons l'ajouter sur l'interface, dans la liste déroulante prévue à cet effet pour pouvoir laisser le choix à l'utilisateur au démarrage de l'application. Dans le même élan, on va peupler la liste déroulante des vitesses avec quelques-unes des vitesses les plus courantes. Voici le code de cet ensemble. Personnellement je l'ai ajouté dans la méthode `InitializeComponent()` qui charge les composants.

```
private void Form1_Load(object sender, EventArgs e)
{
    // on commence par lister les voies séries présentes
    String[] ports = SerialPort.GetPortNames(); // fonction statique
    // on ajoute les ports au combo box
    foreach (String s in ports)
        comboPort.Items.Add(s);

    // on ajoute les vitesses au combo des vitesses
    comboVitesse.Items.Add("300");
    comboVitesse.Items.Add("1200");
    comboVitesse.Items.Add("2400");
    comboVitesse.Items.Add("4800");
    comboVitesse.Items.Add("9600");
    comboVitesse.Items.Add("14400");
    comboVitesse.Items.Add("19200");
    comboVitesse.Items.Add("38400");
    comboVitesse.Items.Add("57600");
    comboVitesse.Items.Add("115200");
}
```

Code : Intialisation du combobox des vitesses

Si vous lancez votre programme maintenant avec la carte Arduino connectée, vous devriez avoir le choix des vitesses mais aussi d'au moins un port série. Si ce n'est pas le cas, il faut trouver pourquoi avant de passer à la suite (Vérifiez que la carte est bien connectée par exemple).

4.4.2.1.3 Gérer une connexion Une fois que la carte est reconnue et que l'on voit bien son port dans la liste déroulante, nous allons pouvoir ouvrir le port pour établir le canal de communication entre Arduino et l'ordinateur. Comme vous vous en doutez sûrement, la fonction que nous allons écrire est celle du clic sur le bouton. Lorsque nous cliquons sur le bouton de connexion, deux actions peuvent être effectuées selon l'état précédent. Soit nous nous connectons, soit nous nous déconnectons. Les deux cas seront gérés en regardant le texte contenu dans le bouton ("Connecter" ou "Déconnecter"). Dans le cas de la déconnexion, il suffit de fermer le port à l'aide de la méthode `close()`. Dans le cas de la connexion, plusieurs choses sont à faire. Dans l'ordre, nous allons commencer par instancier un nouvel objet de type `BaudRate` et ainsi de suite. Voici le code commenté pour faire tout cela. Il y a cependant un dernier point évoqué rapidement juste après et sur lequel nous reviendrons plus tard.

```
private void btnConnexion_Click(object sender, EventArgs e)
{
    // on gère la connexion/déconnexion
    if (btnConnexion.Text == "Connecter") // alors on connecte
    {
        // crée un nouvel objet voie série
        port = new SerialPort();
        // règle la voie série
        // parse en int le combo des vitesses
        port.BaudRate = int.Parse(comboVitesse.SelectedItem.ToString());
        port.DataBits = 8;
        port.StopBits = StopBits.One;
        port.Parity = Parity.None;
        // récupère le nom sélectionné
        port.PortName = comboPort.SelectedItem.ToString();

        // ajoute un gestionnaire de réception
        // pour la réception de donnée sur la voie série
        port.DataReceived +=
            new SerialDataReceivedEventHandler(DataReceivedHandler);

        port.Open(); // ouvre la voie série

        btnConnexion.Text = "Déconnecter";
    }
    else // sinon on déconnecte
    {
        port.Close(); // ferme la voie série
        btnConnexion.Text = "Connecter";
    }
}
```

Code : Gestion du bouton de connexion

Le point qui peut paraître étrange est la ligne 16, avec la propriété `Handler()` qui devra être appelée lorsque des données arriveront. Je vais vous demander d'être patient, nous en reparlerons

plus tard lorsque nous verrons la réception de données. A ce stade du développement, lorsque vous lancez votre application vous devriez pouvoir sélectionner une voie série, une vitesse, et cliquer sur “Connecter” et “Déconnecter” sans aucun bug.

4.4.2.2 Émettre et recevoir des données

La voie série est prête à être utilisée ! La connexion est bonne, il ne nous reste plus qu'à envoyer les données et espérer avoir quelque chose en retour. ;)

4.4.2.2.1 Envoyer des données Pour envoyer des données, une fonction toute prête existe pour les objets char qui serait envoyé un par un. Dans notre cas d'utilisation, c'est ce deuxième cas qui nous intéresse. Nous allons donc implémenter la méthode `TextChanged` du composant “`boxEmission`” afin de détecter chaque caractère entré par l'utilisateur. Ainsi, nous enverrons chaque nouveau caractère sur la voie série, un par un. Le code suivant, commenté, vous montre la voie à suivre.

```
// lors d'un envoi de caractère
private void boxEmission_TextChanged(object sender, EventArgs e)
{
    // met le dernier caractère dans un tableau avec une seule case le contenant
    char[] car = new char[] {boxEmission.Text[boxEmission.TextLength-1]};
    // on s'assure que le port est existant et ouvert
    if(port!=null && port.IsOpen)
    {
        // envoie le tableau de caractère,
        // depuis la position 0, et envoie 1 seul élément
        port.Write(car,0,1);
    }
}
```

Code : Envoi de données

4.4.2.2.2 Recevoir des données La dernière étape pour pouvoir gérer de manière complète notre voie série est de pouvoir afficher les caractères reçus. Cette étape est un petit peu plus compliquée. Tout d'abord, revenons à l'explication commencée un peu plus tôt. Lorsque nous démarrons la connexion et créons l'objet `boxReception`. Dans l'idéal nous aimerions faire de la façon suivante :

```
// gestionnaire de la réception de caractère
private void DataReceivedHandler(object sender, SerialDataReceivedEventArgs e)
{
    String texte = port.ReadExisting();
    boxReception.Text += texte;
}
```

Code : Réception de données

Cependant, les choses ne sont pas aussi simples cette fois-ci. En effet, pour des raisons de sécurité sur les processus, C# interdit que le texte d'un composant (boxReception) soit modifié de manière asynchrone, quand les données arrivent. Pour contourner cela, nous devons créer une méthode "déléguée" à qui on passera notre texte à afficher et qui se chargera d'afficher le texte quand l'interface sera prête. Pour créer cette déléguée, nous allons commencer par rajouter une méthode dite de *callback* pour gérer la mise à jour du texte. La ligne suivante est donc à ajouter dans la classe, comme membre :

```
// une déléguée pour pouvoir mettre à jour le texte de la boîte de réception
// de manière "thread-safe"
delegate void SetTextCallback(string text);
```

Le code de la réception devient alors le suivant :

```
// gestionnaire de la réception de caractère
private void DataReceivedHandler(object sender, SerialDataReceivedEventArgs e)
{
    String texte = port.ReadExisting();
    // boxReception.Text += texte;
    SetText(texte);
}

private void SetText(string text)
{
    if (boxReception.InvokeRequired)
    {
        SetTextCallback d = new SetTextCallback(SetText);
        boxReception.Invoke(d, new object[] { text });
    }
    else
    {
        boxReception.Text += text;
    }
}
```

Code : Réception de données avec la déléguée

[[information]] | Je suis désolé si mes informations sont confuses. Je ne suis malheureusement pas un maître dans l'art des threads UI de C#. Cependant, un tas de documentation mieux expliquée existe sur internet si vous voulez plus de détails.

Une fois tout cela instancié, vous devriez avoir un terminal voie série tout beau fait par vous-même ! Libre à vous maintenant toutes les cartes en main pour créer des applications qui communiqueront avec votre Arduino et feront des échanges d'informations avec.

4.4.3 En Python

Comme ce langage à l'air d'être en vogue, je me suis un peu penché dessus pour vous fournir une approche de comment utiliser python pour se servir de la voie série. Mon niveau en python

étant équivalent à “grand débutant”, je vais vous proposer un code simple reprenant les fonctions utiles à savoir le tout sans interface graphique. Nul doute que les pythonistes chevronnés sauront creuser plus loin :)

[[information]] | Comme pour les exemples dans les autres langages, on utilisera l’exercice “Attention à la casse” dans l’Arduino pour tester notre programme.

Pour communiquer avec la voie série, nous allons utiliser une librairie externe qui s’appelle `pySerial`.

4.4.3.1 Installation

4.4.3.1.1 Ubuntu Pour installer `pySerial` sur votre machine Ubuntu c’est très simple, il suffit de lancer une seule commande :

```
sudo apt-get install python3-serial
```

Vous pouvez aussi l’installer à partir des sources à l’adresse suivante : <https://pypi.python.org/pypi/pyserial> . Ensuite, décompresser l’archive et exécuter la commande : (pour python 2)

```
python setup.py install
```

(pour python 3)

```
python3 setup.py install
```

4.4.3.1.2 Windows Si vous utilisez Windows, il vous faudra un logiciel capable de décompresser les archives de types `tar.gz` (comme 7-zip par exemple). Ensuite vous devrez récupérer les sources à la même adresse que pour Linux : <https://pypi.python.org/pypi/pyserial> . Enfin, comme pour Linux encore il vous suffira d’exécuter la commande qui va bien :

```
python setup.py install
```

4.4.3.2 Utiliser la librairie

Pour utiliser la librairie, il vous faudra tout d’abord l’importer. Pour cela, on utilise la commande `import` :

```
import serial
```

mais comme seule une partie du module nous intéresse vraiment (`Serial`) on peut restreindre :

```
from serial import Serial
```

(notez l’importance des majuscules/minuscules)

4.4.3.2.1 Ouvrir un port série Maintenant que le module est bien chargé, nous allons pouvoir commencer à l’utiliser. La première chose importante à faire est de connaître le port série à utiliser. On peut obtenir une liste de ces derniers grâce à la commande :

```
python -m serial.tools.list_ports
```

Si comme chez moi cela ne fonctionne pas, vous pouvez utiliser d’autres méthodes.

- Sous Windows : en allant dans le gestionnaire de périphériques pour trouver le port série concerné (COMx)
- Sous Linux : en utilisant la commande `ls /dev`, vous pourrez trouver le nom du port série sous le nom “ttyACMx” par exemple

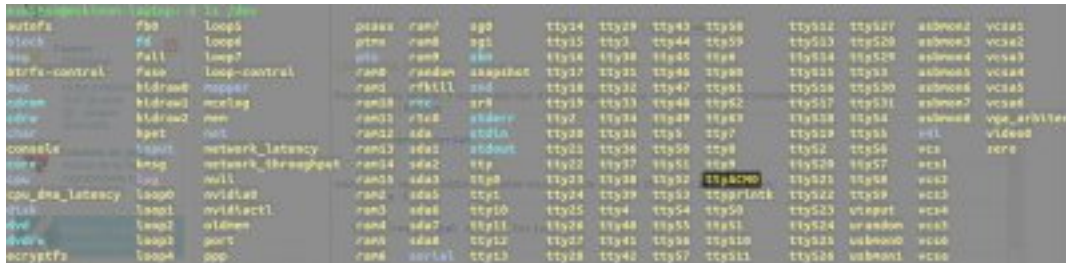


Figure 4.20 – Le port USB de l’Arduino

Lorsque le port USB est identifié, on peut créer un objet de type `Serial`. Le constructeur que l’on va utiliser prend deux paramètres, le nom du port série et la vitesse à utiliser (les autres paramètres (parité...) conviennent par défaut).

```
port = Serial('/dev/ttyACM0', 9600)
```

Une fois cet objet créé, la connexion peut-être ouverte avec la fonction `open()`

```
port.open()
```

Pour vérifier que la voie série est bien ouverte, on utilisera la méthode “`isOpen()`” qui retourne un booléen vrai si la connexion est établie.

4.4.3.2.2 Envoyer des données Maintenant que la voie série est ouverte, nous allons pouvoir lui envoyer des données à traiter. Pour le bien de l’exercice, il nous faut récupérer un (des) caractère(s) à envoyer et retourner avec la casse inversée. Nous allons donc commencer par récupérer une chaîne de caractère de l’utilisateur :

```
chaine = input("Que voulez vous transformer? ")
```

Puis nous allons simplement l’envoyer avec la fonction “`write`”. Cette fonction prend en paramètre un tableau de bytes. Nous allons donc transformer notre chaîne pour convenir à ce format avec la fonction python “`bytes()`” qui prend en paramètres la chaîne de caractères et le format d’encodage.

```
bytes(chaine, 'UTF-8')
```

Ce tableau peut directement être envoyé dans la fonction `write()` :

```
port.write(bytes(chaine, 'UTF-8'))
```

4.4.3.2.3 Recevoir des données La suite logique des choses voudrait que l’on réussisse à recevoir des données. C’est ce que nous allons voir maintenant. ;) Nous allons tout d’abord vérifier que des données sont arrivées sur la voie série via la méthode `inWaiting()`. Cette dernière nous renvoie le nombre de caractères dans le buffer de réception de la voie série. S’il est différent de 0, cela signifie qu’il y a des données à lire. S’il y a des caractères, nous allons utiliser la fonction “`read()`” pour les récupérer. Cette fonction retourne les caractères (byte) un par un dans l’ordre où il sont arrivés. Un exemple de récupération de caractère pourrait-être :

```
while(port.inWaiting() != 0):
    car = port.read() #on lit un caractère
    print(car) #on l'affiche
```

Code : Lecture des données en python

Vous en savez maintenant presque autant que moi sur la voie série en python :P! Je suis conscient que c'est assez maigre comparé aux autres langages, mais je ne vais pas non plus apprendre tout les langages du monde :D

4.4.3.3 Code exemple complet et commenté

```
####!/usr/bin/python3
# -*-coding:Utf-8 -*

from serial import Serial
import time

port = Serial('/dev/ttyACM0', 9600)

port.open()

#### test que le port est ouvert
if (port.isOpen()):
    # demande de la chaîne à envoyer
    chaine = input("Que voulez vous transformer? ")
    # on écrit la chaîne en question
    port.write(bytes(chaine, 'UTF-8'))
    # attend que des données soit revenues
    while(port.inWaiting() == 0):
        # on attend 0.5 seconde pour que les données arrive
        time.sleep(0.5)

    # si on arrive là, des données sont arrivées
    while(port.inWaiting() != 0):
        # il y a des données!
        car = port.read() #on lit un caractère
        print(car) #on l'affiche
else :
    print ("Le port n'a pas pu être ouvert!")
```

Code : Code complet d'utilisation de la liaison série en python

Cette annexe vous aura permis de comprendre un peu comment utiliser la voie série en général avec un ordinateur. Avec vos connaissances vous êtes dorénavant capable de créer des interfaces graphiques pour communiquer avec votre arduino. De grandes possibilités s'offrent à vous, et de plus grandes vous attendent avec les parties qui suivent...

5 Les grandeurs analogiques

Dans cette partie, je vais introduire la notion de grandeurs analogiques, qui sont opposées aux grandeurs logiques. Grâce à ce chapitre, vous serez capable d'utiliser des capteurs pour interagir avec l'environnement autour de votre carte Arduino, ou bien des actionneurs tels que les moteurs.

5.1 Les entrées analogiques de l'Arduino

Ce premier chapitre va vous faire découvrir comment gérer des tensions analogiques avec votre carte Arduino. Vous allez d'abord prendre en main le fonctionnement d'un certain composant essentiel à la mise en forme d'un signal analogique, puis je vous expliquerai comment vous en servir avec votre Arduino. Rassurez-vous, il n'y a pas besoin de matériel supplémentaire pour ce chapitre! ;)

5.1.1 Un signal analogique : petits rappels

Faisons un petit rappel sur ce que sont les signaux analogiques. D'abord, jusqu'à présent nous n'avons fait qu'utiliser des grandeurs numériques binaires. Autrement dit, nous n'avons utilisé que des états logiques HAUT et BAS. Ces deux niveaux correspondent respectivement aux tensions de 5V et 0V. Cependant, une valeur analogique ne se contentera pas d'être exprimée par 0 ou 1. Elle peut prendre *une infinité de valeurs* dans un intervalle donné. Dans notre cas, par exemple, la grandeur analogique pourra varier aisément de 0 à 5V en passant par 1.45V, 2V, 4.99V, etc. Voici un exemple de signal analogique, le très connu signal sinusoïdal :

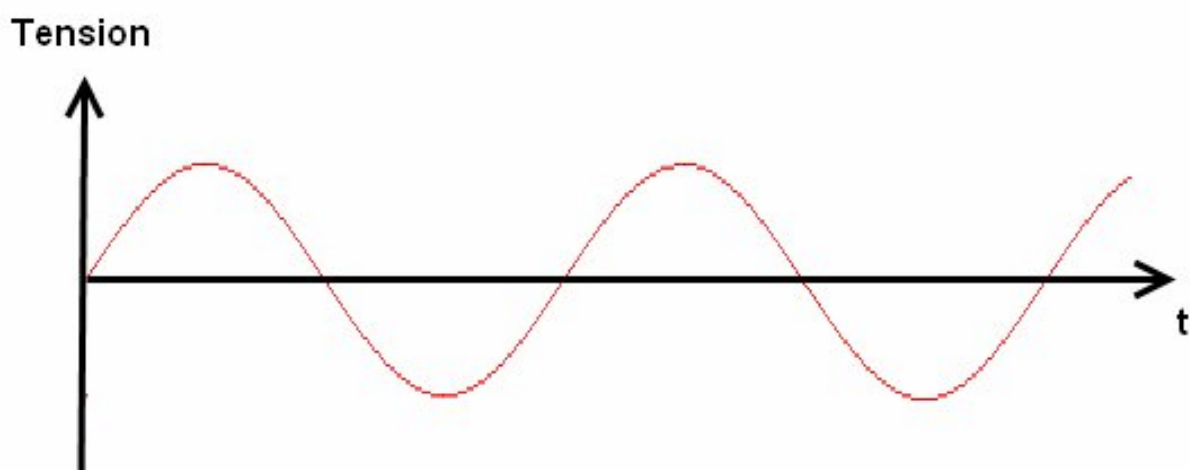


Figure 5.1 – Un signal analogique périodique

[[information]] | On retiendra que l'on ne s'occupe que de la tension et non du courant, en fonction du temps.

Si on essaye de mettre ce signal (ce que je vous déconseille de faire) sur une entrée numérique de l'Arduino et qu'on lit les valeurs avec `digitalRead()`, on ne lira que 0 ou 1. Les broches numériques de l'Arduino étant incapable de lire les valeurs d'un signal analogique.

5.1.1.0.1 Signal périodique Dans la catégorie des signaux analogiques et même numériques (dans le cas d'horloge de signal pour le cadencement des micro-contrôleurs par exemple) on a les **signaux dits périodiques**. La période d'un signal est en fait un motif de ce signal qui se répète et qui donne ainsi la forme du signal. Prenons l'exemple d'un signal binaire qui prend un niveau logique 1 puis un 0, puis un 1, puis un 0, ...

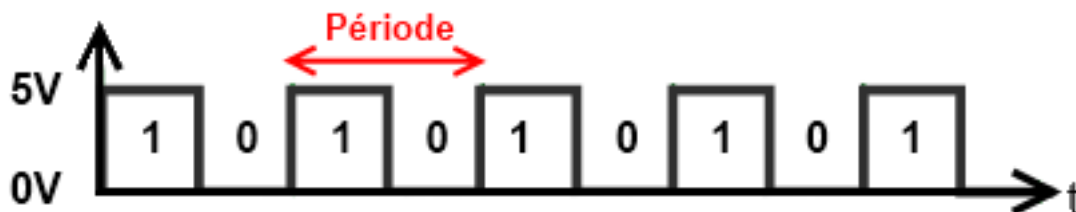


Figure 5.2 – Représentation d'une période

La période de ce signal est le motif qui se répète tant que le signal existe. Ici, c'est le niveau logique 1 et 0 qui forme le motif. Mais cela aurait pu être 1 1 et 0, ou bien 0 1 1, voir 0 0 0 1 1, les possibilités sont infinies ! Pour un signal analogique, il en va de même. Reprenons le signal de tout à l'heure :

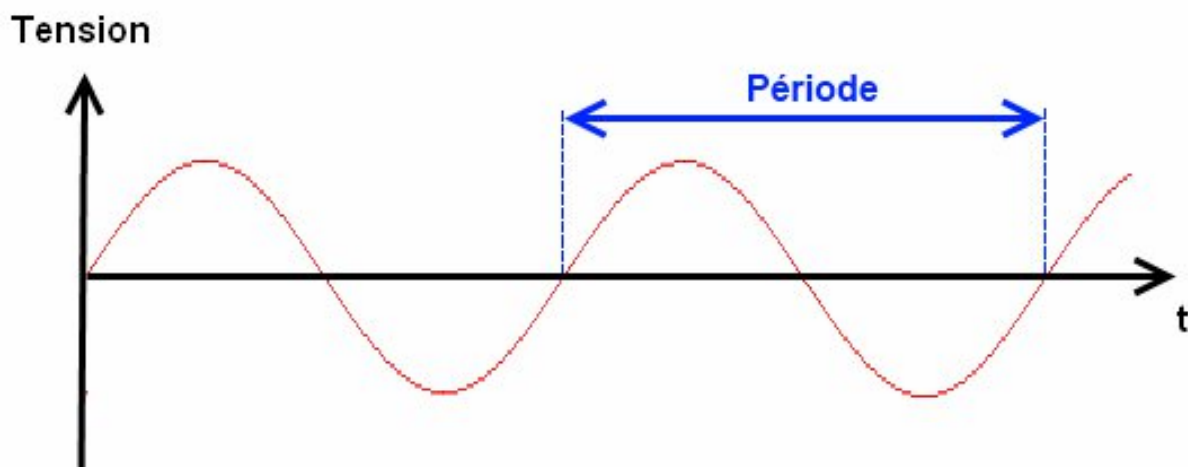


Figure 5.3 – Période d'un signal analogique

Ici le motif qui se répète est "la bosse de chameau" ou plus scientifiquement parlant : une période d'un signal sinusoïdal. ^^ Pour terminer, la période désigne aussi le temps que met un motif à se répéter. Si j'ai une période de 1ms, cela veut dire que le motif met 1ms pour se dessiner complètement avant de commencer le suivant. Et en sachant le nombre de fois que se répète le motif en 1 seconde, on peut calculer la fréquence du signal selon la formule suivante : $F = \frac{1}{T}$; avec F la fréquence, en hertz, du signal et T la période, en seconde, du signal.

5.1.1.0.2 Notre objectif L'objectif va donc être double. Tout d'abord, nous allons devoir être capables de convertir cette valeur analogique en une valeur numérique, que l'on pourra ensuite

manipuler à l'intérieur de notre programme. Par exemple, lorsque l'on mesurera une tension de 2,5V nous aurons dans notre programme une variable nommée "tension" qui prendra la valeur 250 lorsque l'on fera la conversion (ce n'est qu'un exemple). Ensuite, nous verrons avec Arduino ce que l'on peut faire avec les signaux analogiques. Je ne vous en dis pas plus...

5.1.2 Les convertisseurs analogiques → numérique ou CAN

[[question]] | Qu'est-ce que c'est ?

C'est un dispositif qui va convertir des grandeurs analogiques en grandeurs numériques. La valeur numérique obtenue sera proportionnelle à la valeur analogique fournie en entrée, bien évidemment. Il existe différentes façons de convertir une grandeur analogique, plus ou moins faciles à mettre en œuvre, plus ou moins précises et plus ou moins onéreuses. Pour simplifier, je ne parlerai que des tensions analogiques dans ce chapitre.

5.1.2.0.1 La diversité Je vais vous citer quelques types de convertisseurs, sachez cependant que nous n'en étudierons qu'un seul type.

- **Convertisseur à simple rampe** : ce convertisseur "fabrique" une tension qui varie proportionnellement en un court laps de temps entre deux valeurs extrêmes. En même temps qu'il produit cette tension, il compte. Lorsque la tension d'entrée du convertisseur devient égale à la tension générée par ce dernier, alors le convertisseur arrête de compter. Et pour terminer, avec la valeur du compteur, il détermine la valeur de la grandeur d'entrée. Malgré sa bonne précision, sa conversion reste assez lente et dépend de la grandeur à mesurer. Il est, de ce fait, peu utilisé.
- **Convertisseur flash** : ce type de convertisseur génère lui aussi des tensions analogiques. Pour être précis, il en génère plusieurs, chacune ayant une valeur plus grande que la précédente (par exemple 2V, 2.1V, 2.2V, 2.3V, etc.) et compare la grandeur d'entrée à chacun de ces paliers de tension. Ainsi, il sait entre quelle et quelle valeur se trouve la tension mesurée. Ce n'est pas très précis comme mesure, mais il a l'avantage d'être rapide et malheureusement cher.
- **Convertisseur à approximations successives** : Pour terminer, c'est ce convertisseur que nous allons étudier...

5.1.2.1 Arduino dispose d'un CAN

Vous vous doutez bien que si je vous parle des CAN, c'est qu'il y a une raison. Votre carte Arduino dispose d'un tel dispositif intégré dans son cœur : le micro-contrôleur. Ce convertisseur est un convertisseur "à approximations successives". Je vais détailler un peu plus le fonctionnement de ce convertisseur par rapport aux autres dont je n'ai fait qu'un bref aperçu de leur fonctionnement (bien que suffisant).

[[information]] | Ceci rentre dans le cadre de votre culture générale électronique, ce n'est pas nécessaire de lire comment fonctionne ce type de convertisseur. Mais je vous recommande vivement de le faire, car il est toujours plus agréable de comprendre comment fonctionnent les outils qu'on utilise ! :)

5.1.2.1.1 Principe de dichotomie La dichotomie, ça vous parle ? Peut-être que le nom ne vous dit rien, mais il est sûr que vous en connaissez le fonctionnement. Peut-être alors connaissez-vous le jeu “plus ou moins” en programmation ? Si oui alors vous allez pouvoir comprendre ce que je vais expliquer, sinon lisez le principe sur le lien que je viens de vous donner, cela vous aidera un peu. La dichotomie est donc une méthode de recherche conditionnelle qui s’applique lorsque l’on recherche une valeur comprise entre un minimum et un maximum. L’exemple du jeu “plus ou moins” est parfait pour vous expliquer le fonctionnement. Prenons deux joueurs. Le joueur 1 choisit un nombre compris entre deux valeurs extrêmes, par exemple 0 et 100. Le joueur 2 ne connaît pas ce nombre et doit le trouver. La méthode la plus rapide pour que le joueur 2 puisse trouver quel est le nombre choisi par le joueur 1 est :

Joueur 1 dit : "quel est le nombre mystère?"
> 40

Joueur 1 dit : "Ce nombre est plus grand"
> 80

Joueur 1 dit : "Ce nombre est plus petit"
> 60

Joueur 1 dit : "Ce nombre est plus grand"
> 70

Joueur 1 dit : "Ce nombre est plus grand"
> 75

Joueur 1 dit : "Ce nombre est plus petit"
> 72

Bravo, Joueur 2 a trouvé le nombre mystère !

Je le disais, le joueur 2, pour arriver le plus rapidement au résultat, doit choisir une méthode rapide. Cette méthode, vous l’aurez deviné, consiste à couper en deux l’espace de recherche. Au début, cet espace allait de 0 à 100, puis au deuxième essai de 40 à 100, au troisième essai de 40 à 80, etc.

[[attention]] | Cet exemple n’est qu’à titre indicatif pour bien comprendre le concept.

En conclusion, cette méthode est vraiment simple, efficace et rapide ! Peut-être l’aurez-vous observé, on est pas obligé de couper l’espace de recherche en deux parties égales.

5.1.2.2 Le CAN à approximations successives

On y vient, je vais pouvoir vous expliquer comment il fonctionne. Voyez-vous le rapport avec le jeu précédent ? Pas encore ? Alors je m’explique. Prenons du concret avec une valeur de tension de 3.36V que l’on met à l’entrée d’un CAN à approximations successives (j’abrègerai par CAN dorénavant).

[[information]] | Notez le symbole du CAN qui se trouve juste au-dessus de l’image. Il s’agit d’un “U” renversé et du caractère #.

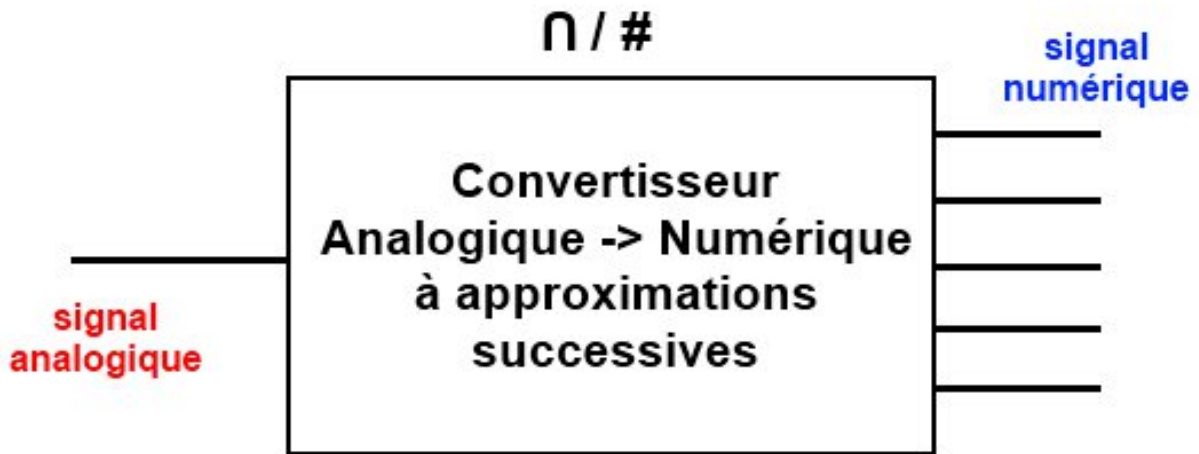


Figure 5.4 - Le CAN à approximations successives

Cette tension analogique de 3.36V va rentrer dans le CAN et va ressortir sous forme numérique (avec des 0 et 1). Mais que se passe-t-il à l'intérieur pour arriver à un tel résultat ? Pour que vous puissiez comprendre correctement comment fonctionne ce type de CAN, je vais être obligé de vous apprendre plusieurs choses avant.

5.1.2.2.1 Le comparateur Commençons par le **comparateur**. Comme son nom le laisse deviner, c'est quelque chose qui compare. Ce quelque chose est un composant électronique. Je ne rentrerai absolument pas dans le détail, je vais simplement vous montrer comment il fonctionne.

[[question]] | Comparer, oui, mais quoi ?

Des tensions ! :D Regardez son symbole, je vous explique ensuite...

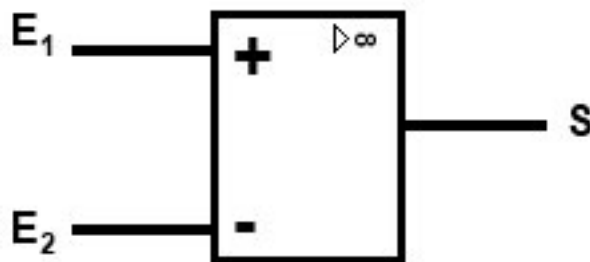


Figure 5.5 - Le comparateur

Vous observez qu'il dispose de deux entrées E_1 et E_2 et d'une sortie S . Le principe est simple :

- Lorsque la tension $E_1 > E_2$ alors $S = +V_{cc}$ ($+V_{cc}$ étant la tension d'alimentation positive du comparateur)
- Lorsque la tension $E_1 < E_2$ alors $S = -V_{cc}$ ($-V_{cc}$ étant la tension d'alimentation négative, ou la masse, du comparateur)
- $E_1 = E_2$ est une condition quasiment impossible, si tel est le cas (si on relie E_1 et E_2) le comparateur donnera un résultat faux

Parlons un peu de la tension d'alimentation du comparateur. Le meilleur des cas est de l'alimenter entre 0V et +5V. Comme cela, sa sortie sera soit égale à 0V, soit égale à +5V. Ainsi, on rentre dans le domaine des tensions acceptées par les micro-contrôleurs et de plus il verra soit un état logique BAS, soit un état logique HAUT. On peut réécrire les conditions précédemment énoncées comme ceci :

- $E_1 > E_2$ alors $S = 1$
- $E_1 < E_2$ alors $S = 0$
- $E_1 = E_2$, alors $S = \textit{indefini}$

Simple n'est-ce pas ?

5.1.2.2.2 Le démultiplexeur Maintenant, je vais vous parler du **démultiplexeur**. C'est en fait un nom un peu barbare pour désigner un composant électronique qui fait de l'aiguillage de niveaux logiques (il en existe aussi qui font de l'aiguillage de tensions analogiques). Le principe est là encore très simple. Le démultiplexeur à plusieurs sorties, une entrée et des entrées de sélection :

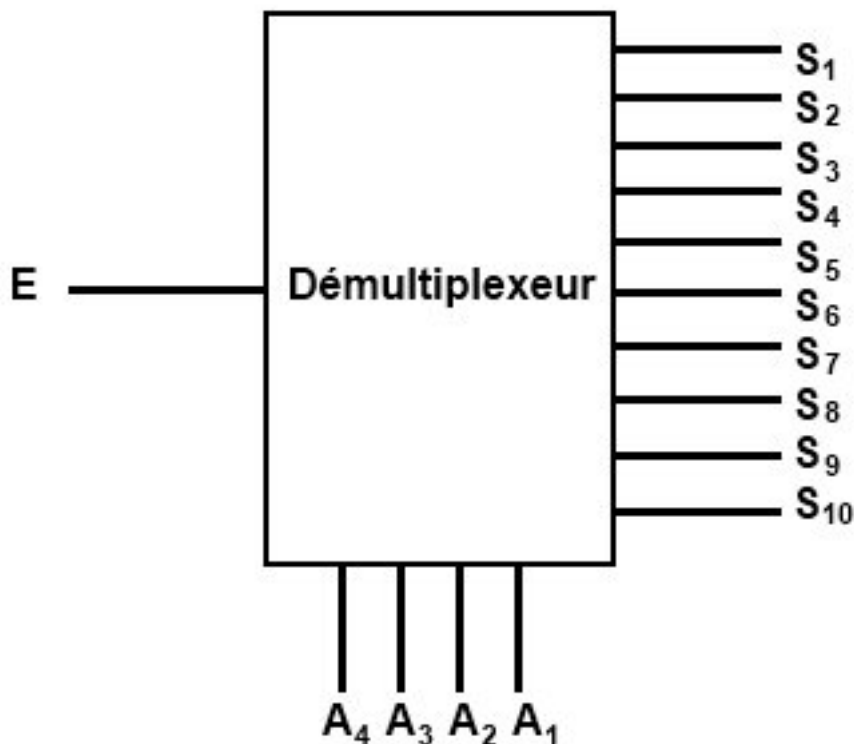


Figure 5.6 – Le démultiplexeur

- E est l'entrée où l'on impose un niveau logique 0 ou 1.
- Les sorties S sont là où se retrouve le niveau logique d'entrée. UNE seule sortie peut être active à la fois et recopier le niveau logique d'entrée.
- Les entrées A permettent de sélectionner quelle sera la sortie qui est active. La sélection se fait grâce aux combinaisons binaires. Par exemple, si je veux sélectionner la sortie 4, je vais écrire le code 0100 (qui correspond au chiffre décimal 4) sur les entrées A_1 à A_4

[[attention]] | Je rappelle que, pour les entrées de sélection, le bit de poids fort est A_4 et le bit de poids faible A_1 . Idem pour les sorties, S_1 est le bit de poids faible et S_{10} , le bit de poids fort.

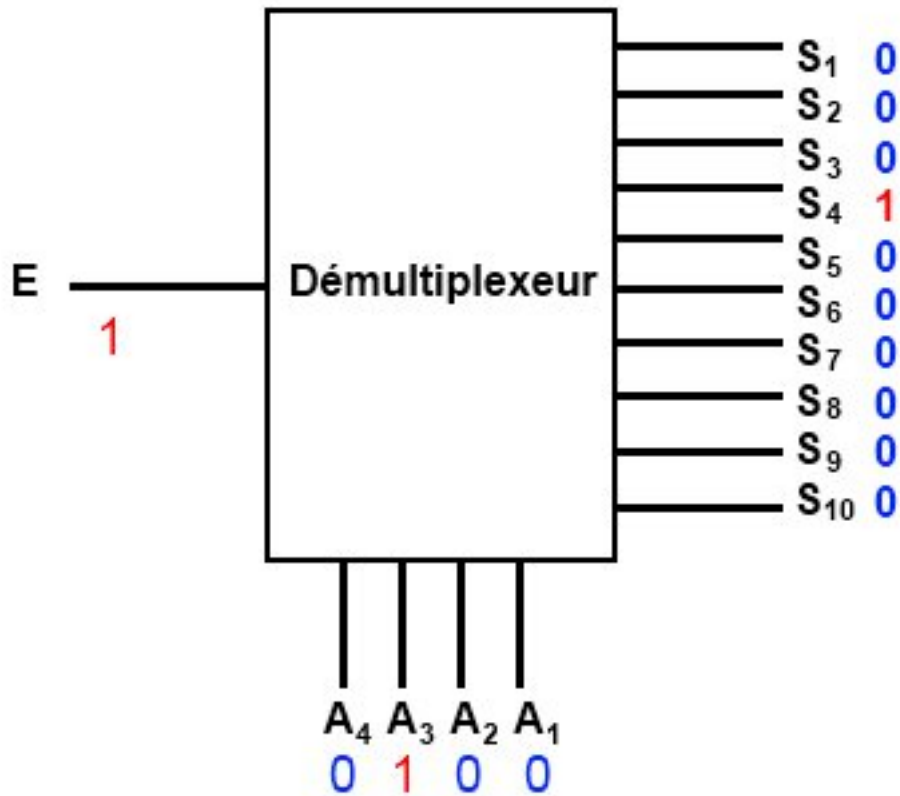


Figure 5.7 – Le démultiplexeur en action

5.1.2.2.3 La mémoire Ce composant électronique sert simplement à stocker des données sous forme binaire.

5.1.2.2.4 Le convertisseur numérique analogique Pour ce dernier composant avant l'acte final, il n'y a rien à savoir si ce n'est que c'est l'opposé du CAN. Il a donc plusieurs entrées et une seule sortie. Les entrées reçoivent des valeurs binaires et la sortie donne le résultat sous forme de tension.

5.1.2.2.5 Fonctionnement global Revenons dans les explications du fonctionnement d'un CAN à approximations successives. Je vous ai fait un petit schéma rassemblant les éléments précédemment présentés :

Voilà donc comment se compose le CAN. Si vous avez compris le fonctionnement de chacun des composants qui le constituent, alors vous n'aurez pas trop de mal à suivre mes explications. Dans le cas contraire, je vous recommande de relire ce qui précède et de bien comprendre et rechercher sur internet de plus amples informations si cela vous est nécessaire.

En premier lieu, commençons par les conditions initiales :

- V_e est la tension analogique d'entrée, celle que l'on veut mesurer en la convertissant en signal numérique.
- **La mémoire** contient pour l'instant que des 0 sauf pour le bit de poids fort (S_{10}) qui est à 1. Ainsi, le convertisseur numérique → analogique va convertir ce nombre binaire en une tension analogique qui aura pour valeur 2.5V.

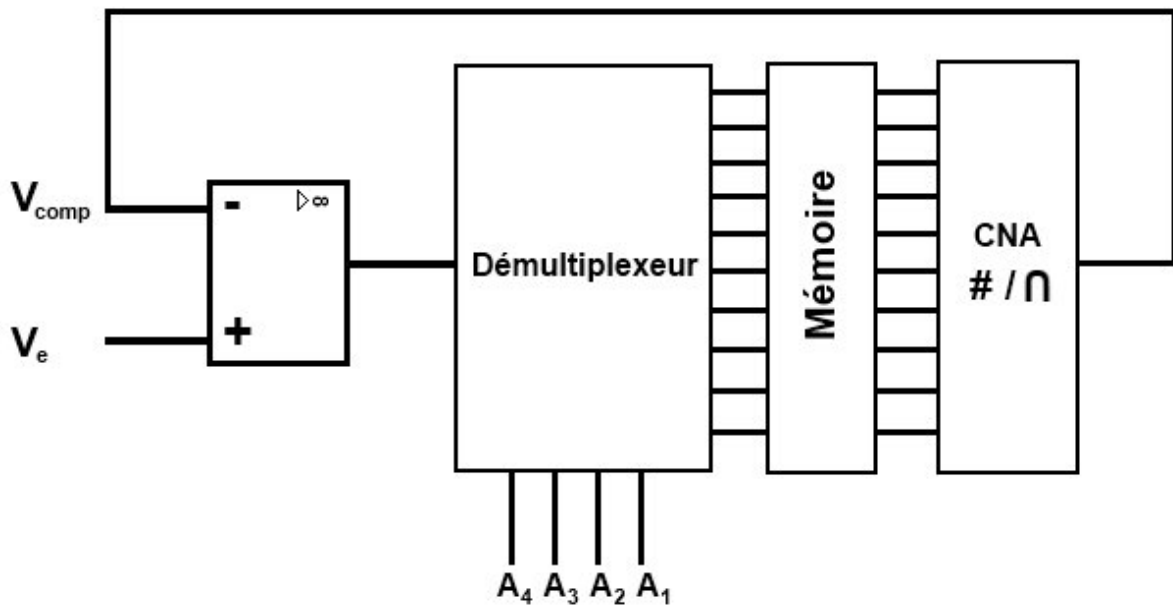


Figure 5.8 – Chaîne de fonctionnement du CAN

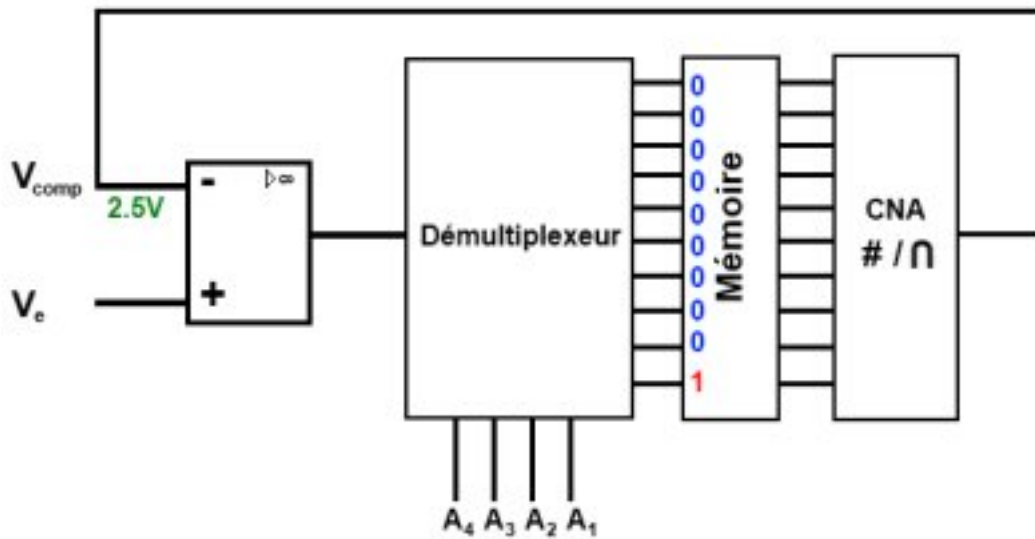


Figure 5.9 – Chaîne de fonctionnement du CAN en action

— Pour l'instant, le démultiplexeur n'entre pas en jeu.

Suivons le fonctionnement étape par étape :

Étape 1 :

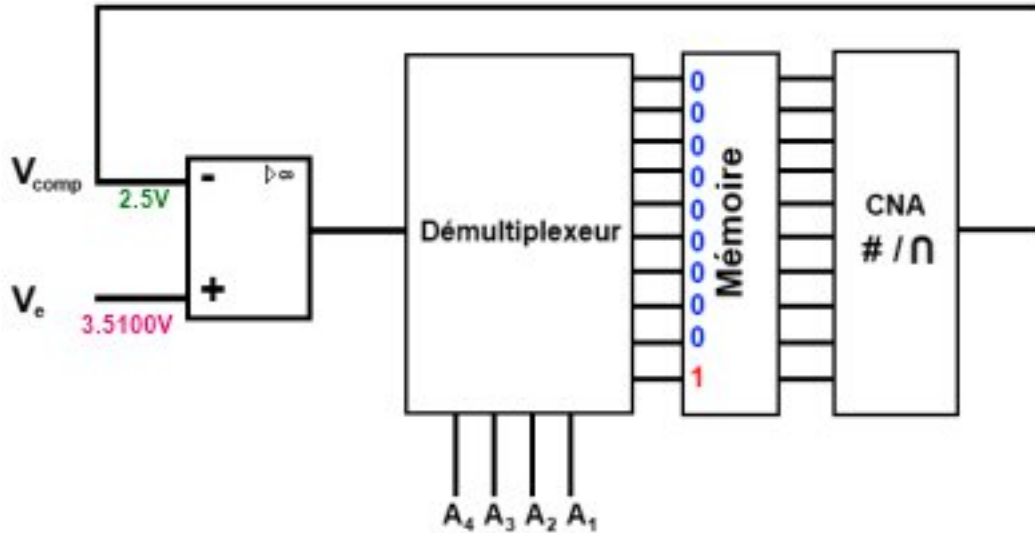


Figure 5.10 – Chaîne de fonctionnement du CAN - Exemple - Etape 1

- J'applique une tension $V_e = 3.5100V$ précisément.
- Le comparateur compare la tension $V_{comp} = 2.5V$ à la tension $V_e = 3.5100V$. Étant donné que $V_e > V_{comp}$, on a un Niveau Logique **1** en sortie du comparateur.
- Le multiplexeur entre alors en jeu. Avec ses signaux de sélections, il va sélectionner la sortie ayant le poids le plus élevé, soit S_{10} .
- La mémoire va alors enregistrer le niveau logique présent sur la broche S_{10} , dans notre cas c'est **1**.

Étape 2 :

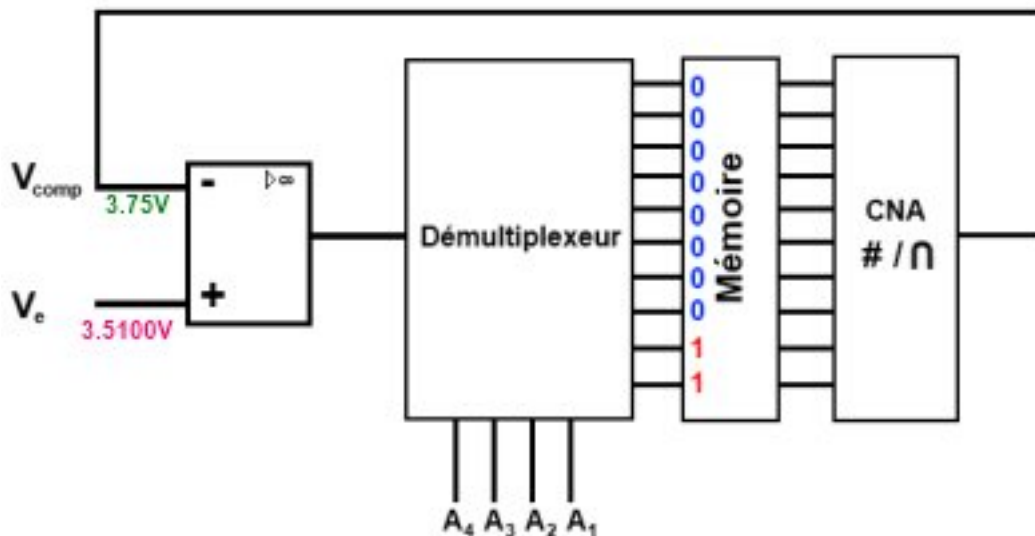


Figure 5.11 – Chaîne de fonctionnement du CAN - Exemple - Etape 2

- Au niveau de la mémoire, on change le deuxième bit de poids fort (mais moins fort que le premier) correspondant à la broche S_9 en le passant à **1**.
- En sortie du CNA, on aura alors une tension de $3.75V$
- Le comparateur compare, il voit $V_{comp} > V_e$ donc il donne un état logique **0**.
- La mémoire enregistre alors le niveau sur la broche S_9 qui est à **0**.

Étape 3 :

Redondante aux précédentes

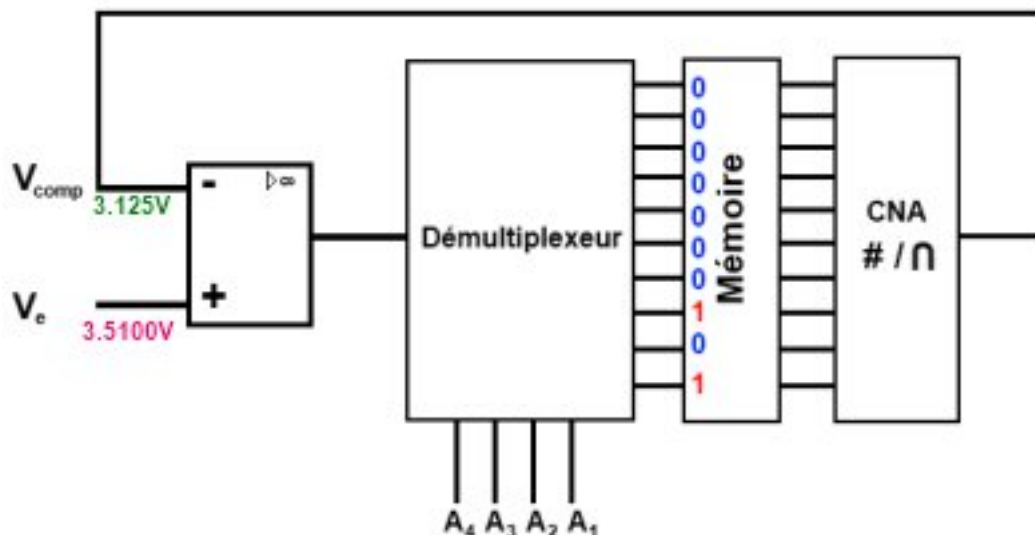


Figure 5.12 – Chaîne de fonctionnement du CAN - Exemple - Etape 3

- On passe le troisième bit le plus fort (broche S_8) à **1**.
- Le CNA converti le nombre binaire résultant en une tension de $3.125V$.
- Le comparateur voit $V_e > V_{comp}$, sa sortie passe à **1**.
- La mémoire enregistre l'état logique de la broche S_8 qui est à **1**.

Le CAN continue de cette manière pour arriver au dernier bit (celui de poids faible). En mémoire, à la fin de la conversion, se trouve le résultat. On va alors lire cette valeur binaire que l'on convertira ensuite pour l'exploiter. Bon, j'ai continué les calculs à la main (n'ayant pas de simulateur pour le faire à ma place), voici le tableau des valeurs :

->

Poids du bit	NL en sortie du comparateur	Bits stockés en mémoire	Tension en sortie du convertisseur CN
10	1	1	2.5
9	0	0	3.75
8	1	1	3.125
7	1	1	3.4375
6	0	0	3.59375
5	0	0	3.515625
4	1	1	3.4765625
3	1	1	3.49609375

Poids du bit	NL en sortie du comparateur	Bits stockés en mémoire	Tension en sortie du convertisseur CN
2	1	1	3.505859375
1	0	0	3.5107421875

Table 5.1 – Lien entre tension et bits

<-

Résultat : Le résultat de la conversion donne :

->

Résultat de conversion (binaire)	Résultat de conversion (décimale)	Résultat de conversion (Volts)
1011001110	718	3,505859375

Table 5.2 – Exemple de résultat d'une conversion

<- Observez la précision du convertisseur. Vous voyez que la conversion donne un résultat (très) proche de la tension réelle, mais elle n'est pas exactement égale. Ceci est dû au pas du convertisseur.

5.1.2.2.6 Pas de calcul du CAN Qu'est-ce que le **pas de calcul**? Eh bien il s'agit de la tension minimale que le convertisseur puisse "voir". Si je mets le bit de poids le plus faible à 1, quelle sera la valeur de la tension V_{comp} ? Le convertisseur a une tension de référence de 5V. Son nombre de bit est de 10. Donc il peut "lire" : 2^{10} valeurs pour une seule tension. Ainsi, sa précision sera de : $\frac{5}{2^{10}} = 0,0048828125V$ La formule à retenir sera donc :

$$\frac{V_{ref}}{2^N}$$

Avec :

- V_{ref} : tension de référence du convertisseur
- N : nombre de bit du convertisseur

Il faut donc retenir que, pour ce convertisseur, sa précision est de $4.883mV$. Donc, si on lui met une tension de $2mV$ par exemple sur son entrée, le convertisseur sera incapable de la voir et donnera un résultat égal à 0V.

5.1.2.2.7 Les inconvénients Pour terminer avant de passer à l'utilisation du CNA avec Arduino, je vais vous parler de ses inconvénients. Il en existe trois principaux :

- **la plage de tension d'entrée** : le convertisseur analogique de l'Arduino ne peut recevoir à son entrée que des tensions comprises entre 0V et +5V. On verra plus loin comment améliorer la précision du CAN.
- **la précision** : la précision du convertisseur est très bonne sauf pour les deux derniers bits de poids faible. On dit alors que la précision est de $\pm 2LSB$ (à cause du pas de calcul que je viens de vous expliquer).

- **la vitesse de conversion** : le convertisseur N/A de la carte Arduino n'a pas une très grande vitesse de conversion par rapport à un signal audio par exemple. Ainsi, si l'on convertit un signal audio analogique en numérique grâce à la carte Arduino, on ne pourra entendre que les fréquences en dessous de 10kHz. Dans bien des cas cela peut être suffisant, mais d'en d'autre il faudra utiliser un convertisseur A/N externe (un composant en plus) qui sera plus rapide afin d'obtenir le spectre audio complet d'un signal sonore.

*[CAN] : Convertisseur Analogique Numérique

5.1.3 Lecture analogique, on y vient...

5.1.3.1 Lire la tension sur une broche analogique

Un truc très sympa avec Arduino, c'est que c'est facile à prendre en main. Et ça se voit une fois de plus avec l'utilisation des convertisseurs numérique -> analogique ! En effet, vous n'avez qu'une seule nouvelle fonction à retenir : `analogRead()` !

5.1.3.1.1 `analogRead(pin)` Cette fonction va nous permettre de lire la valeur lue sur une entrée analogique de l'Arduino. Elle prend un argument et retourne la valeur lue :

- L'argument est le numéro de l'entrée analogique à lire (explication ci-dessous)
- La valeur retournée (un `int`) sera le résultat de la conversion analogique->numérique

Sur une carte Arduino Uno, on retrouve 6 CAN. Ils se trouvent tous du même côté de la carte, là où est écrit "Analog IN" :

Ces 6 entrées analogiques sont numérotées, tout comme les entrées/sorties logiques. Par exemple, pour aller lire la valeur en sortie d'un capteur branché sur le convertisseur de la broche analogique numéro 3, on fera : `valeur = analogRead(3);`.

[[e]] | Ne confondez pas les entrées analogiques et les entrées numériques ! Elles ont en effet le même numéro pour certaines, mais selon comment on les utilise, la carte Arduino saura si la broche est analogique ou non.

Mais comme nous sommes des programmeurs intelligents et organisés, on nommera les variables proprement pour bien travailler de la manière suivante :

```
// broche analogique 3 OU broche numérique 3
const int monCapteur = 3;

// la valeur lue sera comprise entre 0 et 1023
int valeurLue = 0;

// fonction setup()
{

}

void loop()
{
    // on mesure la tension du capteur sur la broche analogique 3
```

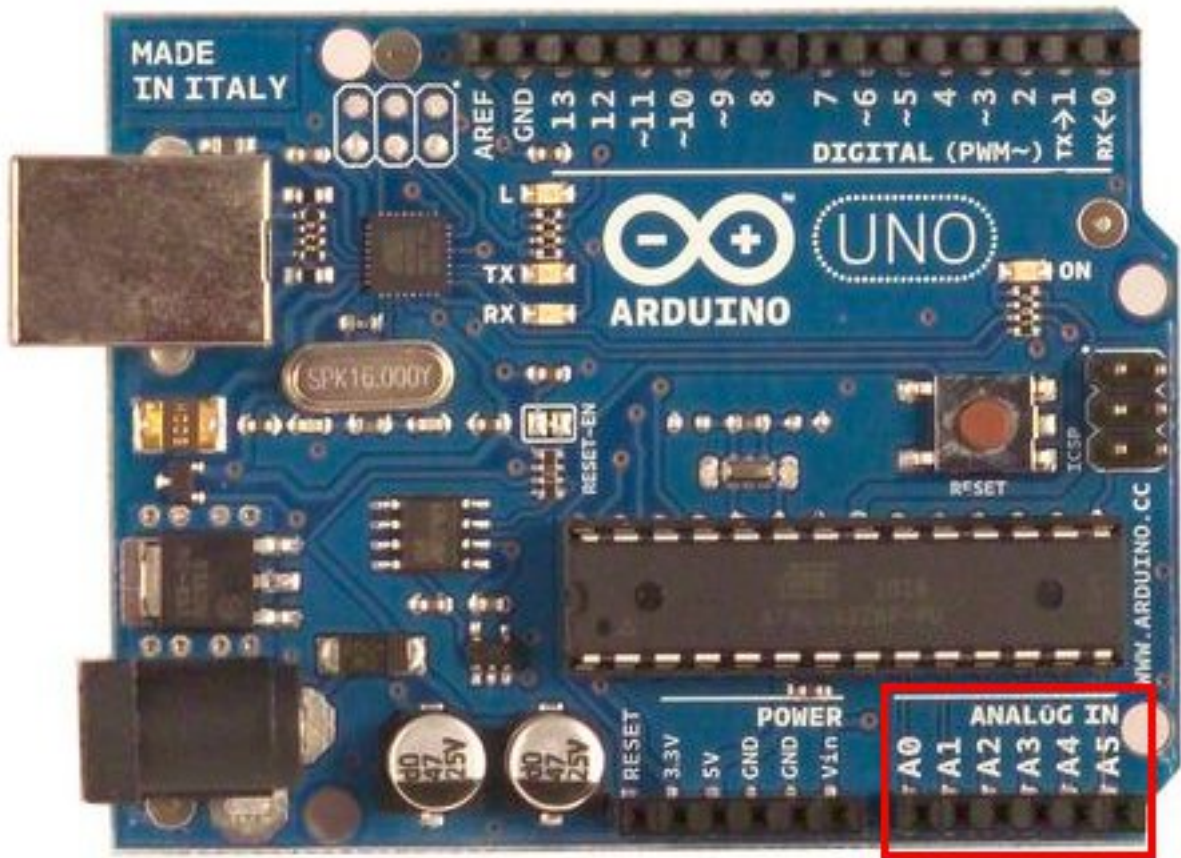


Figure 5.13 – Positions des entrées analogiques

```
    valeurLue = analogRead(monCapteur);  
  
    // du code et encore du code ...  
}
```

Code : Mesure simple d'une entrée analogique

5.1.3.2 Convertir la valeur lue

Bon c'est bien, on a une valeur retournée par la fonction comprise entre 0 et 1023, mais ça ne nous donne pas vraiment une tension ça ! Il va être temps de faire un peu de code (et de math) pour **convertir** cette valeur... Et si vous réfléchissiez un tout petit peu pour trouver la solution sans moi? ^^ ... Trouvée ?

5.1.3.2.1 Conversion Comme je suis super sympa je vais vous donner la réponse, avec en prime : une explication ! Récapitulons. Nous avons une valeur entre 0 et 1023. Cette valeur est l'image de la tension mesurée, elle-même comprise entre 0V et +5V. Nous avons ensuite déterminé que le pas du convertisseur était de 4.88mV par unité. Donc, deux méthodes sont disponibles :

- avec un simple produit en croix
- en utilisant le pas calculé plus tôt

Exemple : La mesure nous retourne une valeur de 458.

- Avec un produit en croix on obtient : $\frac{458 \times 5}{1023} = 2.235V$;
- En utilisant le pas calculé plus haut on obtient : $458 \times 4.88 = 2.235V$.

[[a]] | Les deux méthodes sont valides, et donnent les mêmes résultats. La première à l'avantage de faire ressortir l'aspect "physique" des choses en utilisant les tensions et la résolution du convertisseur.

Voici une façon de le traduire en code :

```
// variable stockant la valeur lue sur le CAN  
int valeurLue;  
// résultat stockant la conversion de valeurLue en Volts  
float tension;  
  
void loop()  
{  
    valeurLue = analogRead(uneBrocheAvecUnCapteur);  
    // produit en croix, ATTENTION, donne un résultat en mV!  
    tension = valeurLue * 4.88;  
    // formule à aspect "physique", donne un résultat en V!  
    tension = valeurLue * (5 / 1023);  
}
```

Code : Conversion d'une mesure en tension

[[q]] | Mais il n'existe pas une méthode plus "automatique" que faire ce produit en croix ?

Eh bien SI ! En effet, l'équipe Arduino a prévu que vous aimeriez faire des conversions facilement et donc une fonction est présente dans l'environnement Arduino afin de vous faciliter la tâche ! Cette fonction se nomme `map()`. À partir d'une valeur d'entrée, d'un intervalle d'entrée et d'un intervalle de sortie, la fonction vous retourne la valeur équivalente comprise entre le deuxième intervalle. Voici son prototype de manière plus explicite :

```
sortie = map(valeur_d_entree,
            valeur_extreme_basse_d_entree,
            valeur_extreme_haute_d_entree,
            valeur_extreme_basse_de_sortie,
            valeur_extreme_haute_de_sortie
);
// cette fonction retourne la valeur calculée équivalente
// entre les deux intervalles de sortie
```

Code : La fonction `map`

Prenons notre exemple précédent. La valeur lue se nomme "valeurLue". L'intervalle d'entrée est la gamme de la conversion allant de 0 à 1023. La gamme (ou intervalle) de "sortie" sera la tension réelle à l'entrée du micro-contrôleur, donc entre 0 et 5V. En utilisant cette fonction nous écrirons donc :

```
// conversion de la valeur lue en tension en mV
tension = map(valeurLue, 0, 1023, 0, 5000);
```

Code : Utilisation de la fonction `map`

[[q]] | Pourquoi tu utilises 5000mV au lieu de mettre simplement 5V ?

Pour la simple et bonne raison que la fonction `map` utilise des entiers. Si j'utilisais 5V au lieu de 5000mV j'aurais donc seulement 6 valeurs possibles pour ma tension (0, 1, 2, 3, 4 et 5V). Pour terminer le calcul, il sera donc judicieux de rajouter une dernière ligne :

```
// conversion de la valeur lue en tension en mV
tension = map(valeurLue, 0, 1023, 0, 5000);
// conversion des mV en V
tension = tension / 1000;
```

Au retour de la liaison série (seulement si on envoie les valeurs par la liaison série) on aurait donc (valeurs à titre d'exemple) :

```
valeurLue = 458
tension = 2.290V
```

[[a]] | On est moins précis que la tension calculée plus haut, mais on peut jouer en précision en modifiant les valeurs de sortie de la fonction `map()`. Ou bien garder le calcul théorique et le placer dans une "fonction maison".

5.1.4 Exemple d'utilisation

5.1.4.1 Le potentiomètre

[[q]] | Qu'est-ce que c'est que cette bête-là encore ?

Le **potentiomètre** (ou “potar” pour les (très) intimes) est un composant très fréquemment employé en électronique. On le retrouve aussi sous le nom de résistance variable. Comme ce dernier nom l’indique si bien, un potentiomètre nous permet *entre autres* de réaliser une résistance variable. En effet, on retrouve deux applications principales que je vais vous présenter juste après. Avant toute chose, voici le symbole du potentiomètre :

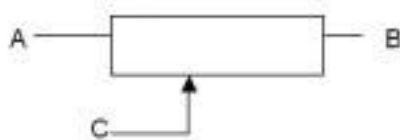


Figure 5.14 – Symbole du potentiomètre

5.1.4.1.1 Cas n°1 : le pont diviseur de tension On y remarque une première chose importante, le potentiomètre a trois broches. Deux servent à borner les tensions maximum (A) et minimum (B) que l’on peut obtenir à ses bornes, et la troisième (C) est reliée à un curseur mobile qui donne la tension variable obtenue entre les bornes précédemment fixées. Ainsi, on peut représenter notre premier cas d’utilisation comme un “diviseur de tension réglable”. En effet, lorsque vous déplacez le curseur, en interne cela équivaut à modifier le point milieu. En termes électroniques, vous pouvez imaginer avoir deux résistances en série (R1 et R2 pour être original). Lorsque vous déplacez votre curseur vers la borne basse, R1 augmente alors que R2 diminue et lorsque vous déplacez votre curseur vers la borne haute, R2 augmente alors que R1 diminue. Voici un tableau montrant quelques cas de figure de manière schématique :

->

Schéma équivalent	Position du curseur	Tension sur
	Curseur à la moitié	$V_{signal} = (1$
	Curseur à 25% du départ	$V_{signal} = (1$
	Curseur à 75% du départ	$V_{signal} = (1$

Table 5.3 – Illustrations de la position du curseur et sa tension résultante

<-

[[i]] | Si vous souhaitez avoir plus d'informations sur les résistances et leurs associations ainsi que sur les potentiomètres, je vous conseille d'aller jeter un œil sur [ce chapitre](#). ;)

5.1.4.1.2 Cas n°2 : la résistance variable Le deuxième cas d'utilisation du potentiomètre est la **résistance variable**. Cette configuration est très simple, il suffit d'utiliser le potentiomètre comme une simple résistance dont les bornes sont A et C ou B et C. On pourra alors faire varier la valeur ohmique de la résistance grâce à l'axe du potentiomètre.

[[a]] | Attention, il existe des potentiomètres **linéaires** (la valeur de la tension évolue de manière proportionnelle au déplacement du curseur), mais aussi des potentiomètres **logarithmique/anti-logarithmique** (la valeur de la tension évolue de manière logarithmique ou anti-logarithmique par rapport à la position du curseur). Choisissez-en dont un qui est linéaire si vous souhaitez avoir une proportionnalité.

5.1.4.2 Utilisation avec Arduino

Vous allez voir que l'utilisation avec Arduino n'est pas vraiment compliquée. Il va nous suffire de raccorder les alimentations sur les bornes extrêmes du potentiomètre, puis de relier la broche du milieu sur une entrée analogique de la carte Arduino :

Une fois le raccordement fait, nous allons faire un petit programme pour tester cela. Ce programme va simplement effectuer une mesure de la tension obtenue sur le potentiomètre, puis envoyer la valeur lue sur la liaison série (ça nous fera réviser ^^). Dans l'ordre, voici les choses à faire :

- Déclarer la broche analogique utilisée (pour faire du code propre
- Mesurer la valeur
- L'afficher!

Je vous laisse chercher? Aller, au boulot! :diable : ... Voici la correction, c'est le programme que j'ai fait, peut-être que le vôtre sera mieux :

```
// le potentiomètre, branché sur la broche analogique 0
const int potar = 0;
// variable pour stocker la valeur lue après conversion
int valeurLue;
// on convertit cette valeur en une tension
float tension;

void setup()
{
    // on se contente de démarrer la liaison série
    Serial.begin(9600);
}

void loop()
{
    // on convertit en nombre binaire la tension lue en sortie du potentiomètre
    valeurLue = analogRead(potar);
```

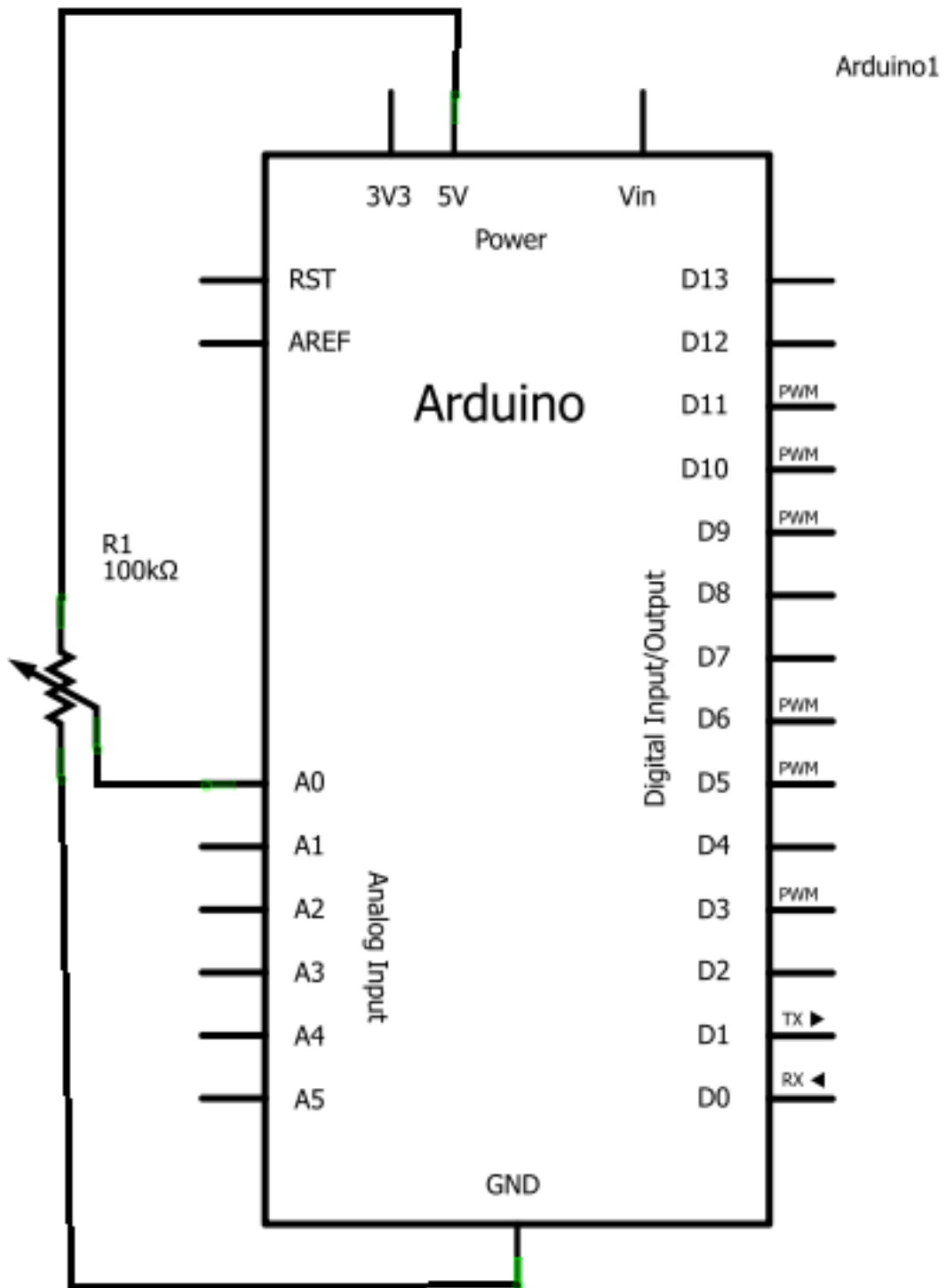


Figure 5.15 - Potentiomètre - Schéma

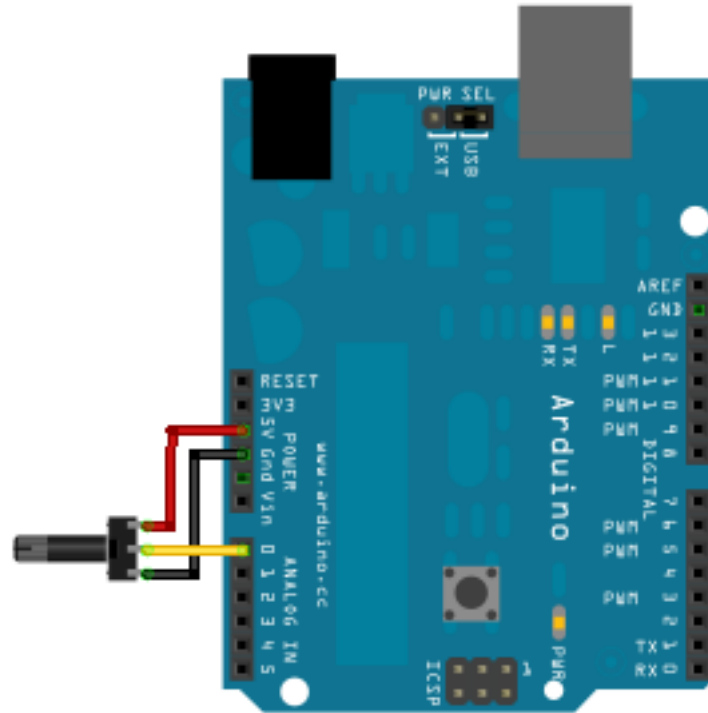


Figure 5.16 – Potentiomètre - Montage

```

// on traduit la valeur brute en tension (produit en croix)
tension = valeurLue * 5.0 / 1023;

// on affiche la valeur lue sur la liaison série
Serial.print("valeurLue = ");
Serial.println(valeurLue);

// on affiche la tension calculée
Serial.print("Tension = ");
Serial.print(tension,2);
Serial.println(" V");

// on saute une ligne entre deux affichages
Serial.println();
// on attend une demi-seconde pour que l'affichage ne soit pas trop rapide
delay(500);
}

```

Code : Voltmètre simple

Vous venez de créer votre premier voltmètre! :D

5.1.5 Une meilleure précision ?

[[q]] | Est-il possible d'améliorer la précision du convertisseur ?

Voilà une question intéressante à laquelle je répondrai qu'il existe deux solutions plus ou moins faciles à mettre en œuvre.

5.1.5.1 Solution 1 : modifier la plage d'entrée du convertisseur

C'est la solution la plus simple ! Voyons deux choses...

5.1.5.1.1 Tension de référence interne Le micro-contrôleur de l'Arduino possède plusieurs tensions de référence utilisables selon la plage de variation de la tension que l'on veut mesurer. Prenons une tension, en sortie d'un capteur, qui variera entre 0V et 1V et jamais au delà. Par défaut, la mesure se fera entre 0 et 5V sur 1024 niveaux (soit une précision de 4.88 mV). Ce qui veut dire qu'on aura seulement les 204 premiers niveaux d'utiles puisque tout le reste correspondra à plus d'un volt. Pour améliorer la précision de lecture, on va réduire la plage de mesure d'entrée du convertisseur analogique - > numérique en réduisant la tension de référence utilisée (initialement 5V). Pour cela, rien de matériel, tout se passe au niveau du programme puisqu'il y a une fonction qui existe : `analogReference()`. Cette fonction prend en paramètre le nom de la référence à utiliser :

- DEFAULT : La référence de 5V par défaut (ou 3,3V pour les cartes Arduino fonctionnant sous cette tension, telle la Due)
- INTERNAL : Une référence interne de 1.1V (pour la Arduino Uno)
- INTERNAL1V1 : Comme ci-dessus mais pour la Arduino Mega
- INTERNAL2V56 : Une référence de 2.56V (uniquement pour la Mega)
- EXTERNAL : La référence sera celle appliquée sur la broche ARef

Dans notre cas, le plus intéressant sera de prendre la valeur INTERNAL pour pouvoir faire des mesures entre 0 et 1.1V. Ainsi, on aura 1024 niveaux ce qui nous fera une précision de 1.07mV. C'est bien meilleur ! Le code est à placer dans la fonction `setup()` de votre programme :

```
void setup()
{
    // permet de choisir une tension de référence de 1.1V
    analogReference(INTERNAL);
}
```

Code : Changement de la référence : INTERNAL

5.1.5.1.2 Tension de référence externe Maintenant que se passe t'il si notre mesure devait être faite entre 0 et 3V ? On ne pourrait plus utiliser INTERNAL1V1 puisqu'on dépasse les 1.1V. On risquerait alors de griller le comparateur. Dans le cas d'une Arduino Mega, on ne peut pas non plus utiliser INTERNAL2V56 puisqu'on dépasse les 2.56V. Nous allons donc ruser en prenant une référence externe à l'aide de la valeur EXTERNAL comme ceci :

```
void setup()
{
    // permet de choisir une tension de référence externe à la carte
```

```

    analogReference(EXTERNAL);
}

```

Code : Changement de la référence : EXTERNAL

Ensuite, il ne restera plus qu'à apporter une tension de référence supérieure à 3V sur la broche ARef de la carte pour obtenir notre nouvelle référence.

[[i]] | Astuce : la carte Arduino produit une tension de 3.3V (à côté de la tension 5V). Vous pouvez donc utiliser cette tension directement pour la tension de référence du convertisseur. ;) Il suffit pour cela de relier avec un fil la sortie indiquée 3.3V à l'entrée AREF.

[[e]] | Attention cependant, la tension maximale de référence **ne peut être supérieure à +5V** et **la minimale inférieure à 0V**. En revanche, toutes les tensions comprises entre ces deux valeurs sont acceptables.

[[q]] | Mais, si je veux que ma tension d'entrée puisse varier au-delà de +5V, comment je fais ? Y a-t-il un moyen d'y parvenir ? o_o

Oui, voyez ce qui suit...

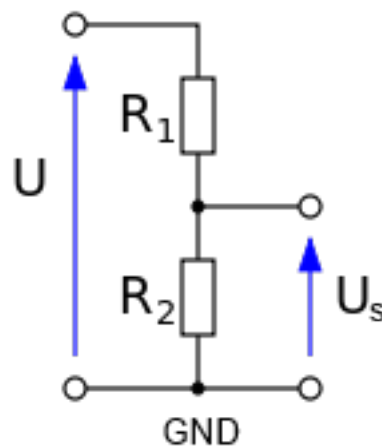


Figure 5.17 – Schéma du pont diviseur de tension

5.1.5.2 Solution 2 : utiliser un pont diviseur de tension

Nous avons vu cela à la partie précédente avec le potentiomètre. Il s'agit en fait de diviser votre signal par un certain ratio afin de l'adapter aux contraintes d'entrées du convertisseur. Par exemple, imaginons que nous ayons une mesure à faire sur un appareil qui délivre une tension comprise entre 0 et 10V. Cela ne nous arrange pas puisque nous ne sommes capable de lire des valeurs qu'entre 0 et 5V. Nous allons donc diviser cette mesure par deux afin de pouvoir la lire sans risque. Pour cela, on utilisera deux résistances de valeur identique (2 fois $10k\Omega$ par exemple) pour faire un pont diviseur de tension. La tension à mesurer rentre dans le pont (U sur le schéma ci-contre) et la tension mesurable sort au milieu du pont (tension U_s). Il ne reste plus qu'à connecter la sortie du pont à une entrée analogique de la carte Arduino et lire la valeur de la tension de sortie.

[[i]] | Libre à vous de modifier les valeurs des résistances du pont diviseur de tension pour faire rentrer des tensions différentes (même au delà de 10V!). Attention cependant à ne pas dépasser les +5V en sortie du pont !

5.1.5.3 Solution 3 : utiliser un CAN externe

La deuxième solution consisterait simplement en l'utilisation d'un convertisseur analogique -> numérique externe. A vous de choisir le bon. Il en existe beaucoup, ce qu'il faut principalement regarder c'est :

- la **précision** (10 bits pour Arduino, existe d'autre en 12bits, 16bits, ...)
- la **vitesse d'acquisition** (celui d'Arduino est à une vitesse de 100µs)
- le **mode de transfert** de la lecture (liaison série, I²C, ...)
- le **nombre d'entrées** (6 sur Arduino Uno)
- la **tension d'entrée** maximale et minimale (max +5V et min 0V)

Au programme :

- Le prochain chapitre est un TP faisant usage de ces voies analogiques
- Le chapitre qui le suit est un chapitre qui vous permettra de créer des tensions analogiques avec votre carte Arduino, idéal pour mettre en œuvre la deuxième solution d'amélioration de la précision de lecteur du convertisseur !

En somme, ce chapitre vous a permis de vous familiariser un peu avec les tensions analogiques, ce qui vous permettra par la suite de gérer plus facilement les grandeurs renvoyées par des capteurs quelconques.

5.2 [TP] Vu-mètre à LED

On commence cette partie sur l'analogique sur les chapeaux de roues en réalisant tout de suite notre premier TP. Ce dernier n'est pas très compliqué, à condition que vous ayez suivi correctement le tuto et que vous n'ayez pas oublié les bases des parties précédentes ! :p

5.2.1 Consigne

5.2.1.0.1 Vu-mètre, ça vous parle ? Dans ce TP, nous allons réaliser un **vu-mètre**. Même si le nom ne vous dit rien, je suis sûr que vous en avez déjà rencontré. Par exemple, sur une chaîne hi-fi ou sur une table de mixage on voit souvent des loupottes s'allumer en fonction du volume de la note joué. Et bien c'est ça un vu-mètre, c'est un système d'affichage sur plusieurs LED, disposées en ligne, qui permettent d'avoir un retour visuel sur une information analogique (dans l'exemple, ce sera le volume).

5.2.1.0.2 Objectif Pour l'exercice, nous allons réaliser la visualisation d'une tension. Cette dernière sera donnée par un potentiomètre et sera affichée sur 10 LED. Lorsque le potentiomètre sera à 0V, on allumera 0 LED, puis lorsqu'il sera au maximum on les allumera toutes. Pour les valeurs comprises entre 0 et 5V, elles devront allumer les LED proportionnellement. Voilà, ce n'est pas plus compliqué que ça. Comme d'habitude voici une petite vidéo vous montrant le résultat attendu et bien entendu ...

-> **BON COURAGE!** <-

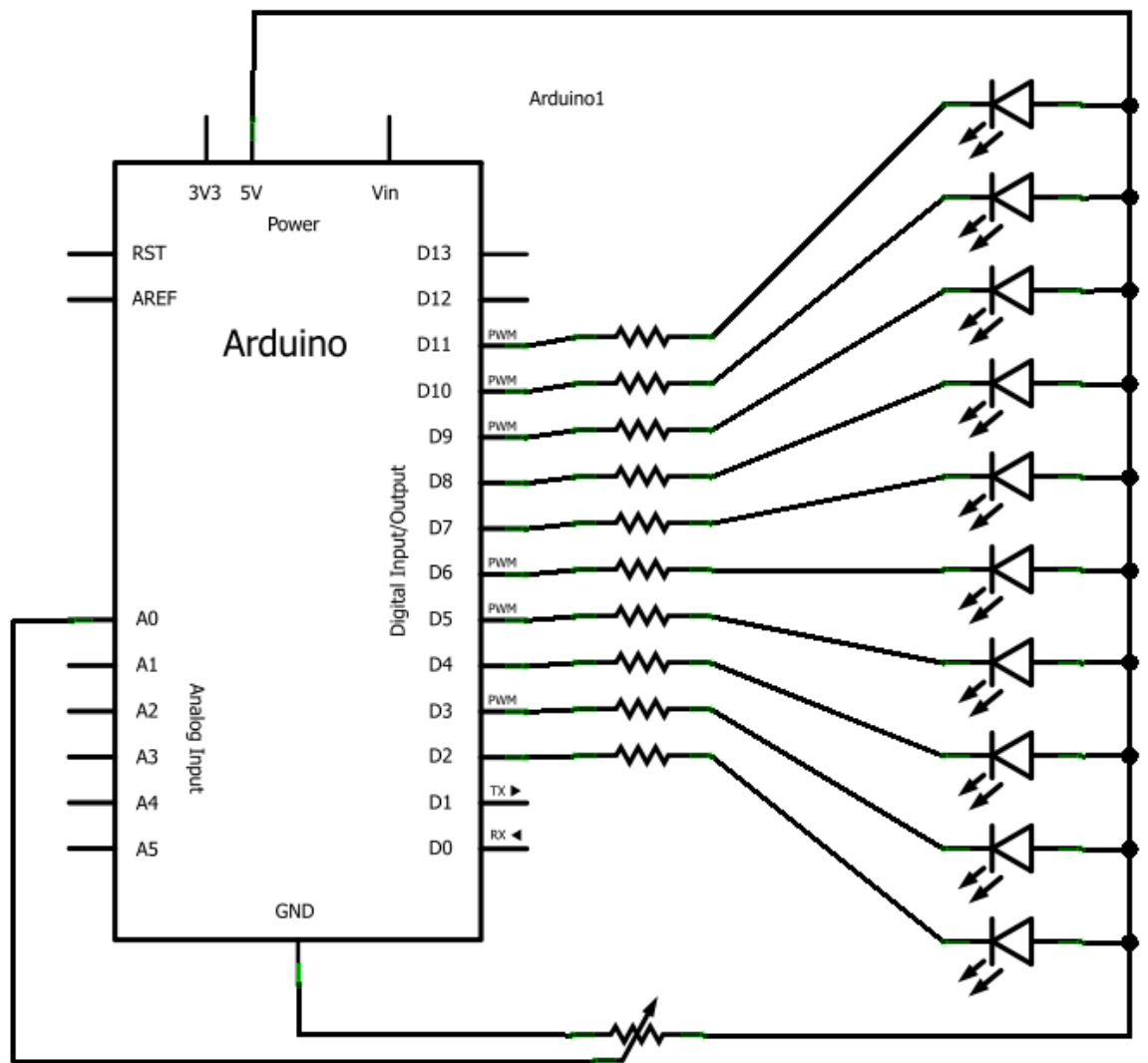
->!(https://www.youtube.com/watch?v=UkmQEM_5ZIE)<-

5.2.2 Correction !

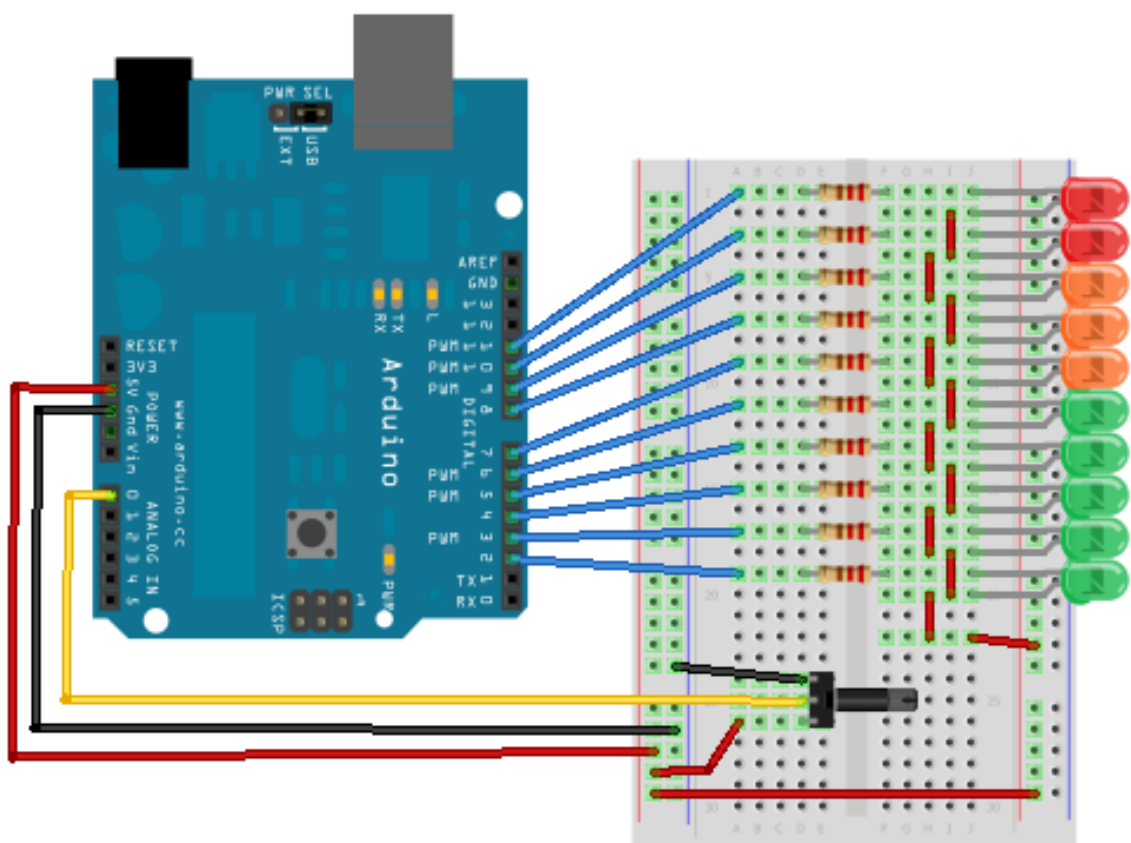
J'espère que tout c'est bien passé pour vous et que l'affichage cartonne ! Voici maintenant venu l'heure de la correction, en espérant que vous n'en aurez pas besoin et que vous la consulterez juste pour votre culture. ;) Comme d'habitude nous allons commencer par voir le schéma puis ensuite nous étudierons le code.

5.2.2.1 Schéma électronique

Le schéma n'est pas très difficile. Nous allons retrouver 10 LEDs et leurs résistances de limitations de courant branchées sur 10 broches de l'Arduino (histoire d'être original nous utiliserons 2 à 11). Ensuite, nous brancherons un potentiomètre entre le +5V et la masse. Sa broche centrale, qui donne la tension variable sera connectée à l'entrée analogique 0 de l'Arduino. Voici le schéma obtenu :



[[s]] | ->



| <-

5.2.2.2 Le code

Là encore vous commencez à avoir l'habitude, nous allons d'abord étudier le code des variables globales (pourquoi elles existent ?), voir la fonction `setup()`, puis enfin étudier la boucle principale et les fonctions annexes utilisées.

5.2.2.2.1 Variables globales et setup Dans ce TP nous utilisons 10 LEDs, ce qui représente autant de sorties sur la carte Arduino donc autant de "const int ..." à écrire. Afin de ne pas se fatiguer de trop, nous allons déclarer un tableau de "const int" plutôt que de copier/coller 10 fois la même ligne. Ensuite, nous allons déclarer la broche analogique sur laquelle sera branché le potentiomètre. Enfin, nous déclarons une variable pour stocker la tension mesurée sur le potentiomètre. Et c'est tout pour les déclarations !

```
// Déclaration et remplissage du tableau...
// ...représentant les broches des LEDs
const int leds[10] = {2,3,4,5,6,7,8,9,10,11};
// le potentiomètre sera branché sur la broche analogique 0
const int potar = 0;
// variable stockant la tension mesurée
int tension = 0;
```

Code : TP Vumètre, les variables

Une fois que l'on a fait ces déclarations, il ne nous reste plus qu'à déclarer les broches en sortie et à les mettre à l'état HAUT pour éteindre les LEDs. Pour faire cela de manière simple (au lieu de 10 copier/coller), nous allons utiliser une boucle `for` pour effectuer l'opérations 10 fois (afin d'utiliser la puissance du tableau).

```
void setup()
{
    int i = 0;
    for(i = 0; i < 10; i++)
    {
        // déclaration de la broche en sortie
        pinMode(leds[i], OUTPUT);
        // mise à l'état haut
        digitalWrite(leds[i], HIGH);
    }
}
```

Code : TP Vumètre, le setup

5.2.2.2.2 Boucle principale Alors là vous allez peut-être être surpris mais nous allons avoir une fonction principale `super light`. En effet, elle ne va effectuer que deux opérations : Mesurer la tension du potentiomètre, puis appeler une fonction d'affichage pour faire le rendu visuel de cette tension. Voici ces deux lignes de code :

```
void loop()
{
    // on récupère la valeur de la tension du potentiomètre
    tension = analogRead(potar);
    // et on affiche sur les LEDs cette tension
    afficher(tension);
}
```

Code : TP Vumètre, la loop

Encore plus fort, la même écriture mais en une seule ligne !

```
void loop()
{
    // la même chose qu'avant même en une seule ligne !
    afficher(analogRead(potar));
}
```

5.2.2.2.3 Fonction d’affichage Alors certes la fonction principale est très légère, mais ce n’est pas une raison pour ne pas avoir un peu de code autre part. En effet, le gros du traitement va se faire dans la fonction d’affichage, qui, comme son nom et ses arguments l’indiquent, va servir à afficher sur les LEDs la tension mesurée. Le but de cette dernière sera d’allumer les LEDs de manière proportionnelle à la tension mesurée. Par exemple, si la tension mesurée vaut 2,5V (sur 5V max) on allumera 5 LEDs (sur 10). Si la tension vaut 5V, on les allumera toutes. Je vais maintenant vous montrer une astuce toute simple qui va tirer pleinement parti du tableau de broches créé tout au début. Tout d’abord, mettons-nous d’accord. Lorsque l’on fait une mesure analogique, la valeur retournée est comprise entre 0 et 1023. Ce que je vous propose, c’est donc d’allumer une LED par tranche de 100 unités. Par exemple, si la valeur est comprise entre 0 et 100, une seule LED est allumée. Ensuite, entre 100 et 200, on allume une LED supplémentaire, etc. Pour une valeur entre 700 et 800 on allumera donc... 8 LEDs, bravo à ceux qui suivent ! :s Ce comportement va donc s’écrire simplement avec une boucle for, qui va incrémenter une variable i de 0 à 10. Dans cette boucle, nous allons tester si la valeur (image de la tension) est inférieure à i multiplié par 100 (ce qui représentera nos différents pas). Si le test vaut VRAI, on allume la LED i, sinon on l’éteint. Démonstration :

```
void afficher(int valeur)
{
    int i;
    for(i=0; i < 10; i++)
    {
        if(valeur < (i*100))
            digitalWrite(leds[i], LOW); // on allume la LED
        else
            digitalWrite(leds[i], HIGH); // ou on éteint la LED
    }
}
```

Code : TP Vumètre, fonction d’affichage

5.2.3 Amélioration

Si jamais vous avez trouvé l'exercice trop facile, pourquoi ne pas faire un peu de zèle en réalisant carrément un mini-voltmètre en affichant sur deux afficheurs 7 segments une tension mesurée (un afficheur pour les Volts et un autre pour la première décimale) ? Ceci n'est qu'une idée d'amélioration, la solution sera donnée, commentée, mais pas expliquée en détail car vous devriez maintenant avoir tout le savoir pour la comprendre. L'exercice est juste là pour vous entraîner et pour vous inspirer avec un nouveau montage.

->!(<https://www.youtube.com/watch?v=mInJ5Uz7BH8>)<-

```
[[s]] | cpp | // les broches du décodeur 7 segments | const int bit_A = 2; |
const int bit_B = 3; | const int bit_C = 4; | const int bit_D = 5; | | //
les broches des transistors pour l'afficheur des dizaines et des unités |
const int alim_dizaine = 6; | const int alim_unite = 7; | | // la broche
du potar | const int potar = 0; | | float tension = 0.0; // tension mise
en forme | int val = 0; // tension brute lue (0 à 1023) | bool afficheur
= false; | long temps; | | void setup() | { | // Les broches sont toutes
des sorties (sauf les boutons) | pinMode(bit_A, OUTPUT); | pinMode(bit_B,
OUTPUT); | pinMode(bit_C, OUTPUT); | pinMode(bit_D, OUTPUT); | pinMode(alim_dizaine,
OUTPUT); | pinMode(alim_unite, OUTPUT); | | // Les broches sont toutes
mise à l'état bas (sauf led rouge éteinte) | digitalWrite(bit_A, LOW); |
digitalWrite(bit_B, LOW); | digitalWrite(bit_C, LOW); | digitalWrite(bit_D,
LOW); | digitalWrite(alim_dizaine, LOW); | digitalWrite(alim_unite, LOW);
| temps = millis(); // enregistre "l'heure" | } | | void loop() | { | //
on fait la lecture analogique | val = analogRead(potar); | // mise en forme
de la valeur lue | tension = val * 5; // simple règle de trois pour la
conversion ( *5/1023) | tension = tension / 1023; | // à ce stade on a
une valeur de type 3.452 Volts... | // que l'on va multiplier par 10 pour
afficher avec les vieilles fonctions | tension = tension*10; | | // si ça
fait plus de 10 ms qu'on affiche, on change de 7 segments | if((millis() -
temps) > 10) | { | // on inverse la valeur de "afficheur" | // pour changer
d'afficheur (unité ou dizaine) | afficheur = !afficheur; | // on affiche |
afficher_nombre(tension, afficheur); | temps = millis(); // on met à jour
le temps | } | } | | // fonction permettant d'afficher un nombre | void
afficher_nombre(float nombre, bool afficheur) | { | long temps; | char
unite = 0, dizaine = 0; | if(nombre > 9) | dizaine = nombre / 10; // on
recupere les dizaines | unite = nombre - (dizaine*10); // on recupere les
unités | | if(afficheur) | { | // on affiche les dizaines | digitalWrite(alim_unite,
LOW); | digitalWrite(alim_dizaine, HIGH); | afficher(dizaine); | } | else
| { | // on affiche les unités | digitalWrite(alim_dizaine, LOW); | digitalWrite(alim_
HIGH); | afficher(unite); | } | } | | // fonction écrivant sur un seul
afficheur | void afficher(char chiffre) | { | // on commence par écrire 0,
donc tout à l'état bas | digitalWrite(bit_A, LOW); | digitalWrite(bit_B,
LOW); | digitalWrite(bit_C, LOW); | digitalWrite(bit_D, LOW); | | if(chiffre
>= 8) | { | digitalWrite(bit_D, HIGH); | chiffre = chiffre - 8; | } | if(chiffre
>= 4) | { | digitalWrite(bit_C, HIGH); | chiffre = chiffre - 4; | } | if(chiffre
>= 2) | { | digitalWrite(bit_B, HIGH); | chiffre = chiffre - 2; | } | if(chiffre
>= 1) | { | digitalWrite(bit_A, HIGH); | chiffre = chiffre - 1; | } | //
```

Et voilà!! | } |

Vous savez maintenant comment utiliser et afficher des valeurs analogiques externes à la carte Arduino. En approfondissant vos recherches et vos expérimentations, vous pourrez certainement faire pas mal de choses telles qu'un robot en associant des capteurs et des actionneurs à la carte, des appareils de mesures (Voltmètre, Ampèremètre, Oscilloscope, etc.). Je compte sur vous pour créer par vous-même! ;) Direction, le prochain chapitre où vous découvrirez comment faire une conversion numérique -> analogique...

5.3 Et les sorties “analogiques”, enfin... presque !

Vous vous souvenez du premier chapitre de cette partie ? Oui, lorsque je vous parlais de convertir une grandeur analogique (tension) en une donnée numérique. Eh bien là, il va s'agir de faire l'opération inverse. Comment ? C'est ce que nous allons voir. Je peux vous dire que ça à un rapport avec la PWM...

5.3.1 Convertir des données binaires en signal analogique

Je vais vous présenter deux méthodes possibles qui vont vous permettre de convertir des données numériques en grandeur analogique (je ne parlerai là encore de tension). Mais avant, plaçons-nous dans le contexte.

[[q]] | Convertir du binaire en analogique, pour quoi faire ? C'est vrai, avec la conversion analogique->numérique il y avait une réelle utilité, mais là, qu'en est-il ?

L'utilité est tout aussi pesante que pour la conversion A->N. Cependant, les applications sont différentes, à chaque outil un besoin dirais-je. En effet, la conversion A->N permettait de transformer une grandeur analogique non-utilisable directement par un système à base numérique en une donnée utilisable pour une application numérique.

Ainsi, on a pu envoyer la valeur lue sur la liaison série. Quant à la conversion opposée, conversion N->A, les applications sont différentes, je vais en citer une plus ou moins intéressante : par exemple commander une, ou plusieurs, LED tricolore (Rouge-Vert-Bleu) pour créer un luminaire dont la couleur est commandée par le son (nécessite une entrée analogique :-°). Tiens, en voilà un projet intéressant ! Je vais me le garder sous la main... :ninja :

[[q]] | Alors ! alors ! alors !! Comment on fait !? :D

Serait-ce un léger soupçon de curiosité que je perçois dans vos yeux frétilants ? :p Comment fait-on ? Suivez -le guide !

5.3.1.1 Convertisseur Numérique->Analogique

La première méthode consiste en l'utilisation d'un convertisseur Numérique->Analogique (que je vais abrégé CNA). Il en existe, tout comme le CAN, de plusieurs sortes :

- **CNA à résistances pondérées** : ce convertisseur utilise un grand nombre de résistances qui ont chacune le double de la valeur de la résistance qui la précède. On a donc des résistances de valeur R , $2R$, $4R$, $8R$, $16R$, ..., $256R$, $512R$, $1024R$, etc. Chacune des résistances sera connectée grâce au micro-contrôleur à la masse ou bien au $+5V$. Ces niveaux logiques correspondent aux bits de données de la valeur numérique à convertir. Plus le bit est de poids fort, plus la résistance à laquelle il est adjoint est grande (maximum R). À l'inverse, plus il est de poids faible, plus il verra sa résistance de sortie de plus petite valeur. Après, grâce à un petit montage électronique, on arrive à créer une tension proportionnelle au nombre de bit à 1.
- **CNA de type $R/2R$** : là, chaque sortie du micro-contrôleur est reliée à une résistance de même valeur ($2R$), elle-même connectée au $+5V$ par l'intermédiaire d'une résistance de valeur R . Toujours avec un petit montage, on arrive à créer une tension analogique proportionnelle au nombre de bit à 1.

Cependant, je n'expliquerai pas le fonctionnement ni l'utilisation de ces convertisseurs car ils doivent être connectés à autant de broches du micro-contrôleur qu'ils ne doivent avoir de précision. Pour une conversion sur 10 bits, le convertisseur doit utiliser 10 sorties du microcontrôleur !

5.3.1.2 PWM ou MLI

Bon, s'il n'y a pas moyen d'utiliser un CNA, alors on va le créer utiliser ce que peut nous fournir la carte Arduino : la **PWM**. Vous vous souvenez que j'ai évoqué ce terme dans le chapitre sur la conversion $A \rightarrow N$? Mais concrètement, c'est quoi ?

[[a]] | Avant de poursuivre, je vous conseille d'aller [relire cette première partie](#) du chapitre sur les entrées analogiques pour revoir les rappels que j'ai faits sur les signaux analogiques. ;)

5.3.1.2.1 Définition N'ayez point peur, je vais vous expliquer ce que c'est au lieu de vous donner une définition tordue comme on peut en trouver parfois dans les dictionnaires. ;) D'abord, la PWM sa veut dire : **Pulse Width Modulation** et en français cela donne **Modulation à Largeur d'Impulsion** (MLI). La PWM est en fait un signal numérique qui, à une **fréquence** donnée, a un **rapport cyclique** qui change.

[[q]] | Y'a plein de mots que je comprends pas, c'est normal ? o_O

Oui, car pour l'instant je n'en ai nullement parlé. Voilà donc notre prochain objectif.

5.3.1.2.2 La fréquence et le rapport cyclique La *fréquence* d'un signal périodique correspond au nombre de fois que la période se répète en UNE seconde. On la mesure en **Hertz**, noté **Hz**. Prenons l'exemple d'un signal logique qui émet un 1, puis un 0, puis un 1, puis un 0, etc. autrement dit un signal créneaux, on va mesurer sa période (en temps) entre le début du niveau 1 et la fin du niveau 0 :

Ensuite, lorsque l'on aura mesuré cette période, on va pouvoir calculer sa fréquence (le nombre de périodes en une seconde) grâce à la formule suivante :

$$F = \frac{1}{T}$$

Avec :

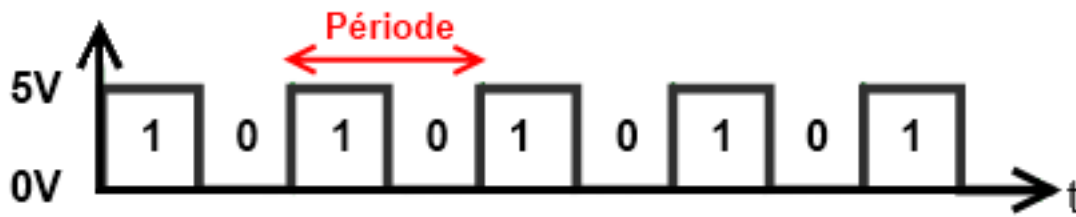


Figure 5.18 – Représentation d'une période

- F : fréquence du signal en Hertz (Hz)
- T : temps de la période en seconde (s)

Le *rapport cyclique*, un mot bien particulier pour désigner le fait que le niveau logique 1 peut ne pas durer le même temps que le niveau logique 0. C'est avec ça que tout repose le principe de la PWM. C'est-à-dire que la PWM est un signal de fréquence fixe qui a un rapport cyclique qui varie avec le temps suivant "les ordres qu'elle reçoit" (on reviendra dans un petit moment sur ces mots).

Le rapport cyclique est mesuré en pourcentage (%). Plus le pourcentage est élevé, plus le niveau logique 1 est présent dans la période et moins le niveau logique 0 l'est. Et inversement. Le rapport cyclique du signal est donc le pourcentage de temps de la période durant lequel le signal est au niveau logique 1. En somme, cette image extraite de la [documentation officielle](#) d'Arduino nous montre quelques exemples d'un signal avec des rapports cycliques différents :

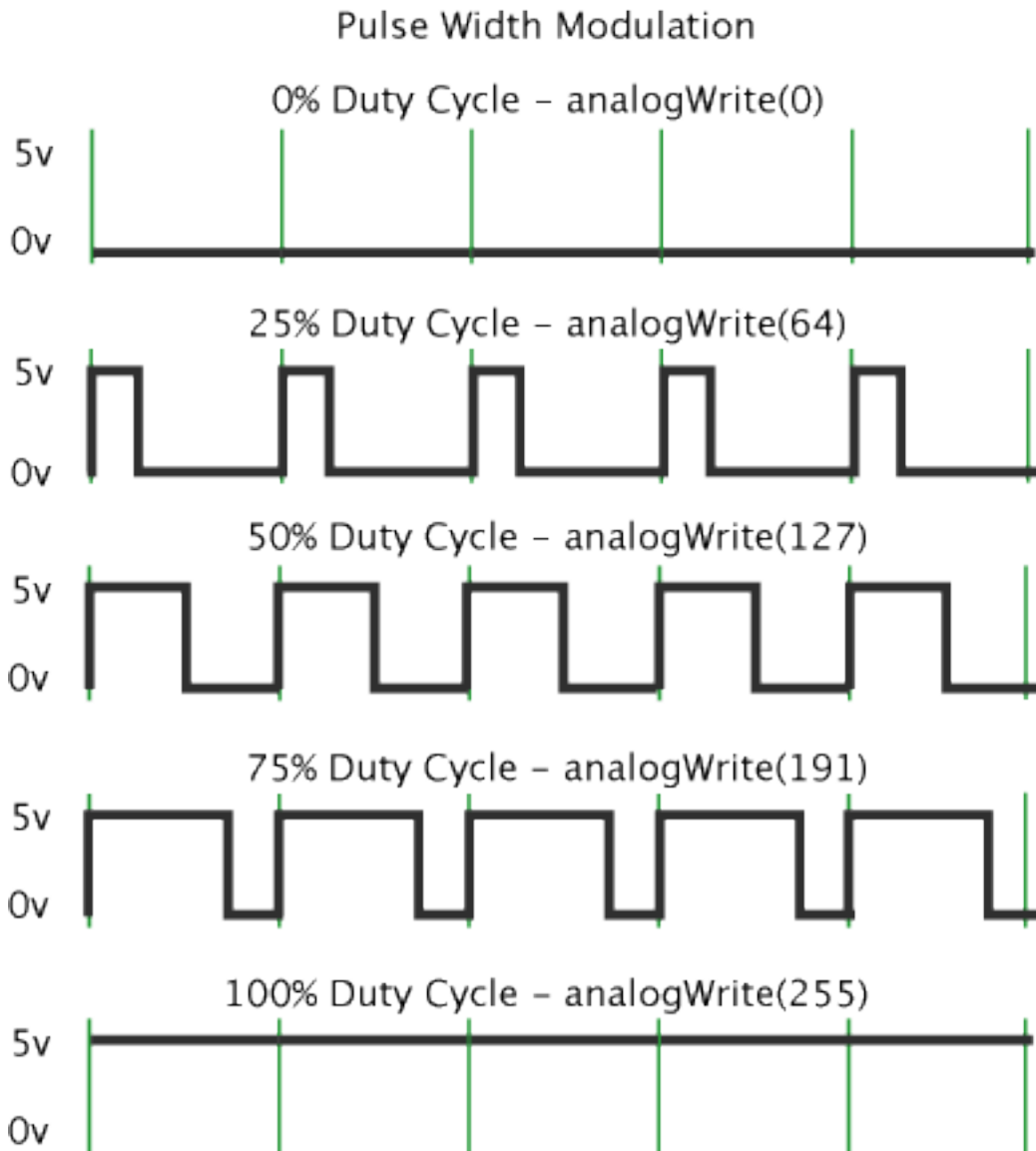


Figure :

Des signaux aux rapports cycliques différents - (CC-BY-SA - [Timothy Hirzel](#))

[[i]] | Astuce : Rapport cyclique ce dit **Duty Cycle** en anglais.

Ce n'est pas tout ! Après avoir généré ce signal, il va nous falloir le transformer en signal analogique. Et oui ! Pour l'instant ce signal est encore constitué d'états logiques, on va donc devoir le transformer en extrayant sa *valeur moyenne*... Je ne vous en dis pas plus, on verra plus bas ce que cela signifie.

5.3.2 La PWM de l'Arduino

5.3.2.1 Avant de commencer à programmer

5.3.2.1.1 Les broches de la PWM Sur votre carte Arduino, vous devriez disposer de 6 broches qui sont compatibles avec la génération d'une PWM. Elles sont repérées par le symbole *tilde* ~ . Voici les broches générant une PWM : 3, 5, 6, 9, 10 et 11.

5.3.2.1.2 La fréquence de la PWM Cette fréquence, je le disais, est fixe, elle ne varie pas au cours du temps. Pour votre carte Arduino elle est de environ 490Hz.

5.3.2.1.3 La fonction analogWrite() Je pense que vous ne serez pas étonné si je vous dis que Arduino intègre une fonction toute prête pour utiliser la PWM ? Plus haut, je vous disais ceci :

la PWM est un signal de fréquence fixe qui a un rapport cyclique qui varie avec le temps suivant “les ordres qu’elle reçoit” Source : Moi :P

C’est sur ce point que j’aimerais revenir un instant. En fait, les ordres dont je parle sont les paramètres passés dans la fonction qui génère la PWM. Ni plus ni moins. Étudions maintenant la fonction permettant de réaliser ce signal : `analogWrite()`. Elle prend deux arguments :

- Le premier est le numéro de la broche où l’on veut générer la PWM
- Le second argument représente la valeur du rapport cyclique à appliquer. Malheureusement on n’exprime pas cette valeur en pourcentage, mais avec un nombre entier compris entre 0 et 255

Si le premier argument va de soi, le second mérite quelques précisions. Le rapport cyclique s’exprime de 0 à 100 % en temps normal. Cependant, dans cette fonction il s’exprimera de 0 à 255 (sur 8 bits). Ainsi, pour un rapport cyclique de 0% nous enverrons la valeur 0, pour un rapport de 50% on enverra 127 et pour 100% ce sera 255. Les autres valeurs sont bien entendu considérées de manière proportionnelle entre les deux. Il vous faudra faire un petit calcul pour savoir quel est le pourcentage du rapport cyclique plutôt que l’argument passé dans la fonction.

5.3.2.1.4 Utilisation Voilà un petit exemple de code illustrant tout ça :

```
// une sortie analogique sur la broche 6
const int sortieAnalogique = 6;

void setup()
{
    pinMode(sortieAnalogique, OUTPUT);
}

void loop()
{
    // on met un rapport cyclique de 107/255 = 42 %
    analogWrite(sortieAnalogique, 107);
}
```

Code : Exemple simple d’utilisation de la PWM

5.3.2.2 Quelques outils essentiels

Savez-vous que vous pouvez d’ores et déjà utiliser cette fonction pour allumer plus ou moins intensément une LED ? En effet, pour un rapport cyclique faible, la LED va se voir parcourir par un courant moins longtemps que lorsque le rapport cyclique est fort. Or, si elle est parcourue moins longtemps par le courant, elle s’éclairera également moins longtemps. En faisant varier le rapport cyclique, vous pouvez ainsi faire varier la luminosité de la LED.

5.3.2.2.1 La LED RGB ou RVB RGB pour Red-Green-Blue en anglais. Cette LED est composée de trois LED de couleurs précédemment énoncées. Elle possède donc 4 broches et existe sous deux modèles : à anode commune et à cathode commune. Exactement comme les afficheurs 7 segments. Choisissez-en une à *anode commune*.

5.3.2.2.2 Mixer les couleurs Lorsque l’on utilise des couleurs, il est bon d’avoir quelques bases en arts plastiques. Révisons les fondements. La lumière, peut-être ne le savez-vous pas, est composée de trois couleurs primaires qui sont :

- Le **rouge**
- Le **vert**
- Le **bleu**

À partir de ces trois couleurs, il est possible de créer n’importe quelle autre couleur du spectre lumineux visible en mélangeant ces trois couleurs primaires entre elles. Par exemple, pour faire de l’orange on va mélanger du rouge (2/3 du volume final) et du vert (à 1/3 du volume final). Je vous le disais, la fonction `analogWrite()` prend un argument pour la PWM qui va de 0 à 255. Tout comme la proportion de couleur dans les logiciels de dessin ! On parle de “norme RGB” faisant référence aux trois couleurs primaires. Pour connaître les valeurs RGB d’une couleur, je vous propose de regarder avec le logiciel **Gimp** (gratuit et multiplateforme). Pour cela, il suffit de deux observations/clics :

1. Tout d’abord on sélectionne la “boîte à couleurs” dans la boîte à outils
2. Ensuite, en jouant sur les valeurs R, G et B on peut voir la couleur obtenue

Afin de faire des jolies couleurs, nous utiliserons `analogWrite()` trois fois (une pour chaque LED). Prenons tout de suite un exemple avec du **orange** et regardons sa composition sous Gimp :

À partir de cette image nous pouvons voir qu’il faut :

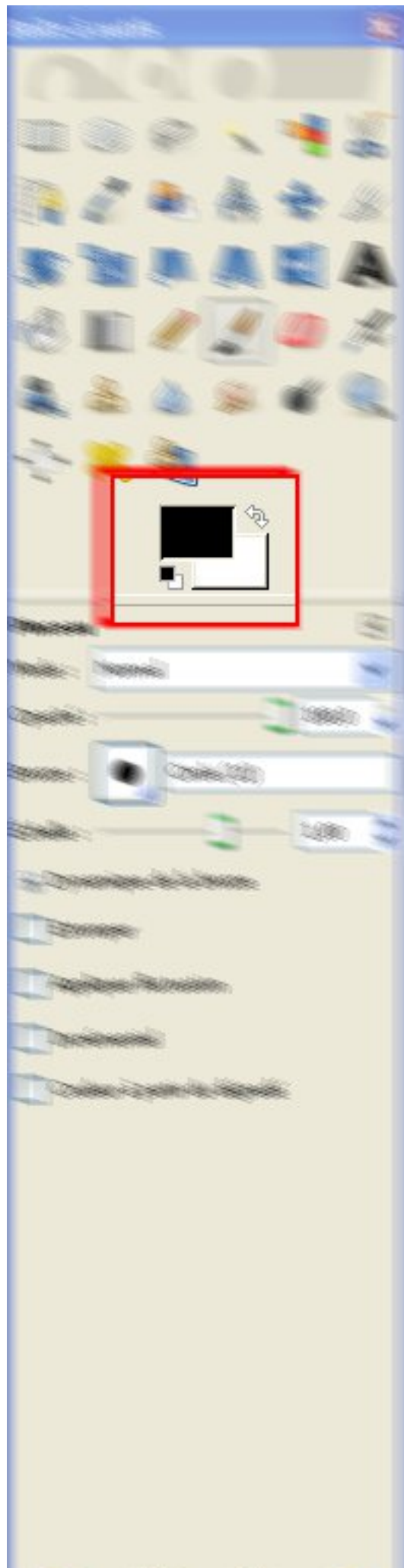
- 100 % de rouge (255)
- 56 % de vert (144)
- 0% de bleu (0)

Nous allons donc pouvoir simplement utiliser ces valeurs pour faire une jolie couleur sur notre LED RGB :

```
const int ledRouge = 11;
const int ledVerte = 9;
const int ledBleue = 10;

void setup()
{
  // on déclare les broches en sorties
  pinMode(ledRouge, OUTPUT);
  pinMode(ledVerte, OUTPUT);
  pinMode(ledBleue, OUTPUT);

  // on met la valeur de chaque couleur
  analogWrite(ledRouge, 255);
  analogWrite(ledVerte, 144);
}
```



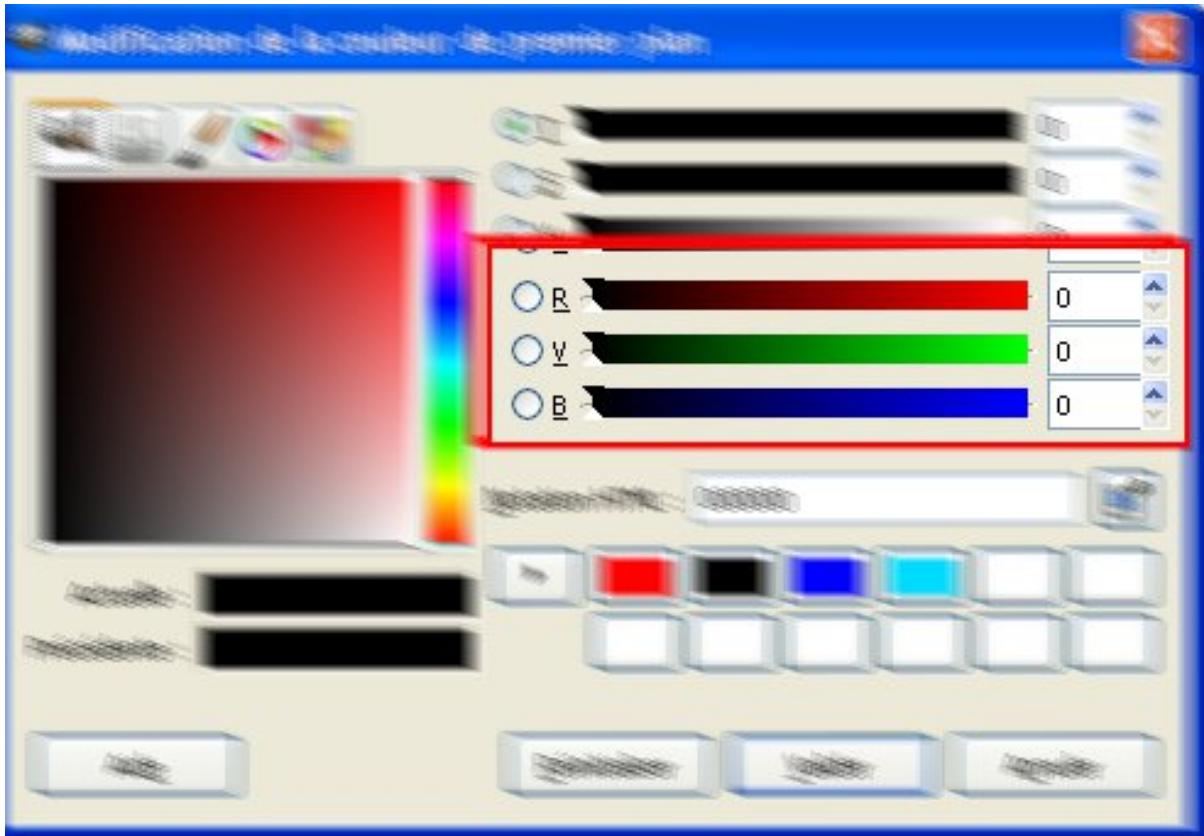


Figure 5.20 – Le ColorPicker de Gimp

```

    analogWrite(ledBleue, 0);
}

void loop()
{
    // on ne change pas la couleur donc rien à faire dans la boucle principale
}

```

Code : Utilisation d'une led RGB

[[q]] | Moi j'obtiens pas du tout de l'orange ! Plutôt un bleu étrange...

C'est exact. Souvenez-vous que c'est une LED à anode commune, or lorsqu'on met une tension de 5V en sortie du microcontrôleur, la LED sera éteinte. Les LED sont donc pilotées à l'état bas. Autrement dit, ce n'est pas la durée de l'état haut qui est importante mais plutôt celle de l'état bas. Afin de pallier cela, il va donc falloir mettre la valeur “inverse” de chaque couleur sur chaque broche en faisant l'opération $ValeurReelle = 255 - ValeurTheorique$. Le code précédent devient donc :

```

const int ledRouge = 11;
const int ledVerte = 9;
const int ledBleue = 10;

void setup()
{

```

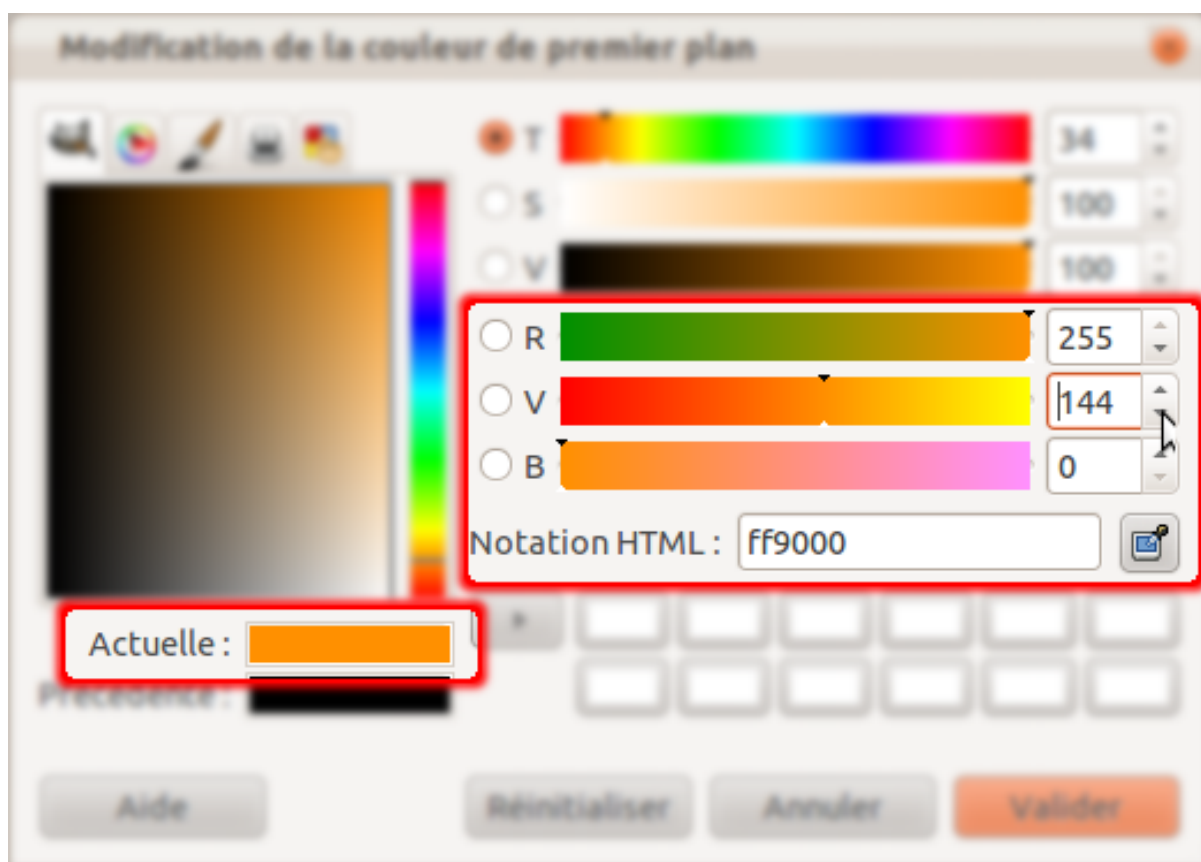


Figure 5.21 – La couleur orange avec Gimp

```

// on déclare les broches en sorties
pinMode(ledRouge, OUTPUT);
pinMode(ledVerte, OUTPUT);
pinMode(ledBleue, OUTPUT);

// on met la valeur de chaque couleur
analogWrite(ledRouge, 255-255);
analogWrite(ledVerte, 255-144);
analogWrite(ledBleue, 255-0);
}

```

Code : Utilisation d’une led RGB à anode commune

On en a fini avec les rappels, on va pouvoir commencer un petit exercice.

5.3.2.3 À vos claviers, prêt... programmez !

5.3.2.3.1 L’objectif L’objectif est assez simple, vous allez générer trois PWM différentes (une pour chaque LED de couleur) et créer 7 couleurs (le noir ne compte pas ! :P) distinctes qui sont les suivantes :

- rouge
- vert
- bleu
- jaune
- bleu ciel
- violet
- blanc

Ces couleurs devront “défiler” une par une (dans l’ordre que vous voudrez) toutes les 500ms.

5.3.2.3.2 Le montage à réaliser Vous allez peut-être être surpris car je vais utiliser pour le montage une LED à anode commune, afin de bien éclairer les LED avec la bonne proportion de couleur. Donc, lorsqu’il y aura la valeur 255 dans analogWrite(), la LED de couleur rouge, par exemple, sera complètement illuminée.

**** C’est parti ! ;)**

5.3.2.3.3 Correction Voilà le petit programme que j’ai fait pour répondre à l’objectif demandé :

```

// définition des broches utilisée (vous êtes libre de les changer)
const int led_verte = 9;
const int led_bleue = 10;
const int led_rouge = 11;

int compteur_defilement = 0; // variable permettant de changer de couleur

void setup()

```

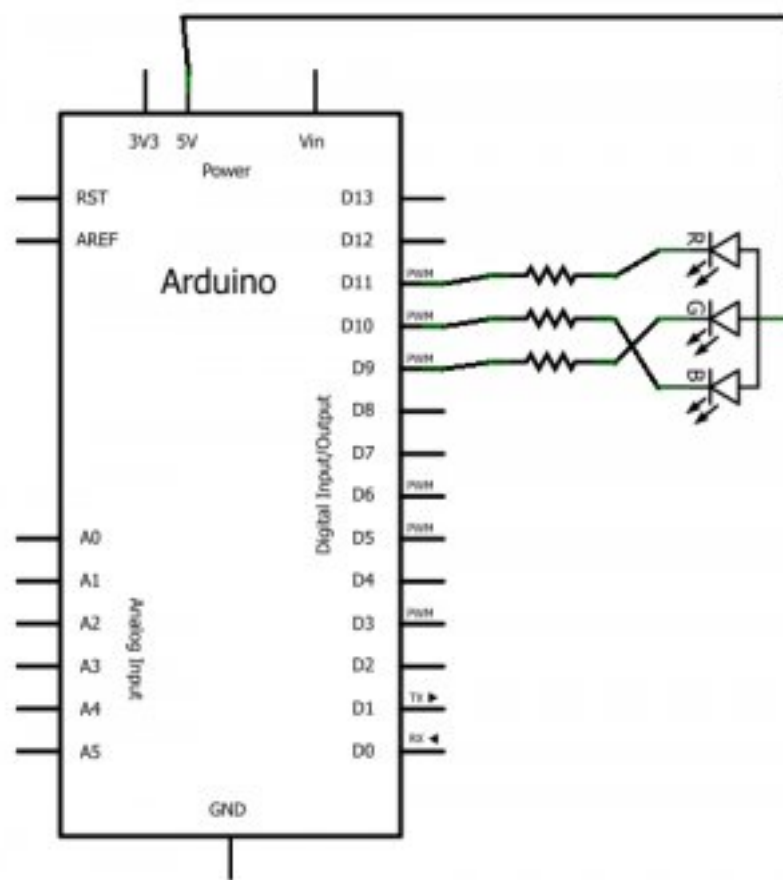


Figure 5.22 – RGB schéma

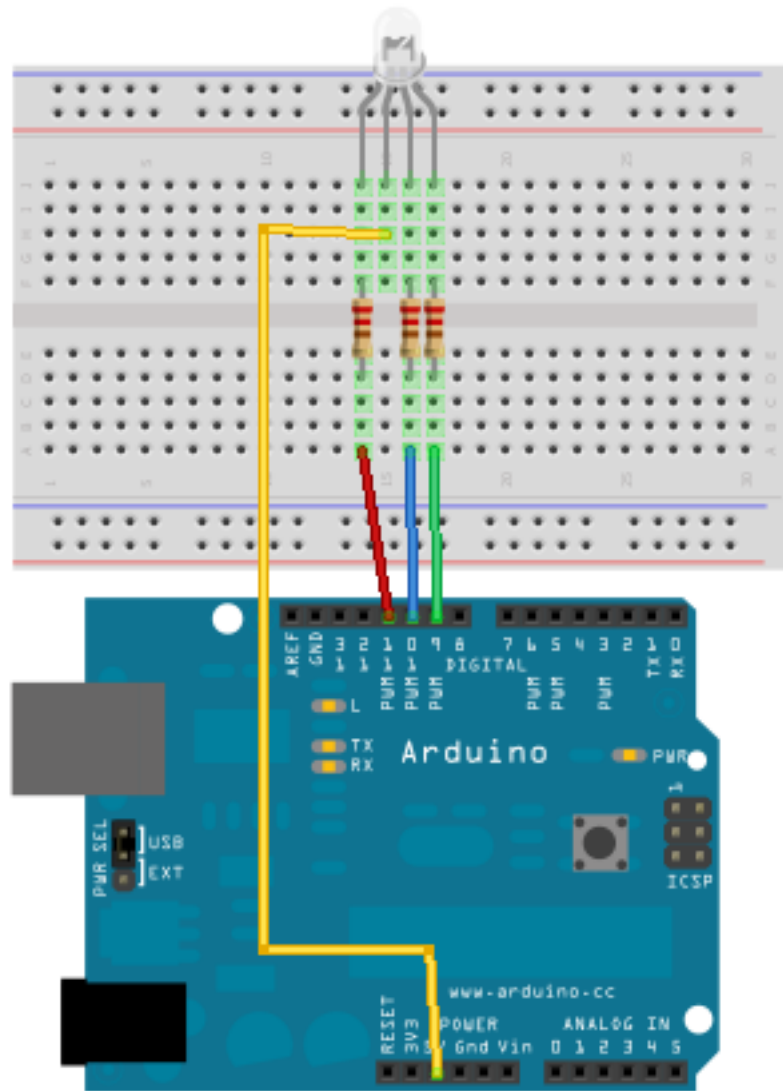


Figure 5.23 – RGB montage

```

{
    // définition des broches en sortie
    pinMode(led_rouge, OUTPUT);
    pinMode(led_verte, OUTPUT);
    pinMode(led_bleue, OUTPUT);
}

void loop()
{
    couleur(compteur_defilement); // appel de la fonction d'affichage
    compteur_defilement++; // incrémentation de la couleur à afficher

    // si le compteur dépasse 6 couleurs
    if(compteur_defilement > 6)
        compteur_defilement = 0;

    delay(500);
}

void couleur(int numeroCouleur)
{
    switch(numeroCouleur)
    {
        case 0 : // rouge
            // rapport cyclique au minimum pour une meilleure luminosité de la LED
            analogWrite(led_rouge, 0);
            // qui je le rappel est commandée en "inverse"
            // (0 -> LED allumée; 255 -> LED éteinte)
            analogWrite(led_verte, 255);
            analogWrite(led_bleue, 255);
            break;
        case 1 : // vert
            analogWrite(led_rouge, 255);
            analogWrite(led_verte, 0);
            analogWrite(led_bleue, 255);
            break;
        case 2 : // bleu
            analogWrite(led_rouge, 255);
            analogWrite(led_verte, 255);
            analogWrite(led_bleue, 0);
            break;
        case 3 : // jaune
            analogWrite(led_rouge, 0);
            analogWrite(led_verte, 0);
            analogWrite(led_bleue, 255);
            break;
        case 4 : // violet
            analogWrite(led_rouge, 0);

```



```

    analogWrite(led_verte, 255);
    analogWrite(led_bleue, 0);
    break;
case 5 : // bleu ciel
    analogWrite(led_rouge, 255);
    analogWrite(led_verte, 0);
    analogWrite(led_bleue, 0);
    break;
case 6 : // blanc
    analogWrite(led_rouge, 0);
    analogWrite(led_verte, 0);
    analogWrite(led_bleue, 0);
    break;
default : // "noir"
    analogWrite(led_rouge, 255);
    analogWrite(led_verte, 255);
    analogWrite(led_bleue, 255);
    break;
}
}

```

Code : Exercice, affichage de plusieurs couleurs

Bon ben je vous laisse lire le code tout seul, vous êtes assez préparé pour le faire, du moins j’espère. Pendant ce temps je vais continuer la rédaction de ce chapitre. :-°

5.3.3 Transformation PWM -> signal analogique

Bon, on est arrivé à modifier les couleurs d’une LED RGB juste avec des “impulsions”, plus exactement en utilisant directement le signal PWM.

[[q]] | Mais comment faire si je veux un signal complètement analogique ?

C’est justement l’objet de cette sous-partie : créer un signal analogique à partir d’un signal numérique.

[[a]] | Cependant, avant de continuer, je tiens à vous informer que l’on va aborder des notions plus profondes en électronique et que vous n’êtes pas obligé de lire cette sous-partie si vous ne vous en sentez pas capable. Revenez plus tard si vous le voulez. Pour ceux qui cela intéresserait vraiment, je ne peux que vous encourager à vous accrocher et éventuellement lire [ce chapitre](#) pour mieux comprendre certains points essentiels utilisés dans cette sous-partie.

5.3.3.1 La valeur moyenne d’un signal

Sur une période d’un signal périodique, on peut calculer sa valeur moyenne. En fait, il faut faire une moyenne de toutes les valeurs que prend le signal pendant ce temps donné. C’est une peu lorsque l’on fait la moyenne des notes des élèves dans une classe, on additionne toutes les notes et on divise le résultat par le nombre total de notes. Je ne vais prendre qu’un seul exemple, celui dont nous avons besoin : le signal carré.

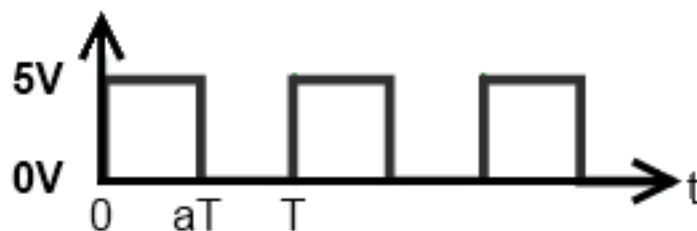


Figure 5.24 - Un signal carré

5.3.3.1.1 Le signal carré

Reprenons notre signal carré :

J'ai modifié un peu l'image pour vous faire apparaître les temps. On observe donc que du temps 0 (l'origine) au temps T , on a une période du signal. aT correspond au moment où le signal change d'état. En somme, il s'agit du temps de l'état haut, qui donne aussi le temps à l'état bas et finalement permet de calculer le rapport cyclique du signal. Donnons quelques valeurs numériques à titre d'exemple :

- $T = 1ms$
- $a = 0.5$ (correspond à un rapport cyclique de 50%)

La formule permettant de calculer la valeur moyenne de cette période est la suivante :

$$\langle U \rangle = \frac{U_1 \times aT + U_2 \times (T - aT)}{T}$$

[[i]] | La valeur moyenne d'un signal se note avec des chevrons \langle, \rangle autour de la lettre indiquant de quelle grandeur physique il s'agit.

5.3.3.1.2 Explications Premièrement dans la formule, on calcule la tension du signal sur la première partie de la période, donc de 0 à aT . Pour ce faire, on multiplie U_1 , qui est la tension du signal pendant cette période, par le temps de la première partie de la période, soit aT . Ce qui donne : $U_1 \times aT$. Deuxièmement, on fait de même avec la deuxième partie du signal. On multiplie le temps de ce bout de période par la tension U_2 pendant ce temps. Ce temps vaut $T - aT$. Le résultat donne alors : $U_2 \times (T - aT)$ Finalement, on divise le tout par le temps total de la période après avoir additionné les deux résultats précédents. Après simplification, la formule devient : $\langle U \rangle = a \times U_1 + U_2 - a \times U_2$ Et cela se simplifie encore en :

$$\langle U \rangle = a \times (U_1 - U_2) + U_2$$

[[i]] | Dans notre cas, comme il s'agit d'un signal carré ayant que deux valeurs : 0V et 5V, on va pouvoir simplifier le calcul par celui-ci : $\langle U \rangle = a \times U_1$, car $U_2 = 0$

[[e]] | Les formules que l'on vient d'apprendre ne s'appliquent que pour **une seule** période du signal. Si le signal a toujours la même période et le même rapport cyclique alors le résultat de la formule est admissible à l'ensemble du signal. En revanche, si le signal a un rapport cyclique qui varie au cours du temps, alors le résultat donné par la formule n'est valable que pour un rapport cyclique donné. Il faudra donc calculer la valeur moyenne pour chaque rapport cyclique que possède le signal.

De ce fait, si on modifie le rapport cyclique de la PWM de façon maîtrisée, on va pouvoir créer un signal analogique de la forme qu'on le souhaite, compris entre 0 et 5V, en extrayant la valeur moyenne du signal. On retiendra également que, **dans cette formule uniquement**, le temps n'a pas d'importance.

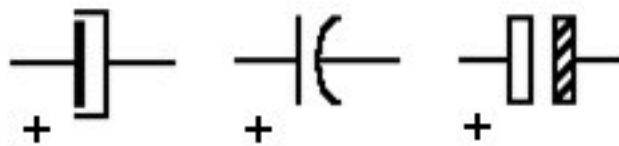
5.3.3.2 Extraire cette valeur moyenne

Alors, mais comment faire pour extraire la valeur moyenne du signal de la PWM, me direz-vous. Eh bien on va utiliser les propriétés d'un certain couple de composants très connu : le **couple RC** ou **résistance-condensateur**.

[[q]] | La résistance on connaît, mais, le condensateur... tu nous avais pas dit qu'il servait à supprimer les parasites ? o_O

Si, bien sûr, mais il possède plein de caractéristiques intéressantes. C'est pour cela que c'est un des composants les plus utilisés en électronique. Cette fois, je vais vous montrer une de ses caractéristiques qui va nous permettre d'extraire cette fameuse valeur moyenne.

5.3.3.2.1 Le condensateur Je vous ai déjà parlé de la résistance, vous savez qu'elle limite le courant suivant la loi d'Ohm. Je vous ai aussi parlé du condensateur, je vous disais qu'il absorbait les parasites créés lors d'un appui sur un bouton poussoir. À présent, on va voir un peu plus en profondeur son fonctionnement car on est loin d'avoir tout vu ! Le condensateur, je rappelle ses



symboles :

est constitué de deux plaques métalliques, des **armatures**, posées face à face et isolées par... un isolant ! :P Donc, en somme le condensateur **est équivalent** à un interrupteur ouvert puisqu'il n'y a pas de courant qui peut passer entre les deux armatures. Chaque armature sera mise à un potentiel électrique. Il peut être égal sur les deux armatures, mais l'utilisation majoritaire fait que les deux armatures ont un potentiel différent.

5.3.3.2.2 Le couple RC Bon, et maintenant ? Maintenant on va faire un petit montage électrique, vous pouvez le faire si vous voulez, non en fait faites-le vous comprendrez mes explications en même temps que vous ferez l'expérience qui va suivre. Voilà le montage à réaliser :

Les valeurs des composants sont :

- $U = 5V$ (utilisez la tension 5V fournie par votre carte Arduino)
- $C = 1000\mu F$
- $R_{charge} = 1k\Omega$
- $R_{decharge} = 1k\Omega$

Le montage est terminé ? Alors fermez l'interrupteur...

[[q]] | Que se passe-t-il ?

Lorsque vous fermez l'interrupteur, le courant peut s'établir dans le circuit. Il va donc aller allumer la LED. Ceci fait abstraction du condensateur. Mais, justement, dans ce montage il y a

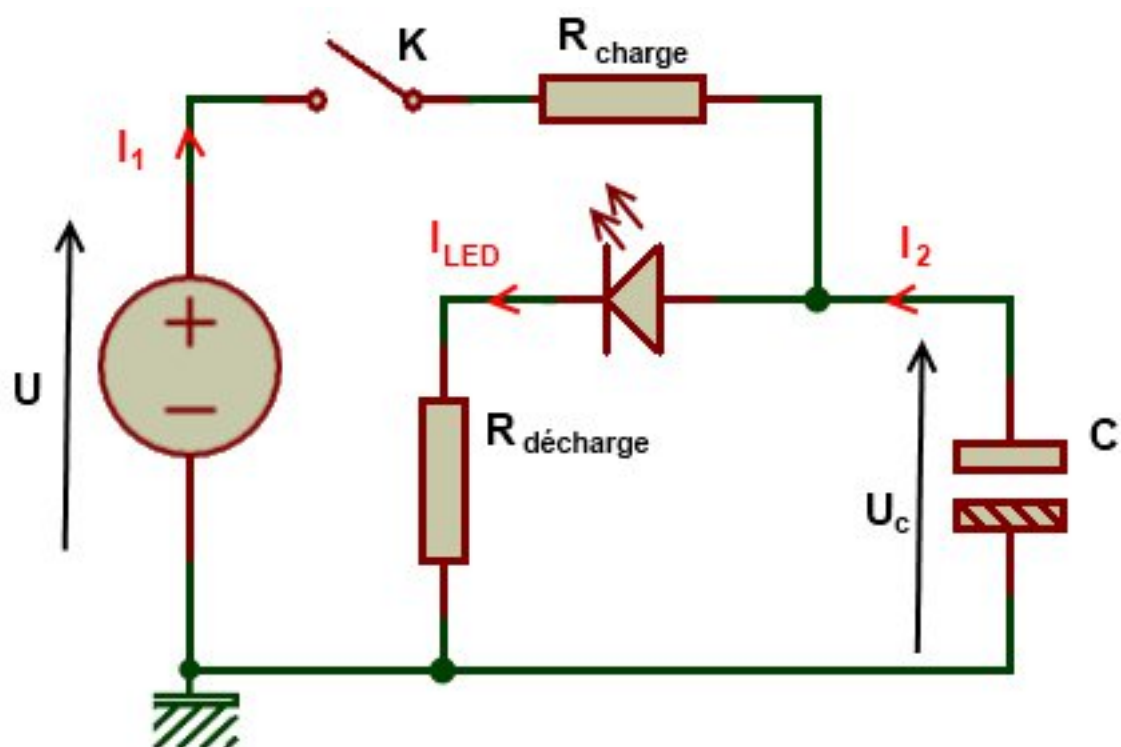


Figure 5.25 – Un couple RC en action

un condensateur. Qu'observez-vous ? La LED ne s'allume pas immédiatement et met un peu de temps avant d'être complètement allumée. Ouvrez l'interrupteur. Et là, qu'y a-t-il de nouveau ? En théorie, la LED devrait être éteinte, cependant, le condensateur fait des siennes. On voit la LED s'éteindre tout doucement et pendant plus longtemps que lorsqu'elle s'allumait. Troublant, n'est-ce pas ? :D

[[i]] | Vous pouvez réitérer l'expérience en changeant la valeur des composants, sans jamais descendre en dessous de 220 Ohm pour la résistance de décharge.

5.3.3.2.3 Explications Je vais vous expliquer ce phénomène assez étrange. Vous l'aurez sans doute deviné, c'est le condensateur qui joue le premier rôle ! En fait, lorsque l'on applique un potentiel différent sur chaque armature, le condensateur n'aime pas trop ça. Je ne dis pas que ça risque de l'endommager, simplement qu'il n'aime pas ça, comme si vous on vous forçait à manger quelque chose que vous n'aimez pas. Du coup, lorsqu'on lui applique une tension de 5V sur une des ses armatures et l'autre armature est reliée à la masse, il met du temps à accepter la tension. Et plus la tension croît, moins il aime ça et plus il met du temps à l'accepter. Si on regarde la tension aux bornes de ce pauvre condensateur, on peut observer ceci :

La tension augmente d'abord très rapidement, puis de moins en moins rapidement aux bornes du condensateur lorsqu'on le **charge** à travers une résistance. Oui, on appelle ça la **charge** du condensateur. C'est un peu comme si la résistance donnait un mauvais goût à la tension et plus la résistance est grande, plus le goût est horrible et moins le condensateur se charge vite. C'est l'explication de pourquoi la LED s'est éclairée lentement. Lorsque l'on ouvre l'interrupteur, il se passe le phénomène inverse. Là, le condensateur peut se débarrasser de ce mauvais goût qu'il a accumulé, sauf que la résistance et la LED l'en empêchent. Il met donc du temps à se **décharger** et la LED s'éteint doucement :

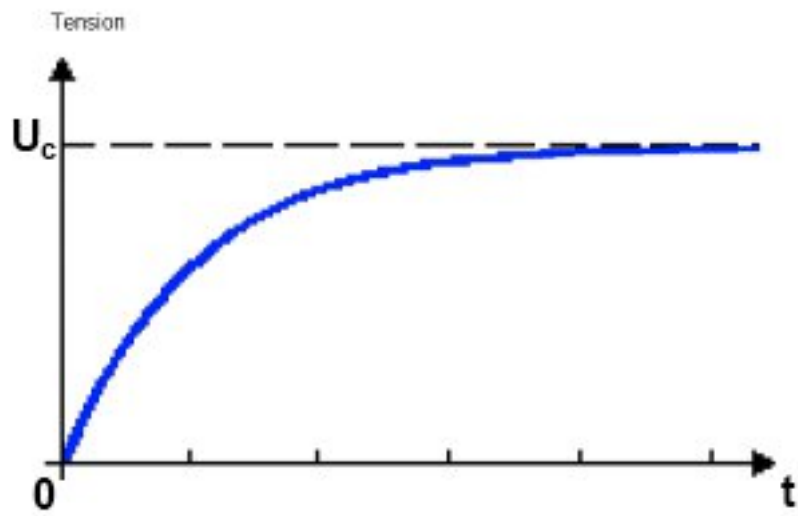


Figure 5.26 - La courbe de charge d'un condensateur

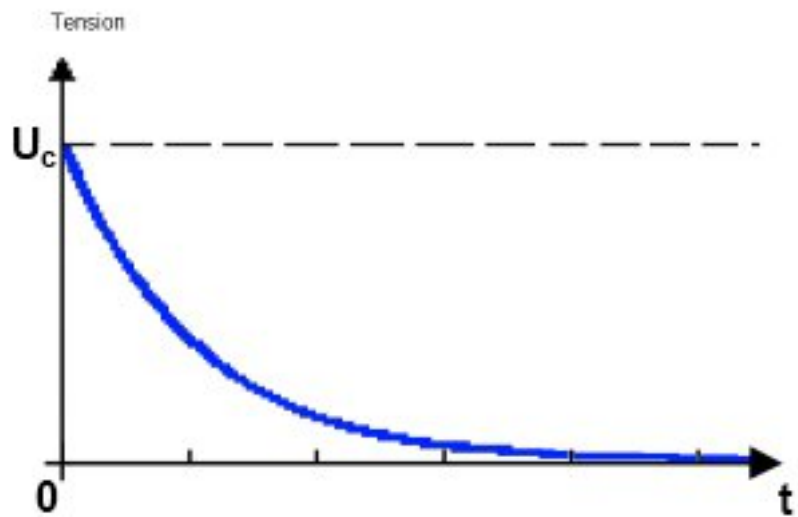


Figure 5.27 - La courbe de décharge d'un condensateur

Pour terminer, on peut déterminer le temps de charge et de décharge du condensateur à partir d'un paramètre très simple, que voici :

$$\tau = R \times C$$

Avec :

- τ : (prononcez "to") temps de charge/décharge en secondes (s)
- R : valeur de la résistance en Ohm (Ω)
- C : valeur de la capacité du condensateur en Farad (F)

Cette formule donne le temps τ qui correspond à 63% de la charge à la tension appliquée au condensateur. On considère que le condensateur est complètement chargé à partir de 3τ (soit 95% de la tension de charge) ou 5τ (99% de la tension de charge).

5.3.3.2.4 Imposons notre PWM! [[q]] | Bon, très bien, mais quel est le rapport avec la PWM?

Ha, haa! Alors, pour commencer, vous connaissez la réponse.

[[q]] | Depuis quand? o_O

Depuis que je vous ai donné les explications précédentes. Dès que l'on aura imposé notre PWM au couple RC, il va se passer quelque chose. Quelque chose que je viens de vous expliquer. À chaque fois que le signal de la PWM sera au NL 1 (Niveau Logique 1), le condensateur va se charger. Dès que le signal repasse au NL 0, le condensateur va se décharger. Et ainsi de suite. En somme, cela donne une variation de tension aux bornes du condensateur semblable à celle-ci :

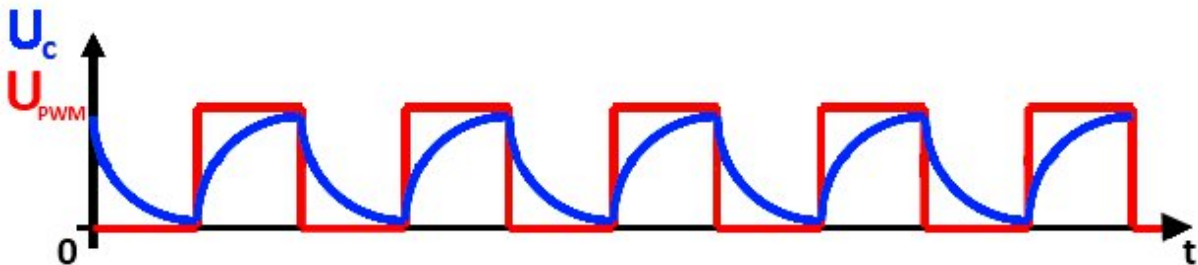


Figure 5.28 – Charge et décharge successive

[[q]] | Qu'y a-t-il de nouveau par rapport au signal carré, à part sa forme bizarroïde!?

Dans ce cas, rien de plus, si on calcule la valeur moyenne du signal bleu, on trouvera la même valeur que pour le signal rouge. (Ne me demandez pas pourquoi, c'est comme ça, c'est une formule très compliquée qui le dit :P). Précisons que dans ce cas, encore une fois, le temps de charge/décharge 3τ du condensateur est choisi de façon à ce qu'il soit égal à une demi-période du signal. Que se passera-t-il si on choisit un temps de charge/décharge plus petit ou plus grand?

5.3.3.2.5 Constante de temps τ supérieure à la période Voilà le chronogramme lorsque la constante de temps de charge/décharge du condensateur est plus grande que la période du signal :

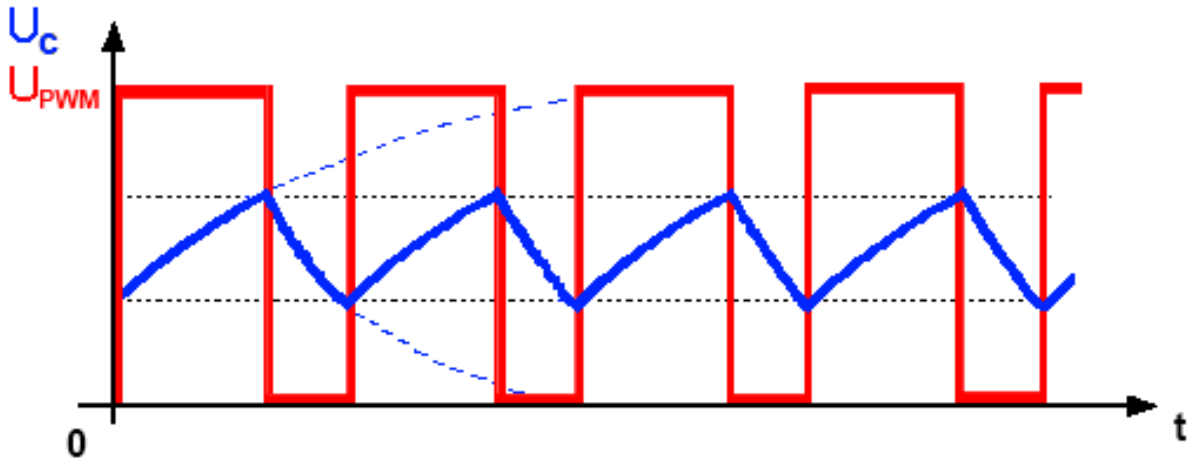


Figure 5.29 – Charge et décharge successive

Ce chronogramme permet d’observer un phénomène intéressant. En effet, on voit que la tension aux bornes du condensateur n’atteint plus le +5V et le 0V comme au chronogramme précédent. Le couple RC étant plus grand que précédemment, le condensateur met plus de temps à se charger, du coup, comme le signal “va plus vite” que le condensateur, ce dernier ne peut se charger/décharger complètement. Si on continue d’augmenter la valeur résultante du couple RC, on va arriver à un signal comme ceci :

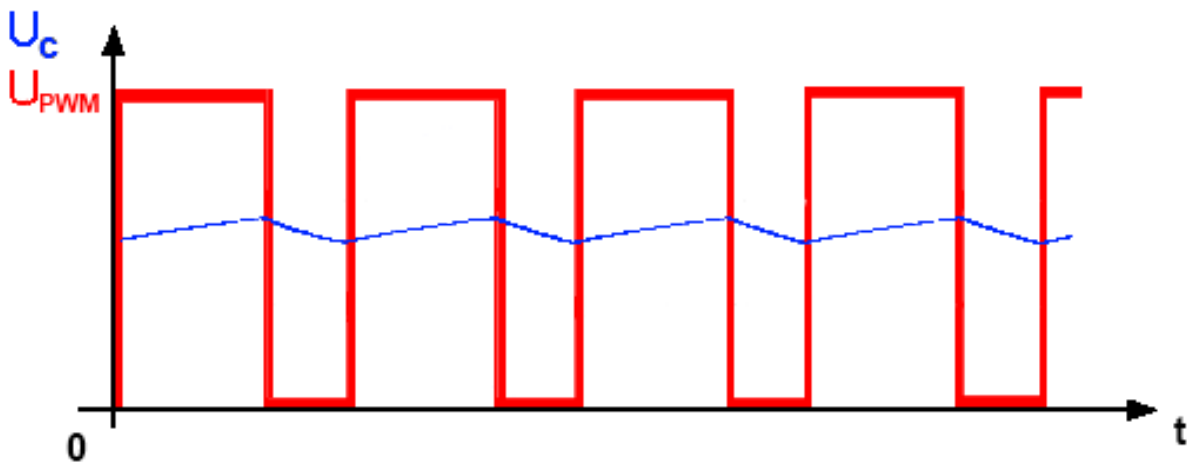


Figure 5.30 – Charge et décharge successive

Et ce signal, Mesdames et Messieurs, c’est la valeur moyenne du signal de la PWM!! :D

5.3.3.3 Calibrer correctement la constante RC

Je vous sens venir avec vos grands airs en me disant : “Oui, mais là le signal il est pas du tout constant pour un niveau de tension. Il arrête pas de bouger et monter descendre ! Comment on fait si on veut une belle droite ?” “Eh bien, dirais-je, cela n’est pas impossible, mais se révèle être une tâche difficile et contraignante. Plusieurs arguments viennent conforter mes dires”.

5.3.3.3.1 Le temps de stabilisation entre deux paliers Je vais vous montrer un chronogramme qui représente le signal PWM avec deux rapports cycliques différents. Vous allez pouvoir observer un phénomène “qui se cache” :

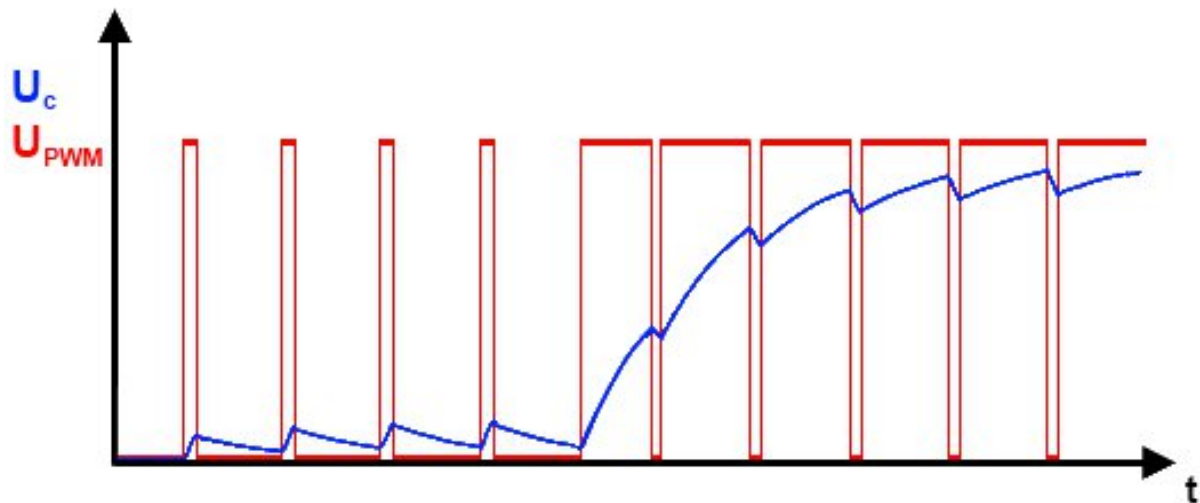


Figure 5.31 – Charge et décharge successive

Voyez donc ce fameux chronogramme. Qu’en pensez-vous ? Ce n’est pas merveilleux hein ! :(Quelques explications : pour passer d’un palier à un autre, le condensateur met un certain temps. Ce temps est grosso modo celui de son temps de charge (constante RC). C’est-à-dire que plus on va augmenter le temps de charge, plus le condensateur mettra du temps pour se stabiliser au palier voulu. Or si l’on veut créer un signal analogique qui varie assez rapidement, cela va nous poser problème.

5.3.3.3.2 La perte de temps en conversion C’est ce que je viens d’énoncer, plus la constante de temps est grande, plus il faudra de périodes de PWM pour stabiliser la valeur moyenne du signal à la tension souhaitée. À l’inverse, si on diminue la constante de temps, changer de palier sera plus rapide, mais la tension aux bornes du condensateur aura tendance à suivre le signal. C’est le premier chronogramme que l’on a vu plus haut.

5.3.3.3.3 Finalement, comment calibrer correctement la constante RC ? Cela s’avère être délicat. Il faut trouver le juste milieu en fonction du besoin que l’on a.

- Si l’on veut un signal qui soit le plus proche possible de la valeur moyenne, il faut une constante de temps très grande.
- Si au contraire on veut un signal qui soit le plus rapide et que la valeur moyenne soit une approximation, alors il faut une constante de temps faible.
- Si on veut un signal rapide et le plus proche possible de la valeur moyenne, on a deux solutions qui sont :
 - mettre un deuxième montage ayant une constante de temps un peu plus grande, en cascade du premier (on perd quand même en rapidité)
 - changer la fréquence de la PWM

5.3.4 Modifier la fréquence de la PWM

On l’a vu, avec la fréquence actuelle de la PWM en sortie de l’Arduino, on va ne pouvoir créer que des signaux “lents”. Lorsque vous aurez besoin d’aller plus vite, vous vous confronterez à ce problème. C’est pourquoi je vais vous expliquer comment modifier la fréquence de cette PWM.

[[e]] | Nouveau message d’avertissement : cette fois, on va directement toucher aux registres du microcontrôleur, donc si vous comprenez pas tout, ce n’est pas très grave car cela requiert un niveau encore plus élevé que celui que vous avez actuellement.

Commençons cette très courte sous-partie.

5.3.4.0.1 Pourquoi changer la fréquence de la PWM? Oui, pourquoi? Tout simplement pour essayer de créer un signal qui se rapproche le plus de la valeur moyenne de la PWM à chaque instant. L’objectif est de pouvoir maximiser l’avantage de la structure ayant une faible constante de temps tout en éliminant au mieux son désavantage. Vous verrez peut-être mieux avec des chronogrammes. En voici deux, le premier est celui où la fréquence de la PWM est celle fournie d’origine par l’Arduino, le second est la PWM à une fréquence deux fois plus élevée, après modification du programme :

Pour une constante de temps identique pour chaque courbe réalisée, on relève que le temps de stabilisation du signal est plus rapide sur le chronogramme où la fréquence est deux fois plus élevée qu’avec la fréquence standard d’Arduino. Ici donc, on a : $t_2 - t_1 = 2 \times (t_4 - t_3)$. En effet car le temps (T) est inversement proportionnel à la fréquence (F) selon cette formule : $F = \frac{1}{T}$. Avec quelques mots pour expliquer cela, le temps de charge du condensateur, pour se stabiliser au nouveau palier de tension, est plus rapide avec une fréquence plus élevée. À comparaison, pour le premier signal, le temps de charge est deux fois plus grand que celui pour le deuxième signal où la fréquence est deux fois plus élevée.

[[a]] | Mes dessins ne sont pas très rigoureux, mais mes talents de graphistes me limitent à ça. Soyez indulgent à mon égard. :-° Quoi qu’il en soit, il s’agissait, ici, simplement d’illustrer mes propos et donner un exemple concret.

5.3.4.0.2 Utilisation du registre [[q]] | Bigre! Je viens de comprendre pourquoi on avait besoin de changer la fréquence de la PWM. :D Mais euh... comment on fait? C’est quoi les registres? :-°

Les registres..... eh bien..... c’est compliqué! :ninja : Non, je n’entrerai pas dans le détail en expliquant ce qu’est un registre, de plus c’est un sujet que je ne maîtrise pas bien et qui vous sera certainement inutile dans le cas présent. Disons pour l’instant que le registre est une variable très spéciale.

5.3.4.0.3 Code de modification de la fréquence de la PWM Alors, pour modifier la fréquence de la PWM de l’Arduino on doit utiliser le code suivant :

```
// on définit une variable de type byte
// qui contiendra l'octet à donner au registre pour diviser la fréquence de la PWM

// division par : 1, 8, 64, 256, 1024
byte division_frequence=0x01;
// fréquence : 62500Hz, 7692Hz, ...
```

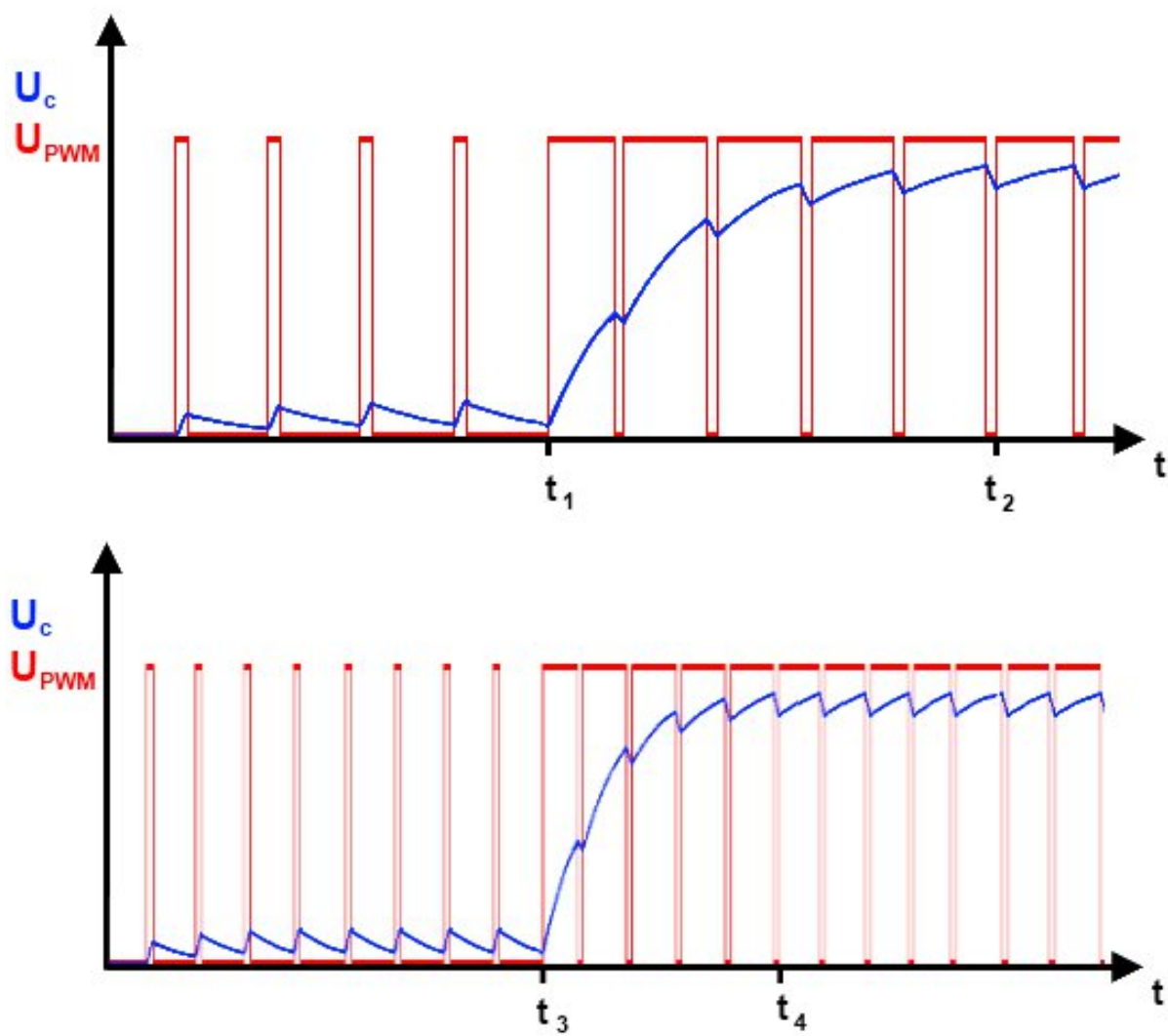


Figure 5.32 – L'impact de la modification de la PWM

```
// temps de la période : 16µs, 130µs, ...

void setup()
{
  pinMode(6, OUTPUT); // broche de sortie

  // TCCR0B c'est le registre, on opère un masquage sur lui même
  TCCR0B = TCCR0B & 0b11111000 | division_frequence;
  // ce qui permet de modifier la fréquence de la PWM
}

void loop ()
{
  // on écrit simplement la valeur de 0 à 255 du rapport cyclique du signal
  analogWrite(6, 128);
  // qui est à la nouvelle fréquence choisit
}
```

Code : Modification de la fréquence de la PWM

[[i]] | Vous remarquerez que les nombres binaires avec Arduino s'écrivent avec les caractères **ob** devant.

[[a]] | Cette sous-partie peut éventuellement être prise pour un *truc et astuce*. C'est quelque peu le cas, malheureusement, mais pour éviter que cela ne le soit complètement, je vais vous expliquer des notions supplémentaires, par exemple la ligne 14 du code.

5.3.4.0.4 Traduction s'il vous plait ! Je le disais donc, on va voir ensemble comment fonctionne la ligne 14 du programme :

```
cpp linenostart=14 TCCR0B = TCCR0B & 0b11111000 | division_frequence;
```

Très simplement, TCCR0B est le nom du registre utilisé. Cette ligne est donc là pour modifier le registre puisqu'on fait une opération avec et le résultat est inscrit dans le registre. Cette opération est, il faudra l'avouer, peu commune. On effectue, ce que l'on appelle en programmation, un **masquage**. Le masquage est une opération logique que l'on utilise avec des données binaires. On peut faire différents masquages en utilisant les opérations logiques ET, OU, OU exclusif, ... Dans le cas présent, on a la variable TCCR0B qui est sous forme binaire et on effectue une opération **ET** (symbole **&**) puis une opération **OU** (symbole **|**). Les opérations ET et OU sont définies par une **table de vérité**. Avec deux entrées, on a une sortie qui donne le résultat de l'opération effectuée avec les deux niveaux d'entrée.

->

A	B	A ET B	A OU B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Table : Table de vérité de l'opérateur ET et OU

<-

Les opérations de type ET et OU ont un niveau de priorité comme la multiplication et l'addition. On commence toujours par effectuer l'opération ET, puis on termine avec l'opération OU. On pourrait donc écrire ceci :

```
TCCR0B = (TCCR0B & 0b11111000) | division_frequence;
```

Prenons maintenant un exemple où la variable spéciale TCCR0B serait un nombre binaire égal à :

```
TCCR0B = 0b10011101; // valeur du registre
```

À présent, on lui fait un masquage de type ET :

```
TCCR0B = 0b10011101;
```

```
TCCR0B = TCCR0B & 0b11111000; // masquage de type ET
```

On fait l'opération à l'aide de la table de vérité du ET (voir tableau ci-dessus) et on trouve le résultat :

```
TCCR0B = 0b10011101;
```

```
TCCR0B = TCCR0B & 0b11111000;
```

```
// TCCR0B vaut maintenant : 0b10011000
```

En somme, on conclut que l'on a gardé la valeur des 5 premiers bits, mais l'on a effacé la valeur des 3 derniers bits pour les mettre à zéro. Ainsi, quelle que soit la valeur binaire de TCCR0B, on met les bits que l'on veut à l'état bas. Ceci va ensuite nous permettre de changer l'état des bits mis à l'état bas en effectuant l'opération OU :

```
byte division_frequence = 0x01; // nombre hexadécimal qui vaut 0b00000001
```

```
TCCR0B = 0b10011101;
```

```
TCCR0B = TCCR0B & 0b11111000;  
// TCCR0B vaut maintenant : 0b10011000
```

```
TCCR0B = TCCR0B | division_frequence;  
// TCCR0B vaut maintenant : 0b10011001
```

D'après la table de vérité du OU logique, on a modifié la valeur de TCCR0B en ne changeant que le ou les bits que l'on voulait.

[[e]] | La valeur de TCCR0B que je vous ai donnée est bidon. C'est un exemple qui vous permet de comprendre comment fonctionne un masquage.

Ce qu'il faut retenir, pour changer la fréquence de la PWM, c'est que pour la variable `division_frequence`, il faut lui donner les valeurs hexadécimales suivantes :

```
0x01 // la fréquence vaut 62500Hz (fréquence maximale fournie par la PWM => provient d  
// effectue une division par 1 de la fréquence max
```

```
0x02 // f = 7692Hz (division par 8 de la fréquence maximale)
```

```
0x03 // f = 976Hz, division par 64
```

```
0x04 // f = 244Hz, division par 256
0x05 // f = 61Hz, division par 1024
```

[[i]] | Vous trouverez plus de détails sur [cette page](#) (en anglais).

5.3.4.0.5 Test de vérification Pour vérifier que la fréquence a bien changé, on peut reprendre le montage que l’on a fait plus haut et enlever l’interrupteur en le remplaçant par un fil. On ne met plus un générateur de tension continue, mais on branche une sortie PWM de l’arduino avec le programme qui va bien. Pour deux fréquences différentes, on devrait voir la LED s’allumer plus ou moins rapidement. On compare le temps à l’état au lorsque l’on écrit 1000 fois un niveau de PWM à 255 à celui mis par le même programme avec une fréquence de PWM différente, grâce à une LED.

À partir de maintenant, vous allez pouvoir faire des choses amusantes avec la PWM. Cela va nous servir pour les moteurs pour ne citer qu’eux. Mais avant, car on en est pas encore là, je vous propose un petit TP assez sympa. Rendez-vous au prochain chapitre ! :)

5.4 [Exercice] Une animation “YouTube”

Dans ce petit exercice, je vous propose de faire une animation que vous avez tous vu au moins une fois dans votre vie : le .gif de chargement YouTube ! Pour ceux qui se posent des questions, nous n’allons pas faire de Photoshop ou quoi que ce soit de ce genre. Non, nous (vous en fait ;)) allons le faire ... avec des LED ! Alors place à l’exercice !

5.4.1 Énoncé

Pour clôturer votre apprentissage avec les voies analogiques, nous allons faire un petit exercice pour se détendre. Le but de ce dernier est de réaliser une des animations les plus célèbres de l’internet : le .gif de chargement YouTube (qui est aussi utilisé sur d’autres plateformes et applications).

Nous allons le réaliser avec des LED et faire varier la vitesse de défilement grâce à un potentiomètre. Pour une fois, plutôt qu’une longue explication je vais juste vous donner une liste de composants utiles et une vidéo qui parle d’elle même !

-> **Bon courage!** <-

- 6 LED + leurs résistances de limitation de courant
- Un potentiomètre
- Une Arduino, une breadboard et des fils !

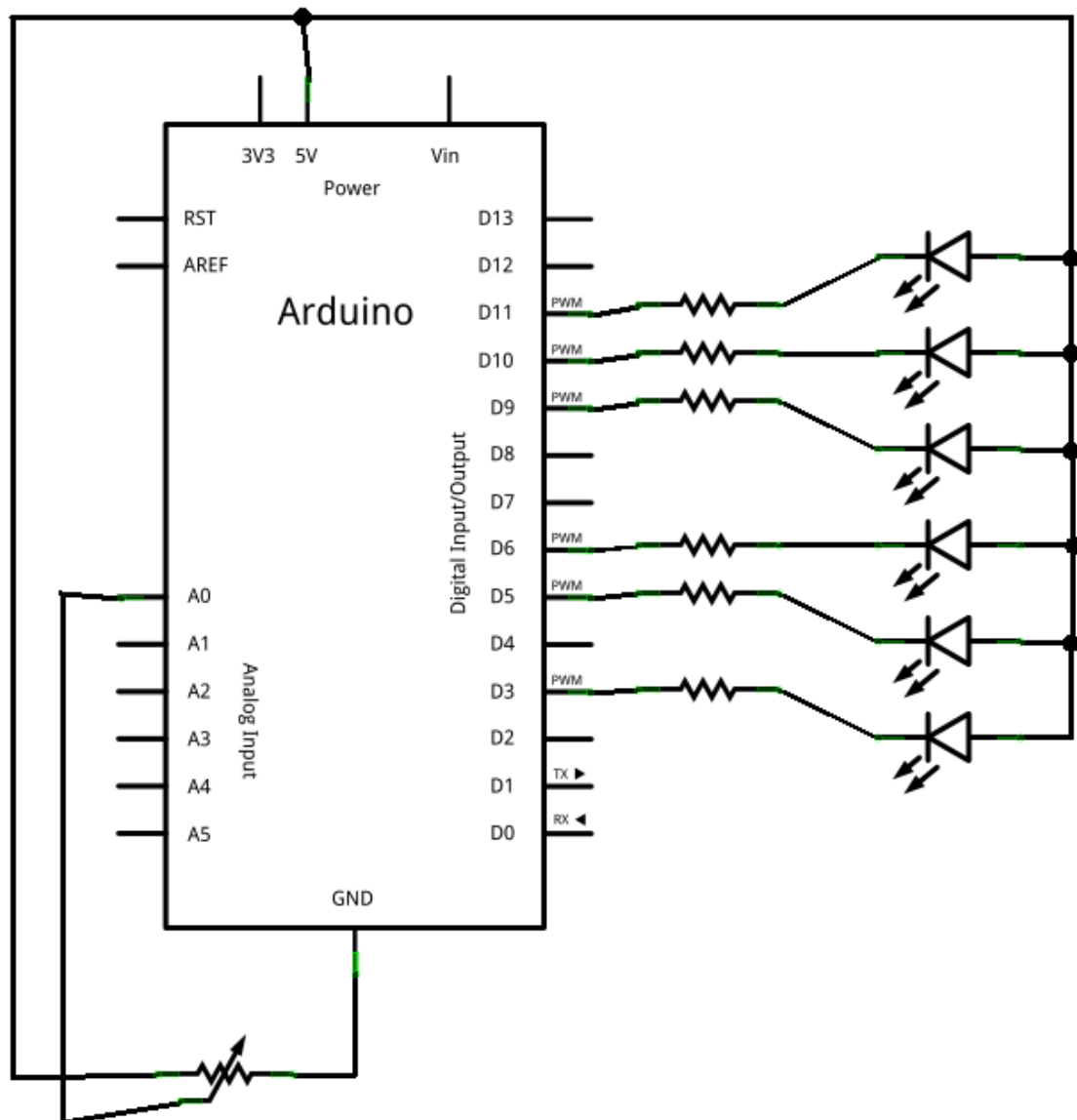
->!(<https://www.youtube.com/watch?v=SyxXHZzCoHY>)<-

5.4.2 Solution

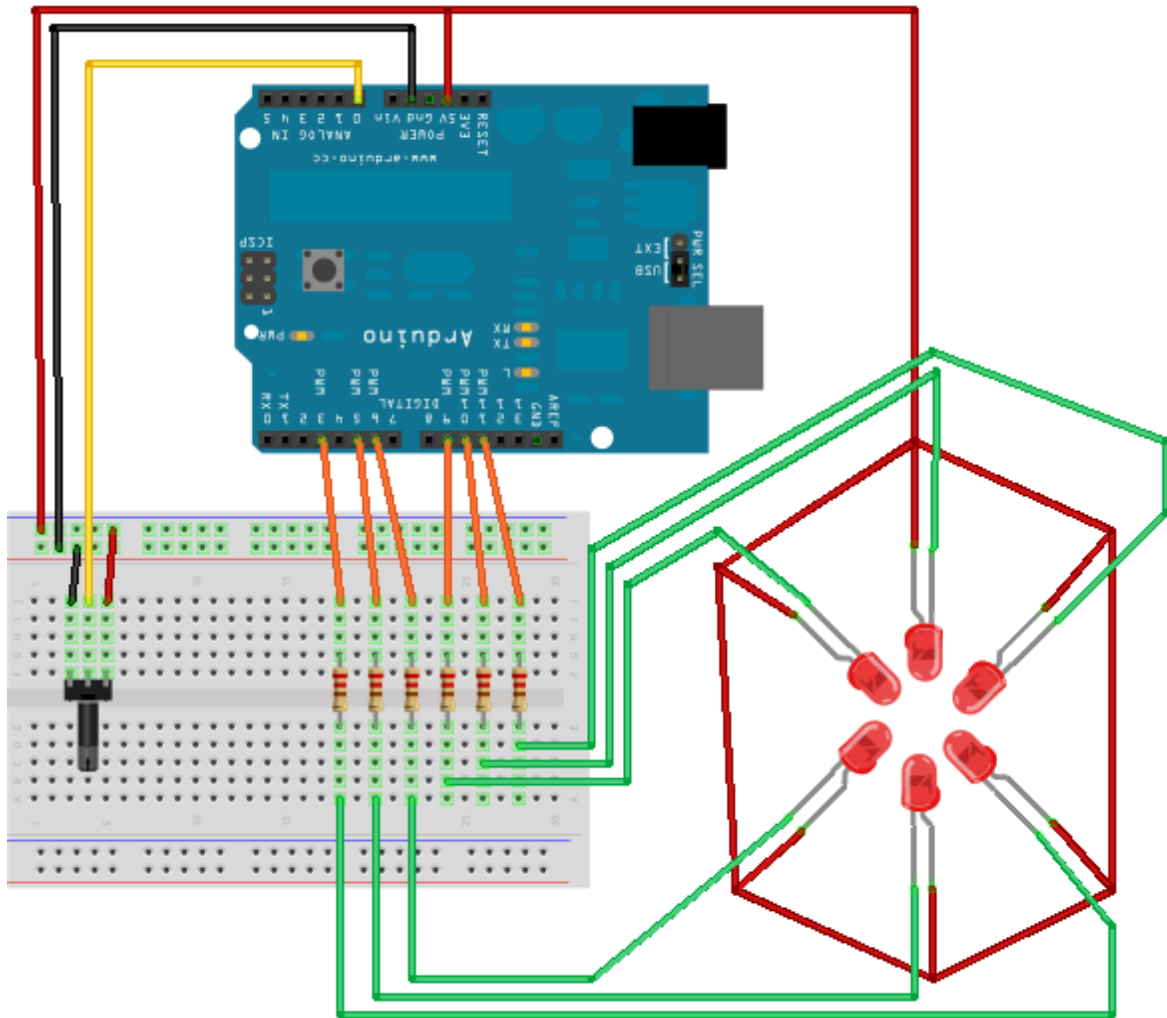
5.4.2.1 Le schéma

Voici tout d'abord le schéma, car une bonne base électronique permettra de faire un beau code ensuite. Pour tout les lecteurs qui ne pensent qu'aux circuits et ne regardent jamais la version "photo" du montage, je vous invite pour une fois à y faire attention, surtout pour l'aspect géométrique du placement des LED. En passant, dans l'optique de faire varier la luminosité des LED, il faudra les connecter sur les broches PWM (notées avec un '~').

Le potentiomètre quant à lui sera bien entendu connecté à une entrée analogique (la 0 dans mon cas). Comme toujours, les LED auront leur anode reliées au +5V et seront pilotées par état bas (important de le rappeler pour le code ensuite).



[[s]] |



||

5.4.2.2 Le code

Alors petit défi avant de regarder la solution... En combien de ligne avez vous réussi à écrire votre code (proprement, sans tout mettre sur une seule ligne, pas de triche!) ? Personnellement je l’ai fait en 23 lignes, en faisant des beaux espaces propres. ;) Bon allez, trêve de plaisanterie, voici la solution, comme à l’accoutumé dans des balises secrètes...

5.4.2.2.1 Les variables globales Comme vous devez vous en douter, nous allons commencer par déclarer les différentes broches que nous allons utiliser. Il nous en faut six pour les LED et une pour le potentiomètre de réglage de la vitesse d’animation. Pour des fins de simplicité dans le code, j’ai mis les six sorties dans un tableau. Pour d’autres fins de facilité, j’ai aussi mis les “niveaux” de luminosité dans un tableau de char que j’appellerai “pwm”. Dans la balise suivante vous trouverez l’ensemble de ces données :

```
[[s]]|cpp | // sortie LEDs | const int LED[6] = {3,5,6,9,10,11}; | // niveaux
de luminosité utilisé | const char pwm[6] = {255,210,160,200,220,240}; |
// potentiometre sur la broche 0 | const int potar = 0; |
```

5.4.2.2.2 Le setup Personne ne devrait se tromper dans cette fonction, on est dans le domaine du connu, vu et revu! Il nous suffit juste de mettre en entrée le potentiomètre sur son convertisseur analogique et en sortie mettre les LED (une simple boucle for suffit grace au tableau;)).

```
[[s]]|cpp | void setup() | { | // le potentiomètre en entrée | pinMode(potar,
INPUT); | // les LEDs en sorties | for(int i=0; i<6; i++) | pinMode(LED[i],
OUTPUT); | } |
```

5.4.2.2.3 La loop Passons au cœur du programme, la boucle loop() ! Je vais vous la divulguer dès maintenant puis l'expliquer ensuite :

```
[[s]] | cpp | void loop() | { | // étape de l'animation | for(int i=0; i<6;
i++) | { | // mise à jour des LEDs | for(int n=0; n<6; n++) | { | analogWrite(LED[n],
pwm[(n+i)%6]); | } | // récupère le temps | int temps = analogRead(potar);
| // tmax = 190ms, tmin = 20ms | delay(temps/6 + 20); | } | } |
```

Comme vous pouvez le constater, cette fonction se contente de faire deux boucle. L'une sert à mettre à jour les "phases de mouvements" et l'autre met à jour les PWM sur chacune des LED.

5.4.2.2.4 Les étapes de l'animation Comme expliqué précédemment, la première boucle concerne les différentes phases de l'animation. Comme nous avons six LED nous avons six niveaux de luminosité et donc six étapes à appliquer (chaque LED prenant successivement chaque niveau). Nous verrons la seconde boucle après.

Avant de passer à la phase d'animation suivante, nous faisons une petite pause. La durée de cette pause détermine la vitesse de l'animation. Comme demandé dans le cahier des charges, cette durée sera réglable à l'aide d'un potentiomètre. La ligne 9 nous permet donc de récupérer la valeur lue sur l'entrée analogique. Pour rappel, elle variera de 0 à 1023.

Si l'on applique cette valeur directement au délai, nous aurions une animation pouvant aller de très très très rapide (potar au minimum) à très très très lent (delay de 1023 ms) lorsque le potar est dans l'autre sens.

Afin d'obtenir un réglage plus sympa, on fait une petite opération sur cette valeur. Pour ma part j'ai décidé de la diviser par 6, ce qui donne $0ms \leq temps \leq 170ms$. Estimant que 0 ne permet pas de faire une animation (puisque'on passerait directement à l'étape suivante sans attendre), j'ajoute 20 à ce résultat. Le temps final sera donc compris dans l'intervalle : $20ms \leq temps \leq 190ms$.

5.4.2.2.5 Mise à jour des LED La deuxième boucle possède une seule ligne qui est la clé de toute l'animation !

Cette boucle sert à mettre à jour les LED pour qu'elles aient toute la bonne luminosité. Pour cela, on utilisera la fonction analogWrite() (car après tout c'est le but du chapitre!). Le premier paramètre sera le numéro de la LED (grâce une fois de plus au tableau) et le second sera la valeur du PWM. C'est pour cette valeur que toute l'astuce survient.

En effet, j'utilise une opération mathématique un peu particulière que l'on appelle **modulo**. Pour ceux qui ne se rappelle pas de ce dernier, nous l'avons vu il y a très longtemps dans la première partie, deuxième chapitres sur [les variables](#). Cet opérateur permet de donner le résultat de la

division euclidienne (mais je vous laisse aller voir le cours pour plus de détail). Pour obtenir la bonne valeur de luminosité il me faut lire la bonne case du tableau `pwm[]`.

Ayant six niveaux de luminosité, j'ai six case dans mon tableau. Mais comment obtenir le bonne ? Eh bien simplement en additionnant le numéro de la LED en train d'être mise à jour (donné par la seconde boucle) et le numéro de l'étape de l'animation en cours (donné par la première boucle). Seulement imaginons que nous mettions à jour la sixième LED (indice 5) pour la quatrième étape (indice 3). Ça nous donne 8. Hors 8 est plus grand que 6 (L'index 5 est la sixième led). En utilisant le modulo, nous prenons le reste de la division de 8/6 soit 2. Il nous faudra donc utiliser la case numéro 2 du tableau `pwm[]` pour cette utilisation. *Tout simplement* :p

[[i]] | Je suis conscient que cette écriture n'est pas simple. Il est tout a fait normal de ne pas l'avoir trouvé et demande une certaine habitude de la programmation et ses astuces pour y penser.

Pour ceux qui se demande encore quel est l'intérêt d'utiliser des tableaux de données, voici deux éléments de réponse.

- Admettons j'utilise une Arduino Mega qui possède 15 pwm, j'aurais pu allumer 15 LEDs dans mon animation. Mais si j'avais fait mon setup de manière linéaire, il m'aurait fallu rajouter 9 lignes. Grâce au tableau, j'ai juste besoin de les ajouter à ce dernier et de modifier l'indice de fin pour l'initialisation dans la boucle `for`.
- La même remarque s'applique à l'animation. En modifiant simplement les tableaux je peux changer rapidement l'animation, ses niveaux de luminosité, le nombre de LEDs, l'ordre d'éclairage etc...

5.4.2.2.6 Le programme complet Et pour tout ceux qui doute du fonctionnement du programme, voici dès maintenant le code complet de la machine ! (Attention lorsque vous faites vos branchement à mettre les LED dans le bon ordre, sous peine d'avoir une séquence anarchique).

```
[[s]]|cpp | // sortie LEDs | const int LED[6] = {3,5,6,9,10,11}; | // niveaux
de luminosité utilisé | const char pwm[6] = {255,210,160,200,220,240}; |
// potentiometre sur la broche 0 | const int potar = 0; | | void setup()
| { | pinMode(potar, INPUT); | for(int i=0; i<6; i++) | pinMode(LED[i],
OUTPUT); | } | | void loop() | { | // étape de l'animation | for(int i=0;
i<6; i++) | { | // mise à jour des LEDs | for(int n=0; n<6; n++) | { |
analogWrite(LED[n], pwm[(n+i)%6]); | } | // récupère le temps | int temps
= analogRead(potar); | // tmax = 190ms, tmin = 20ms | delay(temps/6 + 20);
| } | } | }
```

La mise en bouche des applications possibles avec les entrées/sortie PWM est maintenant terminée. Je vous laisse réfléchir à ce que vous pourriez faire avec. Tenez, d'ailleurs les chapitres de la partie suivante utilisent ces entrées/sorties et ce n'est pas par hasard... ;)

6 Les capteurs et l'environnement autour d'Arduino

Savoir programmer c'est bien, mais créer des applications qui prennent en compte les événements de leur environnement, c'est mieux ! Cette partie va donc vous faire découvrir les capteurs couramment utilisés dans les systèmes embarqués.

Alors oui, le bouton poussoir et le potentiomètre peuvent, en quelque sorte, être définis comme étant des capteurs, mais vous allez voir qu'il en existe plein d'autres, chacun mesurant une grandeur physique distincte et plus ou moins utile selon l'application souhaitée.

6.1 Généralités sur les capteurs

Ce premier chapitre va vous présenter un peu ce que sont les capteurs, à quoi ils servent, où les trouver, etc. leur taille, leur forme et j'en passe. Je vais simplement vous faire découvrir le monde des capteurs qui, vous allez le voir, est merveilleux !

6.1.1 Capteur et Transducteur

Un capteur est un dispositif capable de transformer une grandeur physique (telle que la température, la pression, la lumière, etc.) en une autre grandeur physique manipulable. On peut d'ailleurs prendre des exemples : un microphone est un capteur qui permet de transformer une onde sonore en un signal électrique ; un autre capteur tel qu'une photorésistance permet de transformer un signal lumineux en résistance variable selon son intensité.

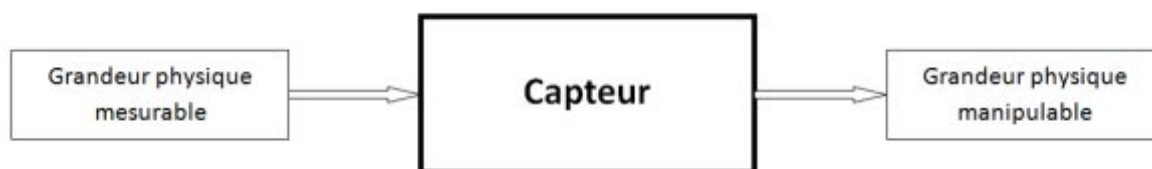


Figure 6.1 – Schéma d'un capteur

Pour nous, utiliser un thermomètre à mercure risque d'être difficile avec Arduino, car ce capteur ne délivre pas d'information électrique ou de résistance qui varie. Il s'agit seulement d'un niveau de liquide. Tandis qu'utiliser un microphone ou une photorésistance sera beaucoup plus facile. On distingue deux types de capteurs.

6.1.1.1 Capteurs Transducteurs passifs



Figure : Une photorésistance - (CC-0)

Ces capteurs ont pour objet de “transformer” ou plus exactement : *donner une image* de la grandeur physique qu’ils mesurent par une **résistance électrique variable** (en fait il s’agit d’une impédance, mais restons simples). Par exemple, le potentiomètre donne une résistance qui varie selon la position de son axe. Pour ce qui est de leur utilisation, il faudra nécessairement les utiliser avec un montage pour pouvoir les utiliser avec Arduino. Nous aurons l’occasion de voir tout cela en détail plus loin. Ainsi, il ne s’agit pas réellement de capteur, mais de **transducteurs** car nous sommes obligés de devoir utiliser un montage additionnel pour assurer une conversion de la grandeur mesurée en un signal électrique exploitable.

[[information]] | Ce sont principalement des transducteurs que nous allons mettre en œuvre dans le cours. |Puisqu’ils ont l’avantage de pouvoir fonctionner seul et donc cela vous permettra de vous exercer au niveau électronique ! :diable :

6.1.1.2 Capteurs actifs



Figure : Un thermocouple - (CC-0)

Cette autre catégorie de capteur est un peu spéciale et ne recense que très peu de capteurs en son sein. Il s’agit de capteur dont la grandeur physique elle-même mesurée va directement établir une relation électrique de sortie. C’est-à-dire qu’en sortie de ce type de capteur, il y aura une grandeur électrique, sans adjonction de tension à ses bornes. On peut dire que la présence de la tension (ou différence de potentiel, plus exactement) est générée par la grandeur physique. Nous n’entrerons pas dans le détail de ces capteurs et resterons dans ce qui est abordable à votre niveau.

6.1.1.3 Les autres capteurs

En fait, il n'en existe pas réellement d'autres...

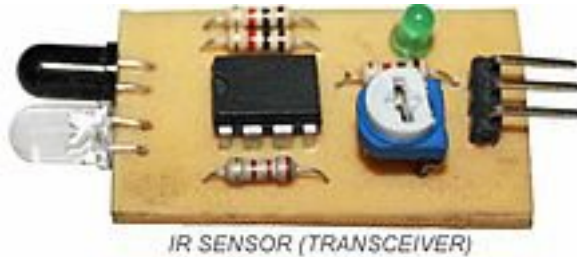


Figure : Détecteur infrarouge - (CC-BY-SA, robot-platform.com)

Ces “autres capteurs”, dont je parle, sont les capteurs ou détecteurs tout prêts que l'on peut acheter dans le commerce, entre autres les détecteurs de mouvements ou capteur de distance. Ils ne font pas partie des deux catégories précédemment citées, puisqu'ils possèdent toute une électronique d'adaptation qui va s'occuper d'adapter la grandeur physique mesurée par le capteur et agir en fonction. Par exemple allumer une ampoule lorsqu'il détecte un mouvement.

Sachez cependant qu'il en existe beaucoup d'autres ! Ce sont donc bien des capteurs qui utilisent un ou des transducteurs. On pourra en fabriquer également, nous serons même obligés afin d'utiliser les transducteurs que je vais vous faire découvrir.

[[attention]] | Retenez donc bien la différence entre transducteur et capteur : un transducteur permet de donner une image de la grandeur physique mesurée par une autre grandeur physique, mais il doit être additionné à un montage pour être utilisé ; un capteur est nécessairement constitué d'un transducteur et d'un montage qui adapte la grandeur physique donnée par le transducteur en une information facilement manipulable.

6.1.2 Un capteur, ça capte !

Un capteur, on l'a vu, est donc constitué d'un transducteur et d'une électronique d'adaptation. Le transducteur va d'abord mesurer la grandeur physique à mesurer, par exemple la luminosité. Il va donner une image de cette grandeur grâce à une autre grandeur, dans ce cas une résistance électrique variable. Et l'électronique d'adaptation va se charger, par exemple, de “transformer” cette grandeur en une tension électrique image de la grandeur mesurée. Attention cependant, cela ne veut pas dire que la sortie sera **toujours** une tension variable. Ainsi, on pourrait par exemple plutôt avoir un courant variable (et tension fixe), ou carrément un message via une liaison de communication (voir série par exemple). Un capteur plus simple par exemple pourrait simplement nous délivrer un niveau logique pour donner une information telle que “obstacle présent/absent”.

A gauche se trouve la grandeur physique mesurable. En sortie du transducteur c'est une autre grandeur physique, manipulable cette fois. Et en sortie de l'électronique d'adaptation, c'est l'information qui peut être sous forme de signal électrique ou d'une simple image de la grandeur physique mesurée par une autre grandeur physique telle qu'une tension électrique ou un courant.

6.1.2.1 Mesure, le rôle du transducteur

Gardons notre exemple avec un capteur, pardon, transducteur qui mesure la luminosité. Le transducteur qui opère avec cette grandeur est une *photorésistance* ou *LDR* (Light Depending Resistor).

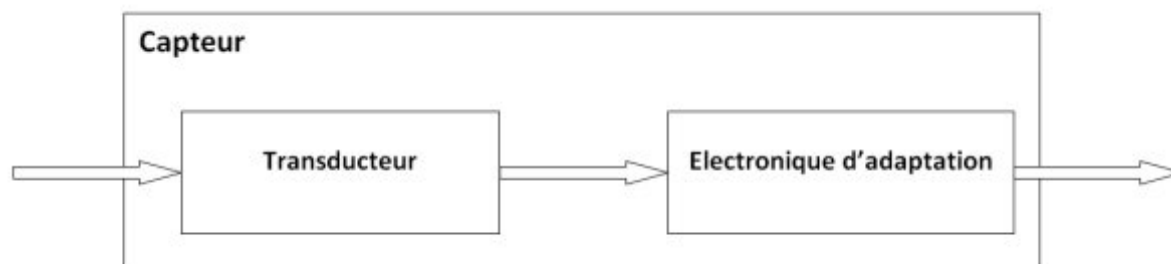


Figure 6.2 – Un capteur c'est quoi ?

C'est une résistance photo-sensible, ou si vous préférez qui réagit à la lumière. La relation établie par la **photorésistance** entre la luminosité et sa résistance de sortie permet d'avoir une image de l'intensité lumineuse par une résistance électrique qui varie selon cette intensité. Voici son symbole électrique et une petite photo d'identité :



Figure : Une photorésistance - (CC-0)



Figure 6.3 – Symbole de la photorésistance

On a donc, en sortie du transducteur, une relation du type y en fonction de x : $y = f(x)$.

Il s'agit simplement du **rapport** entre la grandeur physique d'entrée du capteur et sa grandeur physique de sortie. Ici, le rapport entre la luminosité et la résistance électrique de sortie. Dans les docs techniques, vous trouverez toujours ce rapport exprimé sous forme graphique (on appelle ça une **courbe caractéristique**). Ici, nous avons donc la résistance en fonction de la lumière :



Figure 6.4 – Représentation schématique

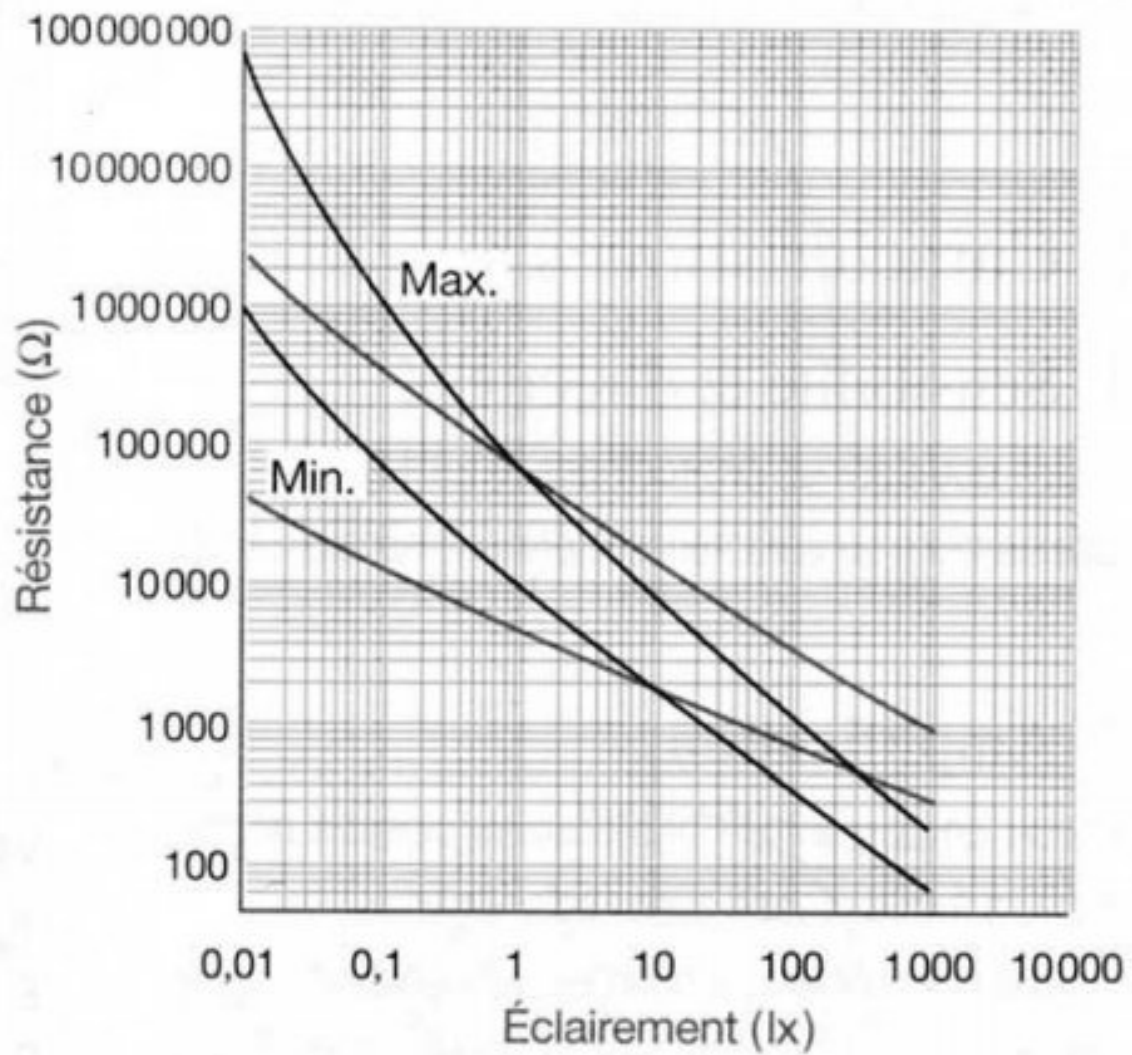


Figure 6.5 – Caractéristique d'une photorésistance

6.1.2.2 L'intérêt d'adapter

Adapter pour quoi faire ? Eh bien je pense déjà avoir répondu à cette question, mais reprenons les explications avec l'exemple ci-dessus. La photorésistance va fournir une résistance électrique qui fluctue selon la luminosité de façon quasi-proportionnelle (en fait ce n'est pas réellement le cas, mais faisons comme si :euh :). Eh bien, que va-t-on faire d'une telle grandeur ? Est-ce que nous pouvons l'utiliser avec notre carte Arduino ? Directement ce n'est pas possible. Nous sommes obligé de l'adapter en une **tension qui varie** de façon proportionnelle à cette résistance, puisque nous ne sommes pas capable de mesurer une résistance directement. Ensuite nous pourrions simplement utiliser la fonction `analogRead()` pour lire la valeur mesurée. Néanmoins, il faudra certainement faire des calculs dans le programme pour donner une réelle image de la luminosité. Et ensuite, éventuellement, afficher cette grandeur ou la transmettre par la liaison série (ou l'utiliser de la manière qui vous fait plaisir :P !). De nouveau, voici la relation établissant le rapport entre les deux grandeurs physiques d'entrée et de sortie d'un transducteur :

$$\rightarrow y = f(x) \leftarrow$$

A partir de cette relation, on va pouvoir gérer l'électronique d'adaptation pour faire en sorte d'établir une nouvelle relation qui soit également une image de la mesure réalisée par le capteur. C'est à dire que l'on va créer une image proportionnelle de la grandeur physique délivrée en sortie du capteur par une nouvelle grandeur physique qui sera, cette fois-ci, bien mieux exploitable. En l'occurrence une tension dans notre cas. La nouvelle relation sera du style y' (noté y') en fonction de y :

$$\rightarrow y' = g(y) \leftarrow$$

Ce qui revient à dire que y' est la relation de sortie du capteur en fonction de la grandeur de mesure d'entrée. Soit :

$$\rightarrow y' = g(y) \longleftrightarrow y' = g(f(x)) \leftarrow$$

Concrètement, nous retrouvons ces formules dans chaque partie du capteur :

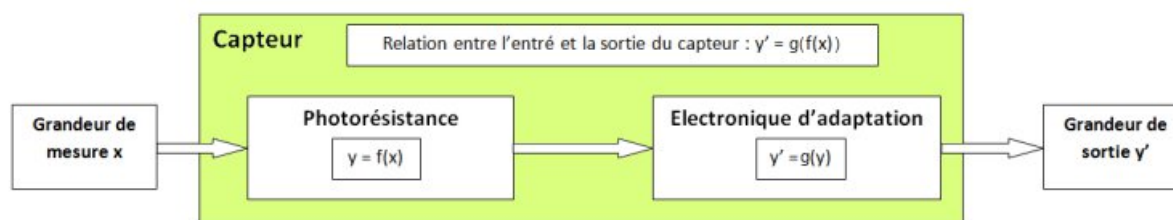


Figure 6.6 – Représentation schématique

6.1.2.3 L'électronique d'adaptation

Elle sera propre à chaque capteur. Cependant, je l'ai énoncé au début de ce chapitre, nous utiliserons principalement des transducteurs qui ont en sortie une résistance électrique variable. L'électronique d'adaptation sera donc quasiment la même pour tous. La seule chose qui changera certainement, c'est le programme. Oui car la carte Arduino fait partie intégrante du capteur puisque c'est avec elle que nous allons "fabriquer" nos capteurs. Le programme sera donc différent pour chaque capteur, d'autant plus qu'ils n'ont pas tous les mêmes relations de sortie... vous l'aurez compris, on aura de quoi s'amuser ! :P

Pour conclure sur l'intérieur du capteur, rentrons dans la partie électronique d'adaptation. La carte Arduino faisant partie de cette électronique, on va avoir un schéma tel que celui-ci :

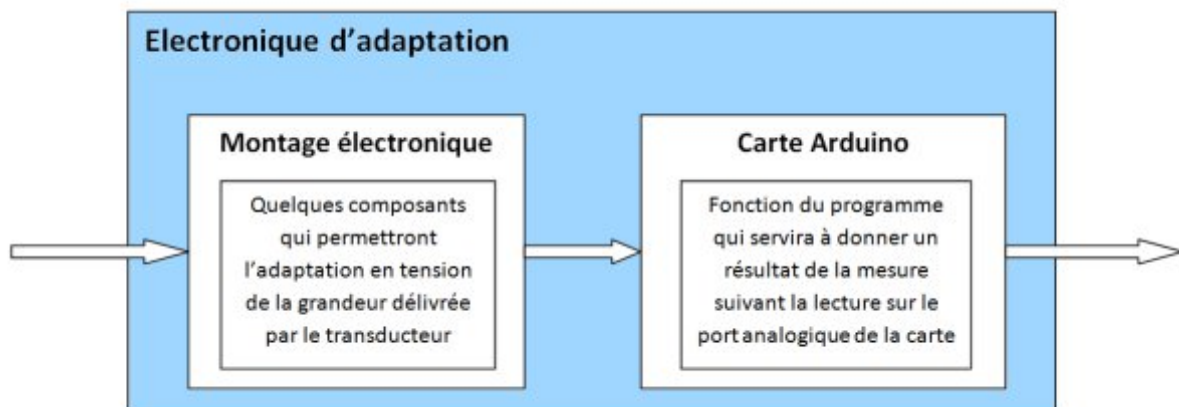


Figure 6.7 – Schéma d'électronique d'adaptation

À l'entrée de l'électronique d'adaptation se trouve la grandeur de sortie du transducteur ; à la sortie de l'électronique d'adaptation se trouve la grandeur de sortie du capteur. On peut après faire ce que l'on veut de la mesure prise par le capteur. Toujours avec la carte Arduino, dans une autre fonction du programme, on pourra alors transformer la valeur mesurée pour la transmettre via la liaison série ou simplement l'afficher sur un écran LCD, voir l'utiliser dans une fonction qui détermine si la mesure dépasse un seuil limite afin de fermer les volets quand il fait nuit...

6.1.3 Les caractéristiques d'un capteur

Pour terminer cette introduction générale sur les capteurs, nous allons aborder les caractéristiques essentielles à connaître.

6.1.3.1 Les critères à ne pas négliger

6.1.3.1.1 La plage de mesure La **plage de mesure**, ou *gamme de mesure*, est la première chose à regarder dans le choix d'un capteur ou d'un transducteur. C'est elle qui définit si vous allez pouvoir mesurer la grandeur physique sur une grande plage ou non. Par exemple pouvoir mesurer une température de -50°C à $+200^{\circ}\text{C}$. Tout dépendra de ce que vous voudrez mesurer.

6.1.3.1.1.1 La précision La précision est le deuxième critère de choix le plus important. En effet, si votre capteur de température a une précision de 1°C , vous aurez du mal à l'utiliser dans un projet qui demande une précision de mesure de températures de 0.1°C ! En règle générale, la précision est plus grande lorsque la plage de mesure est faible et inversement elle devient moins grande lorsque la plage de mesure augmente.

Il est en effet assez difficile de fabriquer des capteurs qui ont une plage de mesure très grande par exemple un voltmètre qui mesurerait jusqu'à 1000V avec une précision de 0.001V ! Et puis, c'est rarement utile d'avoir ces deux paramètres à leur valeur la plus élevée (grande plage de mesure et grande précision).

Dans un cas le plus général, à prix égal un capteur qui mesure une plus grande plage aura sûrement une précision plus faible qu'un capteur mesurant une plage plus réduite.

6.1.3.1.2 Sa tension d'alimentation Il est en effet important de savoir à quelle tension il fonctionne, pour ne pas avoir de mauvaises surprises lorsque l'on veut l'utiliser !

6.1.3.2 D'autres caractéristiques à connaître

6.1.3.2.1 La résolution Certains capteurs proposent une sortie via une communication (série, I²C, SPI...). Du coup, la sortie est dite "numérique" (puisqu'on récupère une information logique plutôt qu'analogique). Un facteur à prendre en compte est la résolution proposée. Certains capteurs seront donc sur 8 bits (la valeur de sortie sera codé sur 256 niveaux), d'autres 10 bits, 16 bits, 32 bits... Il est évident que plus la résolution est élevée et plus la précision offerte est grande.

6.1.3.2.2 La reproductibilité Ce facteur sert à déterminer la fiabilité d'une mesure. Si par exemple vous souhaitez mesurer une température à 0.1°C près, et que le capteur que vous utilisez oscille entre 10.3° et 10.8°C lorsque vous faites une série de mesures consécutives dans un intervalle de temps court, vous n'êtes pas précis.

La reproductibilité est donc le critère servant à exprimer la fiabilité d'une mesure au travers de répétitions consécutives, et le cas échéant exprime l'écart-type et la dispersion de ces dernières. Si la dispersion est élevée, il peut-être utile de faire plusieurs mesures en un court-temps pour ensuite faire une moyenne de ces dernières.

6.1.3.2.3 Le temps de réponse Comme son nom l'indique, cela détermine la vitesse à laquelle le capteur réagit par rapport au changement de l'environnement. Par exemple, les changements de température sont des phénomènes souvent lents à mesurer. Si vous passez le capteur d'un milieu très chaud à un milieu très froid, le capteur va mettre du temps (quelques secondes) pour proposer une information fiable. A contrario, certains capteurs réagissent très vite et ont donc un temps de réponse très faible.

6.1.3.2.4 La bande passante Cette caractéristique est plus difficile à comprendre et est lié au temps de réponse. Elle correspond à la capacité du capteur à répondre aux sollicitations de son environnement. Si sa bande passante est élevée, il peut mesurer aussi bien des phénomènes lents que des phénomènes rapides. Si au contraire elle est faible, sa capacité à mesurer des phénomènes lents **ou** rapides sera réduite sur une certaine plage de fréquences.

6.1.3.2.5 La gamme de température d'utilisation Ce titre est assez explicite. En effet, lorsque l'on mesure certains phénomènes physiques, le capteur doit avoir une certaine réponse. Cependant, il arrive que le phénomène soit conditionné par la température. Le capteur doit donc être utilisé dans certaines conditions pour avoir une réponse correcte (et ne pas être détérioré).

[[information]] | Lorsque vous utilisez un capteur pour la première fois, il est souvent utile de pratiquer un **étalonnage**. Cette opération consiste à prendre quelques mesures pour vérifier/corriger la justesse de sa caractéristique par rapport à la datasheet ou aux conditions ambiantes.

Nous verrons tout au long des chapitres certaines caractéristiques. Après ce sera à vous de choisir vos capteurs en fonction des caractéristiques dont vous aurez besoin.

6.2 Différents types de mesures

Pour commencer ce chapitre sur les capteurs, j'ai rassemblé un petit nombre de capteurs qui sont très utilisés en robotique et en domotique. Vous pourrez ainsi comprendre certains concepts généraux tout en ayant accès à une base de connaissance qui puisse vous aider lorsque vous aurez besoin de mettre en place vos projets personnels. On va commencer en douceur avec les capteurs logiques, aussi appelés **Tout Ou Rien** (TOR). Puis on continuera vers les capteurs utilisant des transducteurs à résistance de sortie variable. On verra ensuite des capteurs un peu particuliers puis pour finir des capteurs un peu compliqués. C'est parti !

6.2.1 Tout Ou Rien, un capteur qui sait ce qu'il veut

On va donc commencer par voir quelques capteurs dont la sortie est un état logique qui indique un état : **Ouvert** ou **Fermé**. Ces capteurs sont appelés des capteur Tout Ou Rien, capteurs **logiques** ou encore capteurs à **rupture de tension**.

6.2.1.1 Deux états

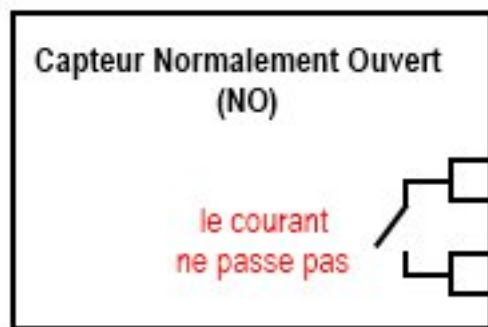


Figure 6.8 – Schéma d'un capteur normalement ouvert



Figure 6.9 – Schéma d'un capteur normalement fermé

Lorsque le capteur est dit ouvert (ou *open* en anglais), le courant ne passe pas entre ses bornes.

S'il s'agit de la **position de repos**, c'est à dire lorsque le capteur ne capte pas la grandeur qu'il doit capter, on dit alors que le capteur a un contact de type **Normalement Ouvert** (ou *Normally Open* en anglais). Tandis que si le capteur est dit fermé (ou *close* en anglais), le courant peut passer

entre ses bornes. S'il s'agit cette fois de sa position de repos, alors on dit qu'il possède un contact **Normalement Fermé** (ou *Normally Closed*).

[[information]] | J'ai schématisé les contacts par des interrupteurs reliés à deux bornes (les carrés à droite) du capteur. | C'est le principe d'un capteur TOR (Tout Ou Rien), mais ce n'est pas forcément des interrupteurs qu'il y a dedans, on va le voir bientôt.

6.2.1.2 Le capteur ILS ou Interrupteur à Lames Souples

6.2.1.2.1 Le principe du TOR Une question qui vous est peut-être passée par la tête : mais comment ce capteur peut mesurer une grandeur physique avec seulement deux états en sortie ? C'est assez facile à comprendre.

Imaginons, dans le cadre d'un de vos projets personnels, que vous ayez l'intention de faire descendre les volets électriques lorsqu'il fait nuit. Vous allez alors avoir recours à un capteur de luminosité qui vous donnera une image, par une autre grandeur physique, de l'intensité lumineuse mesurée. Hors, vous ne voulez pas que ce capteur vous dise s'il fait un peu jour ou un peu nuit, ou entre les deux... Non, il vous faut une réponse qui soit OUI ou NON il fait nuit. Vous aurez donc besoin d'un capteur TOR. Alors, il en existe qui sont capables de donner une réponse TOR, mais lorsque l'on utilisera un transducteur, on devra gérer ça nous même avec Arduino et un peu d'électronique.

6.2.1.2.2 Champ magnétique Ne prenez pas peur, je vais simplement vous présenter le capteur ILS qui utilise le champ magnétique pour fonctionner. ;) En effet, ce capteur, est un capteur TOR qui détecte la présence de champ magnétique. Il est composé de deux lames métalliques souples et sensibles au champ magnétique.

Lorsqu'un champ magnétique est proche du capteur, par exemple un aimant, eh bien les deux lames se mettent en contact et laissent alors passer le courant électrique. D'une façon beaucoup plus simple, c'est relativement semblable à un interrupteur mais qui est actionné par un champ magnétique. Photo d'un interrupteur ILS et image, extraites du site [Wikipédia](#) :



Figure : Interrupteur à

lames souples (reed) - (CC-BY-SA, [Timothée Cognard](#))



teur - (CC-o)

Figure : Action de l'aimant sur l'interrupteur

Dès que l'on approche un aimant, à partir d'un certain seuil de champ magnétique, le capteur agit. Il devient alors un contact fermé et reprend sa position de repos, contact NO, dès que l'on retire le champ magnétique. Ce type de capteur est très utilisé en sécurité dans les alarmes de maison. On les trouve essentiellement au niveau des portes et fenêtres pour détecter leur ouverture.

6.2.1.2.3 Câblage Le câblage de ce capteur peut être procédé de différentes manières. On peut en effet l'utiliser de façon à ce que le courant ne passe pas lorsque rien ne se passe, ou bien qu'il ne passe pas lorsqu'il est actionné.

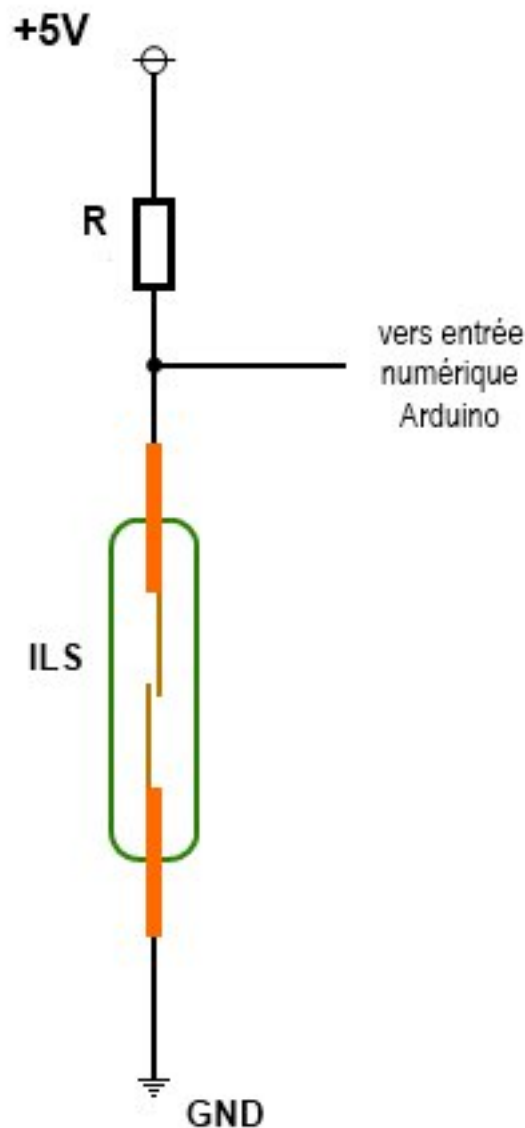


Figure 6.10 – Représentation schématique

1. Dans le premier cas, la sortie vaut HIGH quand l'ILS est au repos et LOW lorsqu'il est actionné par un champ magnétique.
2. Sur la deuxième image, le câblage est différent et fait en sorte que la sortie soit à LOW lorsque le contact ILS est au repos et passe à HIGH dès qu'il est actionné par un champ magnétique.

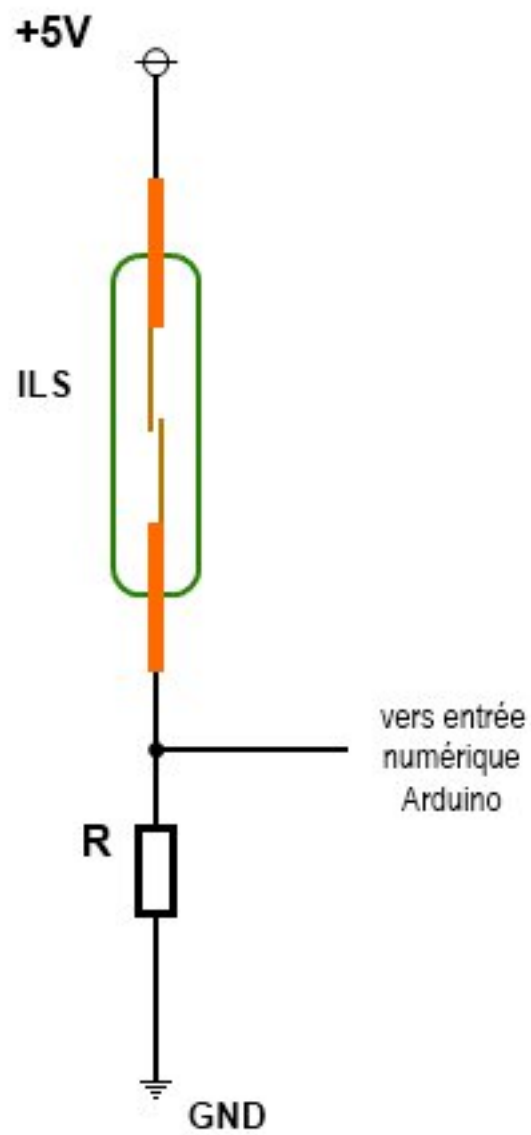


Figure 6.11 – Représentation schématique

Un petit programme tout simple qui permet de voir si l'ILS est actionné ou non, selon le schéma utilisé :

```

const char entree_ils = 2; // utilisation de la broche numérique numéro 2 comme entrée

const char led_indication = 13; // utilisation de la LED de la carte pour indiquer si

unsigned char configuration_ils = 0; // ou 1, dépend du câblage de l'ILS selon les sch
/*
0 pour le premier schéma (gauche)
1 pour le deuxième schéma (droite)
*/

void setup()
{
    // définition des broches utilisées
    pinMode(entree_ils, INPUT);
    pinMode(led_indication, OUTPUT);
}

void loop()
{
    if(configuration_ils) // si c'est le deuxième schéma
    {
        // la LED est éteinte lorsque l'ILS est au repos
        digitalWrite(led_indication, digitalRead(entree_ils));
    }
    else // si c'est le premier schéma
    {
        // la LED est allumée lorsque l'ILS est au repos
        digitalWrite(led_indication, !digitalRead(entree_ils));
    }
}

```

Code : Test des ILS

6.2.1.3 Capteur logique prêt à l'emploi

Bon, là ça va être très très court, puisque vous savez déjà faire ! Les capteurs tout prêts que l'on peut acheter dans le commerce et qui fournissent un état de sortie logique (LOW ou HIGH) sont utilisables tels quels.

Il suffit de connecter la sortie du capteur sur une entrée numérique de la carte Arduino et de lire dans le programme l'état sur cette broche. Vous saurez donc en un rien de temps ce que le capteur indique. On peut citer pour exemple les capteurs de mouvements.

[[information]] | Le programme précédent, utilisé avec l'ILS, est aussi utilisable avec un capteur logique quelconque. Après, à vous de voir ce que vous voudrez faire avec vos capteurs.

6.2.2 Capteurs à résistance de sortie variable

6.2.2.1 La photo-résistance

Nous y voilà, on va enfin voir le transducteur dont j'arrête pas de vous parler depuis tout à l'heure : la photo-résistance ! Je vois que vous commencez à être impatients. ^^

6.2.2.1.1 Petit aperçu La photo-résistance est un composant électronique qui est de type transducteur. Il est donc capable de donner une image de la grandeur physique mesurée, la lumière ou précisément la luminosité, grâce à une autre grandeur physique, la résistance.



Figure : Une photorésistance - (CC-0)

On trouve généralement ce composant en utilisation domotique, pour... devinez quoi?! ... faire monter ou descendre les volets électriques d'une habitation ! :P

Mais on peut également le retrouver en robotique, par exemple pour créer un robot suiveur de ligne noire. Enfin on le trouve aussi dans beaucoup d'autres applications, vous saurez trouver vous-mêmes où est-ce que vous l'utiliserez, je vous fais confiance de ce point de vue là. ;)

6.2.2.1.2 Propriété La photo-résistance suit une relation toute simple entre sa résistance et la luminosité :

$$R = f(E)$$

- R la résistance en Ohm (Ω)
- E l'intensité lumineuse en lux (lx)

Plus l'intensité lumineuse est élevée, plus la résistance diminue. À l'inverse, plus il fait sombre, plus la résistance augmente.

Malheureusement, les photo-résistances ne sont pas des transducteurs très précis. Ils ont notamment des problèmes de linéarité, un temps de réponse qui peut être élevé et une grande tolérance au niveau de la résistance. Nous les utiliserons donc pour des applications qui ne demandent que peu de rigueur. Ce qui ira bien pour ce qu'on veut en faire.



Figure 6.12 – Symbole de la photorésistance

Une **photorésistance** est une résistance qui possède une valeur de base en Ohm. C'est à dire qu'elle est calibrée pour avoir une valeur, par exemple 47 kOhm, à un certain seuil de luminosité. À ce seuil on peut donc mesurer cette valeur, suivant la tolérance qu'affiche la photo-résistance. Si la luminosité augmente, la résistance de base n'est plus vraie et chute. En revanche, dans le noir, la résistance augmente bien au delà de la résistance de base.

[[question]] | Génial!! J'en veux, j'en veux! Comment on l'utilise? :D

La photorésistance est principalement utilisée dans un montage en pont diviseur de tension. Vous le connaissez ce montage, c'est exactement le même principe de fonctionnement que le potentiomètre. Sauf que ce ne sera pas vous qui allez modifier le curseur mais la photorésistance qui, selon la luminosité, va donner une valeur ohmique différente. Ce qui aura pour effet d'avoir une influence sur la tension en sortie du pont diviseur.

6.2.2.1.3 Rappel sur le pont diviseur de tension Je vous rappelle le montage d'un pont diviseur de tension :

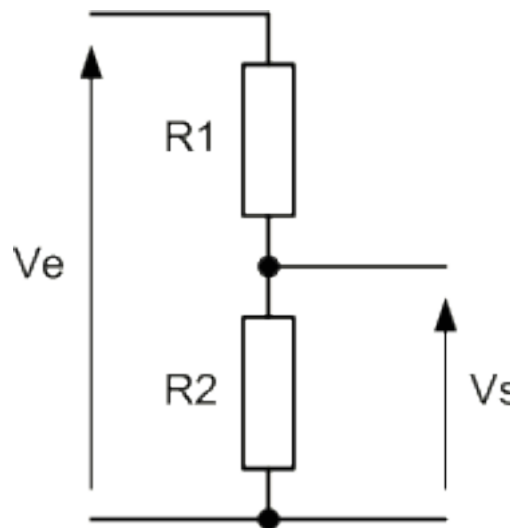


Figure 6.13 – Pont diviseur de tension

La formule associée est la suivante :

$$V_s = V_e \frac{R_2}{R_1 + R_2}$$

[[attention]] | Cette formule s'applique **uniquement** dans le cas où la sortie Vs ne délivre pas de courant (cas des entrées numériques ou analogiques d'un microcontrôleur par exemple). | Dans le cas où il y a justement un courant qui sort de ce pont, cela modifie la valeur de la tension de sortie. C'est comme si vous rajoutiez une résistance en parallèle de la sortie. | Cela a donc pour effet de donner une résistance R2 équivalente plus faible et donc de changer la tension (en appliquant la formule).

Reprenons le tableau présentant quelques exemples :

->

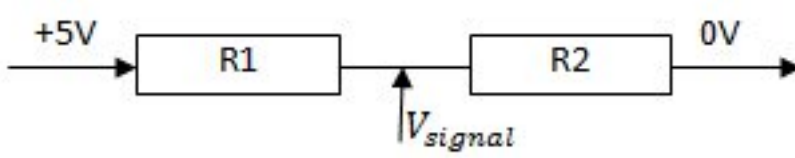
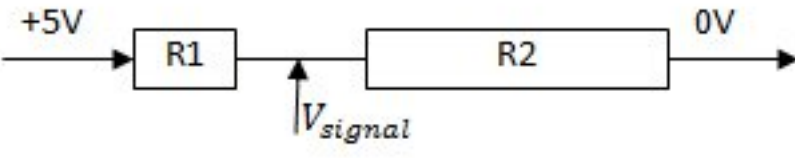
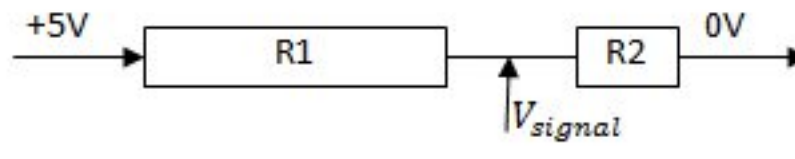
Schéma équivalent	Position du curseur	Tension sur
	Curseur à la moitié	$V_{signal} = (1$
	Curseur à 25% du départ	$V_{signal} = (1$
	Curseur à 75% du départ	$V_{signal} = (1$

Table 6.1 – Illustrations de la position du curseur et sa tension résultante

<-

Vous allez donc maintenant comprendre pourquoi je vais vous donner deux montages pour une utilisation différente de la photorésistance...

6.2.2.1.4 Utilisation n°1 Ce premier montage, va être le premier capteur que vous allez créer ! Facile, puisque je vous fais tout le travail. :P Le principe de ce montage réside sur l'utilisation que l'on va faire de la photo-résistance.

Comme je vous le disais à l'instant, on va l'utiliser dans un pont diviseur de tension. Exactement comme lorsque l'on utilise un potentiomètre. Sauf que dans ce cas, c'est l'intensité lumineuse qui va faire varier la tension en sortie. Voyez plutôt :

On calibre le pont diviseur de tension de manière à ce qu'il soit "équitable" et divise la tension d'alimentation par 2 en sa sortie. Ainsi, lorsque la luminosité fluctuera, on pourra mesurer ces variations avec la carte Arduino. Avec ce montage, **plus l'intensité lumineuse est élevée, plus la**

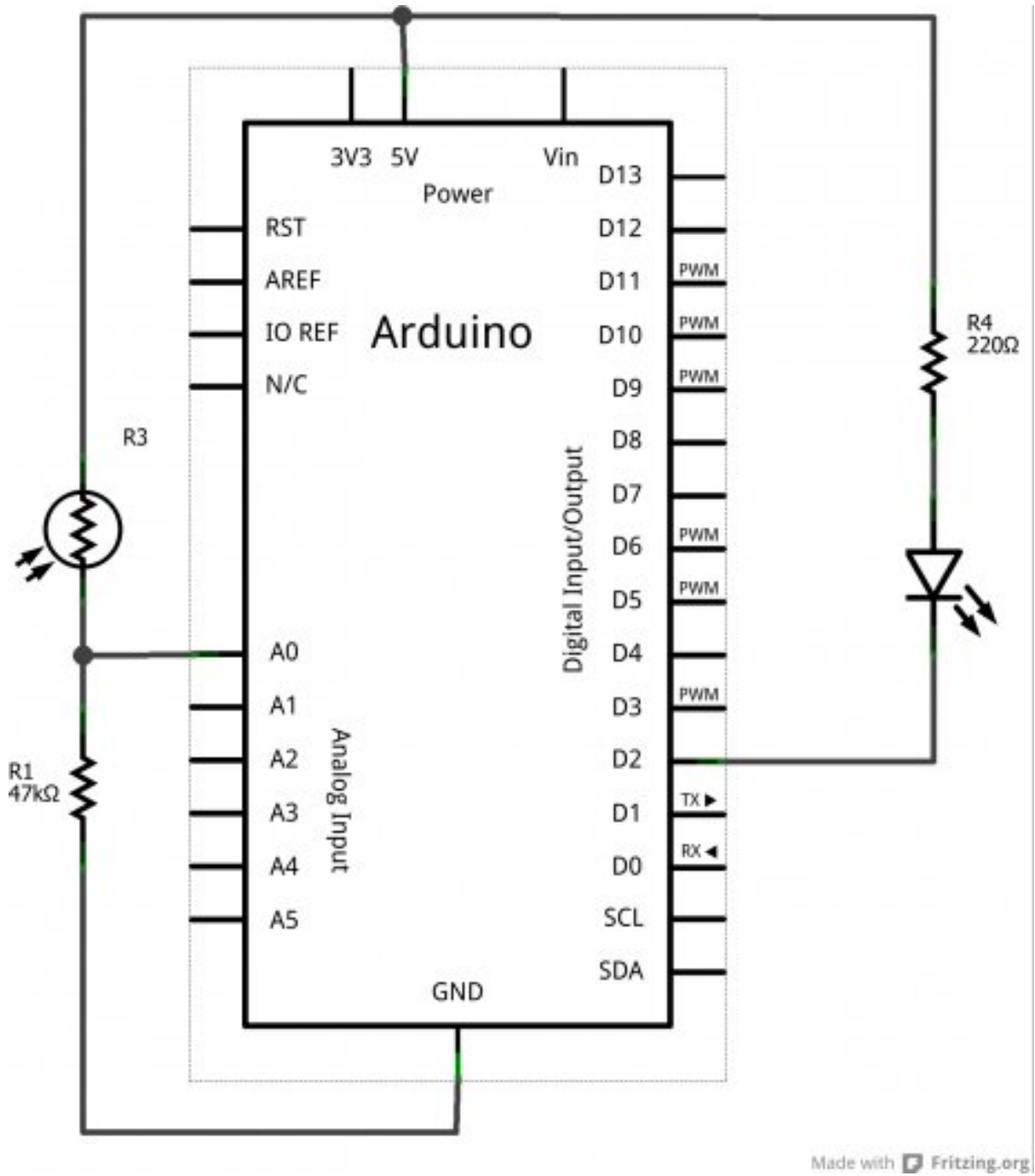


Figure 6.14 – Branchement n°1

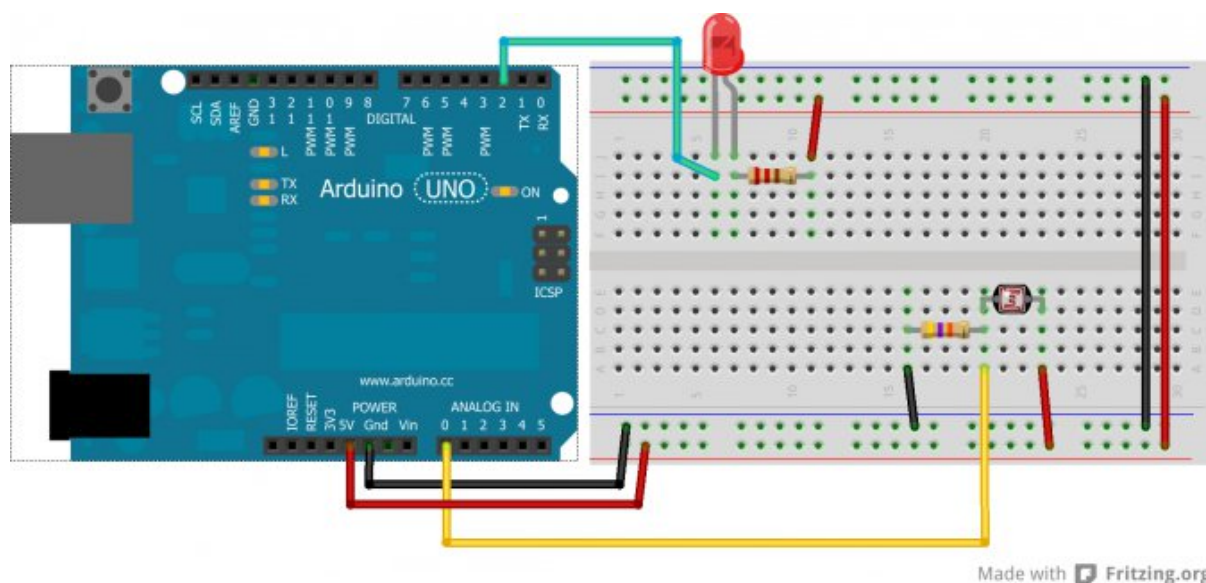


Figure 6.15 – Montage n°1

tension en sortie du pont sera élevée à son tour. Et inversement, plus il fait sombre, moins la tension est élevée.

6.2.2.1.5 Utilisation n°2 Tandis que là, c'est l'effet inverse qui va se produire : **plus il y aura de lumière, moins il y aura de tension en sortie du pont**. Et plus il fera sombre, plus la tension sera élevée.

Rien de bien sorcier. Il suffit de bien comprendre l'intérêt du pont diviseur de tension.

6.2.2.2 Un peu de programmation

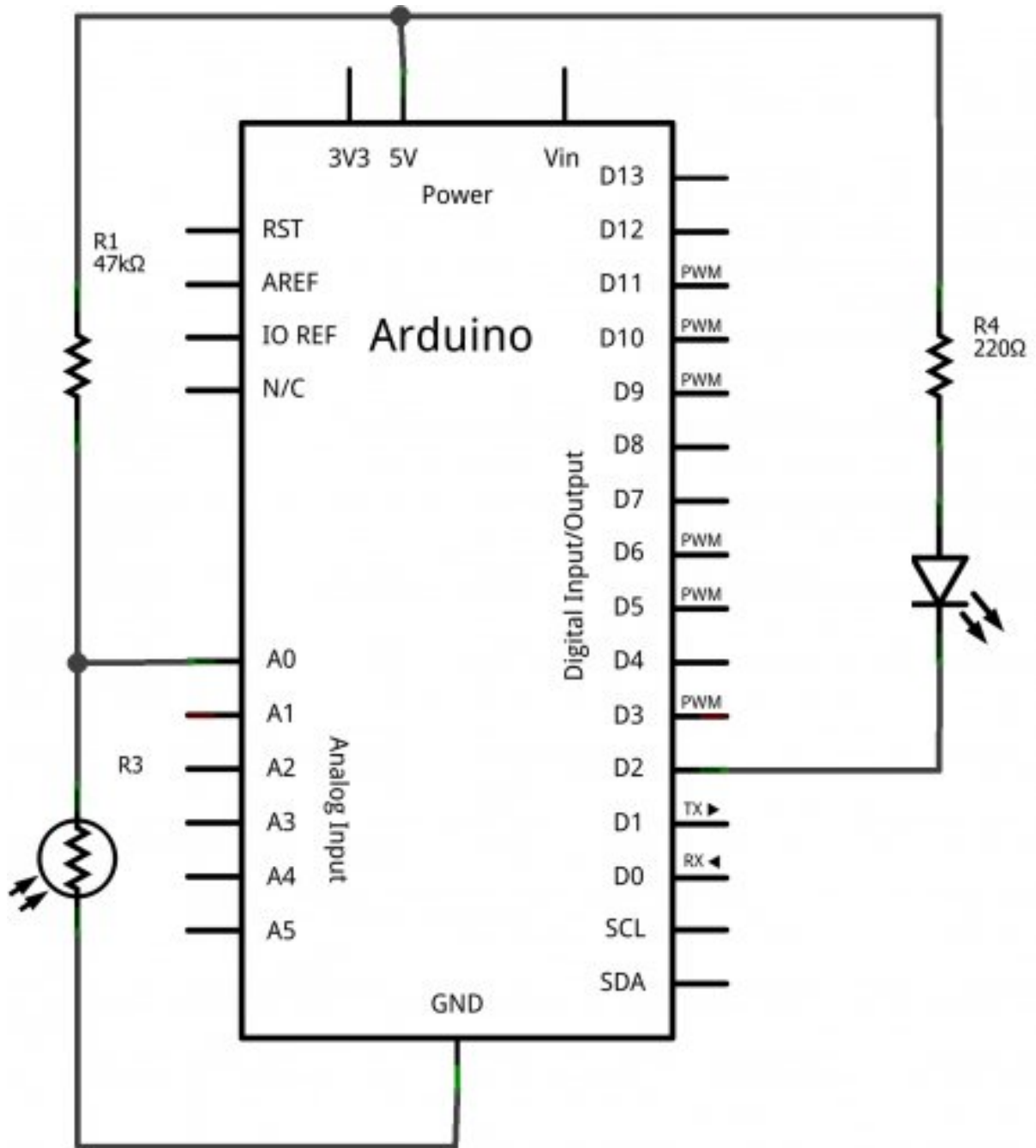
Et si vous aviez un réveil, qui ne vous donne pas l'heure ? Fort utile, n'est-ce pas ! :D Non, sérieusement, il va vous réveiller dès que le jour se lève... ce qui fait que vous dormirez plus longtemps en hiver. C'est vos profs qui vont pas être contents ! ^^ Vous n'aurez qu'à dire que c'est de ma faute. :lol :

Bon allez, un peu de tenue quand même, je ne voudrais pas être la cause de votre échec scolaire. Cette fois, vraiment sérieusement, nous allons faire un tout petit programme qui va simplement détecter la présence ou l'absence de lumière. Lorsque la tension en sortie du pont diviseur de tension créée avec la photorésistance et la résistance fixe chute, c'est que la luminosité augmente. À vous de choisir le schéma correspondant suivant les deux présentés précédemment. Pour commencer, on va initialiser les variables et tout le tralala qui va avec.

```
const char led = 2;           // Une LED pour indiquer s'il fait jour
const char capteur = 0;      // broche A0 sur laquelle va être connecté le pont diviseur de

float tension = 0;           // variable qui va enregistrer la tension lue en sortie
float seuilObscurite = 1.5; // valeur en V, seuil qui détermine le niveau auquel l'

void setup()
{
```



Made with  Fritzing.org

Figure 6.16 – Branchement n°2

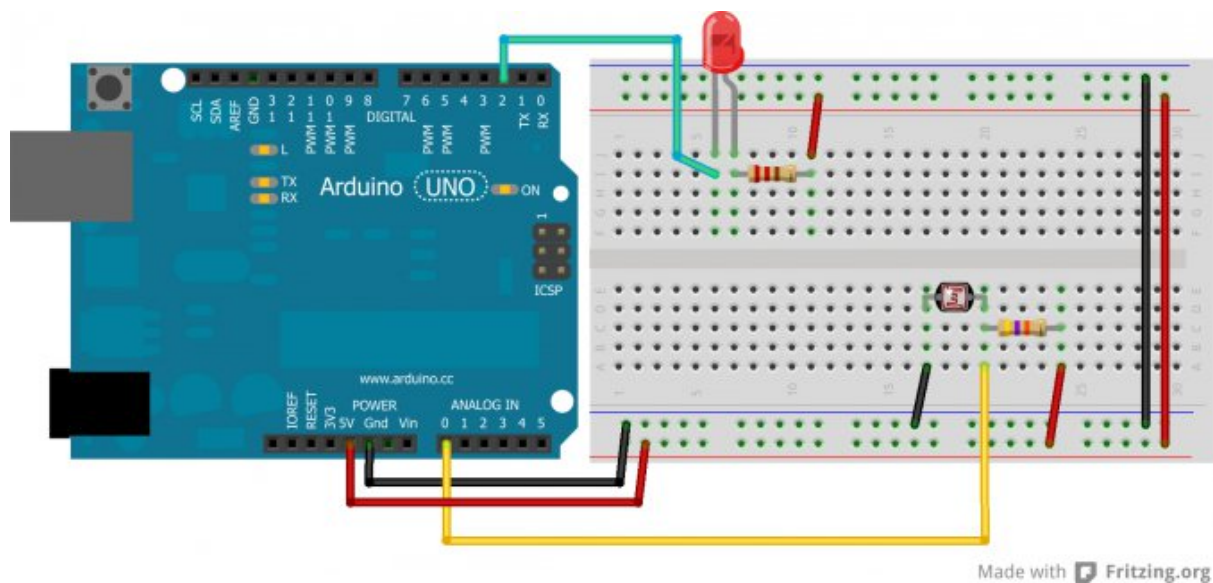


Figure 6.17 – Montage n°2

```
// définition des broches utilisées
pinMode(led, OUTPUT);

Serial.begin(9600); // la voie série pour monitorer
}
```

Code : Mise en place du capteur

Qu'allons-nous retrouver dans la fonction `loop()` ? Eh bien avant tout il va falloir lire la valeur présente en entrée de la broche analogique A0. Puis, on va calculer la tension correspondante à la valeur lue. Enfin, on va la comparer au seuil préalablement défini qui indique le niveau pour lequel l'absence de lumière fait loi.

```
void loop()
{
    // conversion de cette valeur en tension
    tension = (analogRead(capteur) * 5.0) / 1024;

    if(tension >= seuilObscurite)
    {
        digitalWrite(led, LOW); // On allume la LED
    }
    else
    {
        digitalWrite(led, HIGH); // On éteint la LED
    }
    // envoie de la valeur de la tension lue
    // vers l'ordinateur via la liaison série
    Serial.print("Tension = ");
    Serial.print(tension);
    Serial.println(" V");
}
```

```

    // délai pour ne prendre des mesures que toutes les demi-secondes
    delay(500);
}

```

Code : Détecter la nuit

6.2.2.3 Un programme plus évolué

Après tant de difficulté (:euh :), voici un nouveau programme qui vous sera peut-être plus intéressant à faire. En fait ça le deviendra dès que je vous aurais dit l'application qui en est prévue...

6.2.2.3.1 Préparation Cette fois, je vais vous demander d'avoir deux photorésistances identiques. Le principe est simple on va faire une comparaison entre les deux valeurs retournées par les deux capteurs (deux fois le montage précédent).

Si la valeur à droite est plus forte, on allumera une LED en broche 2. Sinon, on allume en broche 3. Si la différence est faible, on allume les deux. Dans tous les cas, il n'y a pas de cas intermédiaire. C'est soit à gauche, soit à droite (selon la disposition des photorésistances).

Ce principe pourrait être appliqué à un petit robot mobile avec un comportement de papillon de nuit. Il cherche la source de lumière la plus intense à proximité.

6.2.2.3.2 Le programme Il parle de lui même, pas besoin d'en dire plus.

```

// déclaration des broches utilisées
const char ledDroite = 2;
const char ledGauche = 3;
const char capteurDroit = 0;
const char capteurGauche = 1;

/* deux variables par capteur qui une stockera la valeur lue sur la broche analogique
et l'autre stockera le résultat de la conversion de la précédente valeur en tension */
float lectureDroite = 0;
float lectureGauche = 0;
float tensionDroite = 0;
float tensionGauche = 0;

void setup()
{
    pinMode(ledDroite, OUTPUT);
    pinMode(ledGauche, OUTPUT);
    Serial.begin(9600);
}

void loop()
{
    // lecture de la valeur en sortie du capteur capteurDroit puis gauche
    lectureDroite = analogRead(capteurDroit);
    lectureGauche = analogRead(capteurGauche);
}

```

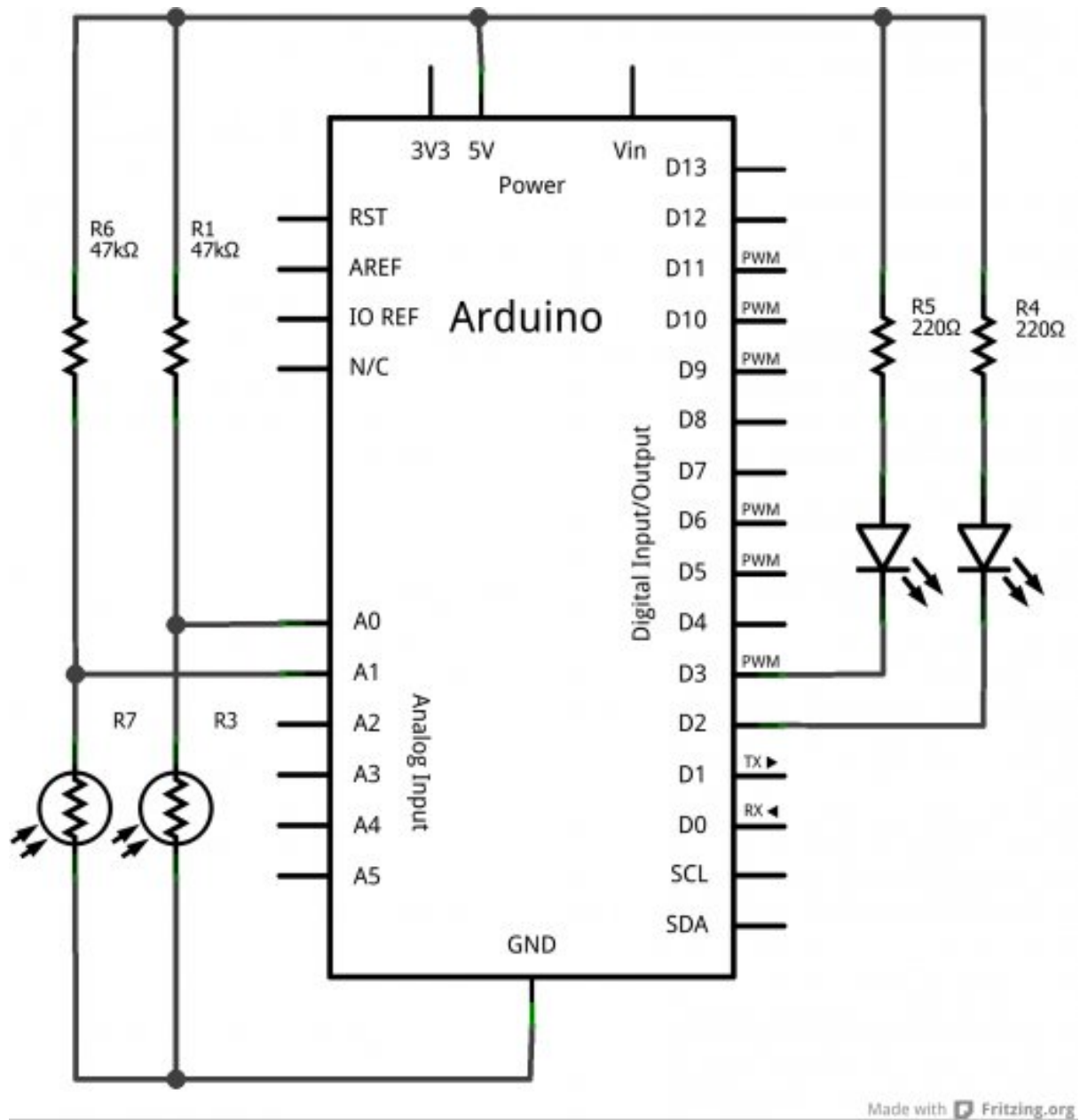


Figure 6.18 – Branchement papillon de lumière

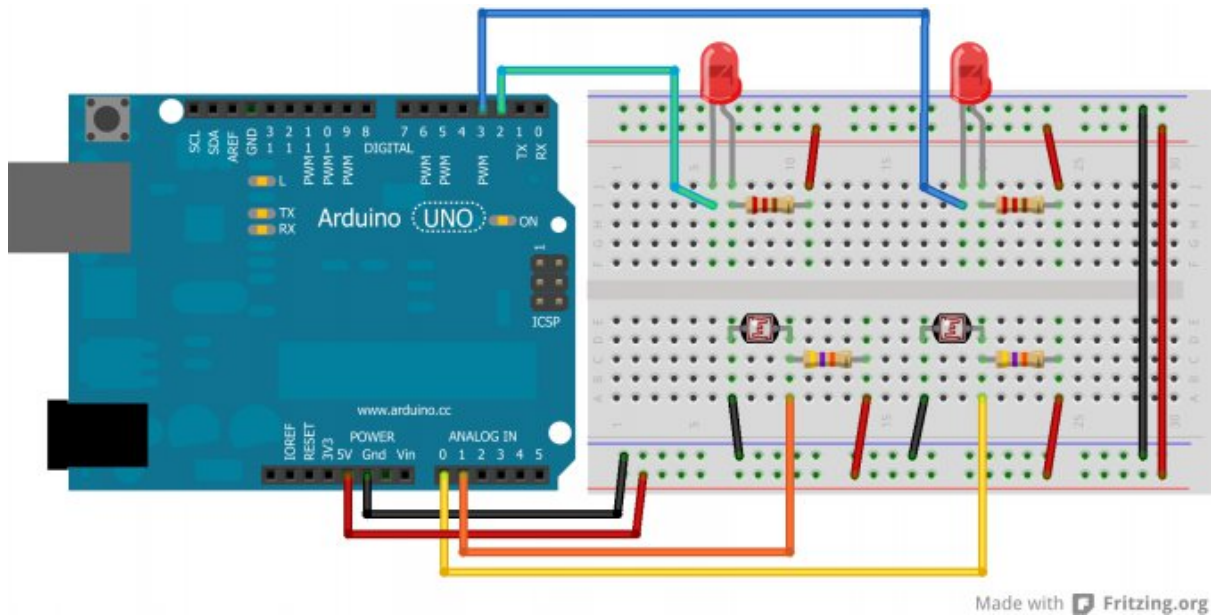


Figure 6.19 – Montage papillon de lumière

```

// conversion en tension de la valeur lue
tensionDroite = (lectureDroite * 5.0) / 1024;
tensionGauche = (lectureGauche * 5.0) / 1024;

// si la tension lue en sortie du capteur 1 est plus grande
// que celle en sortie du capteur 2
if(tensionDroite > tensionGauche)
{
    digitalWrite(ledDroite, LOW); // allumée
    digitalWrite(ledGauche, HIGH); // éteinte
}
else
{
    digitalWrite(ledDroite, HIGH); // éteinte
    digitalWrite(ledGauche, LOW); // allumée
}
// envoi des données lues vers l'ordinateur
Serial.print("Tension Droite = ");
Serial.print(tensionDroite);
Serial.println(" V");
Serial.print("Tension Gauche = ");
Serial.print(tensionGauche);
Serial.println(" V");

delay(100); // délai pour ne prendre une mesure que toutes les 100ms
}

```

Code : Le petit robot mobile.

J'en parlais donc brièvement, ce petit programme peut servir de cerveau à un petit robot mobile qui cherchera alors la source de lumière la plus intense à ses "yeux". Vous n'aurez plus qu'à remplacer les LED par une commande de moteur (que l'on verra dans la prochaine partie sur les moteurs) et alimenter le tout sur batterie pour voir votre robot circuler entre vos pattes. ^^ Bien entendu ce programme pourrait largement être amélioré!

6.2.2.4 Un autre petit robot mobile

On peut renverser la situation pour faire en sorte que le robot suive une ligne noire tracée au sol. C'est un robot suiveur de ligne. Le principe est de "coller" les deux "yeux" du robot au sol.

L'état initial va être d'avoir un œil de chaque côté de la ligne noire. Le robot avance tant qu'il voit du blanc (car la ligne noire est sur une surface claire, blanche en général). Dès qu'il va voir du noir (lorsque la luminosité aura diminué), il va alors arrêter de faire tourner le moteur opposé à l'œil qui a vu la ligne noire. Ainsi, le robot va modifier sa trajectoire et va continuer en suivant la ligne.

À partir de cela, je vous laisse réfléchir à tout ce que vous pouvez faire. Non pas de programme donné tout frais, je viens de définir un cahier des charges, somme toute, assez simple. Vous n'avez donc plus qu'à le suivre pour arriver à vos fins.

6.2.3 Capteurs à tension de sortie variable

Passons à un capteur un petit peu plus amusant et plus étonnant, dont les applications sont très variées!

6.2.3.1 L'élément piézoélectrique

Sous ce nom peu commun se cache un phénomène physique très intéressant. L'élément piézoélectrique, que l'on retrouve dans divers objets du quotidien (montres, certains briquets, raquettes de tennis, ...) présente en effet toute une panoplie de caractéristiques utilisées dans des dizaines voire centaines de domaines. Nous allons voir tout ça en détail. Nous, ce qui va nous intéresser pour le moment, c'est sa propriété à capter des sons.



Figure : Éléments piézoélectriques de montre - (CC-BY-SA, FDominec)



Figure : Allumeur de briquet - (CC-o)

6.2.3.1.1 Constitution Avant de parler de son fonctionnement, voyons un peu sa constitution.

Prenons les éléments piézoélectriques de la première image, à gauche. On observe qu'ils se trouvent sous une forme de pastille composée de plusieurs couches. Généralement c'est une pastille de céramique qui est montée sur une pastille métallique. La fabrication de ces éléments étant très complexe, nous en resterons à ce niveau d'approche.

6.2.3.1.2 Propriété J'ai trouvé amusant de voir sur internet que l'on parlait de sa propriété principale comme étant analogue à celle d'une éponge. Je ne vous épargnerais donc pas cet exemple. ^^

Dès qu'on met une éponge en contact avec de l'eau, elle l'absorbe. Tandis que lorsqu'on la presse, elle se vide de l'eau qu'elle a absorbée. Le rapport avec l'élément piézoélectrique ? Eh bien il agit un peu de la même manière.

Un élément piézoélectrique, lui, subit un phénomène semblable : dès qu'on lui admet une contrainte mécanique, il génère une tension électrique. En revanche, dès qu'on lui administre une tension électrique, il génère alors une contrainte mécanique, restituée par exemple sous forme sonore.

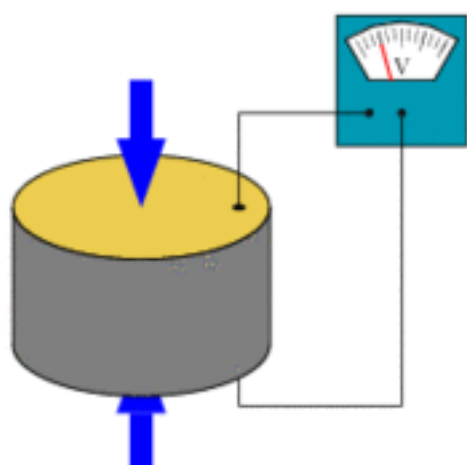


Figure : Génération d'une tension électrique par un élément piézoélectrique sous l'action d'une contrainte mécanique - (CC-BY-SA, Tizeff)

Un exemple d'utilisation dont vous ne vous doutez certainement pas, c'est l'utilisation de cette propriété dans certaines raquettes de tennis. L'élément piézoélectrique se trouve dans le manche de la raquette. Lorsqu'une balle frappe la raquette, elle génère une contrainte mécanique sur l'élément piézoélectrique qui en retour génère une tension électrique. Cette tension est récupérée et injectée à nouveau dans l'élément piézoélectrique qui génère alors une contrainte mécanique opposée à celle générée par la balle. Vous me suivez ? L'intérêt ? Réduire les vibrations causées par le choc et ainsi améliorer la stabilité de la raquette (et de la frappe) tout en réduisant le "stress" provoqué sur le poignet du joueur. Dingue non ?

6.2.3.1.3 Utilisation L'utilisation que nous allons faire de cet élément va nous permettre de capter un choc. Cela peut être un "toc" sur une porte, une déformation de surface, voire même une onde sonore un peu puissante. Ce capteur délivre directement une tension proportionnelle à la contrainte mécanique qu'on lui applique. Il s'agit donc d'un capteur actif. Nous pouvons donc l'utiliser sans rien en le connectant directement avec Arduino.

6.2.3.2 Montage

Vous allez procéder au montage suivant en respectant le schéma de câblage :

La résistance de $1M\Omega$ en parallèle de l'élément piézoélectrique permet d'éviter les courants trop forts qui peuvent être générés par l'élément piézoélectrique.

Il est accompagné par une diode un peu particulière que l'on appelle une diode zener. Cette dernière sert à éviter les surtensions. Si jamais la tension générée par le piezo dépasse son seuil (4.7V en l'occurrence), elle deviendra passante et le courant ira donc vers la masse plutôt que dans le microcontrôleur (évitant ainsi de griller l'entrée analogique). Cette dernière n'est pas indispensable mais fortement conseillée.

6.2.3.3 Programme

Le programme que nous allons associer à ce montage, et qui va être contenu dans la carte Arduino, va exploiter la tension générée par l'élément piézoélectrique, lorsqu'on lui administrera

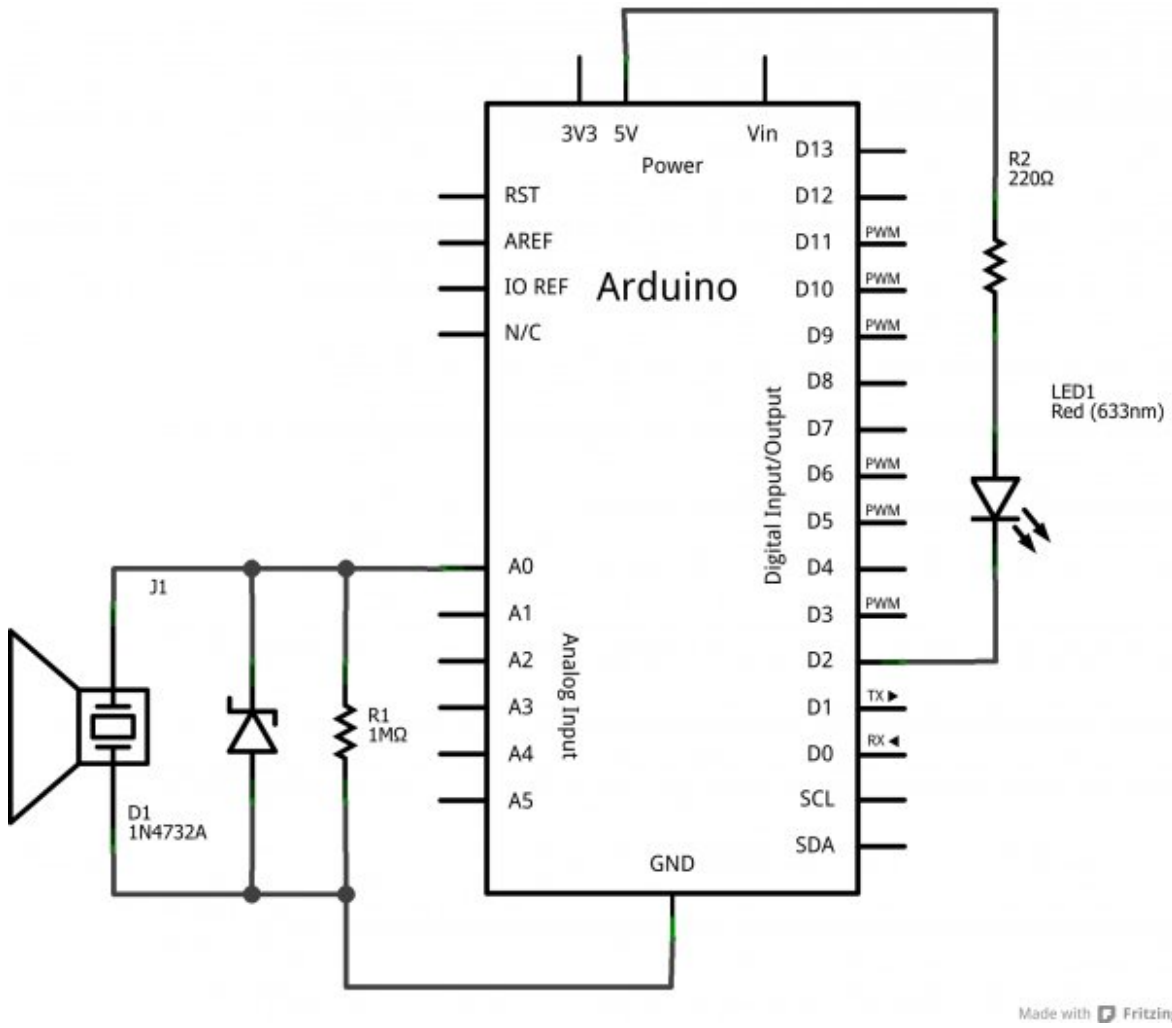


Figure 6.20 – Branchement du piezo

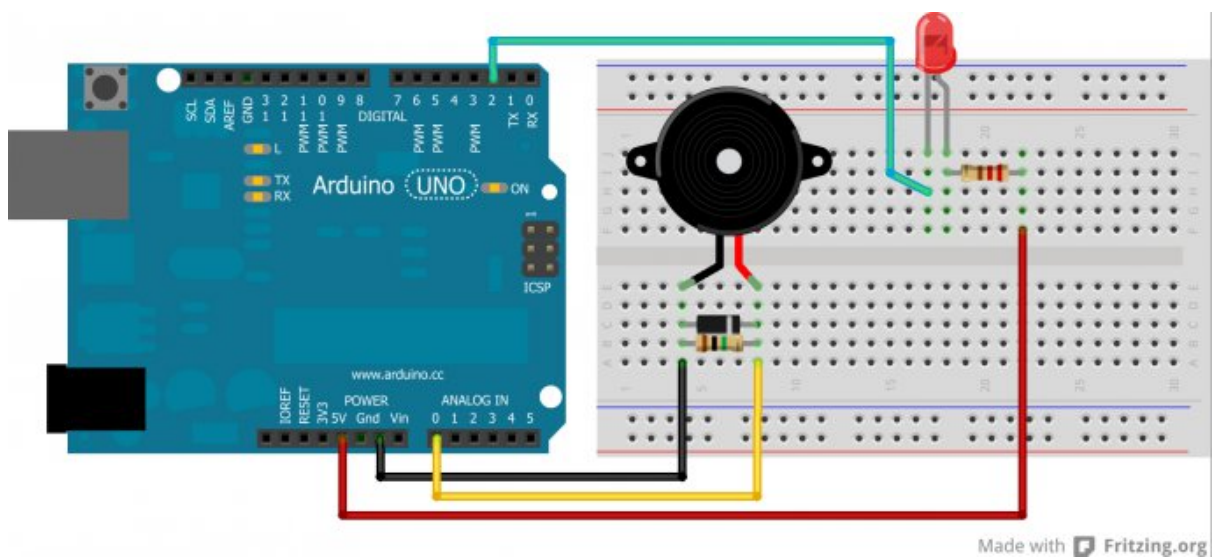


Figure 6.21 – Montage du piezo

une contrainte mécanique, pour allumer ou éteindre une LED présente en broche 2 de la carte Arduino. On pourra s'en servir pour détecter un événement sonore tel que le toc sur une porte.

La condition pour que l'élément piézoélectrique capte correctement le toc d'une porte est qu'il doit être positionné sur la porte de façon à ce que sa surface soit bien plaquée contre elle. Aussi nous utiliserons la liaison série pour indiquer la tension produite par l'élément piézoélectrique.

C'est un petit plus, par forcément utile mais qui vous donnera une idée de la force qu'il faut pour générer une tension particulière. Vous en trouverez certainement une application utile. ;) Allez, un peu de programmation !

6.2.3.3.1 Fonction setup() Au début du programme nous déclarons quelques variables que nous utiliserons par la suite. Aussi nous amorçons l'utilisation de la liaison série et des broches utilisées de la carte Arduino.

```
const char led = 2;           // utilisation de la LED en broche 13 de la carte
const char piezo = 0;        // l'élément piézoélectrique est connecté en broche analog
const int seuil_detection = 100;
/* on peut définir le seuil de détection qui va simplement
permettre de confirmer que c'est bien un évènement sonore suffisant et non parasite */
```

Code : Initialisation des paramètres

Petite parenthèse par rapport au seuil. Ici il est configuré de façon à être comparé à la lecture directe de la valeur en broche analogique (comprise entre 0 et 1023). Mais on peut aussi le définir pour qu'il soit comparé au calcul de la tension en sortie du capteur (par exemple le mettre à 1, pour 1V).

```
float lecture_capteur = 0; // variable qui va contenir la valeur lue en broche analog
float tension = 0;         // variable qui va contenir le résultat du calcul de la te
int etat_led = LOW;        // variable utilisée pour allumer ou éteindre la LED à cha

void setup()
{
  pinMode(led, OUTPUT);    // déclaration de la broche 13 en sortie
  Serial.begin(9600);      // utilisation de la liaison série
}
```

Code : Initialisation du micro contrôleur pour piloter la LED

6.2.3.3.2 Fonction principale Étant donné que le code est plutôt court et simple, nous le laisserons dans la seule fonction loop() plutôt que de le découper en plusieurs petites fonctions. Cependant libre à vous de l'agencer autrement selon vos besoins.

```
void loop()
{
  // lecture de la valeur en sortie du capteur
  lecture_capteur = analogRead(piezo);
  // conversion de cette valeur en tension
  tension = (lecture_capteur * 5.0) / 1024;
```

```

if (lecture_capteur >= seuil_detection)
{
    // on modifie l'état de la LED pour le passer à son état opposé
    etat_led = !etat_led;
    // application du nouvel état en broche 13
    digitalWrite(led, etat_led);

    // envoi vers l'ordinateur, via la liaison série,
    // des données correspondant au Toc et à la tension
    Serial.println("Toc!");
    Serial.print("Tension = ");
    Serial.print(tension);
    Serial.println(" V");
}
}

```

Code : Envoie des données captées.

[[information]] | Ici pas de délai à la fin de la boucle. En effet, si vous mettez un délai (qui est bloquant) vous risqueriez de rater des “toc” puisque cet événement est bref et imprévisible. Si jamais vous tapiez sur votre élément piézoélectrique au moment où le programme est dans la fonction `delay()`, vous ne pourriez pas l’intercepter.

6.2.3.3.3 Seuil de tension Comme je le disais, il est aussi possible et non pas idiot de changer le seuil pour qu’il soit comparé en tant que tension et non valeur “abstraite” comprise entre 0 et 1023. Cela relève de la simplicité extrême, voyez plutôt :

```

// utilisation de la LED en broche 13 de la carte
const char led = 2;
// l'élément piézoélectrique est connecté en broche analogique 0
const char piezo = 0;
// seuil de détection en tension et non plus en nombre entre 0 et 1023
const float seuil_detection = 1.36;

// variable qui va contenir la valeur lue en broche analogique 0
float lecture_capteur = 0;
// variable qui va contenir le résultat du calcul de la tension
float tension = 0;
// variable utilisée pour allumer ou éteindre la LED à chaque "Toc"
int etat_led = LOW;

void setup()
{
    pinMode(led, OUTPUT); // déclaration de la broche 13 en sortie
    Serial.begin(9600);    // utilisation de la liaison série
}

void loop()
{

```

```
// lecture de la valeur en sortie du capteur
lecture_capteur = analogRead(piezo);
// conversion de cette valeur en tension
tension = (lecture_capteur * 5.0) / 1024;

if (tension >= seuil_detection) // comparaison de deux tensions
{
    // on modifie l'état de la LED pour le passer à son état opposé
    etat_led = !etat_led;
    // application du nouvel état en broche 13
    digitalWrite(led, etat_led);

    // envoi vers l'ordinateur, via la liaison série,
    // des données correspondant au Toc et à la tension
    Serial.println("Toc!");
    Serial.print("Tension = ");
    Serial.print(tension);
    Serial.println(" V");
}
}
```

Code : Conversion en Volt

Je n'ai modifié que le type de la variable `seuil_detection`.

6.2.3.4 La réversibilité de l'élément piézoélectrique

Tout à l'heure je vous disais que l'élément piézoélectrique était capable de transformer une contrainte mécanique en une tension électrique. Je vous ai également parlé du "phénomène éponge" en vous disant que l'on pouvait aussi bien absorber que restituer non pas de l'eau comme l'éponge mais une contrainte mécanique à partir d'une tension électrique. On va donc s'amuser à créer du son avec l'élément piézoélectrique ! :P

6.2.3.4.1 Faire vibrer l'élément piézoélectrique ! [[attention]] | Attention tout de même, bien que vous l'aurez très certainement compris, il n'est plus question d'utiliser l'élément piézoélectrique en entrée comme un capteur, mais bien en sortie comme un actionneur.

Tout d'abord, il vous faudra brancher l'élément piézoélectrique. Pour cela, mettez son fil noir à la masse et son fil rouge à une broche numérique, n'importe laquelle. Pas besoin de résistance cette fois-ci. Et voilà les branchements sont faits ! Il ne reste plus qu'à générer un signal pour faire vibrer l'élément piézoélectrique. Selon la fréquence du signal, la vibration générée par l'élément piézoélectrique sera plus ou moins grave ou aiguë. Essayons simplement avec ce petit programme de rien du tout (je ne vous donne que la fonction `loop()`, vous savez déjà tout faire ;)) :

```
void loop()
{
    digitalWrite(piezo, HIGH);
    delay(5);
    digitalWrite(piezo, LOW);
}
```



```

    delay(5);
}

```

Code : Signal 100 Hz

Ce code va générer un signal carré d'une période de 10ms, soit une fréquence de 100Hz. C'est un son plutôt grave. Vous pouvez aisément changer la valeur contenue dans les délais pour écouter les différents sons que vous allez produire. Essayez de générer un signal avec la PWM et soyez attentif au résultat en changeant la valeur de la PWM avec un potentiomètre par exemple.

6.2.3.4.2 Une fonction encore toute prête Maintenant, si vous souhaitez générer ce signal et en même temps faire d'autres traitements, cela va devenir plus compliqué, car le temps sera plus difficile à maîtriser (et les délais ne sont pas toujours les bienvenus :P). Pour contrer cela, nous allons confier la génération du signal à une fonction d'Arduino qui s'appelle `tone()`.

Cette fonction prend en argument la broche sur laquelle vous voulez appliquer le signal ainsi que la fréquence dudit signal à réaliser. Si par exemple je veux émettre un signal de 440Hz (qui correspond au "la" des téléphones) je ferais : `tone(piezo, 440);`. Le son va alors devenir permanent, c'est pourquoi, si vous voulez l'arrêter, il vous suffit d'appeler la fonction `noTone()` qui va alors arrêter la génération du son sur la broche spécifiée en argument.

[[information]] | La fonction `tone()` peut prendre un troisième argument qui spécifie en millisecondes la durée pendant laquelle vous désirez jouer le son, vous évitant ainsi d'appeler `noTone()` ensuite.

Pour les plus motivés d'entre vous, vous pouvez essayer de jouer une petite mélodie avec l'élément piézoélectrique. :) *Ah les joies nostalgiques de l'époque des sonneries monophoniques.* :lol :

6.2.4 Étalonner son capteur

Faisons une petite pause dans notre découverte des capteurs pour parler d'un problème qui peut arriver à tout le monde... Comment faites-vous si vous possédez un capteur mais ne possédez pas sa caractéristique exacte ? Comment feriez-vous pour faire correspondre une valeur analogique lue avec une donnée physique réelle ?

Par exemple, si je vous donne un composant en vous disant "Hey, te voilà un capteur de température, je sais qu'il s'alimente en 5V sur telle et telle broches, je sais que le signal de sortie est une tension en fonction de la température mais je suis incapable de te dire quelle est la caractéristique (la courbe tension en fonction de la température)".

Nous allons maintenant voir comment résoudre ce problème en voyant une méthode pour étalonner son capteur. Nous allons ainsi nous même déterminer la courbe caractéristique du capteur et déterminer son coefficient liant la température et la tension. À la fin, je vous donnerais la vraie courbe constructeur et nous pourrions comparer nos résultats pratiques avec ceux de référence :)

6.2.4.0.1 Le capteur utilisé Pour étudier la méthode que je vous propose ici, nous allons utiliser un capteur de température assez répandu qui se nomme "LM35". Il existe dans différents boîtiers que voici :

Vous aurez deviné le branchement, il est assez simple. Il suffit de relier +VS au 5V et GND à la masse. Le signal sera ensuite lu sur la broche Vout.

6.2.4.1 La méthode

La méthode pour caractériser le capteur est assez simple. À l'aide d'une multitude de mesures et d'un appareil témoin, nous allons pouvoir créer un tableau qui nous servira à calculer la courbe (à l'aide d'un logiciel comme Excel par exemple). Pour cela, en plus de votre capteur vous aurez besoin d'un appareil de mesure "témoin" qui vous servira de référence. Par exemple le bon vieux thermomètre qui traîne accroché à votre fenêtre fera parfaitement l'affaire :D .

6.2.4.1.1 Prise de mesures Vous êtes prêts, alors allons-y, commençons à travailler. Reliez le capteur à l'Arduino et l'Arduino à l'ordinateur, de la manière la plus simple possible, comme ceci par exemple :

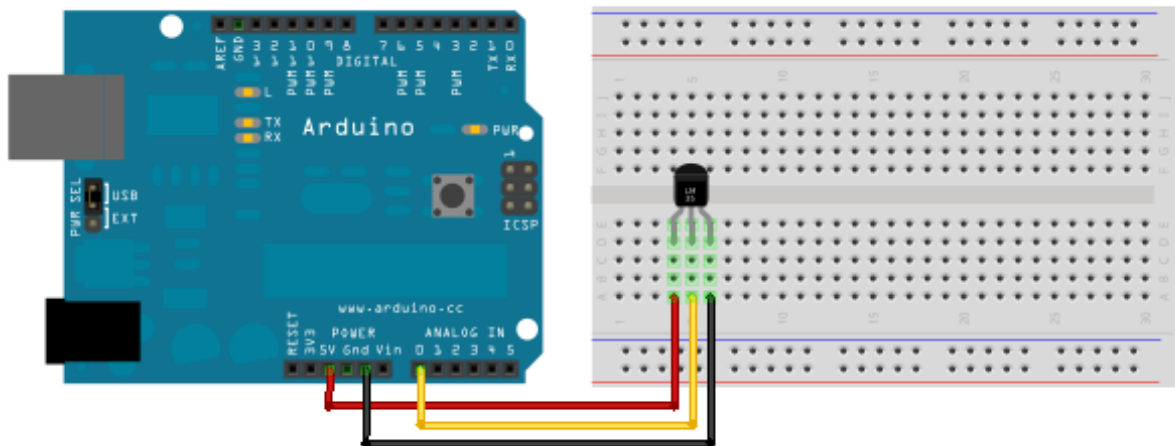


Figure 6.23 – LM35

Ensuite, nous devons récupérer les données envoyées par le capteur de manière régulière (ou rajoutez un bouton et faites des envois lors de l'appui ;). Pour cela, voici un petit programme sans difficulté qui vous enverra les valeurs brutes ou converties en volts toutes les demi-secondes.

```
const int capteur = 0; // capteur branché sur la pin analogique 0
float tension = 0.0;
int valeur = 0;

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  valeur = analogRead(capteur);
  tension = (valeur*5.0)/1024;

  Serial.print("Tension : ");
  Serial.print(tension);
  Serial.println(" V");
}
```

```

    Serial.print("Valeur : ");
    Serial.println(valeur);
    Serial.println("-----");

    delay(500);
}

```

Code : Mesure des valeurs

Maintenant que tout est prêt, il nous faut un banc de test. Pour cela, préparez une casserole avec de l'eau contenant plein de glaçons (l'eau doit être la plus froide possible). Faites une première mesure avec votre capteur plongé dedans (attention, les broches doivent être isolées électriquement ou alors mettez l'ensemble dans un petit sac plastique pour éviter que l'eau n'aille faire un court-circuit). Faites en même temps une mesure de la température réelle observée à l'aide du thermomètre. Une fois cela fait, relevez ces mesures dans un tableau qui possédera les colonnes suivantes :

- Température réelle (en °C)
- Tension selon Arduino (en V)
- Valeur brute selon Arduino

Quand la première mesure est faite, commencez à faire réchauffer l'eau (en la plaçant sur une plaque de cuisson par exemple). Continuez à faire des mesures à intervalle régulier (tous les 5 degrés voire moins par exemple). Plus vous faites de mesure, plus l'élaboration de la courbe finale sera précise. Voici à titre d'exemple le tableau que j'ai obtenu :

->

Température (°C)	Tension (V)	Valeur CAN
2	0,015	3
5	0,054	11
10	0,107	22
16	0,156	32
21	0,210	43
24	0,234	48
29	0,293	60
35	0,352	72
38	0,386	79
43	0,430	88
46	0,459	94
50	0,503	103

Table 6.2 – Relevé de température

<-

6.2.4.1.2 Réalisation de la caractéristique Lorsque vous avez fini de prendre toutes vos valeurs, vous allez pouvoir passer à l'étape suivante qui est :

Calculer la caractéristique de votre courbe !! Sortez vos cahiers, votre calculatrice et en avant ! ... Non je blague (encore que ça ferait un super TP), on va continuer à utiliser notre logiciel tableur pour faire le travail pour nous ! On va donc commencer par regarder un peu l'allure de la courbe.

Je vais en faire deux, une symbolisant les valeurs brutes de la conversion du CAN (entre 0 et 1023) en rouge et l'autre qui sera l'image de la tension en fonction de la température en bleu. Nous pourrions alors déterminer deux caractéristiques, selon ce qui vous arrange le plus.

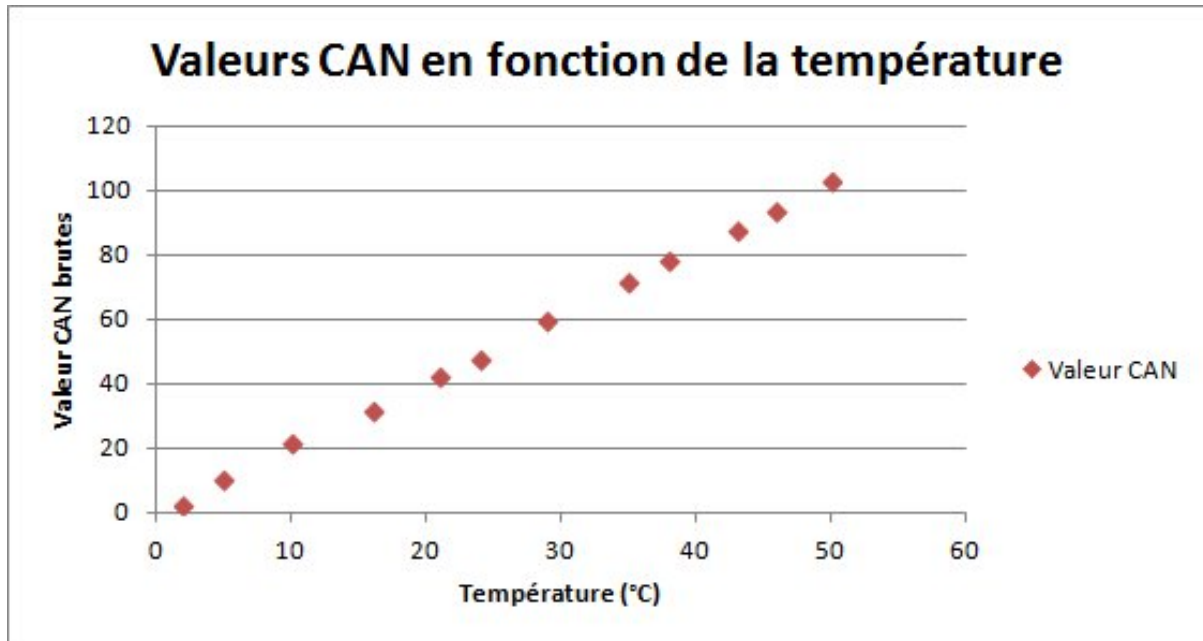


Figure 6.24 – Valeurs CAN en fonction de la température

Une fois cela fait, il ne reste plus qu'à demander gentiment au logiciel de graphique de nous donner la **courbe de tendance** réalisée par ces points. Sous Excel, il suffit de cliquer sur un des points du graphique et choisir ensuite l'option "Ajouter une courbe de tendance...".

Vous aurez alors le choix entre différents types de courbe (linéaire, exponentielle...). Ici, on voit que les points sont alignés, il s'agit donc d'une équation de courbe linéaire, de type $y = ax + b$. Cochez la case "Afficher l'équation sur le graphique" pour pouvoir voir et exploiter cette dernière ensuite.

Voici alors ce que l'on obtient lorsque l'on rajoute notre équation :

Grâce à l'équation, nous pouvons déterminer la relation liant la température et la tension (ou les valeurs du CAN). Ici nous obtenons :

- $y = 0.01x - 0.0003$ (pour la tension)
- $y = 2.056x - 0.0707$ (pour les valeurs du CAN)

Le coefficient constant (-0.003 ou -0.0707) peut ici être ignoré. En effet, il est faible (on dit **négligeable**) comparé aux valeurs étudiées. Dans les équations, x représente la température et y représente la tension ou les valeurs du CAN. On lit donc l'équation de la manière suivante : Tension en Volt égale 0,01 fois la température en degrés Celsius. Ce qui signifie que dorénavant, en ayant une mesure du CAN ou une mesure de tension, on est capable de déterminer la température

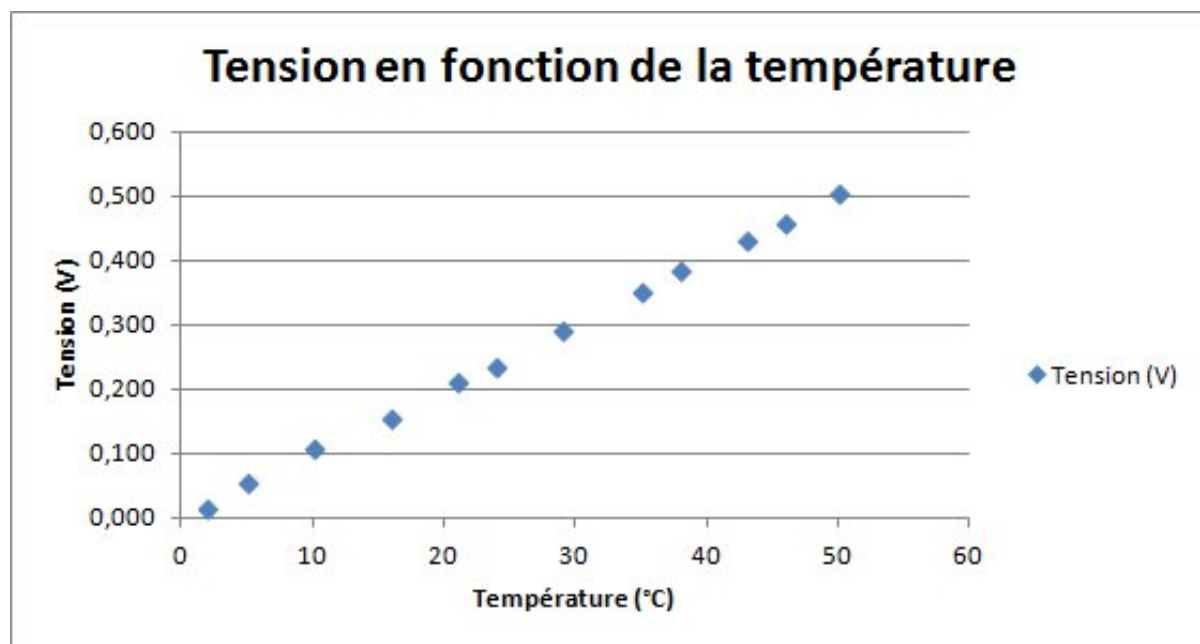


Figure 6.25 – Tension en fonction de la température

en degrés Celsius :) Super non ? Par exemple, si nous avons une tension de 300mV, avec la formule trouvée précédemment on déterminera que l'on a $0.3 = 0.01 \times Temperature$, ce qui équivaut à $Temperature = 0.3/0.01 = 30^{\circ}C$. On peut aisément le confirmer via le graphique ;)

Maintenant j'ai trois nouvelles, deux bonnes et une mauvaise... La bonne c'est que vous êtes capable de déterminer la caractéristique d'un capteur. La deuxième bonne nouvelle, c'est que l'équation que l'on a trouvée est correcte... .. parce qu'elle est marquée dans [la documentation technique](#) qui est super facile à trouver :D (ça c'était la mauvaise nouvelle, on a travaillé pour rien !!) Mais comme c'est pas toujours le cas, c'est toujours bien de savoir comment faire ;)

6.2.4.1.3 Adaptation dans le code Puisque nous savons mesurer les valeurs de notre capteur et que nous avons une équation caractéristique, nous pouvons faire le lien en temps réel dans notre application pour faire une utilisation de la grandeur *physique* de notre mesure. Par exemple, s'il fait 50°C nous allumons le ventilateur. En effet, souvenez-vous, avant nous n'avions qu'une valeur entre 0 et 1023 qui ne signifiait physiquement pas grand chose. Maintenant nous sommes en mesure (oh oh oh :D) de faire la conversion. Il faudra pour commencer récupérer la valeur du signal. Prenons l'exemple de la lecture d'une tension analogique du capteur précédent :

```
int valeur = analogRead(monCapteur); // lit la valeur
```

Nous avons ensuite deux choix, soit nous le transformons en tension puis ensuite en valeur physique grâce à la caractéristique du graphique bleu ci-dessus, soit nous transformons directement en valeur physique avec la caractéristique rouge. Comme je suis fainéant, je vais chercher à économiser une instruction en prenant la dernière solution. Pour rappel, la formule obtenue était : $y = 2.056x - 0.0707$. Nous avons aussi dit que le facteur constant était négligeable, on a donc de manière simplifiée $y = 2.056x$ soit "la température est égale à la valeur lue divisé par 2.056 ($x = \frac{y}{2.056}$). Nous n'avons plus qu'à faire la conversion dans notre programme !

```
float temperature = valeur/2.056;
```

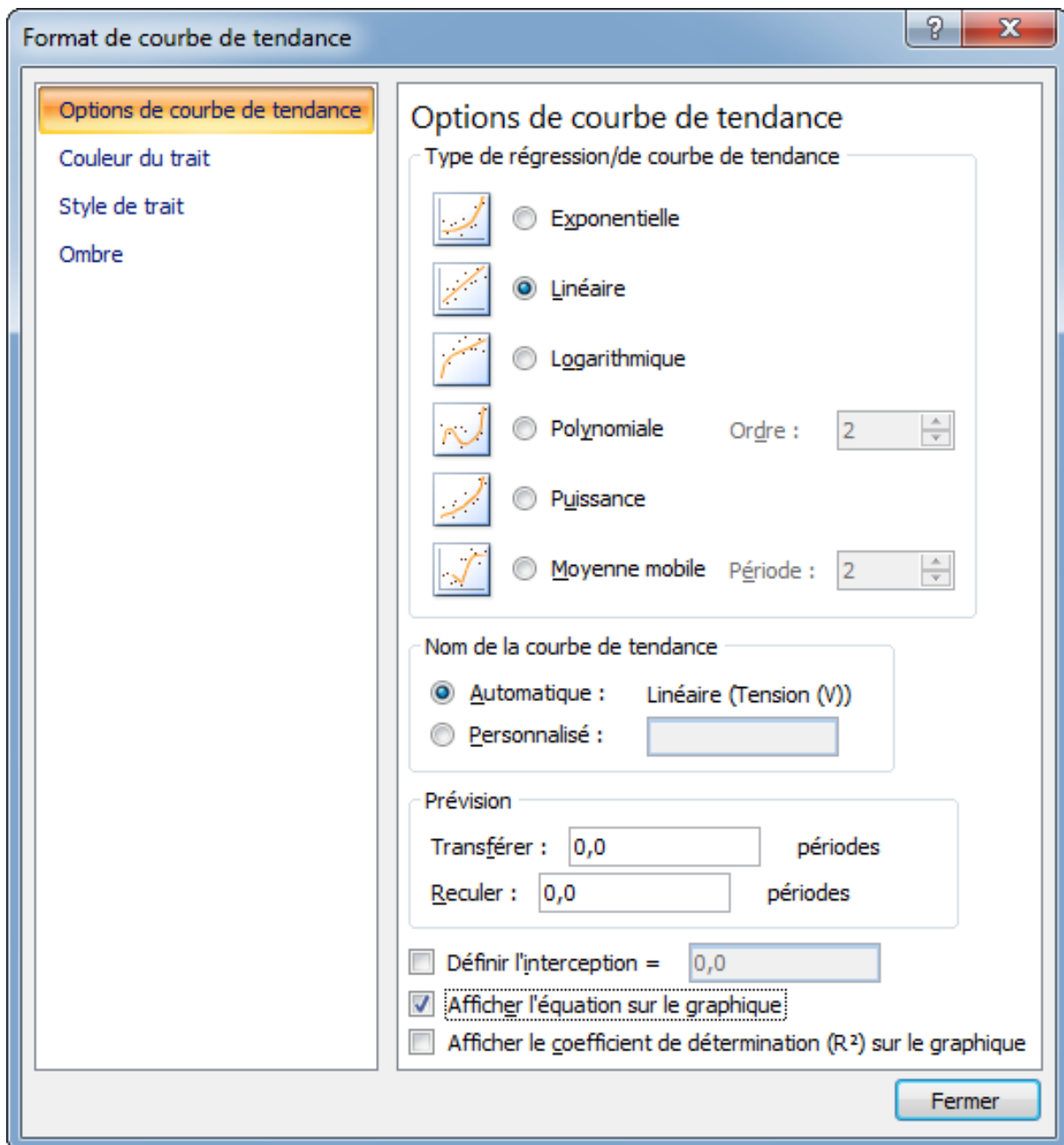


Figure 6.26 – Options de la courbe de tendance

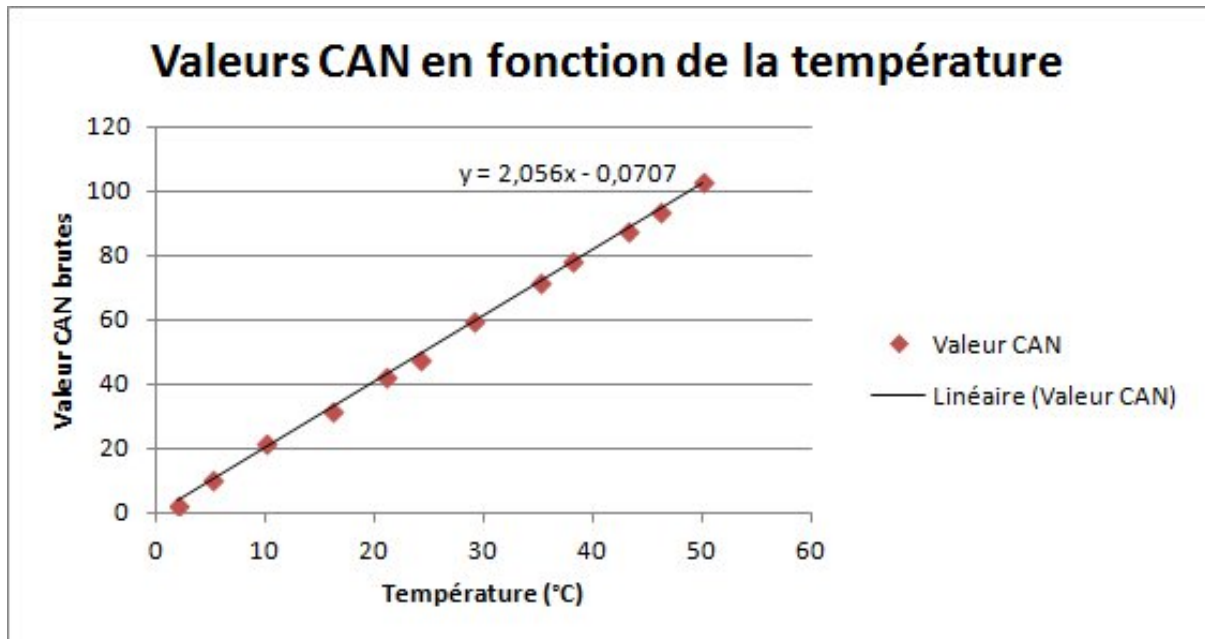


Figure 6.27 – Ajout de la courbe de tendance

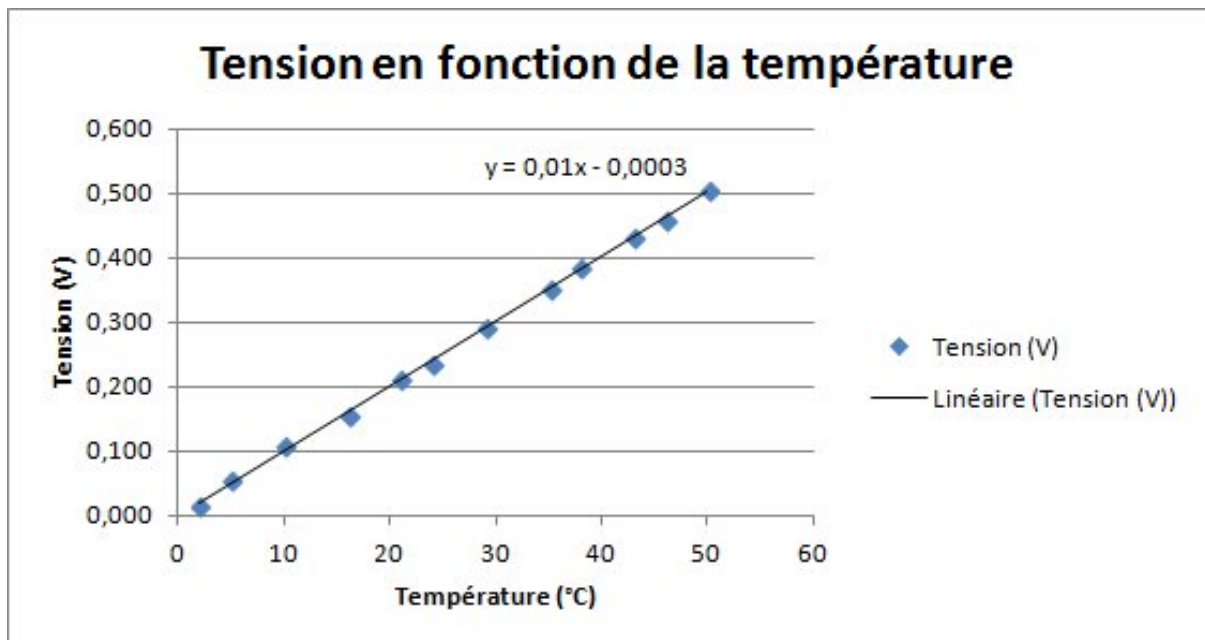


Figure 6.28 – Ajout de la courbe de tendance

Code : La formule de conversion

Et voilà ! Si l'on voulait écrire un programme plus complet, on aurait :

```
int monCapteur = 0; // Capteur sur la broche A0;
int valeur = 0;
float temperature = 0.0;

void setup()
{
  Serial.begin(9600);
}

void loop()
{
  valeur = analogRead(monCapteur);
  temperature = valeur/2.056;

  Serial.println(temperature);

  delay(500);
}
```

Code : Lire la température avec un capteur étalonné

[[information]] | Et si jamais notre coefficient constant n'est pas négligeable ?

Eh bien prenons un exemple ! Admettons qu'on obtienne la caractéristique suivante : $y = 10x + 22$. On pourrait lire ça comme "ma valeur lue par le CAN est égale à 10 fois la valeur physique plus 22". Si on manipule l'équation pour avoir x en fonction de y , on aurait :

$$\rightarrow y = 10x + 22$$

$$y - 22 = 10x$$

$$x = \frac{y-22}{10} \leftarrow$$

Dans le code, cela nous donnerait :

```
void loop()
{
  valeur = analogRead(monCapteur);
  temperature = (valeur-22)/10;

  Serial.println(temperature);

  delay(500);
}
```

Code : Cas d'une constante non négligeable

6.3 Des capteurs plus évolués

Comme nous avons pu le voir plus tôt, certains capteurs transmettent l'information sous forme d'une donnée électrique qui varie : la résistance ou la tension. Cependant, certains capteurs envoient l'information de manière "codée", afin qu'elle soit plus résistante au bruit (perturbations) et garantir le signal transmis. Parmi ces méthodes de transmission, on retrouve l'utilisation d'une PWM, de la fréquence ou d'un protocole de communication.

6.3.1 Capteur à sortie en modulation de largeur d'impulsion (PWM)

6.3.1.1 Principe

Vous vous souvenez de la PWM ? Nous l'avons utilisée dans le chapitre sur [les sorties analogiques](#). Dans ce type de signal, l'information est présente dans la durée de l'état haut par rapport à l'état bas. Ici, notre capteur va donc de la même façon coder l'information via une durée d'état (mais elle ne sera pas forcément relative à son état antagoniste). Il est donc nécessaire de connaître les caractéristiques du capteur pour pouvoir interpréter le signal correctement.

En effet, si on prend le signal sans rien savoir du capteur, comment déterminer ce que 20ms d'état haut signifie par exemple ? Pour cela, ce type de composant doit toujours être utilisé avec sa documentation technique, afin de déterminer des paramètres comme ses bornes inférieure et supérieure de mesure. Mais c'est aussi vrai pour les autres (si vous tombez sur une résistance variable sans rien en connaître, vous devrez vous farcir une belle séance d'étalonnage pour l'identifier !).

6.3.1.2 Utilisation

Prenons un cas simple. Imaginons que nous avons un capteur de température qui nous renvoie l'information suivante : *"La température en °C Celsius est proportionnelle à la durée d'état haut. La plage de mesure va de 0°C à 75°C pour une durée d'état haut de 0ms à 20ms"*.

Nous avons donc une relation proportionnelle entre une température et une durée. Nous pouvons alors déduire une règle mathématique pour faire la conversion de degrés/durée.

En effet, on a les équivalences suivantes :

- $0^{\circ}C \Leftrightarrow 0ms \Leftrightarrow 0\%$
- $75^{\circ}C \Leftrightarrow 20ms \Leftrightarrow 100\%$

Une simple règle de trois nous donne : $x = \frac{75}{20} = 3.75^{\circ}C/ms$ ce qui signifie que pour chaque milliseconde, on a 3.75°C .

Voyons maintenant comment l'utiliser...

6.3.1.2.1 Dans la pratique avec Arduino Bon, c'est pas mal on a le côté théorique de la chose, mais ça ne nous dit toujours pas comment on va l'exploiter avec notre Arduino. En effet, générer une PWM on sait faire, mais mesurer une durée d'état haut ça on ne sait pas ! Et bien rassurez-vous, comme d'habitude c'est assez simple. En effet, il existe une fonction dans le framework Arduino qui sert exactement à cela, mesurer une durée d'état haut ou bas.

Cette fonction s'appelle `pulseIn()`. Elle prend simplement en paramètres la broche sur laquelle vous voulez faire la mesure et l'état que vous voulez mesurer (HIGH ou LOW). En option, un troisième paramètre permettra de spécifier un "timeout", un temps maximal à attendre avant de décider que la mesure n'est pas possible. Si le timeout est de 2 secondes et que l'état à mesurer n'a pas commencé 2 secondes après l'appel de la fonction, alors cette dernière retournera 0. Dernier détail, l'intervalle de mesure de la fonction est de 10µs à 3 minutes et renvoie un `unsigned long` représentant la durée de l'état en **microsecondes**.

Voilà, vous savez tout pour utiliser cette fonction ! Il ne faut pas oublier cependant que la broche sur laquelle nous allons faire la mesure doit être placée en INPUT lors du `setup()` ;).

Reprenons maintenant l'exemple commencé ci-dessus.

Pour mesurer la température, nous allons mesurer l'état haut en sachant que celui-ci sera proportionnel à la température. Un code simple serait donc :

```
const char capteur = 2; // en admettant que le capteur de température soit sur la broche 2

void setup()
{
  pinMode(capteur, INPUT);
  Serial.begin(9600); // pour afficher la température
}

void loop()
{
  unsigned long duree = pulseIn(capteur, HIGH, 25000);
  // dans notre exemple la valeur est dans l'intervalle [0, 20000]
  float temperature = duree*0.00375; // 3.75 °C par ms donc 0.00375 °C par µs

  /* Dans notre cas, on n'utilise pas "map()" car la fonction fait des arrondis,
  ce qui risquerait de nous faire perdre de l'informations */

  Serial.print("Duree lue : ");
  Serial.println(duree, DEC);
  Serial.print("Temperature : ");
  Serial.println(temperature);

  delay(200); // pour ne pas spammer la voie série
}
```

Code : Mise en pratique de `pulseIn`

6.3.1.2.2 Simulation de l'exemple Si comme moi vous n'avez pas de capteur retournant une PWM, voici un petit montage tout simple permettant de tester ce concept.

Pour cela, nous allons utiliser une PWM de l'Arduino ! En effet, on sait depuis le chapitre **Sorties analogiques** faire varier un rapport cyclique dans une PWM, on va donc l'appliquer ici pour tester `pulseIn()`.

Pour cela, reliez une broche PWM à la broche qui vous sert de capteur puis essayez de réaliser ce que l'on vient de voir.

[[i]] | La fréquence de la PWM via Arduino est d'environ 490Hz, ce qui signifie que la durée d'état haut pourra varier entre 0ms et 2,04ms

```
const char capteur = 2; // broche capteur
const char emetteur = 3; // broche PWM

void setup()
{
  pinMode(capteur, INPUT);
  pinMode(emetteur, OUTPUT);

  Serial.begin(9600);
}

void loop()
{
  analogWrite(emetteur, 127); // test avec une valeur moyenne : environ 1ms
  unsigned long duree = pulseIn(capteur, HIGH);

  Serial.print("Duree : ");
  Serial.println(duree, DEC); // vérifie qu'on a bien la durée attendue

  delay(250);
}
```

Code : Simulation d'un capteur PWM et exploitation de pulseIn

6.3.1.3 Étude de cas : le capteur de distance SRF05

[[i]] | Un tutoriel plus complet sur ce capteur peut être trouvé ici : <https://zestedesavoir.com/tutoriels/539/realiser-un-telemetre-a-ultrasons/>

Prenons un exemple, le télémètre ultrason SRF05 dont la doc. technique a été retranscrite [ici](#).

Ce composant est l'exemple classique du capteur renvoyant un créneau codant l'information. En effet, ce dernier mesure ce que l'on appelle un **temps de vol**. Explications ! Le SRF05 est un télémètre ultra-son. Pour mesurer une distance, il compte le temps que met une onde pour faire un aller-retour. Un chronomètre est déclenché lors du départ de l'onde et est arrêté lorsque l'on détecte le retour de l'onde (une fois que celle-ci a "rebondi" sur un obstacle). Puisque l'on connaît la vitesse de propagation (V) de l'onde dans l'air, on peut déterminer la distance (d) nous séparant de l'objet. On a donc la formule : $v = \frac{d}{t}$ soit $d = t \times v$.

[[a]] | Le temps mesuré correspond à l'aller ET au retour de l'onde, on a donc deux fois la distance. Il ne faudra pas oublier de diviser le résultat par deux pour obtenir la distance réelle qui nous sépare de l'objet.

Comme expliqué dans la documentation, pour utiliser le sonar il suffit de générer un état haut pendant 10 µs puis ensuite mesurer l'état haut généré par le sonar.

Ce dernier représente le temps que met l'onde à faire son aller-retour. Si l'onde met plus de 30ms à faire son voyage, elle est alors considérée comme perdue et la ligne repasse à LOW.

Et voilà, vous avez maintenant toutes les informations pour faire un petit programme d'essai pour utiliser ce sonar.

Ah, une dernière information... La vitesse d'une onde sonore dans l'air à 15°C est de 340 mètres par seconde. Ça pourrait être utile!

```
#####define VITESSE 340 // vitesse du son 340 m/s

const int declencheur = 2; // la broche servant à déclencher la mesure
const int capteur = 3; // la broche qui va lire la mesure

void setup()
{
  pinMode(declencheur, OUTPUT);
  pinMode(capteur, INPUT);

  digitalWrite(declencheur, LOW);
  Serial.begin(9600);
}

void loop()
{
  digitalWrite(declencheur, HIGH);
  delayMicroseconds(10); // on attend 10 µs
  digitalWrite(declencheur, LOW);

  // puis on récupère la mesure
  unsigned long duree = pulseIn(capteur, HIGH);

  if(duree > 30000)
  {
    // si la durée est supérieure à 30ms, l'onde est perdue
    Serial.println("Onde perdue, mesure echouee!");
  }
  else
  {
    // l'onde est revenue! on peut faire le calcul
    // on divise par 2 pour n'avoir qu'un trajet (plutôt que l'aller-retour)
    duree = duree/2;
    float temps = duree/1000000.0; // on met en secondes
    float distance = temps*VITESSE; // on multiplie par la vitesse, d=t*v

    Serial.print("Duree = ");
    Serial.println(temps); // affiche le temps de vol d'un trajet en secondes
    Serial.print("Distance = ");
    Serial.println(distance); // affiche la distance mesurée
  }

  delay(250);
}
```

```
}
```

Code : Utilisation d'un capteur à ultrasons

6.3.2 Capteur à signal de sortie de fréquence variable

Voyons maintenant un autre type de sortie très similaire à celui vu ci-dessus, la fréquence. Les capteurs de ce type vont donc vous délivrer une fréquence variable en fonction de la valeur mesurée. Je ne vais pas vous mentir, je n'ai pas d'exemple en tête ! Cependant, il est facile d'imaginer comment les utiliser en prenant en compte ce que l'on vient de voir pour le capteur renvoyant une PWM.

En effet, considérons un capteur nous envoyant un signal de type "créneau" à une fréquence f . Un créneau possède en théorie une durée d'état haut égale à la durée de l'état bas. Si on fait donc une mesure de cette durée d'état haut via `pulseIn()` vue précédemment, on peut aisément déduire la période (T) du signal (qui sera égale à deux fois la valeur lue) et ainsi la fréquence puisque $f = 1/T$.

De manière programmatrice, on obtiendra donc le code suivant :

```
const char capteur = 2; // broche sur laquelle est branchée le capteur

void setup()
{
  pinMode(capteur, INPUT);

  Serial.begin(9600);
}

void loop()
{
  unsigned long duree = pulseIn(capteur, HIGH); // ou LOW, ce serait pareil !
  duree = duree*2; // pour avoir la période complète

  // hop! on calcule la fréquence !
  float frequence = 1.0/duree;
  // passe la fréquence en Hz (car la période était mesurée en µs)
  frequence = frequence*1000000;
  Serial.print("Frequence = ");
  Serial.println(frequence);

  delay(250);
}
```

Code : Mesurer une fréquence

6.3.2.1 Exemple / Exercice

Afin de mettre tout cela en pratique, je vous propose un petit exercice pour mettre en œuvre ce dernier point. Peu de matériel est à prévoir.

6.3.2.1.1 Principe Pour cet exercice, je vous propose d'émuler le comportement d'un capteur générant une fréquence variable.

Nous allons utiliser un potentiomètre qui va nous servir de "variateur". Ensuite nous allons utiliser une fonction propre à Arduino pour générer une fréquence particulière qui sera l'image multipliée par 10 de la valeur mesurée du potentiomètre. Enfin, nous allons "reboucler" la sortie "fréquence" sur une entrée quelconque sur laquelle nous mesurerons cette fréquence.

C'est clair ? J'espère !

6.3.2.1.1.1 La fonction tone() Pour faire cette exercice vous connaissez déjà tout à une chose près : Comment générer une fréquence. Pour cela, je vous propose de partir à la découverte de la fonction `tone()`. Cette dernière génère une fréquence sur une broche, n'importe laquelle. Elle prend en paramètre la broche sur laquelle le signal doit être émis ainsi que la fréquence à émettre. Par exemple, pour faire une fréquence de 100Hz sur la broche 3 on fera simplement : `tone(3, 100);`.

Un troisième argument peut-être utilisé. Ce dernier sert à indiquer la durée pendant laquelle le signal doit être émis. Si on omet cet argument, la fréquence sera toujours générée jusqu'à l'appel de la fonction antagoniste `noTone()` à laquelle on passe en paramètre la broche sur laquelle le signal doit être arrêté.

[[a]] | L'utilisation de la fonction `tone` interfère avec le module PWM des broches 3 et 11. Gardez-le en mémoire ;) . Cette fonction ne peut pas non plus descendre en dessous de 31Hz.

Vous avez toutes les informations, maintenant à vous de jouer !

6.3.2.1.2 Correction Voici ma correction commentée. Comme il n'y a rien de réellement compliqué, je ne vais pas faire des lignes d'explications et vous laisser simplement avec le code et ses commentaires :evil :!

```
const char potar = 0; // potentiomètre sur la broche A0;
const char emetteur = 8; // fréquence émise sur la broche 8
const char recepteur = 2; // fréquence mesurée sur la broche 2

void setup()
{
  pinMode(emetteur, OUTPUT);
  pinMode(recepteur, INPUT);

  Serial.begin(9600);
}

void loop()
{
  // fait la lecture analogique (intervalle [0;1023] )
  unsigned int mesure = 100;

  // applique la mesure comme fréquence (intervalle [0;10230] )
  tone(emetteur, mesure*10);
```

```
// mesure la demi-période
unsigned long periode = pulseIn(recepteur, HIGH);
// pour avoir une période complète on multiplie par 2
periode = periode*2;
// transforme en fréquence
float frequence = 1.0/periode;
// passe la fréquence en Hz (car la période était mesurée en µs)
frequence = frequence*1000000;

Serial.print("Mesure : ");
Serial.println(mesure, DEC);
Serial.print("Période : ");
Serial.println(periode, DEC);
Serial.print("Fréquence : ");
Serial.println(frequence);

delay(250);
}
```

Code : Générateur et mesure de fréquence, correction

6.3.3 Capteur utilisant un protocole de communication

Certains capteurs ne renvoient pas l'information sous forme "physique" dans le sens où ils ne renvoient pas quelque chose de mesurable directement, comme un temps ou une tension. Non, ces derniers préfèrent envoyer l'information encapsulée bien au chaud dans une trame d'un protocole de communication.

Le gros intérêt de cette méthode est très probablement la résistance au "bruit". Je ne parle bien sûr pas des cris des enfants du voisin ou les klaxons dans la rue mais bien de bruit électronique. Ce dernier est partout et peut avoir des conséquences ennuyeuses sur vos mesures. Transmettre l'information par un protocole de communication est donc un moyen fiable de garantir que la donnée arrivera de manière intègre jusqu'au destinataire. De plus, on peut facilement coupler cette transmission avec un protocole de vérification simple ou compliqué (comme la vérification de parité ou un calcul de CRC).

[[a]] | Cette partie ne va pas vous enseigner comment utiliser chacun des moyens de communication que nous allons voir. En effet, il s'agit plutôt d'une introduction/ouverture sur l'existence de ces derniers. Ils seront traités de manière indépendante dans des chapitres dédiés comme le fût [la voie série](#).

6.3.3.1 Quelques protocoles de communication

6.3.3.1.1 Voie série / UART Ce protocole vous devez déjà le connaître par cœur puisque nous l'utilisons presque dans tous les chapitres! Il s'agit en effet de la voie série via `Serial`. Rien de réellement compliqué donc tellement vous êtes habitués à le voir! Ceci est une liaison point-à-point, donc seuls deux composants (l'Arduino et le capteur) peuvent être reliés entre eux directement. Elle est généralement bi-directionnelle, ce qui signifie que les deux composants reliés peuvent émettre en même temps.

6.3.3.1.2 I2C Le protocole I²C (*Inter-Integrated Circuit*) ou TWI (*Two Wire Interface*) permet d'établir une liaison de type "maître/esclave". L'Arduino sera maître et le capteur l'esclave (l'Arduino peut aussi être un esclave dans certains cas). Ainsi, l'Arduino émettra les ordres pour faire les demandes de données et le capteur, lorsqu'il recevra cet ordre, la renverra.

Ce protocole utilise 3 fils. Un pour la masse et ainsi avoir un référentiel commun, un servant à émettre un signal d'horloge (SCL) et un dernier portant les données synchronisées avec l'horloge (SDA).

Chez Arduino, il existe une bibliothèque pour utiliser l'I2C, elle s'appelle `Wire`.

On peut placer plusieurs esclaves à la suite. Un code d'adresse est alors utilisé pour décider à quel composant le maître fait une requête.

6.3.3.1.3 SPI Le SPI (*Serial Peripheral Interface*) est une sorte de combo entre la voie série et l'I2C. Elle prend le meilleur des deux mondes.

Comme en voie série, la liaison est bi-directionnelle et point-à-point¹. Cela signifie que Arduino et le capteur sont reliés directement entre eux et ne peuvent que parler entre eux. Cela signifie aussi que les deux peuvent s'envoyer des données simultanément.

Comme en I2C, la liaison est de type maître/esclave. L'un fait une demande à l'autre, le maître transmet l'horloge à l'esclave pour transmettre les données.

Cette transmission utilise 4 fils. Une masse pour le référentiel commun, un fil d'horloge (SCLK), un fil nommé MOSI (*Master Output, Slave Input*, données partant de l'Arduino et allant vers le capteur) et MISO (*Master Input, Slave Output*, données partant du capteur et allant vers l'Arduino).

Chez Arduino, il existe une bibliothèque pour utiliser le SPI, elle s'appelle ... `SPI`.

6.3.3.1.4 Protocole propriétaire Ici pas de solution miracle, il faudra manger de la documentation technique (très formateur !). En effet, si le constructeur décide d'implémenter un protocole à sa sauce alors ce sera à vous de vous plier et de coder pour réussir à l'implémenter et l'utiliser.

Vous savez maintenant tout sur les capteurs, votre Arduino peut maintenant "sentir" le monde qui l'entoure comme promis en introduction de cette partie. Mais ce n'est pas fini, il existe un grand nombre de capteurs, beaucoup trop important pour en faire une liste exhaustive dans un tutoriel comme celui-ci. Maintenant que vous pouvez percevoir le monde, passons à la suite en essayant d'interagir avec ce dernier grâce à l'utilisation des moteurs.

1. Afin d'améliorer cette voie série, il existe une autre broche nommée SS (*Slave Select*) permettant de choisir à quel composant le maître parle. Ainsi la liaison n'est plus limitée à un esclave seulement.

7 Le mouvement grâce aux moteurs

S'il y a bien une chose sur laquelle on ne peut pas faire abstraction en robotique, c'est le mouvement. Le mouvement permet d'interagir avec l'environnement, il permet par exemple à un robot de rouler ou de prendre un objet, voire d'exprimer des émotions. Bref, ce mouvement est indispensable.

Dans cette partie nous allons apprendre à donner du mouvement à nos applications. Pour cela, nous allons utiliser des moteurs. Ce sont eux qui, principalement, transforment l'énergie qu'ils reçoivent en un mouvement. Nous verrons également comment reconnaître et utiliser les différents types de moteurs.

Chaque chapitre présentera un moteur. Le premier sera à propos du **moteur à courant continu**. Il est **indispensable de le lire** car les deux autres chapitres reprendront des éléments qui seront considérés comme acquis (on ne va pas s'amuser à faire des copier/coller des mêmes notions trois fois de suite... :P)

7.1 Le moteur à courant continu

Nul doute que vous connaissez l'existence des moteurs car il en existe toute une panoplie ! Le premier qui vous viendra certainement à l'esprit sera le moteur de voiture, ou peut-être celui présent dans une perceuse électrique. Voilà deux exemples d'objets dans lesquels on peut trouver un moteur. Bien entendu, ces deux moteurs sont de type différent, il serait en effet peu probable de faire avancer votre voiture avec un moteur de perceuse électrique... et puis l'utilisation d'une perceuse intégrant un moteur de voiture de plusieurs centaines de kilos serait fastidieuse :P . Voyons donc comment fonctionne le moteur électrique le plus répandu : le moteur à courant continu...

7.1.1 Un moteur, ça fait quoi au juste ?

Commençons en douceur par l'explication de ce à quoi sert un moteur et son fonctionnement.

[[information]] |Ce chapitre n'est pas un des plus simples car il va faire apparaître des notions de mécanique qui sont indispensables pour comprendre le mouvement. Il prend en général plusieurs heures de cours pour être bien expliqué. Nous allons donc vous faire ici uniquement une introduction à la mécanique du moteur. Cependant, cette introduction présente des notions très importantes pour bien comprendre la suite, ne la négligez donc pas !

Prenons un moteur électrique des plus basiques qui soient :



Figure : Un moteur classique à courant

continu - (CC-BY-SA, [Dcaldero8983](#))

Vous avez devant vos yeux un moteur électrique tel que l'on peut en trouver dans les engins de modélisme ou dans les voitures téléguidées. Mais sachez qu'il en existe de toute sorte, allant du miniature au gigantesque, adaptés à d'autres types d'applications. Nous nous contenterons ici des moteurs électriques "basiques".

7.1.1.0.1 Transformation de l'énergie électrique en énergie mécanique Un moteur ça fait quoi ? Ça tourne ! On dit qu'un **moteur est un composant de conversion d'énergie électrique en énergie mécanique**. Les moteurs à courant continu (ce terme deviendra plus clair par la suite) transforment l'énergie électrique en énergie mécanique de rotation, pour être précis. Mais ils peuvent également servir de générateur d'électricité en convertissant une énergie mécanique de rotation en énergie électrique. C'est le cas par exemple de la dynamo sur votre vélo !

[[attention]] |Ce dernier point n'est pas à négliger, car même si dans la plupart des applications votre moteur servira à générer un mouvement, il sera possible qu'il soit actionné "à l'envers" et génère alors du courant. Il faudra donc protéger votre circuit pour ne pas l'abîmer à cause de cette "injection" d'énergie non désirée. On va revenir dessus plus loin. ;)

7.1.1.1 Principe de fonctionnement du moteur à courant continu

7.1.1.1.1 Du vocabulaire Tout d'abord, nous allons prendre une bonne habitude. Le moteur à courant continu s'appelle aussi "Machine à Courant Continu", que j'abrégerais en MCC. Le moteur à courant continu est composé de deux parties principales : le **rotor** (partie qui tourne) et le **stator** (partie qui ne tourne pas, statique). En électrotechnique (science traitant l'électricité en tant qu'énergie) le stator s'appelle aussi **inducteur** (qui fait l'action d'induire) et le rotor s'appelle l'**induit** (qui subit l'action d'induction). Sur l'image à droite, vous pouvez observer au milieu - entouré par les aimants bleu et rouge qui constituent le stator - le rotor composé de fils de cuivre enroulés sur un support lui-même monté sur un axe. Cet axe, c'est l'**arbre** de sortie du moteur. C'est lui qui va transmettre le mouvement à l'ensemble mécanique (pignons, chaîne, actionneur...) qui lui est associé en aval. Dans le cas d'un robot sur roues par exemple, on va mettre la roue sur cet axe, bien souvent par l'intermédiaire d'un réducteur qui diminue la vitesse de rotation tout en augmentant le couple. On verra tout à l'heure pour éclaircir ces termes qui doivent,

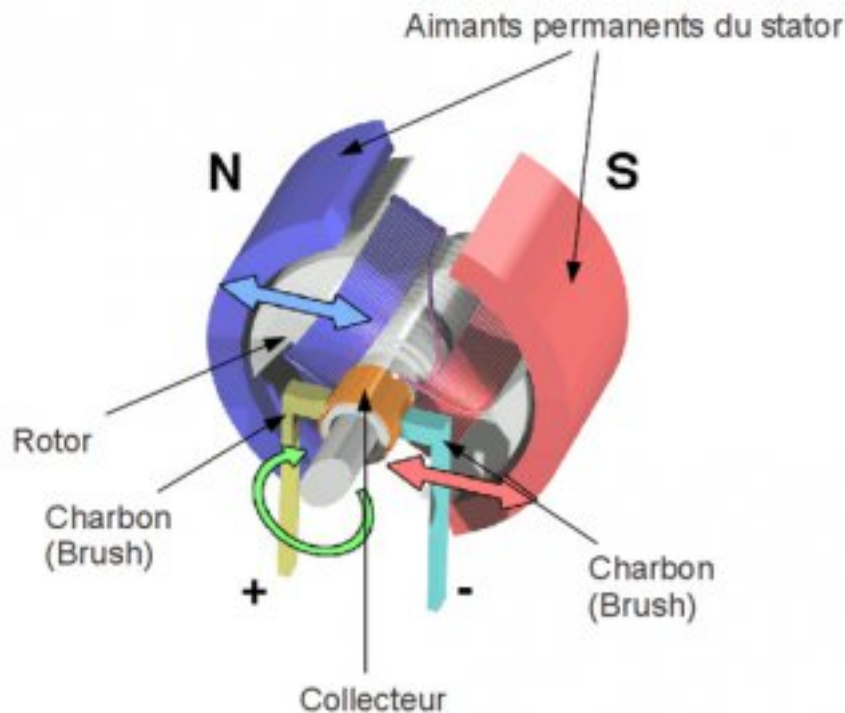


Figure 7.1 – Éclaté d'un MCC

pour l'instant, ne pas vous dire grand chose.

7.1.1.1.2 De nouvelles bases sur l'électricité Vous le savez peut-être, lorsque un courant circule dans un fil il génère un **champ magnétique**. Plus le courant qui circulera dans le fil sera grand, plus l'intensité du champs magnétique sera élevée. Lorsqu'on enroule du fil électrique sur lui même, on forme une **bobine**. Un des avantages de la bobine est que l'on "cumule" ce champ magnétique. Donc plus on a de tours de fil (des **spires**) et plus le champ magnétique sera élevé pour un courant donné.

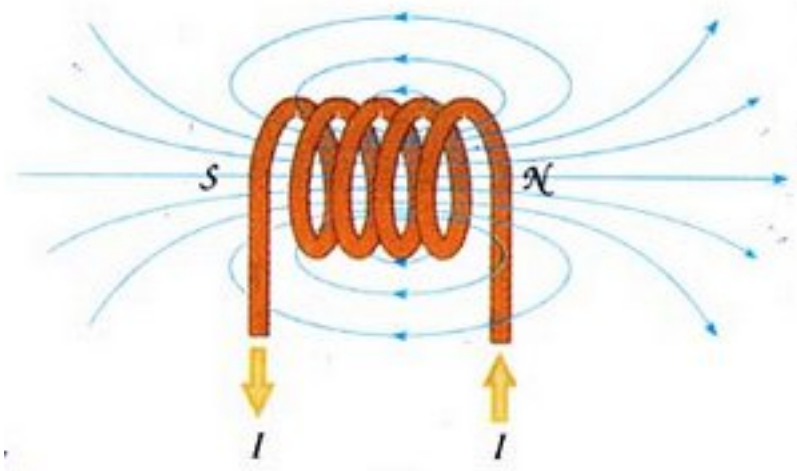


Figure 7.2 – Bobine de cuivre – champ magnétique généré représenté par les lignes bleues

En somme, on retiendra que lorsque l'on crée une bobine de fil électrique, en général du cuivre, on

additionne les champs magnétiques créés par chaque spire de la bobine. Ainsi, vous comprendrez aisément que plus la bobine contient de spires et plus le champ magnétique qu'elle induit est important. Je ne vous ai pas trop perdu, ça va pour le moment ? :) Bon, continuons.

7.1.1.1.3 Le magnétisme Oui, parlons-en. Ce sera bref, rassurez-vous. Je vais faire appel à votre expérience... avec les aimants. Vous avez tous déjà eu l'occasion d'avoir deux aimants dans la main et d'observer la résistance qu'ils émettent lorsque l'on veut les rapprocher l'un de l'autre, ou au contraire lorsqu'ils s'attirent soudainement dès qu'on les met un peu trop près. Ce phénomène est dû au champ magnétique que génèrent les aimants. Voilà un aimant permanent le plus simple soit-il :

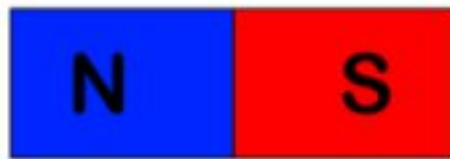


Figure 7.3 – Aimant permanent

Il possède un pôle Nord et un pôle Sud. Cet aimant génère un **champ magnétique permanent**, c'est à dire que le champ magnétique est toujours présent. C'est quelque chose de totalement invisible mais qui permet de faire des choses intéressantes.

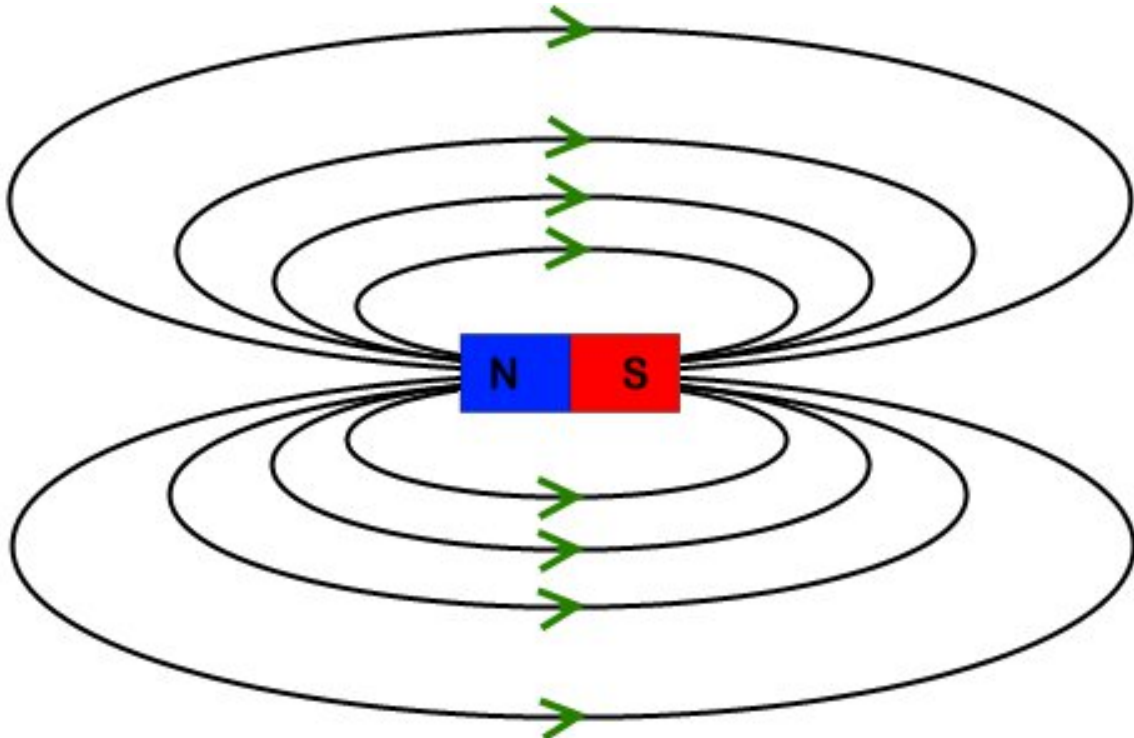


Figure 7.4 – Champ magnétique généré par un aimant permanent

Notez bien que j'ai ajouté des flèches représentatives du sens de parcours du champ magnétique, c'est important pour la suite. Bon, pour terminer mon explication sur le champ magnétique, je

vous propose d’imaginer qu’il s’agisse d’un flux invisible, un peu comme le courant. Pour se rapprocher de l’analogie avec l’eau, on peut imaginer aussi que l’aimant est une fontaine qui propulse de l’eau (champ magnétique) et qui la récupère à l’opposé de là où il l’a éjectée. Tout ça, pour en arriver à vous dire qu’approcher deux aimants avec le même pôle, ils se repoussent mutuellement (les deux fontaines éjectent de l’eau l’une contre l’autre, ce qui a pour effet de les repousser). Et on le comprend bien lorsque l’on regarde le sens du champ magnétique :

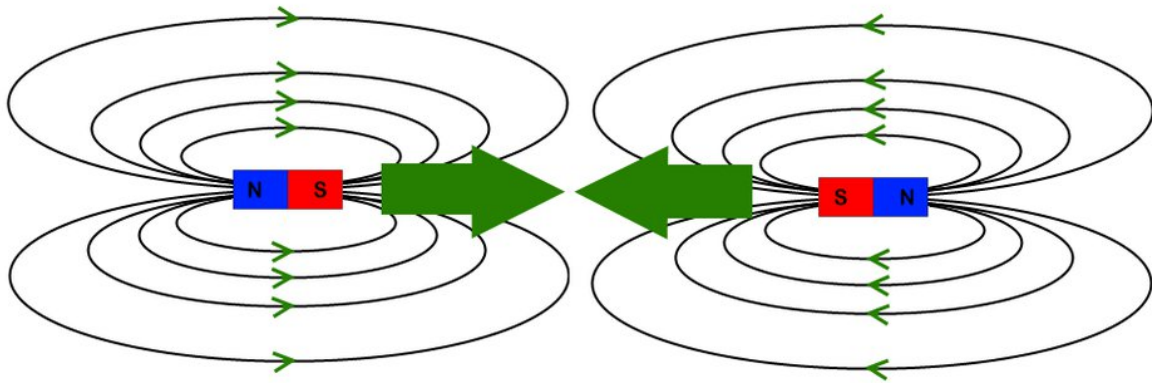


Figure 7.5 – Deux aimants permanents qui se repoussent mutuellement

En revanche, deux aimants orientés dans le même sens se rapprocheront car leur champ magnétique ira dans le sens opposé. La première “fontaine” va aspirer ce que l’autre éjecte, et l’autre va aspirer ce que la première éjecte.

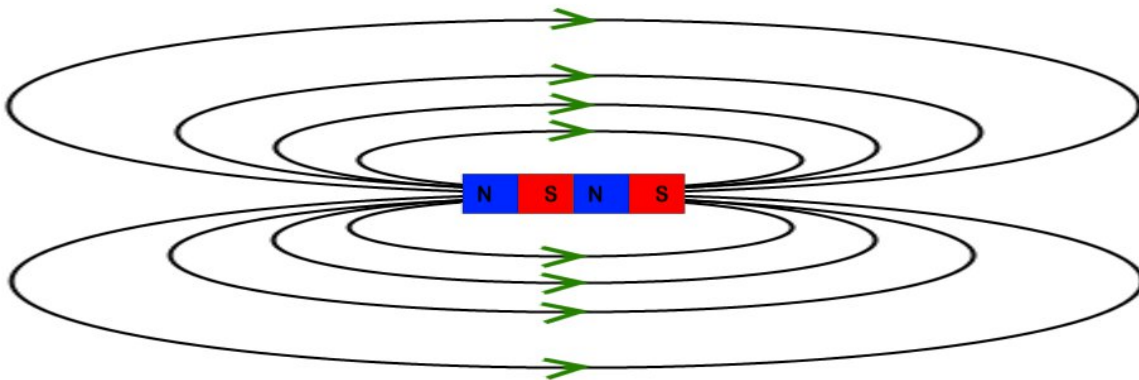


Figure 7.6 – Résultat de la mise en “série” de deux aimants permanents identiques

Par conséquent, le champ magnétique global sera plus intense. On peut alors schématiser le résultat sous la forme d’un seul aimant plus puissant.

[[question]] |Ça nous amène où tout ça ? Je comprends mieux comment fonctionne les aimants, mais pour un moteur électrique, c’est pareil ? :roll :

Eh oui, sans quoi mes explications n’auraient eu aucun sens si je vous avais dit qu’un moteur fonctionnait complètement différemment. :P Décomposons notre explication en deux parties.

7.1.1.1.4 Le stator Le stator, je l’ai dit au début, est une partie immobile du moteur. Sur l’image, il se trouve sur les côtés contre le châssis. Il forme un aimant avec ses pôles Nord et

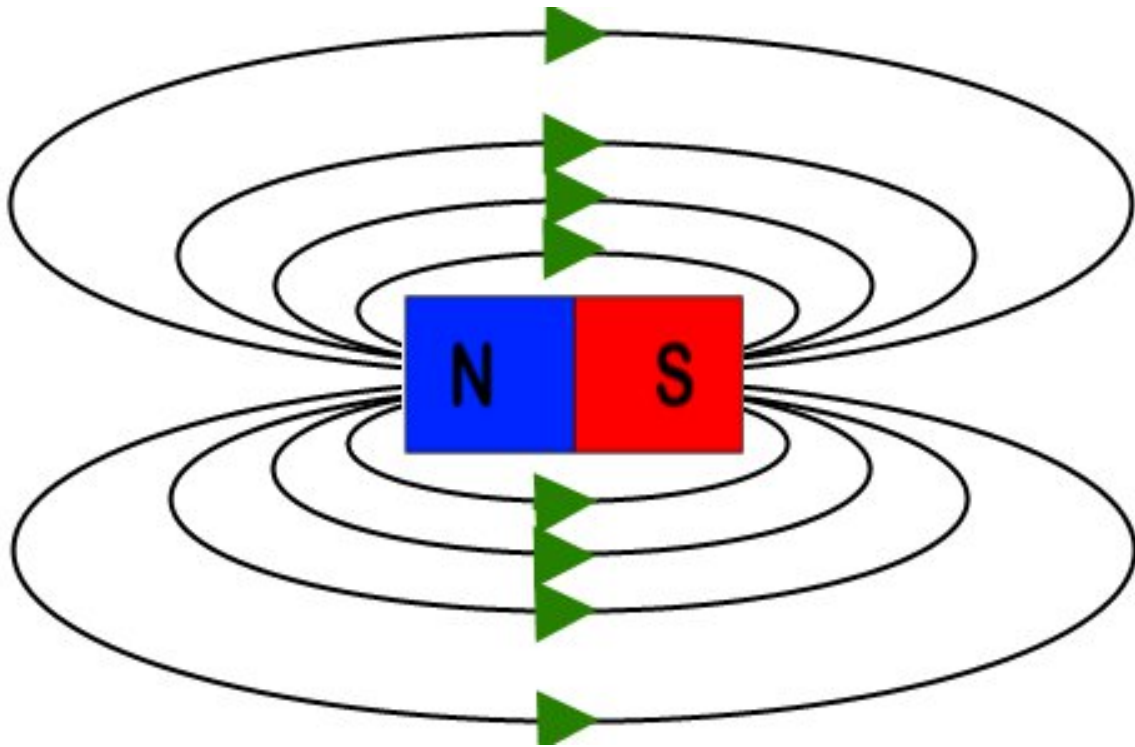


Figure 7.7 – Schématisation du résultat précédent

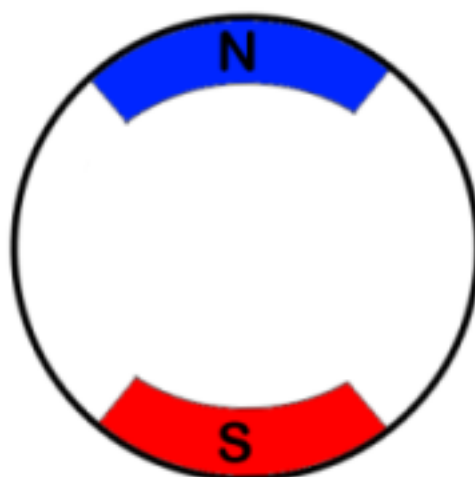


Figure 7.8 – Stator d'une MCC

Sud. Cet ensemble aimant+châssis constitue donc le stator :

Il n'y a pas plus de choses à dire, l'essentiel du phénomène de rotation créé par un moteur électrique va se jouer dans le rotor.

7.1.1.1.5 Le rotor et la mise en mouvement Le rotor, je le rappelle, est situé au centre du stator. Pour faire très simple, je vous donnerai les explications ensuite, le rotor est la pièce maîtresse qui va recevoir un courant continu et va induire un champ magnétique variable pour mettre en rotation l'arbre du rotor. Si l'on veut, oui, il s'auto-met en rotation. :roll :

[[question]] |Waaho! Avec du courant continu il arrive à créer un champ magnétique variable? o_O

Surprenant n'est-ce pas? Eh bien, pour comprendre ce qu'il se passe, je vous propose de regarder comment est constitué un rotor de MCC (j'abrège) :

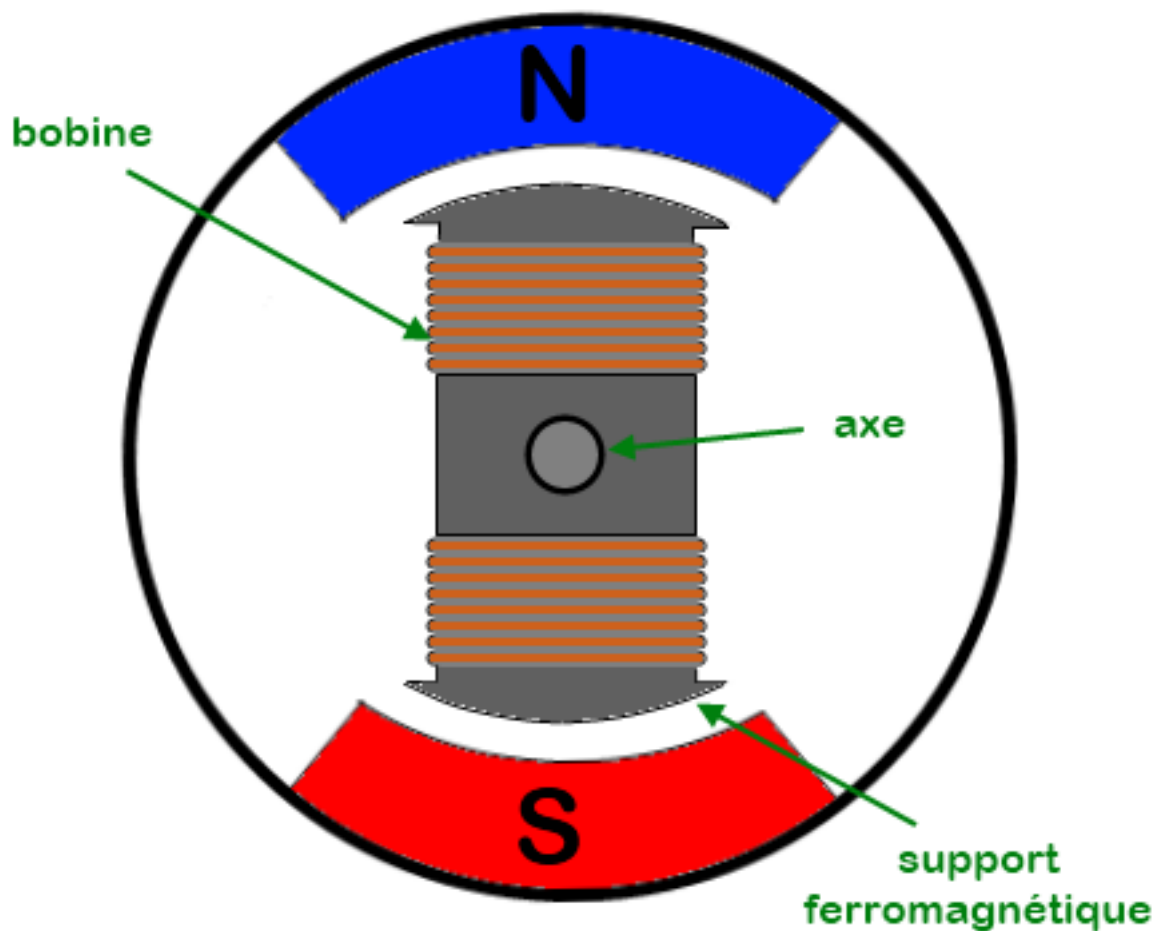


Figure 7.9 – Rotor de MCC

[[attention]] |Il s'agit bien d'un schéma de principe, normalement un moteur à courant continu est constitué de trois bobines sur son rotor. Autrement on pourrait obtenir un équilibre qui empêcherait la rotation de l'arbre du moteur, mais surtout le moteur tournerait dans un sens aléatoire. Ce qui n'est pas très adapté quand on veut faire avancer son robot. ^^

Voilà donc le rotor de notre moteur. Bien, passons à la prati...

[[question]] |Eh oh, attends!! :shock : C'est quoi ces deux bobines, comment on les alimente? o_O

Ha, j'oubliais presque! Merci de me l'avoir rappelé. Il y a en effet un élément dont nous n'avons pas encore évoqué l'existence, il s'agit du **collecteur**. Comme son nom le suggère, c'est un élément du moteur qui se situe sur l'arbre de rotation (ou l'axe du moteur si vous préférez) et qui a pour objectif de récupérer le courant afin de l'amener jusqu'aux bobines. On peut faire le schéma complet du moteur avec les bobines et le collecteur :

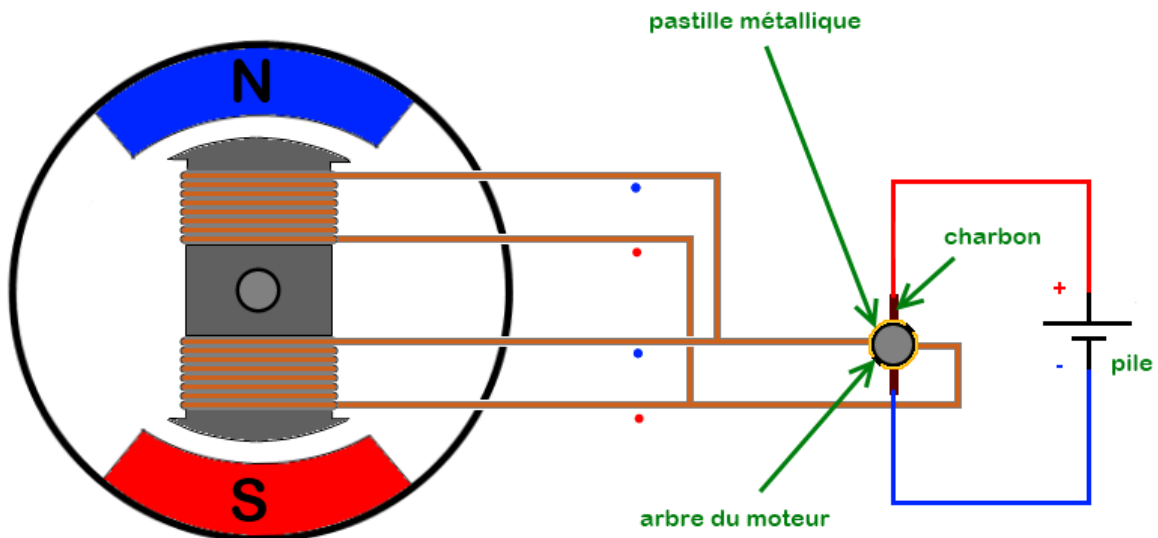


Figure 7.10 – Schéma complet du moteur

[[attention]] |Dites-vous bien qu'il ne s'agit là que d'un schéma de principe simplifié, car je le disais, les moteurs n'ayant que deux bobines n'existent pas.

Le collecteur est représenté ici sur la partie droite de l'image. Il est situé sur l'arbre du moteur (son axe). Ce collecteur est constitué de deux pastilles métalliques auxquelles sont reliées les extrémités des bobines. Le contact électrique entre la pile qui alimente le moteur et les bobines se fait par le collecteur et par des éléments "spéciaux" que l'on appelle les **charbons**. Ces deux éléments servent à amener le courant dans les bobines en faisant un simple contact électrique de toucher. C'est à dire que les charbons frottent sur les pastilles métalliques lorsque le moteur tourne.

[[question]] |Et y tourne comment ce moteur, on le saura un jour? :mad :

Ça vient, patience. ^^ Prenons la configuration du moteur tel qu'il est sur l'image précédente. Faites bien attention au sens des bobines, car si elles sont bobinées dans un sens opposé ou bien si le courant circule dans un sens opposé, le moteur ne tournera pas. J'ai donc pris le soin de mettre un point bleu et rouge, pour indiquer le sens des bobines (vous allez comprendre). Nous y voilà. ;) Sur le schéma précédent, le pôle positif de la pile est relié, via le collecteur, à l'entrée bleue des deux bobines. Leur sortie, en rouge, est donc reliée, toujours via le collecteur, à la borne négative de la pile. Vous admettrez donc, avec ce que l'on a vu plus haut, qu'il y a un courant qui parcourt chaque bobine et que cela génère un champ magnétique. Ce champ est orienté selon le sens du courant qui circule dans la bobine. Dans un premier temps, on va se retrouver avec un champ magnétique tel que celui-ci :

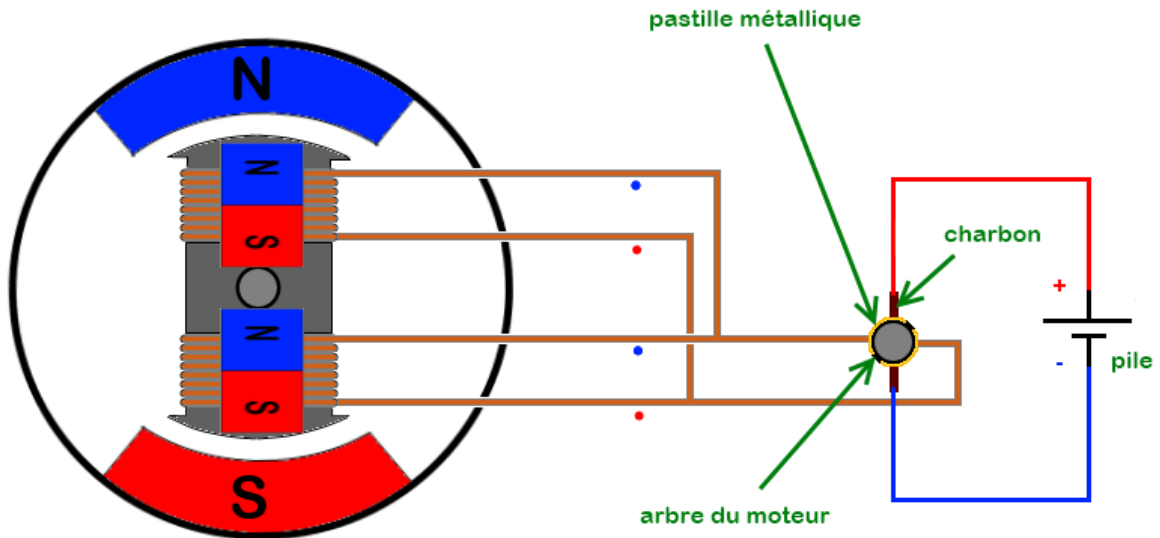


Figure 7.11 – Champ magnétique

Ce champ va être opposé aux deux aimants permanents du stator du moteur, cela va donc mettre en mouvement l'axe du rotor. Et ce mouvement est défini par le fait que deux aimants orientés par leurs pôles opposés (face nord de l'un face au nord du deuxième, idem pour le sud) se repoussent. Par conséquent, l'axe du moteur, je le disais, va se mettre à tourner jusqu'à ce que les aimants permanents du stator se retrouvent face à chacun de leur complément créé par le champ magnétique des bobines :

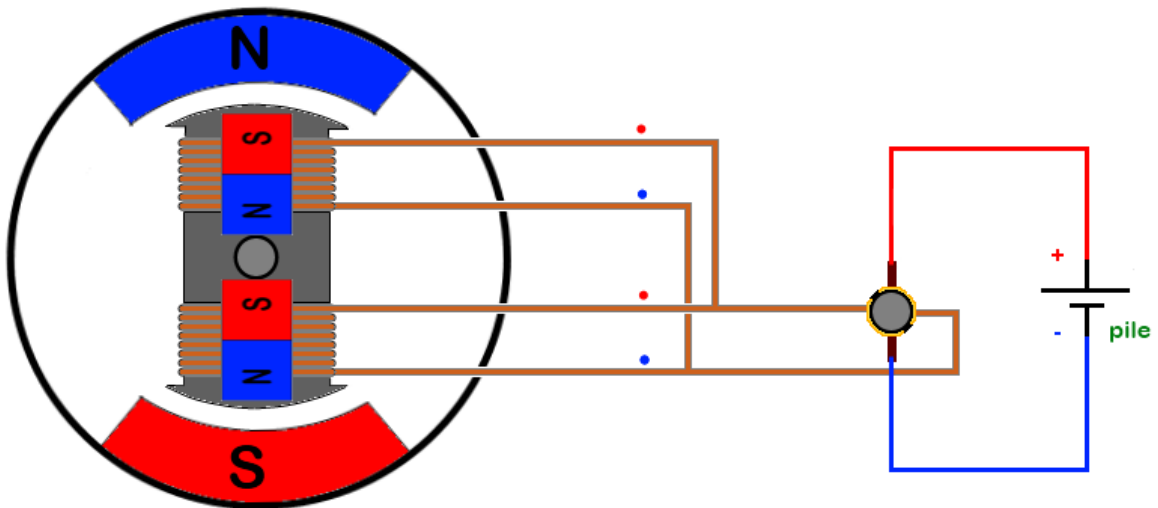


Figure 7.12 – L'axe du moteur se met à tourner

ATTENDEEEEEEZ! Ce n'est pas fini! Non, car dans cette configuration, si rien ne se passe, eh bien... rien ne se passera. ^^ Et oui, puisque le moteur est arrivé dans une phase de stabilité. En effet, chaque aimant est face au champ magnétique opposé, donc ils s'attirent mutuellement ce qui a pour effet de régir cette situation d'équilibre. L'élément qui va s'opposer à cet équilibre est le branchement des bobines du rotor. Vous ne l'avez peut-être pas remarqué, mais les bobines ne

sont plus connectées comme à la situation précédente. Le point rouge des bobines est maintenant relié au pôle positif de la pile et le point bleu au pôle négatif. Le champ magnétique généré par les bobines change alors d'orientation et l'on se retrouve avec des champs opposés. Le moteur est à nouveau en situation de déséquilibre (car les champs magnétiques se repoussent) et cela entraîne un mouvement de rotation de l'axe du moteur. Vous l'aurez compris, ces situations se répètent indéfiniment car **le moteur n'est jamais dans une configuration équilibrée**. C'est cette situation de déséquilibre qui fait que le moteur tourne.

[[attention]] | Alors attention, je le répète une dernière fois, un moteur n'ayant que deux bobines comme sur mes schémas ne peut pas fonctionner, car c'est un modèle simplifié qui engendrerait immédiatement une situation équilibrée à la mise sous tension.

Pour vous prouver que ce que je dis est vrai, voilà des photos du rotor d'un moteur à courant continu que j'avais démonté il y a bien, bien, bieeeeen longtemps : ^^

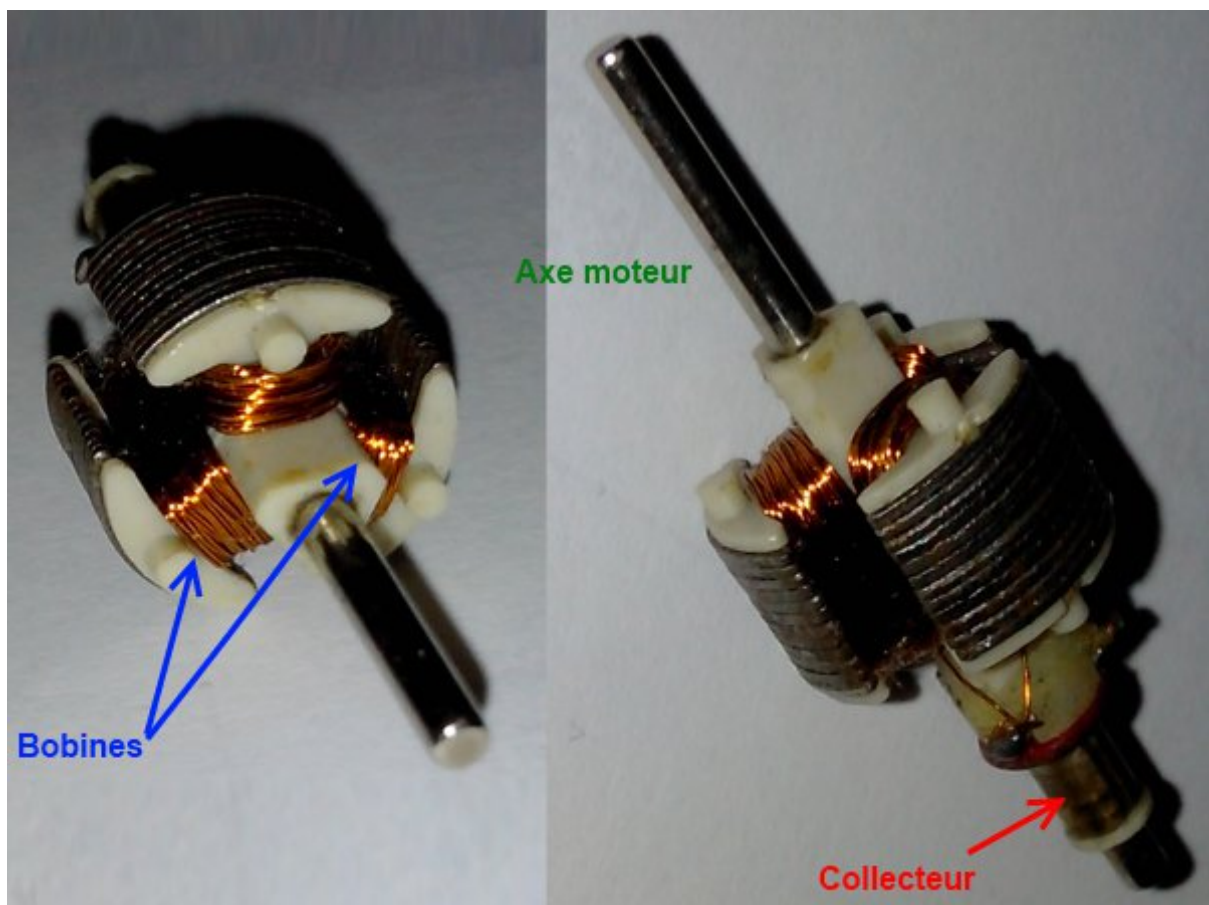


Figure 7.13 – Rotor d'un moteur à courant continu

Vous voyez ? Trois bobines et trois pastilles reliées à chacune, sur le collecteur. Bon, je ne vous refais pas les explications, vous êtes capables de comprendre comment cela fonctionne. ;)

7.1.1.2 La mécanique liée au moteur

A présent, nous allons détailler quelques notions de mécanique liées aux moteurs.

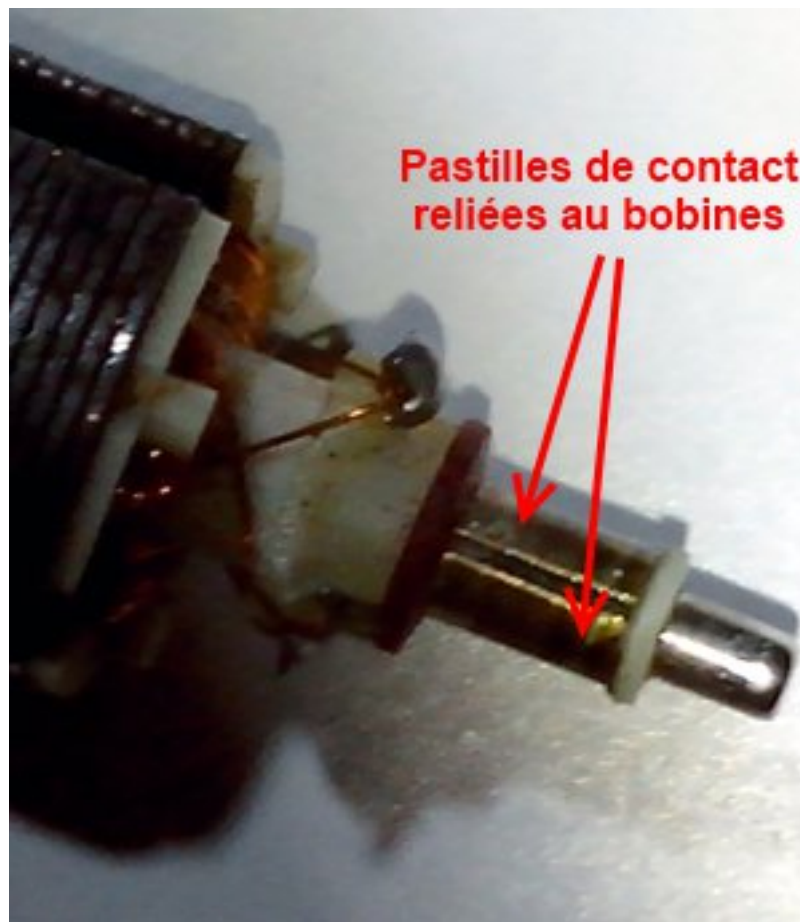


Figure 7.14 – Rotor d'un moteur à courant continu, encore

7.1.1.2.1 Le couple Le couple est une notion un peu dure à comprendre, mais on va y arriver ! Partons de son unité. L'unité du couple est le Newton-Mètre (Nm), attention j'ai bien dit Newton-Mètre et non pas Newton **par** mètre ! Cette unité nous informe de deux choses : le couple est à la fois lié à une distance (le mètre) mais aussi à une force (le Newton). Maintenant je rajoute une information : le couple s'exprime par rapport à un axe. On peut en conclure que le couple est **la capacité du moteur à faire tourner quelque chose sur son axe**. Plus le couple est élevé et plus le moteur sera capable de mettre en mouvement quelque chose de lourd. Exemple : Vous avez peut-être déjà essayé de dévisser un écrou sur une roue de voiture. Vous avez probablement remarqué que plus vous avez une clef avec un bras long (un **effet de levier** important) et plus il était facile de faire bouger l'écrou (pour le premier tour, quand il est bien vissé/coincé). Ce phénomène s'explique simplement par le fait que vous avez plus de couple avec un levier long qu'avec un levier court. Et c'est logique ! Si l'on considère que le couple s'exprime en Newton-mètre, le Newton se sera la force de vos muscles (considérée fixe dans notre cas d'étude, sauf si vous vous appelez Hulk) et le mètre sera la longueur du levier. Plus votre levier est grand, plus la distance est élevée, et plus le couple augmente. Ce qui nous permet d'introduire la formule suivante :

$$C = F \times r$$

Avec :

- C : le couple, en Newton-mètre
- F : la force exercée, en Newton
- r : le rayon de l'action (la longueur du levier si vous préférez), en mètre

On pourra également se souvenir que plus la force exercée sur l'axe de rotation d'un moteur est grande, plus il faudra un couple élevé. Et plus le couple du moteur sera élevé, moins votre futur robot aura de difficultés à supporter de lourdes charges. Cela dit, tout n'est pas parfait car plus la charge est lourde, plus la consommation électrique du moteur va augmenter. On va voir la relation qui recoupe ces deux informations.

[[information]] | Dans le système international, l'expression du couple se fait en N.m (Newton mètre), mais le commun des mortels arrive mieux à interpréter des kilos plutôt que des Newtons, donc les constructeurs prennent des raccourcis. Pour passer des Newtons en kilos, il suffit simplement de les multiplier par la constante gravitationnelle 'g' (qui vaut environ 9.81). Soit $9.81N \simeq 1kg$. Il en équivaut alors la même formule introduisant les mètres : $9.81N.m = 1kg.m$.

7.1.1.2.2 La vitesse de rotation La vitesse de rotation est mesurée par rapport à l'axe de rotation du moteur. Imaginons que le moteur entraîne son axe, lorsqu'il est alimenté par un courant, ce dernier va avoir une vitesse de rotation. Il peut tourner lentement ou rapidement. On mesure une vitesse de rotation en mesurant l'angle en radians parcourus par cet axe pendant une seconde. C'est à dire que le moteur est en fonctionnement, que son axe tourne et que l'on mesure jusqu'où va l'axe de rotation, à partir d'un point de départ fixe, en une seconde. Regardez plutôt l'image suivante pour mieux visualiser ce que je veux vous dire (comprenez que le truc gris et rond c'est le moteur que j'ai dessiné. :roll : On le voit de face et le cercle au milieu c'est son axe) :

Marquage de l'axe du moteur par un point jaune (première image). Au bout d'une seconde (seconde image), mesure de l'angle α entre la position de départ et d'arrivée du point jaune. On obtient alors la vitesse de rotation de l'axe du moteur. Cette mesure est exprimée en angle par seconde.

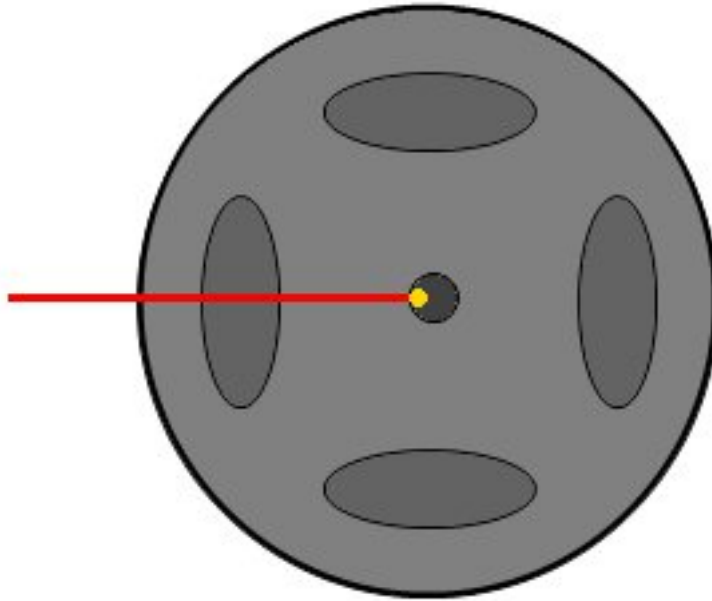


Figure 7.15 – Marquage de l'axe du moteur par un point jaune

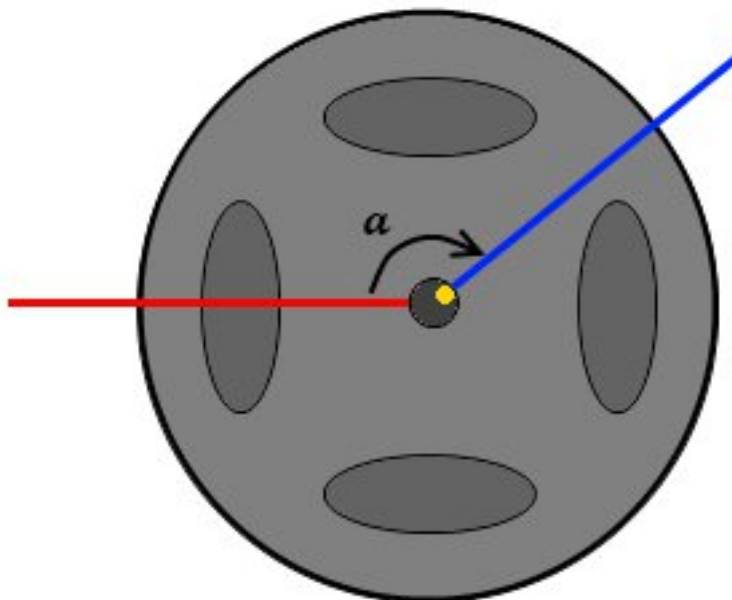


Figure 7.16 – Mesure de l'angle

Savez-vous pourquoi l'on mesure ainsi la vitesse de rotation de l'axe du moteur ? Eh bien car cette mesure est indépendante du diamètre de cet axe. Et oui, car un point éloigné du centre de l'axe du moteur a une distance beaucoup plus grande à parcourir que son homologue proche du centre de l'axe. Du coup, pour aller parcourir une distance plus grande en un temps donné il est obligé d'aller plus vite :

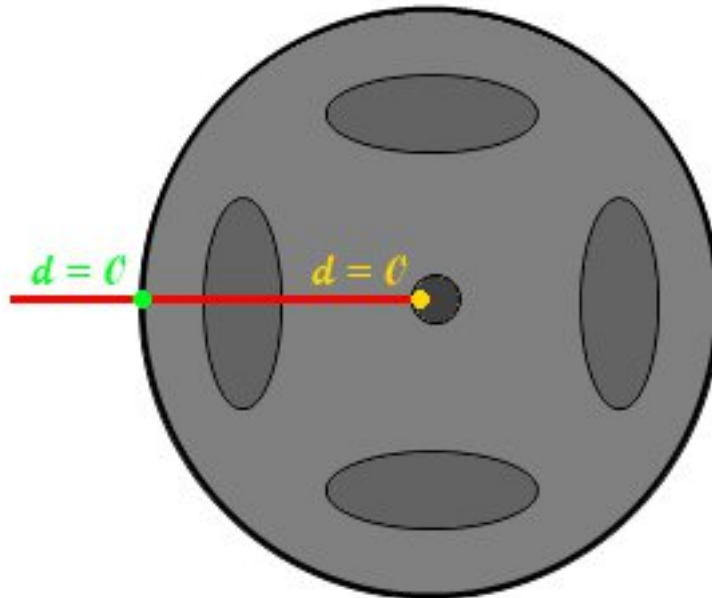


Figure 7.17 – La distance parcourue par le point jaune et vert est nulle

En prenant la mesure à partir d'un point de départ fixe, la distance parcourue par le point jaune et vert est nulle (première image). En faisant tourner l'axe du moteur pendant une seconde, on s'aperçoit que la distance parcourue par chaque point est différente (seconde image). La distance parcourue par le point vert est quasiment 20 fois plus grande que celle parcourue par le point jaune ! Et c'est pourquoi le point vert aura été plus rapide que le point jaune car la distance qu'il parcourt en un même temps est beaucoup plus grande.

En mécanique, comme on aime les choses marrantes on exprime la vitesse de rotation en radians par seconde rad/s et son symbole est le caractère grec ω , prononcez 'oméga'. Pour rappel, 360 est aux degrés ce que 2 pi est aux radians (autrement dit, une vitesse de 2pi/secondes équivaut à dire "l'axe fait un tour par seconde"). Cela se traduit par $360^\circ = 2\pi$ radian. Malheureusement, la vitesse de rotation angulaire n'est pas donnée avec les caractéristiques du moteur. En revanche, on trouve une vitesse en tour/minutes (tr/mn). Vous allez voir que pour passer de cette unité aux rad/s , c'est assez facile. En effet, on sait qu'un tour correspond à une rotation de l'axe sur 360° . Soit $1tr = 360^\circ$. Et dans une minute il y a 60 secondes. Donc l'axe tourne $\frac{1}{60}$ de tour par seconde, s'il fait un tour par minute. On peut alors établir la relation suivante :

$$1tr/mn = 360 \times \frac{1}{60} = 6^\circ/s$$

Hors, on sait que $360^\circ = 2\pi rad$, ce qui donne une nouvelle relation :

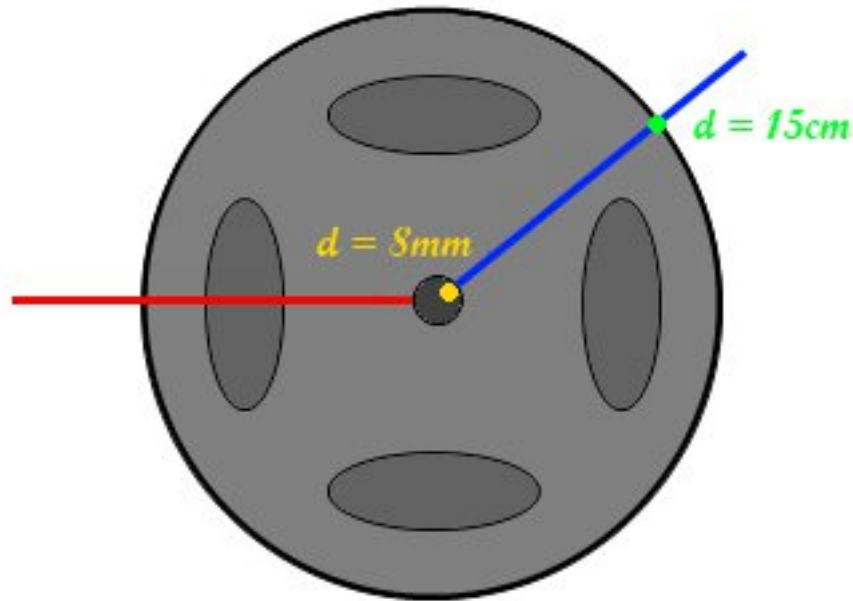


Figure 7.18 – La distance parcourue par chaque point est différente

$$1tr/mn = 2\pi \times \frac{1}{60} = \frac{\pi}{30} rad/s$$

On peut finalement donner la formule qui convertit un radian par seconde en tours par minutes :

$$1rad/s = \frac{1}{\frac{\pi}{30}} = \frac{30}{\pi} \approx 9,55tr/mn$$

[[question]] |Et je fais comment si je veux savoir à quelle vitesse ira mon robot ?

Eh bien comme je vous l'expliquais précédemment, pour répondre à cette question il faut connaître le diamètre de la roue. Prenons l'exemple d'une roue ayant 5cm de diamètre (soit 0.05 mètres) et un moteur qui tourne à 20 rad/s. Le périmètre de la roue vaut donc 15.7 cm (0.157 m) d'après la formule du périmètre d'un cercle qui est $P = 2 \times \pi \times r$, avec r le rayon du cercle. Cela signifie qu'en faisant tourner la roue sur une surface plane et en lui faisant faire un tour sur elle-même, la roue aura parcouru 0,157m sur cette surface. On admet que le moteur tourne à 20 rad/s ce qui représente donc 3.18 tours de l'axe du moteur par seconde (d'après la dernière formule que je vous ai donnée). On peut donc calculer la distance parcourue en une seconde grâce à la formule :

$$V = \frac{d}{t}$$

Avec :

- V : la vitesse en mètre par seconde (m/s)
- d : la distance en mètre (m)

— t : le temps en secondes (s)

On va donc adapter cette formule avec la distance qu'a parcouru la roue en faisant un tour sur elle-même (d_{roue}) et le nombre de tours par seconde de l'axe du moteur (t_{tour}) : $V = \frac{d_{roue}}{t_{tour}}$ On sait que $d_{roue} = 0,157m$ et que $t_{tour} = 3,18tr/s = \frac{1}{3,18}tr.s$ $V = \frac{0,157}{\frac{1}{3,18}} = 0,157 \times 3,18 V = 0,5m/s$
Le robot parcourt donc une distance de 50 centimètres en une seconde (ce qui équivaut à 1800 mètres par heure). Vous avez maintenant toutes les cartes en main pour pouvoir faire avancer votre robot à la vitesse que vous voulez !

7.1.1.2.3 Les réducteurs Un moteur électrique est bien souvent très rapide en rotation. Hors si vous avez besoin de faire un robot qui ne va pas trop vite, il va falloir faire en sorte de réduire sa vitesse de rotation. On peut très bien mettre un "frein" qui va empêcher le moteur de tourner vite, ou bien le piloter (on va voir ça toute à l'heure). Cela dit, même si on réduit sa vitesse de rotation, le moteur ne va pas pouvoir supporter des charges lourdes. Autrement dit, votre robot ne pourra même pas se supporter lui-même ! Nous avons donc besoin de couple. Et pour avoir du couple, tout en réduisant la vitesse de rotation, on va utiliser ce que l'on appelle un **réducteur**. Un réducteur est un ensemble composé d'**engrenages** qui permet de réduire la vitesse de rotation de l'axe du moteur tout en augmentant le couple de sortie. Sur l'image suivante, extraite du site de l'**Académie d'Aix Marseille**, on peut observer un ensemble moteur + réducteur + roue :

Source : http://www.technologie.ac-aix-marseille.fr/spip/spip.php?article35&id_document=51

La règle qui régit son fonctionnement indique qu'entre deux engrenages la puissance est conservée (aux pertes près qui sont dues au frottement des engrenages entre eux). Et comme la puissance mécanique est dépendante du couple et de la vitesse (partie suivante), on peut facilement passer de l'un à l'autre. Reprenons notre roue faisant 5cm de diamètre. Mettez en contact contre elle une grande roue de 10cm de diamètre (deux fois plus grande). Lorsque la petite roue fait un tour, elle va entraîner la deuxième roue plus grande qui va faire... un demi-tour. Oui car le périmètre de la grande roue est deux fois plus grand que celui de la petite. Lorsque la petite parcourt 0,157m en faisant un tour sur elle-même, la grande parcourt elle aussi cette distance mais en ne faisant qu'un demi-tour sur elle-même.

Deux roues en contact, la petite entraîne la grande dont le diamètre est deux fois plus grand que la petite (première image). Le point vert et jaune sert à repérer la rotation de chaque roue. Lorsque la petite roue fait un demi tour, la grande roue fait un quart de tour (seconde image). Si elle fait un tour complet, la grande roue ne fera qu'un demi-tour.

Ce que l'on ne voit pas sur mon dessin, c'est le couple. Hors, ce que vous ne savez peut-être pas, c'est que l'axe de la grande roue bénéficie en fait de deux fois plus de couple que celui de la petite. Car les réducteurs ont pour propriété, je le disais, de modifier le couple de sortie et la vitesse. Et ce selon la relation suivante qui donne le **rapport de réduction** :

$$R = \frac{\omega_{entree}}{\omega_{sortie}} = \frac{C_{sortie}}{C_{entree}}$$

Avec :

- R : le rapport de réduction du réducteur
- ω_{entree} : la vitesse de rotation de l'axe du moteur en entrée du réducteur
- ω_{sortie} : la vitesse de rotation de l'axe du moteur en sortie du réducteur
- C_{sortie} : couple exercé par l'axe de sortie du réducteur

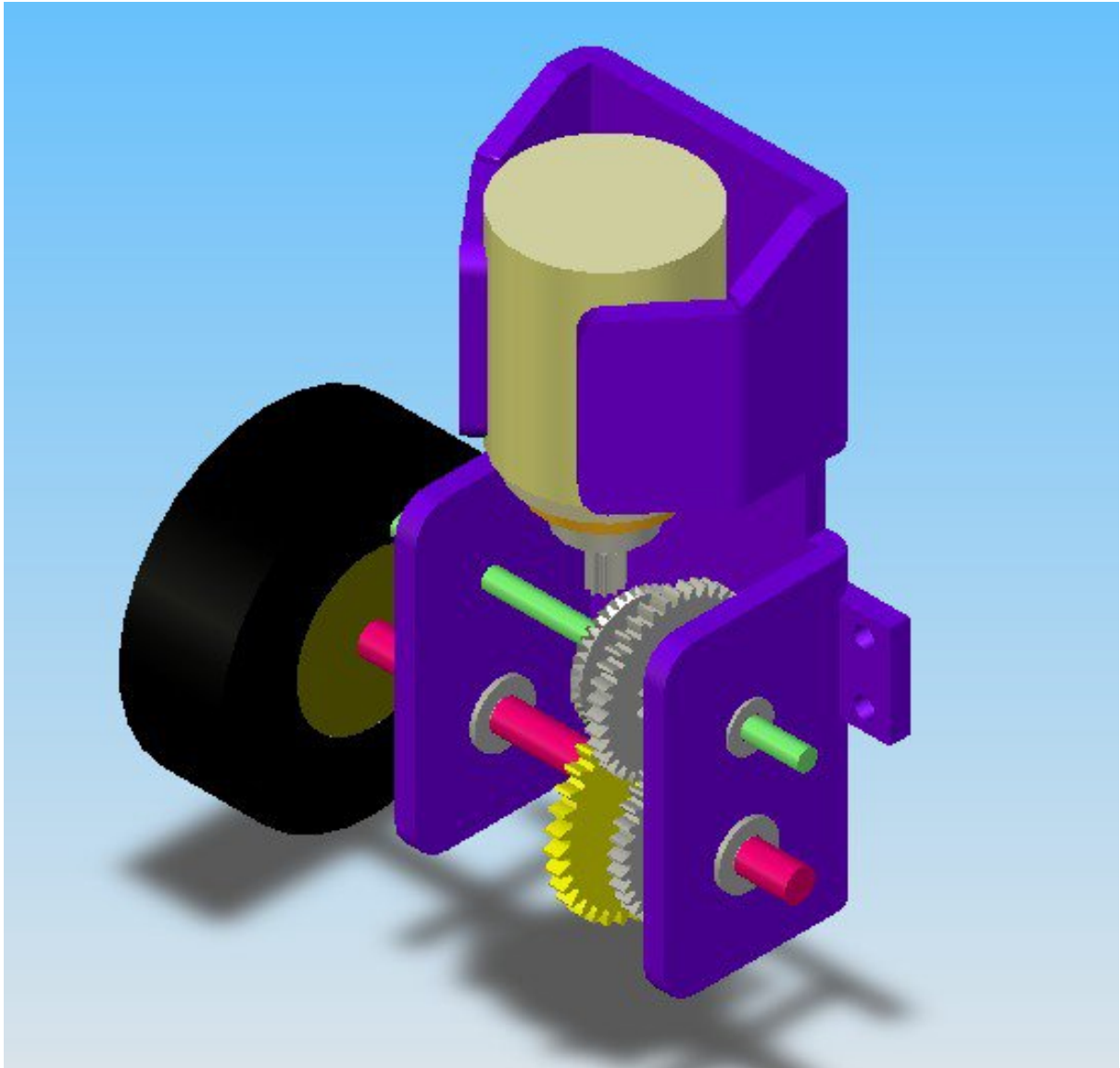


Figure 7.19 – Ensemble moteur + réducteur + roue

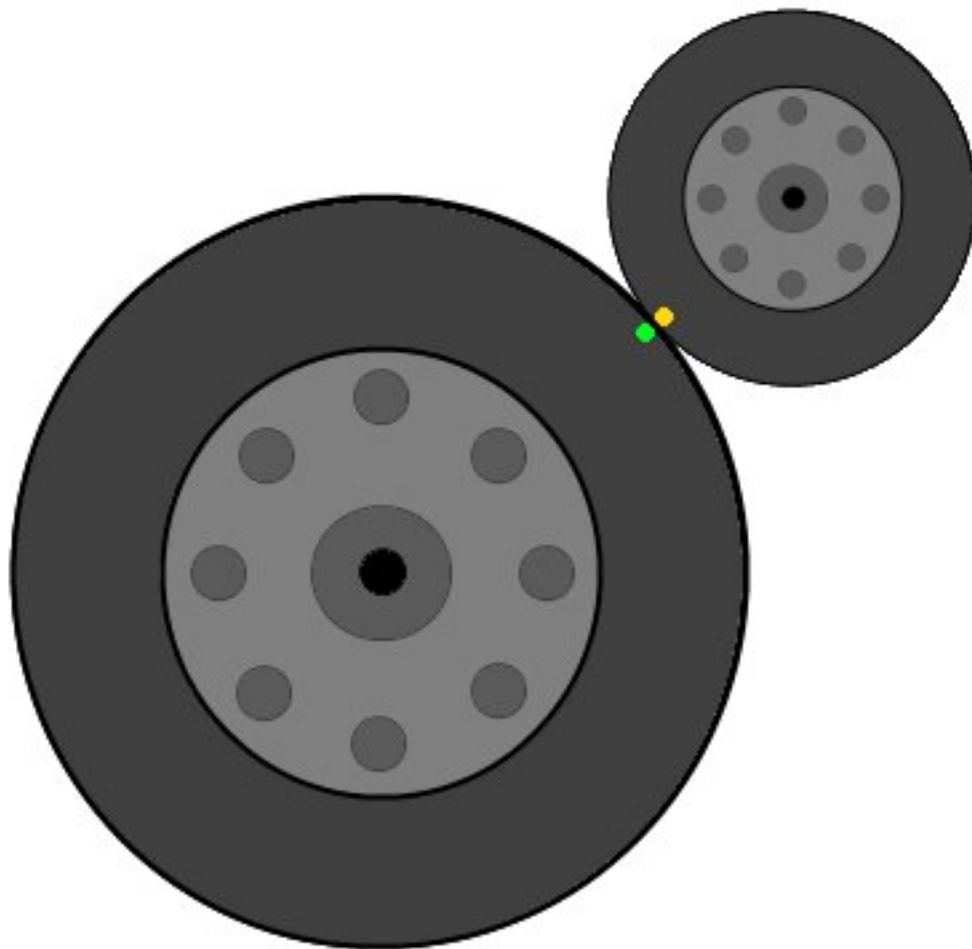


Figure 7.20 - La petite entraîne la grande

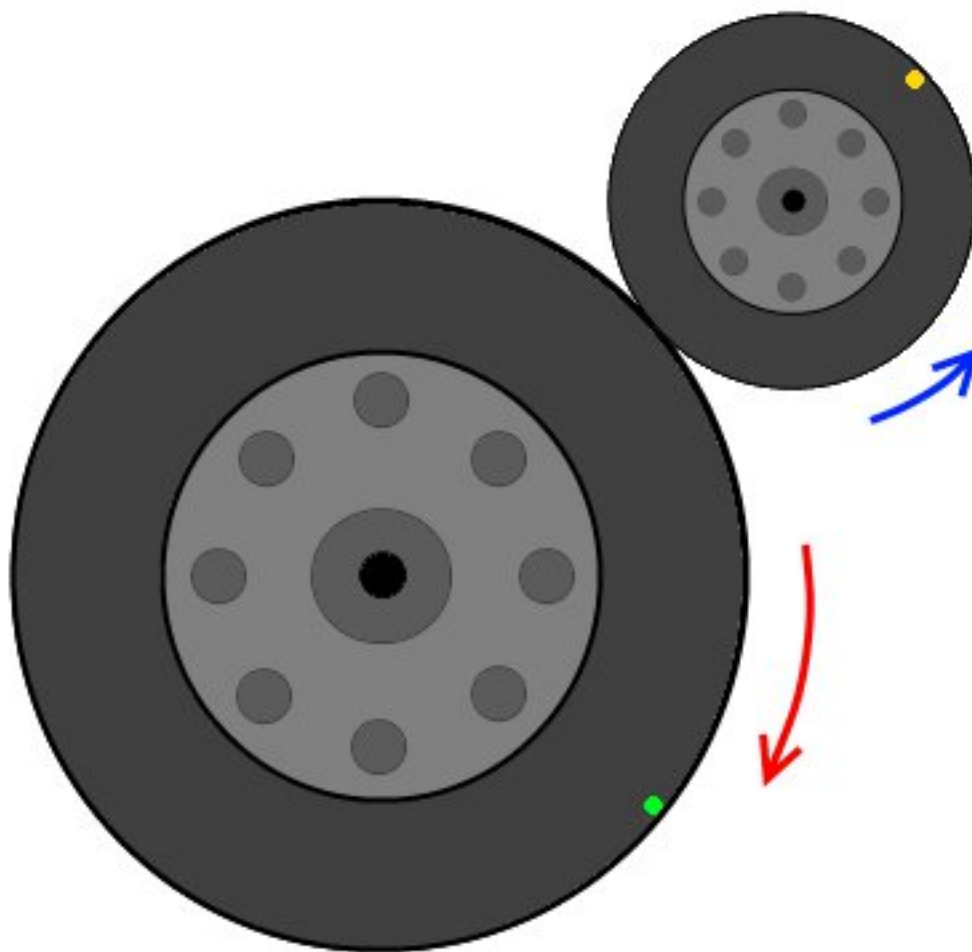


Figure 7.21 – Lorsque la petite roue fait un demi tour, la grande roue fait un quart de tour

- C_{entree} : couple exercé par l'axe du moteur, en entrée du réducteur

Un réducteur s'apparente donc à un système qui modifie deux grandeurs qui sont liées : le couple et la vitesse. On peut schématiser le fonctionnement d'un réducteur de la manière suivante :

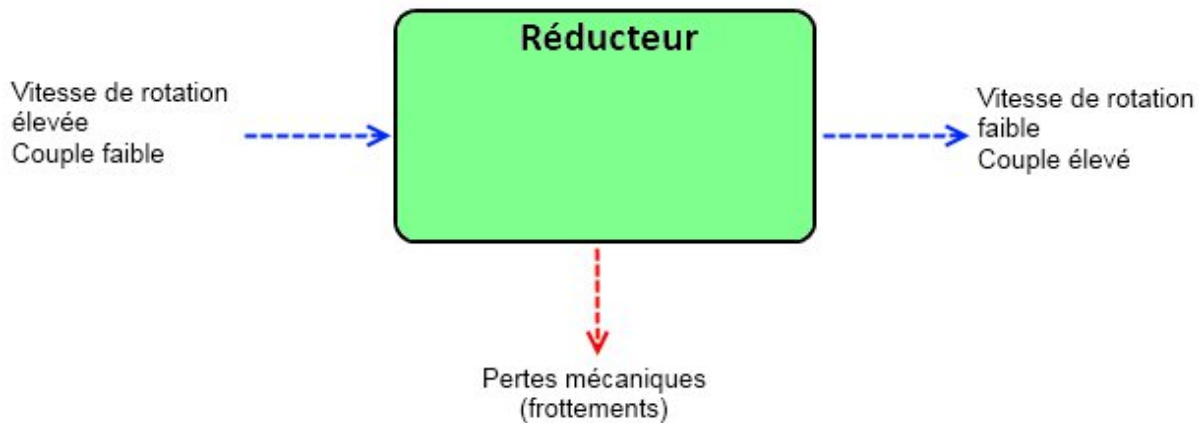


Figure 7.22 – Schéma d'un réducteur

[[question]] |C'est quoi ça, les pertes mécaniques ? :roll :

Justement, venons-en à un autre point que je voudrais aborder.

7.1.1.2.4 La puissance et le rendement Dans un moteur, on trouve deux puissances distinctes :

- La première est la **puissance électrique**. Elle représente la quantité d'énergie électrique dépensée pour faire tourner l'axe du moteur. Elle représente aussi la quantité d'énergie électrique induite lorsque le moteur tourne en générateur, c'est à dire que le moteur transforme une énergie mécanique de rotation en une énergie électrique. Elle se calcule simplement à partir de la formule suivante :

Puissance = Tension x Courant

$$P_{elec} = U \times I$$

Selon les conventions, la tension est exprimée en Volt et le courant en Ampère. Quant à la puissance, elle est exprimée en **Watt (W)**.

- La seconde est la **puissance mécanique**. Elle correspond au couple du moteur multiplié par sa vitesse angulaire :

$$P_{meca} = C \times \omega$$

Le couple doit être exprimé en Newton-Mètre (Nm) et la vitesse en radians par seconde (rad/s). Pour la puissance mécanique, il s'agit encore de Watt.

[[information]] |Une puissance (mécanique ou électrique) s'exprime habituellement en **Watts** (symbole **W**). On retrouve cependant d'autres unités telle que le Cheval Vapeur (CV), avec 1 CV qui vaut (arrondi) 735,5 W.

Mais comme dans tout système, la perfection n'existe pas, on va voir la différence qu'il y a entre la puissance mécanique et électrique, alors que *à priori* elles devraient être équivalentes. Lorsque le moteur est en fonctionnement, il génère des **pertes**. Ces pertes sont dues à différents **phénomènes électriques** ou **thermiques** (échauffement) ou tels que les **frottements mécaniques** (air, pièces en contact, magnétique). **Il y a donc une différence entre la puissance (électrique) en entrée du moteur et la puissance (mécanique) en sa sortie.** Cette différence s'exprime avec la notion de **rendement**. Le rendement est une caractéristique intrinsèque à chaque moteur et permet de définir l'écart entre la puissance d'entrée du moteur et sa puissance de sortie. Il s'exprime sans unité. Il permet également de savoir quel est le pourcentage de pertes provoquées par le moteur. Le rendement se note avec la lettre grecque *eta* (η) et se calcule grâce à la formule suivante :

$$\eta = \frac{P_{sortie}}{P_{entree}}$$

Dans le cas du moteur, on aurait alors les puissances électrique et mécanique telles quelles :

$$\eta = \frac{P_{meca}}{P_{elec}}$$

Et dans le cas où le moteur est utilisé en générateur électrique (on fait tourner l'axe à la main par exemple), la formule reste la même mais la place des puissances électrique et mécanique est inversée :

$$\eta = \frac{P_{elec}}{P_{meca}}$$

[[attention]] |Attention, le rendement est une valeur **sans unité**, on peut en revanche l'exprimer sous forme de pourcentage.

Si l'on prend un exemple : un moteur de puissance électrique 100W, ayant une puissance mécanique de 84W aura un rendement de : $\eta = \frac{P_{meca}}{P_{elec}} \eta = \frac{P_{84}}{P_{100}} \eta = 0,84$ Ce qui correspond à 84%. Sachez toutefois que le rendement ne pourra dépasser les 100% (ou 1), car **il n'existe pas de systèmes capables de fournir plus d'énergie qu'ils n'en reçoivent**. Cela dit, si un jour vous parvenez à en trouver un, vous pourrez devenir le Roi du Monde !! :Pirate :

[[information]] |Les moteurs électriques ont habituellement un bon rendement, entre 80% (0.8) et 95% (0.95). Cela signifie que pour 100W électriques injectés en entrée, on obtiendra en sortie 80 à 95W de puissance mécanique. Tandis qu'un moteur à explosion de voiture dépasse à peine les 30% de rendement !

7.1.1.3 Quelques relations

Une toute dernière chose avant de commencer la suite, il y a deux relations à connaître vis-à-vis des moteurs.

7.1.1.3.1 Lien entre vitesse et tension Dans un moteur CC, quelque soit sa taille et sa puissance, il faut savoir que la tension à ses bornes et la vitesse de sortie sont liées. Plus la tension sera élevée et plus la vitesse sera grande. Nous verrons cet aspect dans la prochaine partie. Faites attention à bien rester dans les plages de tension d'alimentation de votre moteur et ne pas les dépasser. Il pourrait griller ! En effet, vous pouvez dépasser **de manière temporaire** la tension

maximale autorisée pour donner un coup de fouet à votre moteur, mais ne restez jamais dans une plage trop élevée ! Une deuxième conséquence de cette relation concerne le moment du démarrage du moteur. En effet, la relation entre tension et vitesse n'est pas tout à fait linéaire pour les tensions faibles, elle est plutôt "écrasée" à cet endroit. Du coup, cela signifie que le moteur n'arrivera pas à tourner pour une tension trop basse. C'est un peu comme si vous aviez une tension de seuil de démarrage. En dessous de cette tension, le moteur est à l'arrêt, et au dessus il tourne correctement avec une relation de type "100 trs/min/volts" (autrement dit, le moteur tournera à 100 tours par minutes pour 1 volt, puis 200 tours par minutes pour 2 volts et etc etc... bien entendu le 100 est pris comme un exemple purement arbitraire, chaque moteur a sa caractéristique propre).

7.1.1.3.2 Lien entre courant et couple Comme nous venons de le voir, la vitesse est une sorte d'image de la tension. Passons maintenant à une petite observation : Lorsque l'on freine l'axe du moteur, par exemple avec le doigt, on sent que le moteur insiste et essaye de repousser cette force exercée sur son axe. Cela est dû au courant qui le traverse et qui augmente car le moteur, pour continuer de tourner à la même vitesse, doit fournir plus de couple. Hors, le couple et le courant sont liés : si l'un des deux augmente alors l'autre également. Autrement dit, pour avoir plus de couple le moteur consomme plus de courant. Si votre alimentation est en mesure de le fournir, il pourra éventuellement bouger, sinon, comme il ne peut pas consommer plus que ce qu'on lui donne, il restera bloqué et consommera le maximum de courant fourni.

[[attention]] |Si vous faites circuler trop de courant dans un moteur pour trop longtemps, il va chauffer. Les moteurs sont des composants sans protection. Même s'ils chauffent ils ne feront rien pour s'arrêter, bien au contraire. Cela peut mener à une surchauffe et une destruction du moteur (les bobines à l'intérieur sont détruites). Attention donc à ne pas trop le faire forcer sur de longues périodes continues.

7.1.2 Alimenter un moteur

Bon, et si nous voyions un peu comment cela se passe dans la pratique ? Je vais vous montrer comment alimenter les moteurs électriques à courant continu. Vous allez voir que ce n'est pas aussi simple que ça en a l'air, du moins lorsque l'on veut faire quelque chose de propre. Vous allez comprendre de quoi je parle...

7.1.2.1 Connecter un moteur sur une source d'énergie : la pile

Faisons l'expérience la plus simple qui soit : celle de connecter un moteur aux bornes d'une pile de 9V :

[[question]] |C'est tout ? o_O

Ben oui, quoi de plus ? Le moteur est connecté, son axe tourne, la pile débite du courant... Ha ! Voilà ce qui nous intéresse dans l'immédiat : la pile débite du courant. Oui et pas des moindres car les moteurs électriques sont bien généralement de véritables gloutons énergétiques. Si vous avez la chance de posséder un ampèremètre, vous pouvez mesurer le courant de consommation de votre moteur. En général, pour un petit moteur de lecteur CD on avoisine la centaine de milliampères. Pour un moteur un peu plus gros, tel qu'un moteur de modélisme, on trouve plusieurs centaines de milliampères de consommation. Pour des moteurs encore plus gros, on peut se retrouver avec des valeurs dépassant largement l'ampère voire la dizaine d'ampères. Revenons à notre moteur.

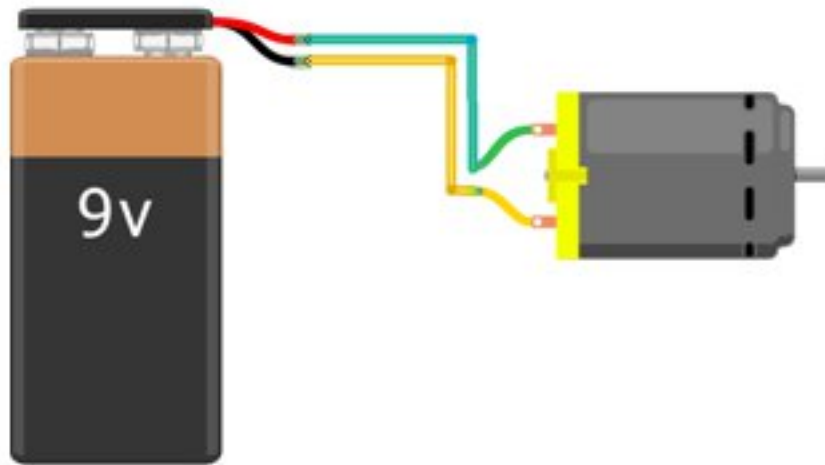


Figure 7.23 – C'est beau, ça tourne.

Lui ne consomme pas plus de 100mA à vide. Mais pour une simple pile c'est beaucoup. Et je vous garantis qu'elle ne tiendra pas longtemps comme ça ! De plus, la vitesse n'est pas réglable, le moteur tourne toujours à son maximum (si c'est un moteur fait pour tourner à 9V). Enfin, pour allumer ou arrêter le moteur, vous êtes obligé de le connecter ou le déconnecter de la pile. En somme, utiliser un moteur dans cette configuration, par exemple pour faire avancer votre petit robot mobile, n'est pas la solution la plus adaptée.

7.1.2.2 Avec la carte Arduino

Vous vous doutez bien que l'on va utiliser la carte Arduino pour faire ce que je viens d'énoncer, à savoir commander le moteur à l'allumage et à l'extinction et faire varier sa vitesse.

[[attention]] | **Ne faites surtout pas le montage qui suit, je vous expliquerai pourquoi !**

Admettons que l'on essaie de brancher le moteur sur une sortie de l'Arduino :

Avec le programme adéquat, le moteur va tourner à la vitesse que l'on souhaite, si l'on veut, réglable par potentiomètre et s'arrêter ou démarrer quand on le lui demande. C'est mieux. C'est la carte Arduino qui pilote le moteur. Malheureux ! Vous ne croyez tout de même pas que l'on va se contenter de faire ça ? ! Non, oulaaaa. C'est hyper ultra dangereux... pour votre carte Arduino ! Il est en effet impensable de réaliser ce montage car les moteurs à courant continu sont de véritables sources de parasites qui pourraient endommager, au point de vue matériel, votre carte Arduino ! Oubliez donc tout de suite cette idée de connecter directement le moteur sur une sortie de votre Arduino. Les moteurs, quand ils tournent, génèrent tout un tas de parasites qui peuvent être des surtensions très grandes par rapport à leur tension d'alimentation. De plus, le courant qu'ils demandent est bien trop grand par rapport à ce que peut fournir une sortie numérique d'une carte Arduino (environ 40 mA). Ce sont deux bonnes raisons de ne pas faire le montage précédent.

[[question]] | Mais alors, on fait comment si on peut pas piloter un moteur avec notre carte Arduino ?

Je n'ai pas dit que l'on ne pouvait pas piloter un moteur avec une carte Arduino. J'ai bien précisé *dans cette configuration*. Autrement dit, il faut faire quelque chose de plus pour pouvoir mener à terme cet objectif.

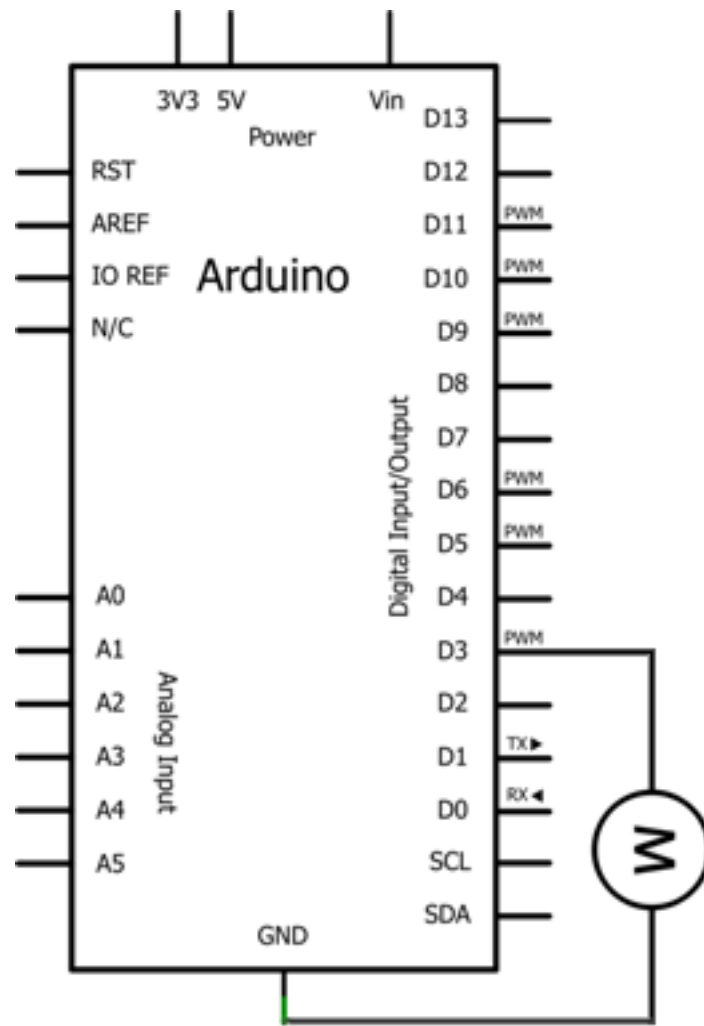


Figure 7.24 – Moteur branché sur une sortie de l'Arduino

7.1.2.3 Une question de puissance : le transistor

Souvenez-vous, nous avons parlé d'un composant qui pourrait convenir dans [ce chapitre](#). Il s'agit du **transistor**. Si vous vous souvenez de ce que je vous avais expliqué, vous devriez comprendre pourquoi je vous en parle ici. Car, à priori, on ne veut pas allumer un afficheur 7 segments. ^^ En fait, le transistor (bipolaire) est comme un interrupteur que l'on commande par un courant. Tout comme on avait fait avec les afficheurs 7 segments, on peut allumer, saturer ou bloquer un transistor pour qu'il laisse passer le courant ou non. Nous avons alors commandé chaque transistor pour allumer ou éteindre les afficheurs correspondants. Essayons de faire de même avec notre moteur :

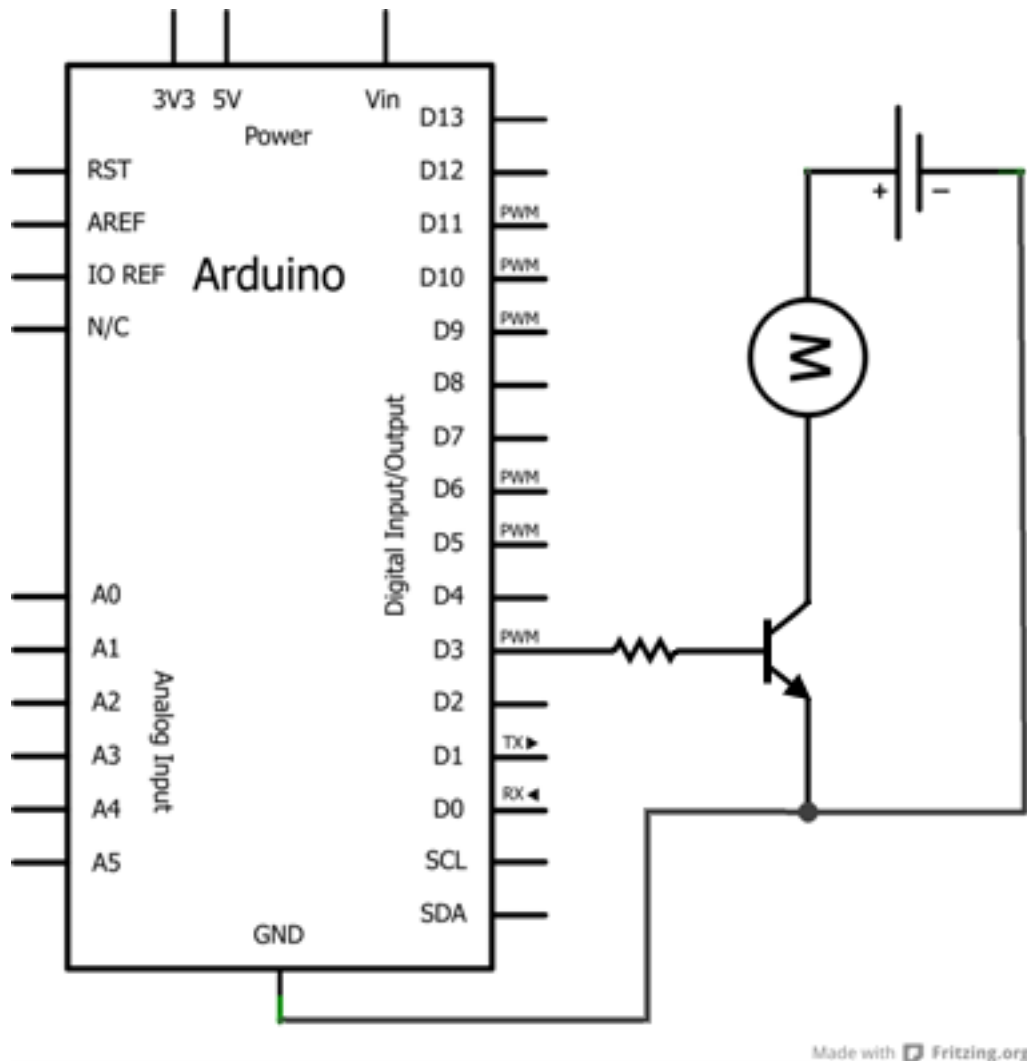


Figure 7.25 – Avec un transistor

Ici, le transistor est commandé par une sortie de la carte Arduino via la résistance sur la base. Lorsque l'état de la sortie est au niveau 0, **le transistor est bloqué et le courant ne le traverse pas**. Le moteur ne tourne pas. Lorsque la sortie vaut 1, le transistor est commandé et devient saturé, c'est-à-dire qu'il laisse passer le courant et le moteur se met à tourner. Le problème, c'est que tout n'est pas parfait et ce transistor cumule des inconvénients qu'il est bon de citer pour éviter d'avoir de mauvaises surprises :

- parcouru par un grand courant, il chauffe et peut être amené à griller s'il n'est pas refroidi

- il est en plus sensible aux parasites et risque d’être endommagé
- enfin, il n’aime pas les “hautes” tensions

Pour répondre à ces trois contraintes, trois solutions. La première consisterait à mettre un transistor qui accepte un courant assez élevé par rapport à la consommation réelle du moteur, ou bien d’adjoindre un *dissipateur* sur le transistor pour qu’il refroidisse. La deuxième solution concernant les parasites serait de mettre un condensateur de filtrage. On en a déjà parlé avec les **boutons poussoirs**. Pour le dernier problème, on va voir que l’on a besoin d’une diode.

7.1.2.3.1 Le “bon” transistor Comme je viens de vous l’expliquer, il nous faut un transistor comme “interface” de puissance. C’est lui qui nous sert d’interrupteur pour laisser passer ou non le courant. Pour l’instant, nous avons beaucoup parlé des transistors “bipolaires”. Ils sont sympas, pas chers, mais il y a un problème : ils ne sont pas vraiment faits pour faire de la commutation, mais plutôt pour faire de l’amplification de courant. Le courant qu’il laisse passer est proportionnel au courant traversant sa base. Pour les petits montages comme celui des 7 segments ce n’est pas vraiment un problème, car les courants sont faibles. Mais pour des montages avec un moteur, où les courants sont bien plus élevés, votre transistor bipolaire va commencer à consommer. On retrouvera jusqu’à plusieurs volts de perdus entre son émetteur et son collecteur, autant de volts qui ne profiteront pas à notre moteur.

[[question]] |Mais alors on fait comment pour pas perdre tout ça ?

Eh bien c’est facile ! On change de transistor ! L’électronique de puissance a donné naissance à d’autres transistors, bien plus optimaux pour les questions de fonctionnement à fort courant et en régime saturé/bloqué. Ce sont les transistors MOSFET (appelés aussi “transistor à effet de champ”). Leur symbole est le suivant :

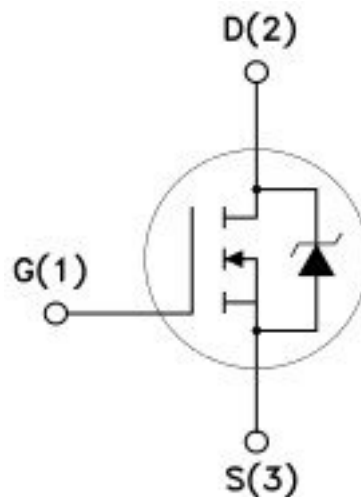


Figure 7.26 – Symbole du transistor MOSFET (canal N)

Il ressemble évidemment à un bipolaire, cela reste un transistor. Par contre il est fait pour faire de l’amplification de tension. Autrement dit, sa broche de commande (que l’on appelle “Gate”) doit recevoir une commande, une tension, donc plus besoin de résistance entre Arduino et le transistor. Son fonctionnement est simple : une différence de potentiel sur la gate et il commute (laisse passer le courant entre D (Drain) et S (Source)) sinon il bloque le courant. Facile non ? Un inconvénient cependant : ils coûtent plus chers que leurs homologues bipolaires (de un à plusieurs euros selon le modèle, le courant qu’il peut laisser passer et la tension qu’il peut bloquer). Mais

en contrepartie, ils n'auront qu'une faible chute de tension lorsqu'ils laissent passer le courant pour le moteur, et ça ce n'est pas négligeable. Il existe deux types de MOSFET, le *canal N* et le *canal P*. Ils font la même chose, mais le comportement est inversé (quand un est passant l'autre est bloquant et vice versa). Voici un schéma d'exemple de branchement (avec une résistance de pull-down, comme ça si le signal n'est pas défini sur la broche Arduino, le transistor sera par défaut bloqué et donc le moteur ne tournera pas) :

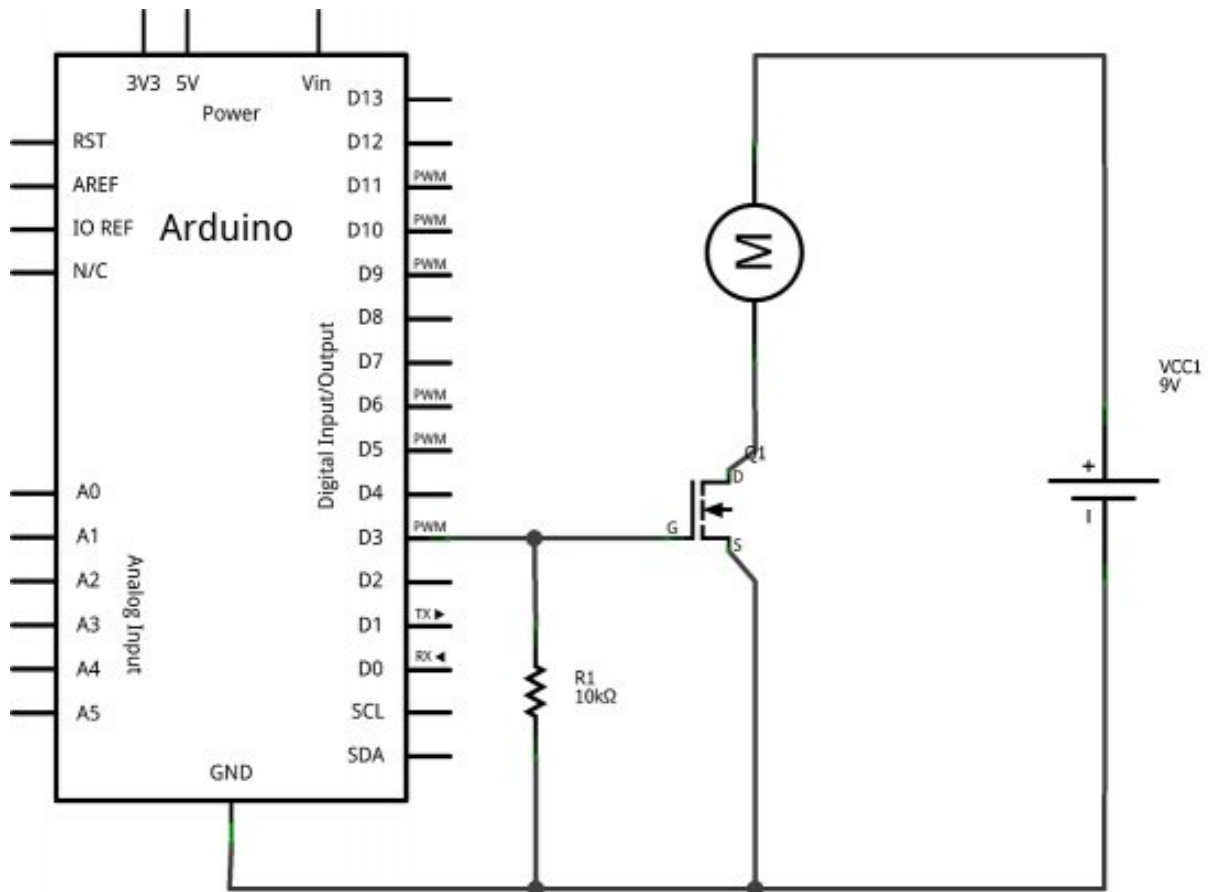


Figure 7.27 – Schéma simple de branchement

7.1.2.4 Protégeons l'ensemble : la diode de roue libre

Une diode, qu'est-ce que c'est ? Nous en avons déjà parlé à vrai dire, il s'agissait des diodes électroluminescentes (LED) mais le principe de fonctionnement reste le même sans la lumière. Une diode, dont voici le symbole :

...est un composant électronique qui ne laisse passer le courant que dans un sens (cf. [ce chapitre](#)). Vos souvenirs sont-ils à nouveau en place ? Alors, on continue ! Reprenons le schéma précédent avec le transistor piloté par l'Arduino et qui commande à son tour le moteur. Saturons le transistor en lui appliquant une tension sur sa base. Le moteur commence à tourner puis parvient à sa vitesse de rotation maximale. Il tourne, il tourne et là... je décide de couper l'alimentation du moteur en bloquant le transistor. Soit. Que va-t-il se passer ?

[[question]] |Le moteur va continuer de tourner à cause de son inertie !



Figure 7.28 – Symbole d'une diode

Très bien. Et que cela va-t-il engendrer ? Une tension aux bornes du moteur. En effet, je l'ai dit plus tôt, un moteur est aussi un générateur électrique car il est capable de convertir de l'énergie mécanique en énergie électrique même si son rôle principal est de faire l'inverse. Et cette tension est très dangereuse pour le transistor, d'autant plus qu'elle est très haute et peut atteindre plusieurs centaines de Volts (phénomène physique lié aux bobines internes du moteur qui vont se charger). En fait, le moteur va générer une tension à ses bornes et un courant, mais comme le transistor bloque la route au courant, cette tension ne peut pas rester la même et est obligée d'augmenter pour conserver la relation de la loi d'Ohm. Le moteur arrive à un phénomène de **charge**. Il va, précisément, se charger en tension. Je ne m'étends pas plus sur le sujet, il y a bien d'autres informations plus complètes que vous pourrez trouver sur internet. La question : comment faire pour que le moteur se décharge et n'atteigne pas des tensions de plusieurs centaines de Volts à ses bornes (ce qui forcerait alors le passage au travers du transistor et détruirait ce dernier) ? La réponse : par l'utilisation d'une diode. Vous vous en doutez, n'est-ce pas ? ;) Il est assez simple de comprendre comment on va utiliser cette diode, je vous donne le schéma. Les explications le suivent :

Reprenons au moment où le moteur tourne. Plus de courant ne circule dans le transistor et la seule raison pour laquelle le moteur continue de tourner est qu'il possède une inertie mécanique. Il génère donc cette fameuse tension qui est orientée vers l'entrée du transistor. Comme le transistor est bloqué, le courant en sortie du moteur va donc aller traverser la diode pour revenir dans le moteur. C'est bien, car la tension induite (celle qui est générée par le moteur) restera proche de la tension d'alimentation du moteur et n'ira pas virevolter au voisinage des centaines de Volts. Mais ça ne s'arrête pas là. Pour ceux qui l'auraient remarqué, la tension induite par le moteur est opposée à celle que fournit l'alimentation de ce dernier. Or, étant donné que maintenant on fait un bouclage de la tension induite sur son entrée (vous me suivez toujours ?), eh bien cela alimente le moteur. Les deux tensions s'opposent et cela a pour effet de ralentir le moteur. La **diode de roue libre**, c'est comme ça qu'on l'appelle, sert donc à deux choses : d'une part elle protège le transistor de la surtension induite par le moteur, d'autre part elle permet au moteur de "s'auto-freiner".

[[question]] |Et on met quoi comme diode ? o_O

Excellente question, j'allais presque oublier ! La diode que nous mettrons sera une diode *Schottky*. Ne vous laissez pas impressionner par ce nom barbare qui signifie simplement que la diode est capable de basculer (passer de l'état bloquant à passant) de manière très rapide. Dès lors qu'il y a une surtension engendrée par le moteur lorsque l'on le coupe de l'alimentation, la diode va l'absorber aussitôt avant que le transistor ait le temps d'avoir des dommages. On pourra également

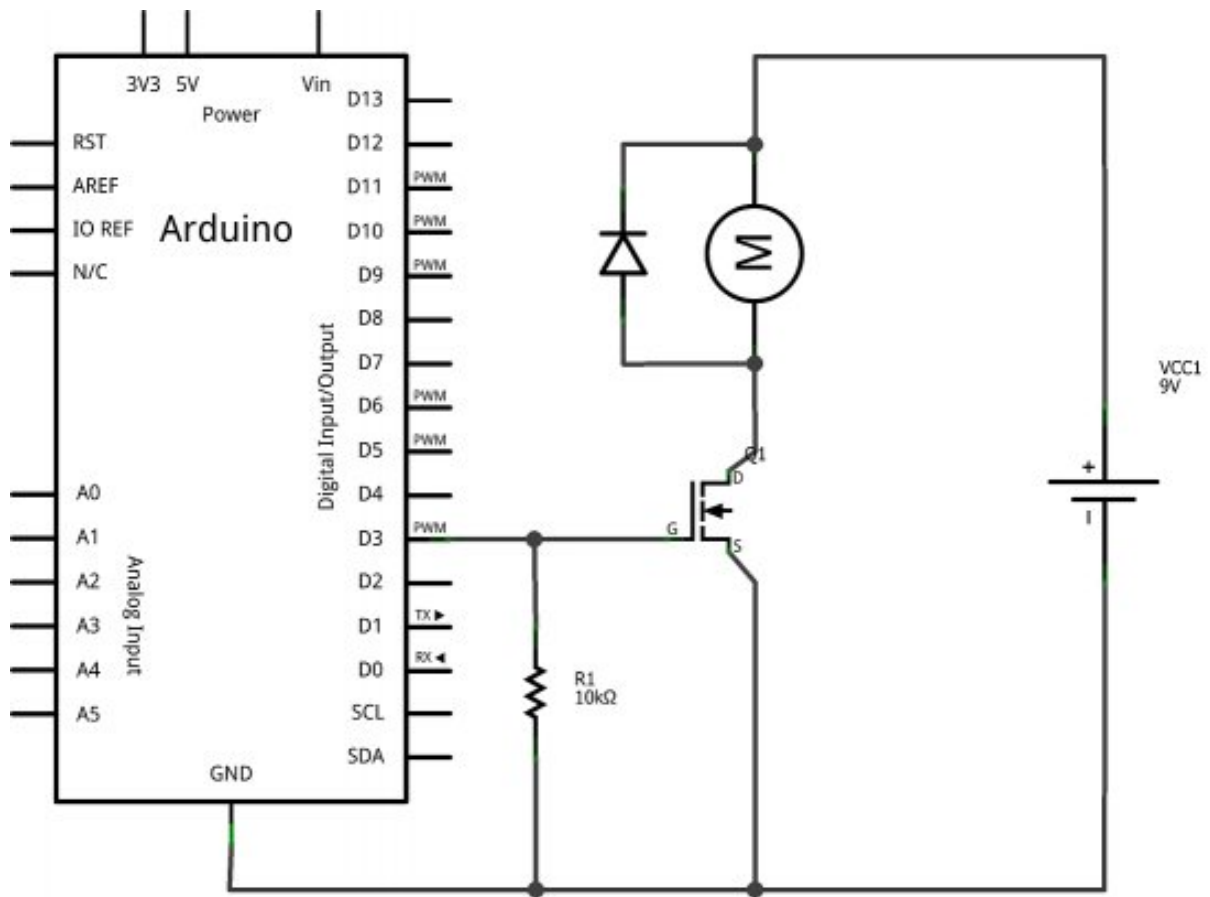


Figure 7.29 – Schéma d'utilisation de la diode

rajouter aux bornes de la diode un condensateur de déparasitage pour protéger le transistor et la diode contre les parasites. Au final, le schéma ressemble à ça :

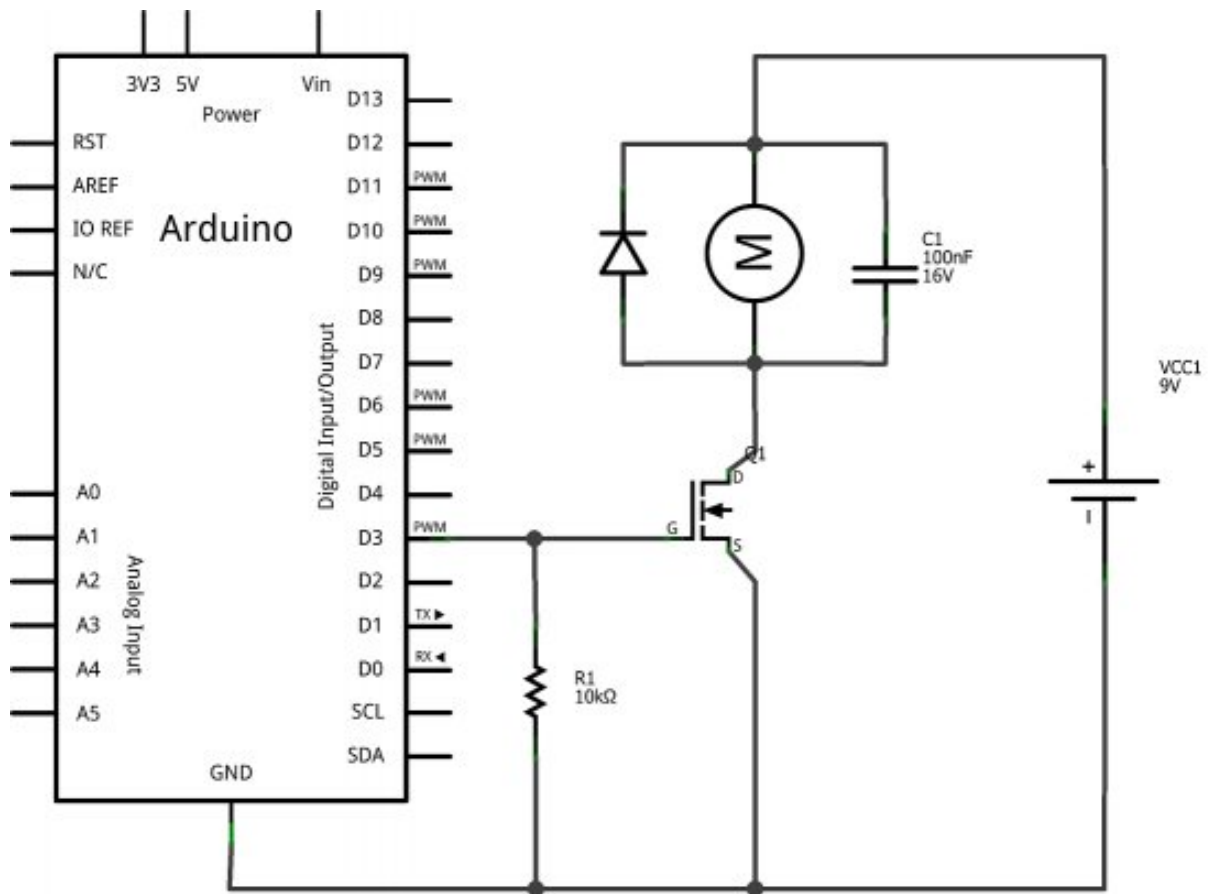


Figure 7.30 – Schéma du montage complet du moteur CC

Sa valeur devra être comprise entre 1nF et 100nF environ. Le but étant de supprimer les petits parasites (pics de tension). Bon, nous allons pouvoir attaquer les choses sérieuses ! :Pirate :

7.1.3 Piloter un moteur

[[information]] | Les montages de cette partie sont importants à connaître. Vous n’êtes pas obligé de les mettre en œuvre, mais si vous le voulez (et en avez les moyens), vous le pouvez. Je dis ça car la partie suivante vous montrera l’existence de shields dédiés aux moteurs à courant continu, vous évitant ainsi quelques maux de têtes pour la réalisation des schémas de cette page. ^^

7.1.3.1 Faire varier la vitesse : la PWM

Maintenant que nous avons les bases fondamentales pour faire tourner notre moteur sans tout faire griller (:roll :), nous allons pouvoir acquérir d’autres connaissances. À commencer par quelque chose de facile : le réglage de la vitesse de rotation du moteur. Comme nous l’expliquons dans le premier morceau de ce chapitre, un moteur à courant continu possède une relation directe entre sa tension d’alimentation et sa vitesse de rotation. En effet, plus la tension à ses bornes est élevée et plus son axe tournera rapidement (dans la limite de ses caractéristiques évidemment).

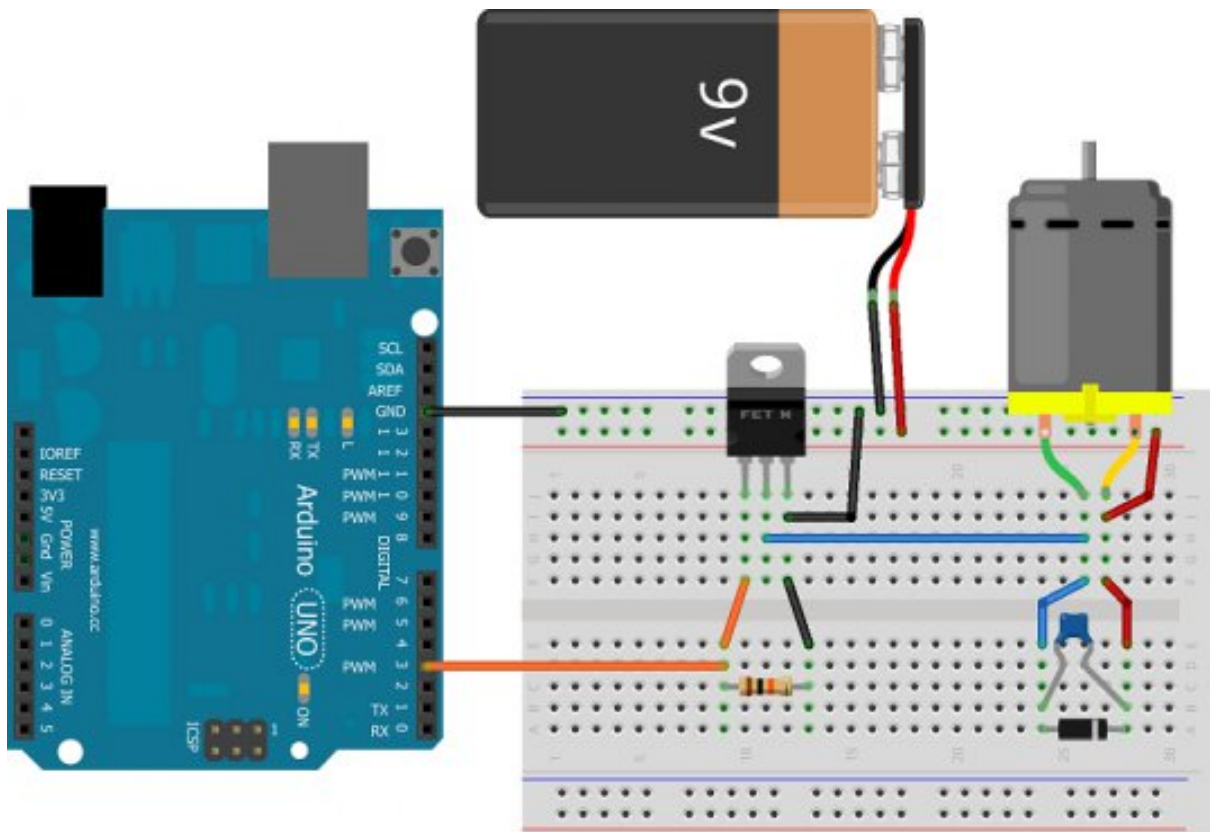


Figure 7.31 – Montage complet du moteur CC

Cependant le microcontrôleur d'Arduino n'est capable de produire que des tensions de 0 ou 5V. En revanche, il peut "simuler" des tensions variables comprises entre 0 et 5V. Encore un petit rappel de cours nécessaire sur la PWM que nous avons déjà [rencontrée ici](#) pour vous rafraîchir la mémoire. Nous sommes en mesure de produire à l'aide de notre microcontrôleur un signal carré dont le rapport cyclique est variable. Et grâce à cela, nous obtenons une tension moyenne (comprise entre 0 et 5V) en sortie de la carte Arduino. Il faut juste bien penser à utiliser les sorties adéquates, à savoir : 3, 5, 6, 9, 10 ou 11 (sur une duemilanove/UNO). Je résume : en utilisant la PWM, on va générer une tension par impulsions plus ou moins grandes. Ce signal va commander le transistor qui va à son tour commander le moteur. Le moteur va donc être alimenté par intermittences à cause des impulsions de la PWM. Ce qui aura pour effet de modifier la vitesse de rotation du moteur.

[[question]] |Mais, si le moteur est coupé par intermittences, il va être en rotation, puis va s'arrêter, puis va recommencer, etc. Ce sera pas beau et ça ne tournera pas moins vite. Je comprends pas trop ton histoire. o_O

Non, puisque le moteur garde une inertie de rotation et comme la PWM est un signal qui va trop vite pour que le moteur ait le temps de s'arrêter puis de redémarrer, on va ne voir qu'un moteur qui tourne à une vitesse réduite. Finalement, nous allons donc pouvoir modifier la vitesse de rotation de notre moteur en modifiant le rapport cyclique de la PWM. Plus il est faible (un état BAS plus long qu'un état HAUT), plus le moteur ira doucement. Inversement, plus le rapport cyclique sera élevé (état HAUT plus long que l'état BAS), plus le moteur ira vite. Tout cela couplé à un transistor pour faire passer de la puissance (et utiliser la tension d'utilisation adaptée au moteur) et nous pouvons faire tourner le moteur à la vitesse que nous voulons. Génial non ? Pour l'instant je ne vous ferai pas de démo (vous pouvez facilement imaginer le résultat), mais cela arrivera très prochainement lors de l'utilisation de l'Arduino dans la prochaine sous-partie. Le montage va être le même que tout à l'heure avec le "nouveau" transistor et sa résistance de base :

Maintenant que le moteur tourne à une vitesse réglable, il pourra être intéressant de le faire tourner aussi dans l'autre sens (si jamais on veut faire une marche arrière, par exemple, sur votre robot), voire même d'être capable de freiner le moteur. C'est ce que nous allons tout de suite étudier dans le morceau suivant en parlant d'un composant très fréquent dans le monde de la robotique : le **pont en H**.

7.1.3.2 Tourner dans les deux sens : le pont en H

Faire tourner un moteur c'est bien. Tourner à la bonne vitesse c'est mieux. Aller dans les deux sens c'est l'idéal. C'est donc ce que nous allons maintenant chercher à faire !

7.1.3.2.1 Découverte du pont en H Tout d'abord une question très simple : pourquoi le moteur tourne dans un seul sens ? Réponse évidente : parce que le courant ne va que dans un seul sens ! Pour pouvoir aller vers l'avant ET vers l'arrière il nous faut donc un dispositif qui serait capable de faire passer le courant dans le moteur dans un sens ou dans l'autre. Vous pouvez faire l'expérience en reprenant le premier montage de ce chapitre où il n'y avait que le moteur connecté sur une pile de 9V. Essayez d'inverser les deux bornes du moteur (ça ne risque rien ;)) pour observer ce qu'il se passe : le moteur change de sens de rotation. C'est dû au champ magnétique créé par les bobines internes du moteur qui est alors opposé. Reprenons notre dispositif de base avec un transistor (que nous symboliserons ici par un interrupteur). Si ce dernier est activé le moteur tourne, sinon le moteur est arrêté. Jusque là rien de nouveau. Rajoutons un deuxième transistor

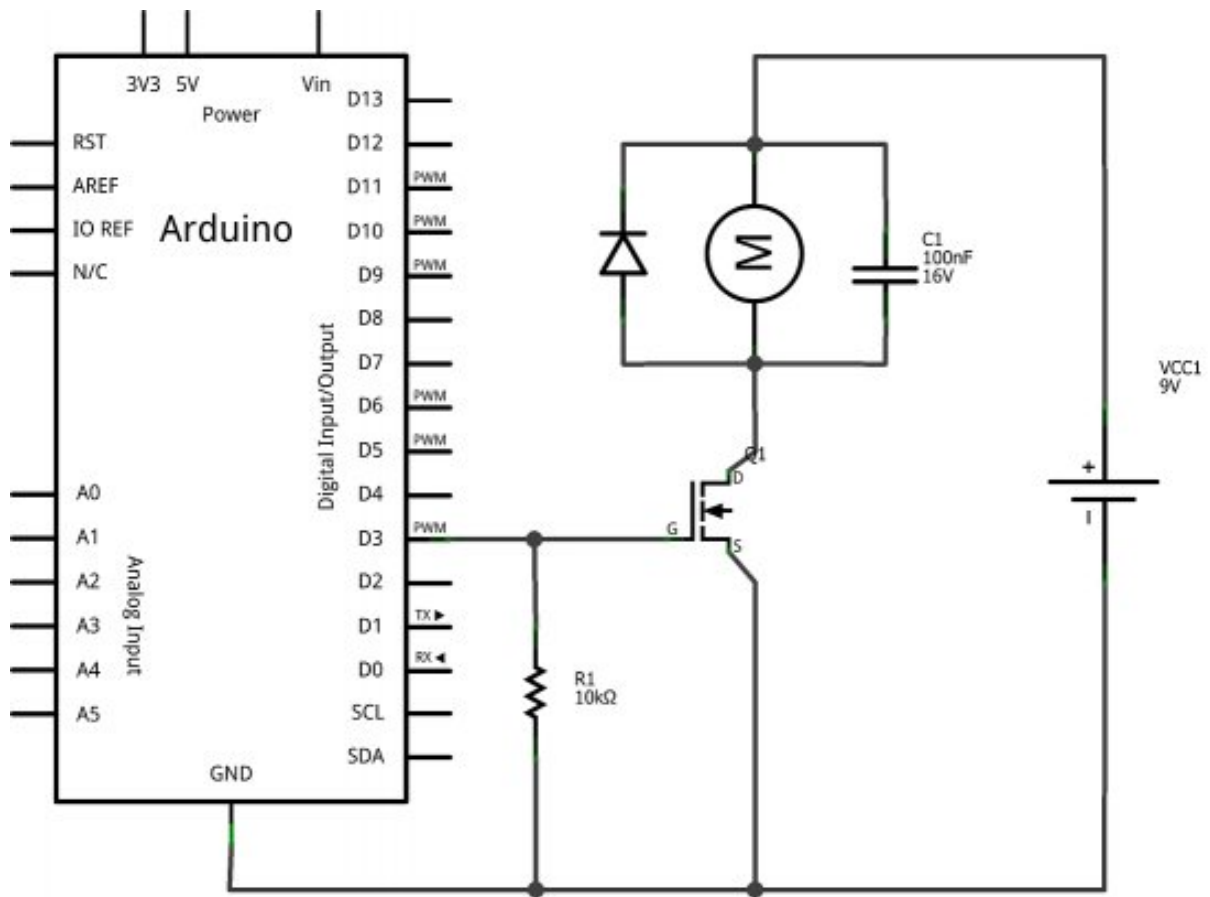


Figure 7.32 – Schéma du montage complet du moteur CC

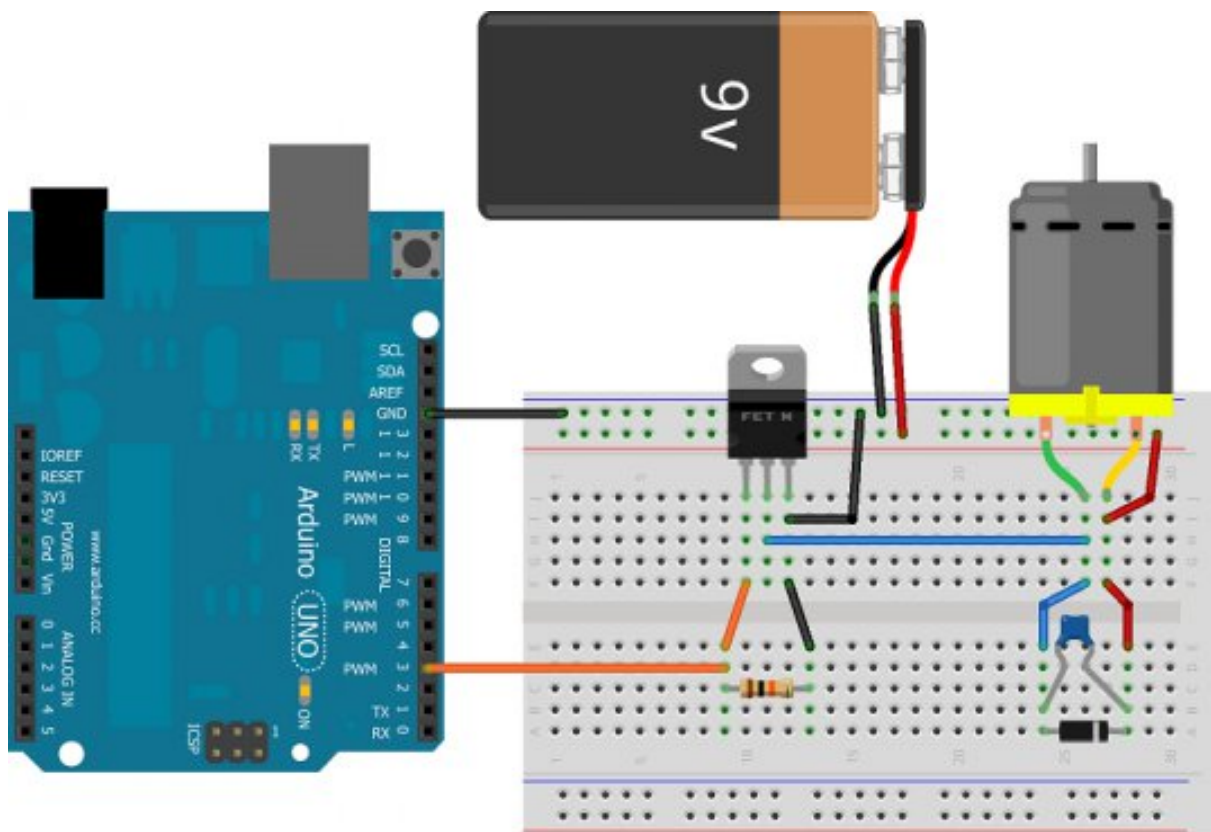


Figure 7.33 – Montage complet du moteur CC

“de l’autre côté” du moteur. Rien ne va changer, mais il va falloir commander les deux transistors pour faire tourner le moteur. Ce n’est pas bon. Essayons avec quatre transistors, soyons fou !

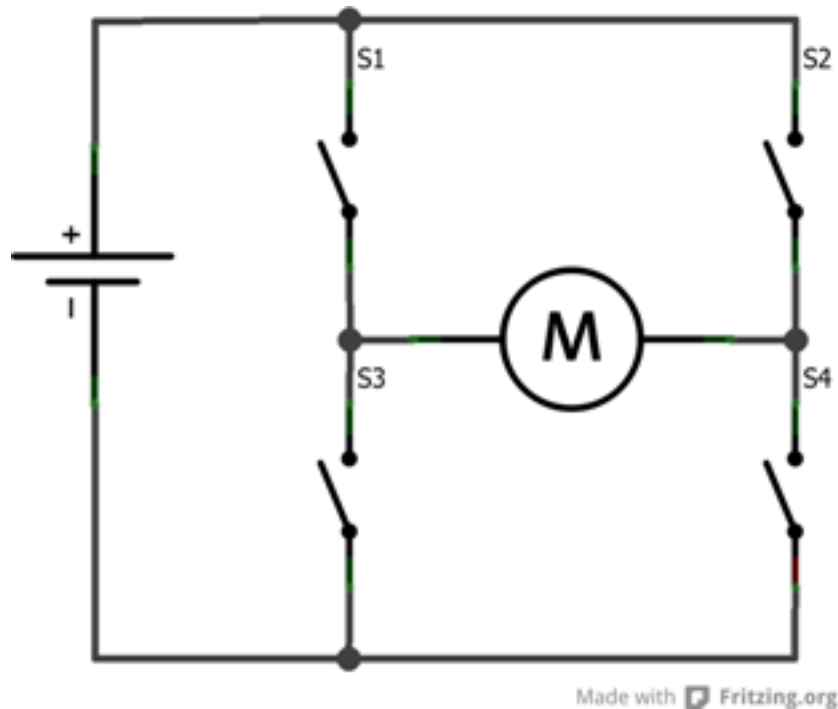


Figure 7.34 – Le pont en H

Eh bien, cela change tout ! Car à présent nous allons piloter le moteur dans les deux sens de rotation. Pour comprendre le fonctionnement de ce pont en H (appelé ainsi par sa forme), imaginons que je ferme les transistors 1 et 4 en laissant ouverts le 2 et le 3. Le courant passe de la gauche vers la droite.

Si en revanche je fais le contraire (2 et 3 fermés et 1 et 4 ouverts), le courant ira dans l’autre sens ! C’est génial non ?

Et ce n’est pas tout !

7.1.3.2.2 Allons plus loin avec le pont en H Comme vous l’aurez sûrement remarqué, les transistors fonctionnent deux par deux. En effet, si on en ferme juste un seul et laisse ouvert les trois autres le courant n’a nulle part où aller et rien ne se passe, le moteur est en roue libre. Maintenant, que se passe-t-il lorsqu’on décide de fermer 1 & 2 en laissant 3 et 4 ouverts ? Cette action va créer ce que l’on appelle un **frein magnétique**. Je vous ai expliqué plus tôt comment cela fonctionnait lorsque l’on mettait une diode de roue libre aux bornes du moteur. Le moteur se retrouve alors court-circuité. En tournant à cause de son inertie, le courant généré va revenir dans le moteur et va le freiner. Attention cependant, c’est différent d’un phénomène de roue libre où le moteur est libre de tourner.

[[attention]] | Ne fermez **jamais** 1 & 3 et/ou 2 & 4 ensemble, cela ferait un court-circuit de l’alimentation et vos transistors risqueraient de griller immédiatement si l’alimentation est capable de fournir un courant plus fort que ce qu’ils ne peuvent admettre.

7.1.3.3 Les protections nécessaires

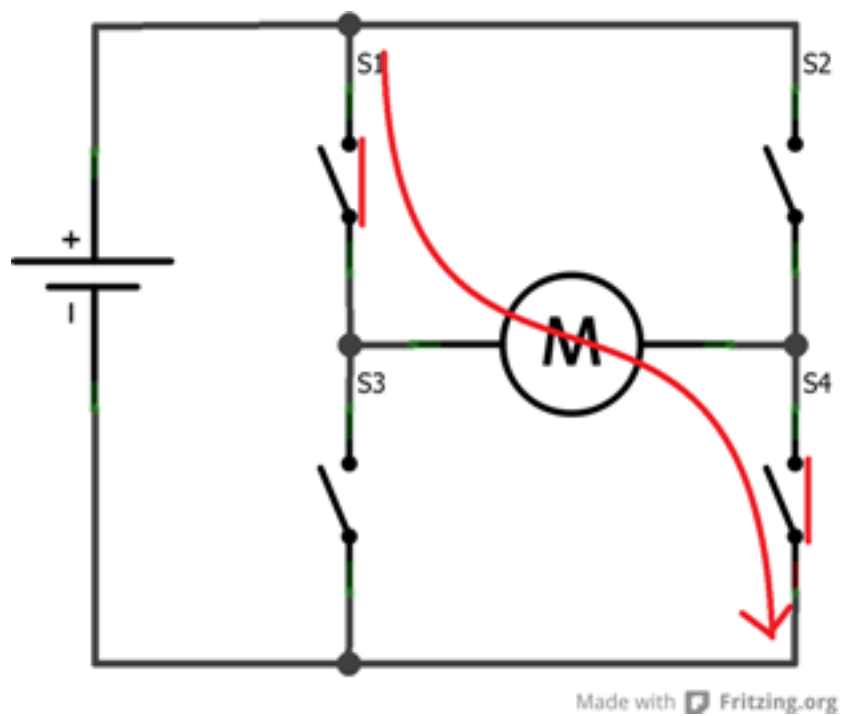


Figure 7.35 – Fonctionnement dans le sens horaire

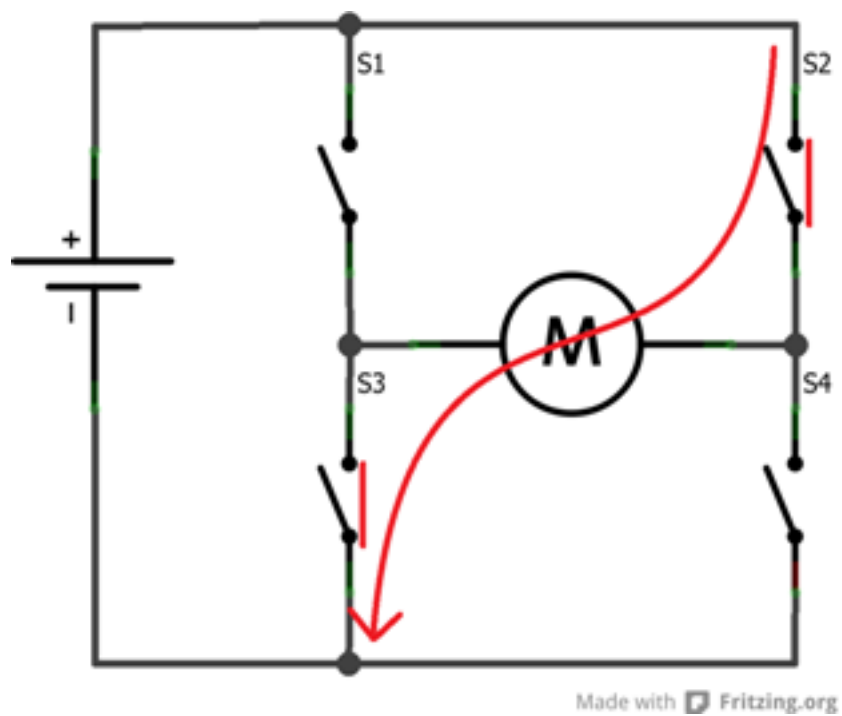


Figure 7.36 – Fonctionnement dans le sens anti-horaire

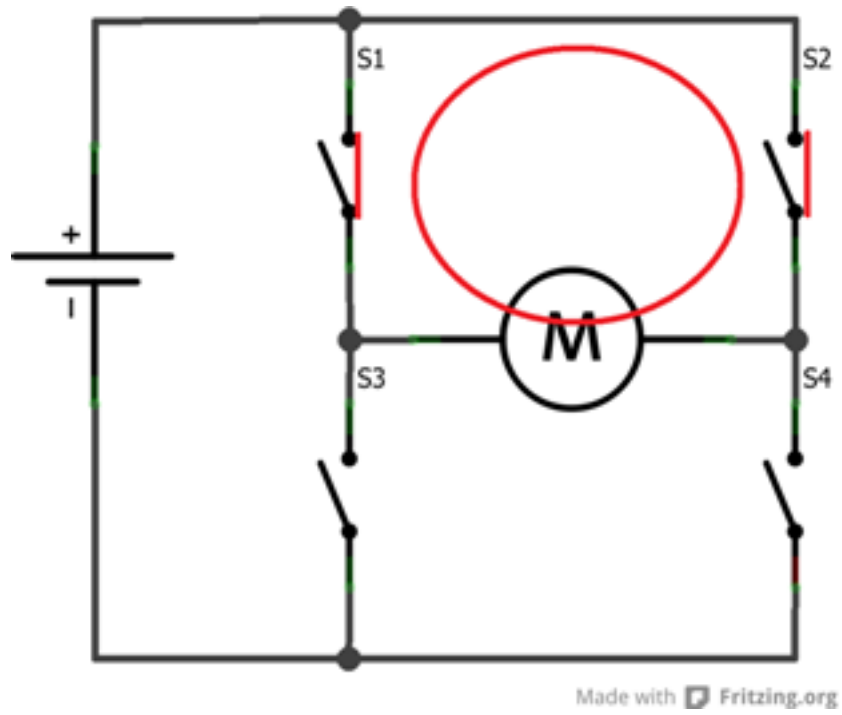


Figure 7.37 – Freinage avec 1 & 2

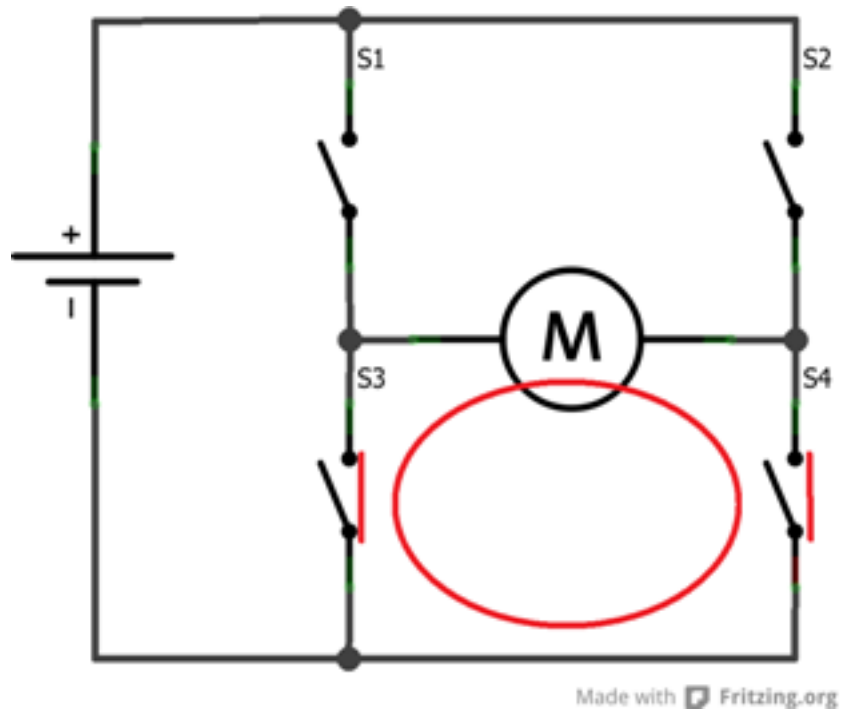


Figure 7.38 – Freinage avec 3 & 4

7.1.3.3.1 Les diodes de roue libre Comme nous l'avons vu plus haut, pour protéger un transistor des parasites ou lors du freinage électronique du moteur, nous plaçons une diode. Dans le cas présent, cette diode devra être en parallèle aux bornes du transistor (regardez le schéma qui suit). Ici nous avons quatre transistors, nous utiliserons donc quatre diodes que nous placerons sur chaque transistor. Ainsi, le courant trouvera toujours un moyen de passer sans risquer de forcer le passage dans les transistors en les grillant. Comme vu précédemment, des diodes de type Shottky sont recommandées pour leurs caractéristiques de tension de seuil faible et commutation rapide.

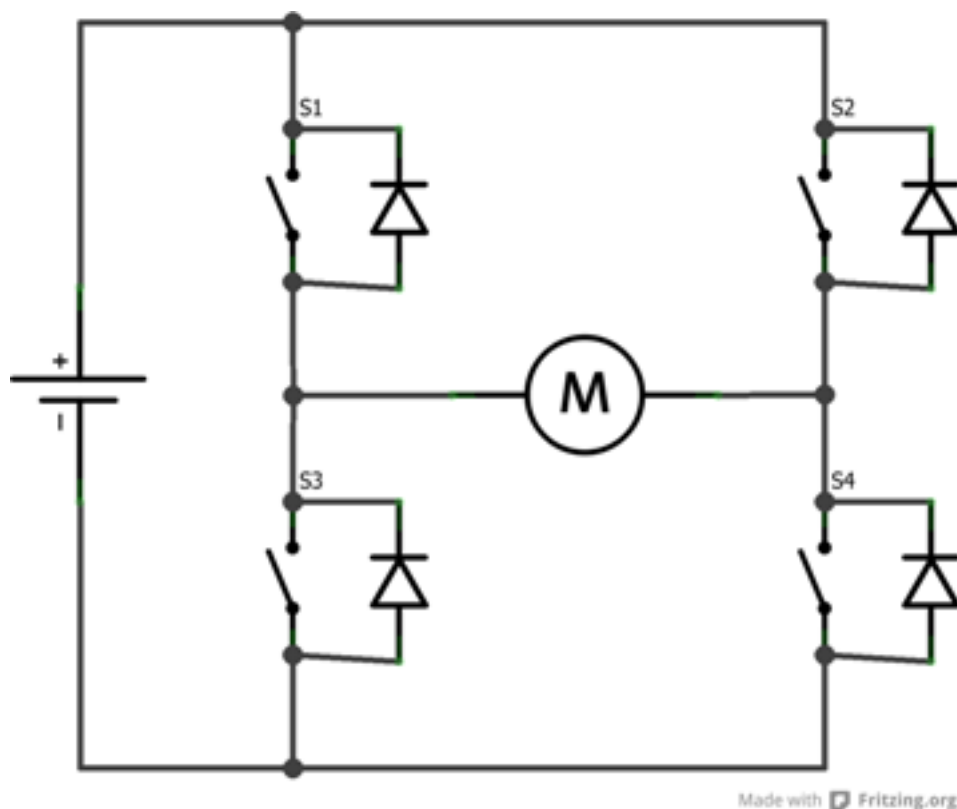


Figure 7.39 – Pont en H avec ses diodes de protection

7.1.3.3.2 Un peu de découplage Lorsque nous utilisons le moteur avec une PWM, nous générons une fréquence parasite. De plus, le moteur qui tourne génère lui même des parasites. Pour ces deux raisons, il est souvent utile d'ajouter des **condensateurs de filtrage** aux bornes du moteur. Comme sur le montage suivant, on peut en placer un en parallèle des deux broches du moteur, et deux autres plus petits entre une broche et la carcasse du moteur.

Ensuite, lorsque le moteur démarre il fera un appel de courant. Pour éviter d'avoir à faire transiter ce courant depuis la source de tension principale (une batterie par exemple), il est de bon usage de mettre un gros condensateur polarisé aux bornes de l'alimentation de puissance du pont en H. Ainsi, au moment du départ l'énergie sera en partie fournie par ce condensateur plutôt qu'en totalité par la batterie (ce qui évitera un échauffement abusif des conducteurs mais aussi une éventuelle baisse de la tension due à l'appel de courant).



Figure 7.40 – Un peu de découplage

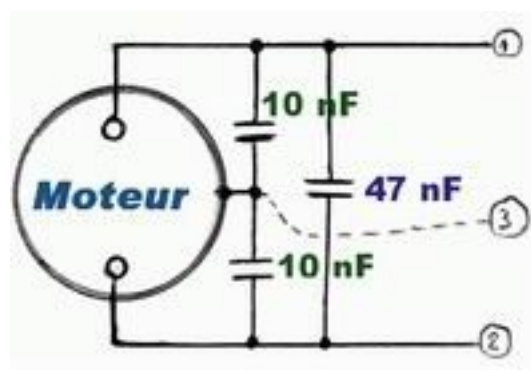


Figure 7.41 – Un peu de découplage, schéma

7.1.3.4 Des solutions intégrées : L293, L298...

Afin d'éviter de vous torturer avec les branchements des transistors et leur logique de contrôle, des composants "clés en main" ont été développés et produits. Nous allons maintenant étudier deux d'entre eux que nous retrouvons dans quasiment tous les shields moteurs Arduino : le L293(D) et son grand frère, plus costaud, le L298.

7.1.3.4.1 Le L293(D) Tout d'abord, voici un lien vers [la datasheet du composant](#). Les premières données nous apprennent que ce composant est un "quadruple demi-pont en H". Autrement formulé, c'est un double pont en H (car oui, 4 fois un demi ça fait 2 !). Ce composant est fait pour fonctionner avec des tensions de 4.5V à 36V et sera capable de délivrer 600 mA par canaux (dans notre cas cela fera 1,2A par moteur puisque nous utiliserons les demi-ponts par paire pour tourner dans les deux sens). Un courant de pic peut être toléré allant jusqu'à 1,2A par canaux (donc 2,4A dans notre cas). Enfin, ce composant existe en deux versions, le L293 et le L293D. La seule différence (non négligeable) entre les deux est que le L293D intègre déjà les diodes en parallèle des transistors. Un souci de moins à se préoccuper ! En revanche, cela implique donc des concessions sur les caractéristiques (le courant max passe à 1A par canaux et 2A pic pour la version sans les diodes). Le branchement de ce composant est assez simple (page 2 de la datasheet), mais nous allons le voir ensemble maintenant. Ce composant a 16 broches et fonctionne selon un système de symétrie assez simple.

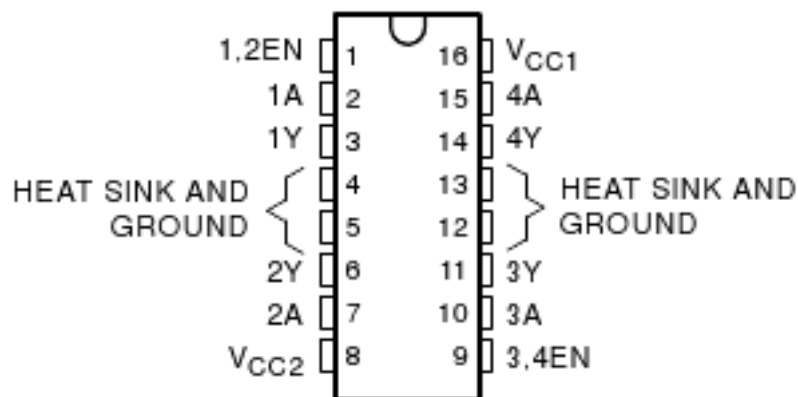


Figure 7.42 – Le L293

De chaque côté les broches du milieu (4, 5, 12 et 13) servent à relier la masse mais aussi à dissiper la chaleur. On trouve les entrées d'activation des ponts (*enable*) sur les broches 1 et 9. Un état HAUT sur ces broches et les ponts seront activés, les transistors pourront s'ouvrir ou se fermer, alors qu'un état BAS désactive les ponts, les transistors restent ouverts. Ensuite, on trouve les broches pour piloter les transistors. Comme un bon tableau vaut mieux qu'un long discours, voici les cas possibles et leurs actions :

->

Input 1 (broche 2 et 10)	Input 2 (broche 7 et 15)	Effet
0	1	Tourne dans le sens horaire
1	0	Tourne dans le sens anti-horaire
0	0	Frein

Input 1 (broche 2 et 10)	Input 2 (broche 7 et 15)	Effet
1	1	Frein

Table 7.1 – Commande et impact sur le moteur

<-

Ainsi, en utilisant une PWM sur la broche d’activation des ponts on sera en mesure de faire varier la vitesse. Il ne nous reste plus qu’à brancher le moteur sur les sorties respectives (2 et 7 ou 11 et 14 selon le pont utilisé) pour le voir tourner. :) Et voilà ! Vous savez à peu près tout ce qu’il faut savoir (pour l’instant :P) sur ce composant.

[[question]] |Attends attends attends, pourquoi il y a deux broches Vcc qui ont des noms différents, c’est louche ça !

Ah oui, c’est vrai et c’est important ! Le composant possède deux sources d’alimentation. Une pour la partie “logique” (contrôle correct des transistors), VCC1 ; et l’autre pour la partie puissance (utile pour alimenter les moteurs à la bonne tension), VCC2. Bien que ces deux entrées respectent les mêmes tensions (4.5V à 36V), nous ne sommes pas obligés de mettre des tensions identiques. Par exemple, la tension pour la logique pourrait venir du +5V de la carte Arduino tandis que la partie puissance pourrait être fournie par une pile 9V par exemple (n’oubliez pas de bien relier les masses entre elles pour avoir un référentiel commun).

[[e]] | N’utilisez **JAMAIS** le +5V de la carte Arduino comme alimentation de puissance (pour la logique c’est OK). Son régulateur ne peut fournir que 250mA ce qui est faible. Si vous l’utilisez pour alimenter des moteurs vous risquez de le griller !

Comme je suis sympa (^^) je vous donne un exemple de branchement du composant avec un moteur et une carte Arduino (j’ai pris le modèle L293D pour ne pas m’embêter à devoir mettre les diodes de protection sur le schéma :roll :) :

Vous noterez la présence du gros condensateur polarisé (100 µF / 25V ou plus selon l’alimentation) pour découpler l’alimentation de puissance du L293D. Comme je n’utilise qu’un seul pont, j’ai relié à la masse les entrées de celui qui est inutilisé afin de ne pas avoir des entrées qui “grésillent” et fassent consommer le montage pour rien. Enfin, vous remarquez que j’utilise trois broches de l’Arduino, deux pour le sens (2 et 4) et une PWM pour la vitesse (3).

7.1.3.4.2 Le L298 Étudions maintenant le grand frère du L293 : **le L298** . Si je parle de grand frère ce n’est pas innocent. En effet, son fonctionnement est très similaire à celui du L293, mais il est capable de débiter des courants jusqu’à 2A nominal par pont et jusqu’à 3A pendant un bref instant. Il propose aussi une fonction pouvant être intéressante qui est la mesure du courant passant au travers du pont (pour vérifier si votre moteur est “rendu en butée”¹ par exemple). Que dire de plus ? On retrouve deux broches d’alimentation, une pour la logique et l’autre pour la puissance. Celle pour la logique peut aller de 4.5 à 7V (là encore on pourra utiliser celle de l’Arduino). L’entrée puissance, en revanche, admet une tension comprise entre 5 et 46V. Pour un fonctionnement optimal, la documentation nous recommande de placer des condensateurs de 100nF sur chaque ligne d’alimentation. Et comme pour le L293, on pourra aussi placer un gros condensateur polarisé de 100µF (tension à choisir selon l’alimentation) sur la ligne d’alimentation de puissance.

1. s’il rencontre un obstacle qui freine sa course

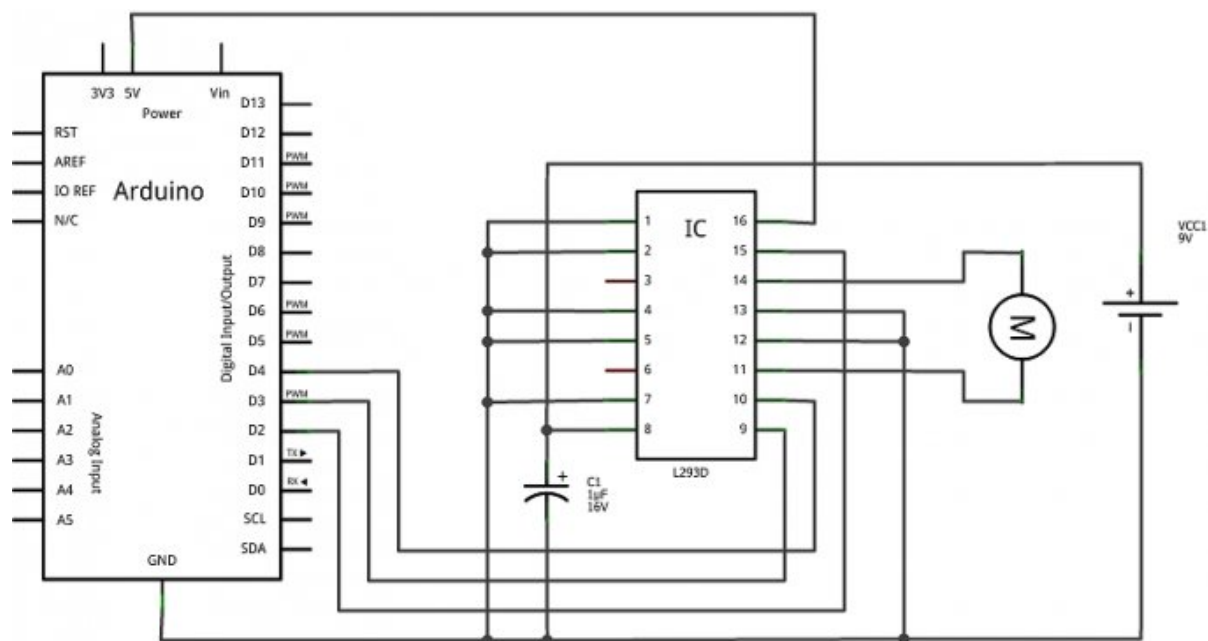


Figure 7.43 – Schéma d'utilisation du L293D

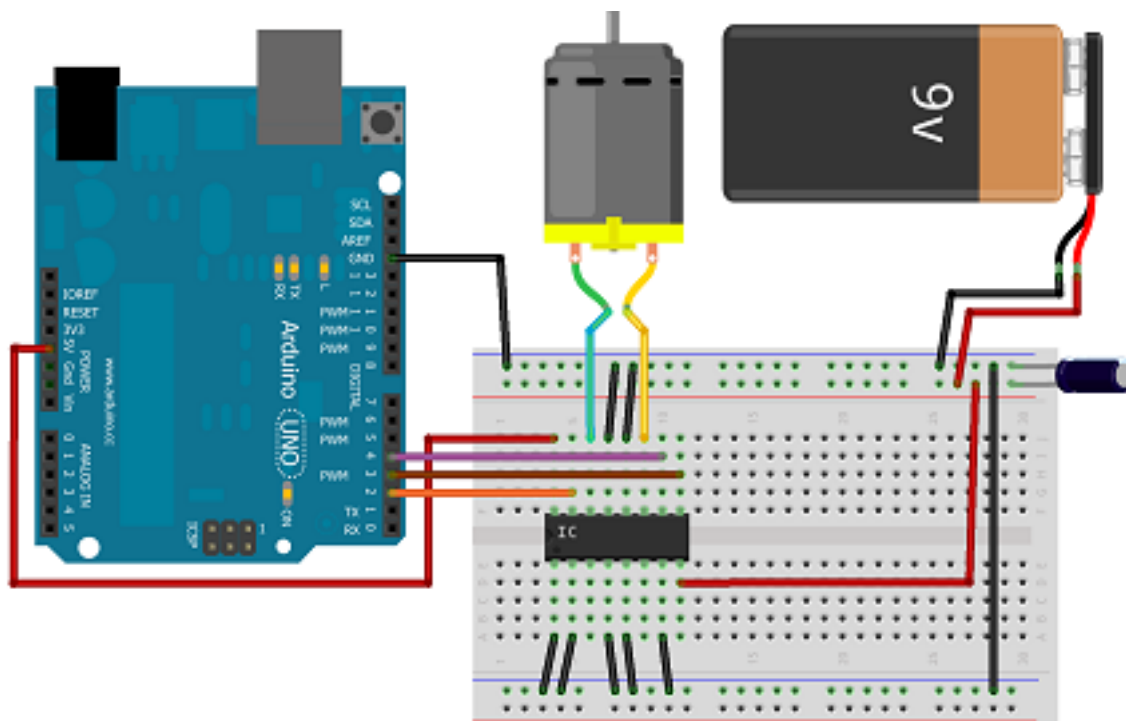


Figure 7.44 – Montage du L293D

Comme le fonctionnement est le même que celui du L293, je vais juste vous proposer une liste des broches utiles (oui je suis fainéant!).

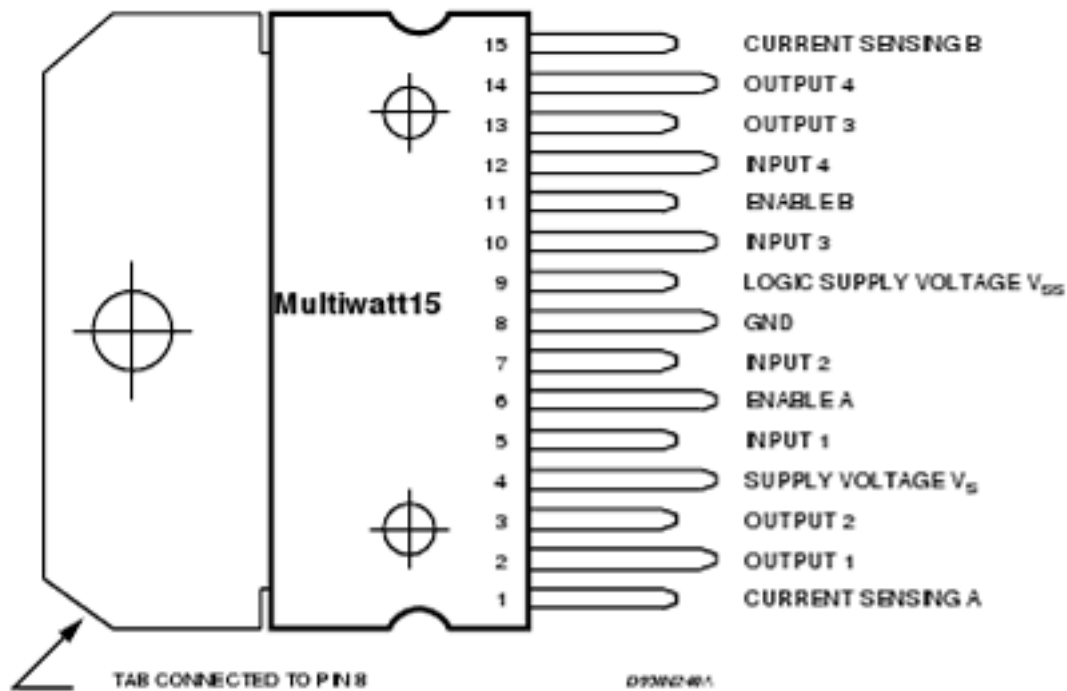


Figure 7.45 - le L298

Pour le premier pont :

- Les sorties sont situées sur les broches 2 et 3.
- Les entrées pour le sens de rotation sont la 5 et 7 et la PWM (*enable*) ira sur la broche 6.

Pour le second pont :

- Les sorties sont situées sur les broches 13 et 14.
- Les entrées pour le sens de rotation sont la 10 et 12 et la PWM (*enable*) ira sur la broche 11.

Pour les deux ponts :

- La masse, qui est au milieu sur la broche 8.
- L'alimentation de la logique de commande (le 5V) sur la broche suivante, la 9.
- Et l'alimentation de la partie puissance sur la broche 4.

Je ne mentionne pas les broches 1 et 15 qui sont celles servant à mesurer le courant traversant les ponts. Je doute que vous vous en serviez dans un premier temps et si vous arrivez jusque là je n'ai aucun doute que vous arriverez à les mettre en oeuvre (indice : il faudra utiliser une résistance;))

[[information]] |Le L298 n'existe pas avec les diodes de roue libre intégrées. Prenez donc garde à bien les rajouter dans votre montage sous peine de voir votre composant griller.

Comme précédemment, voici un schéma d'illustration (l'image représentant le L298 n'est pas exacte, mais le boîtier multiwatt n'existe pas encore dans Fritzing donc j'ai dû feinter) :

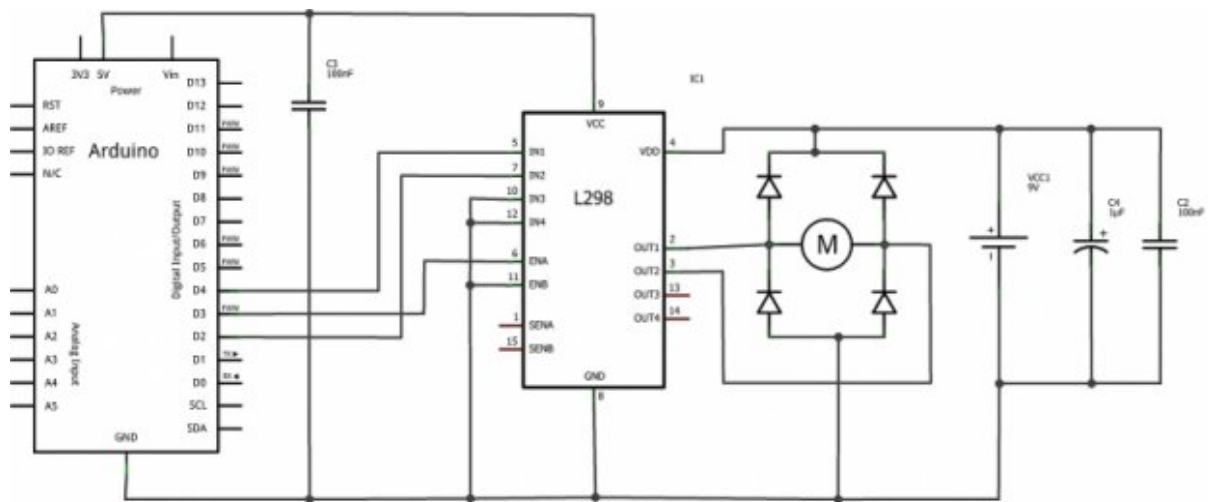


Figure 7.46 – Schéma du L298 avec un moteur et ses diodes

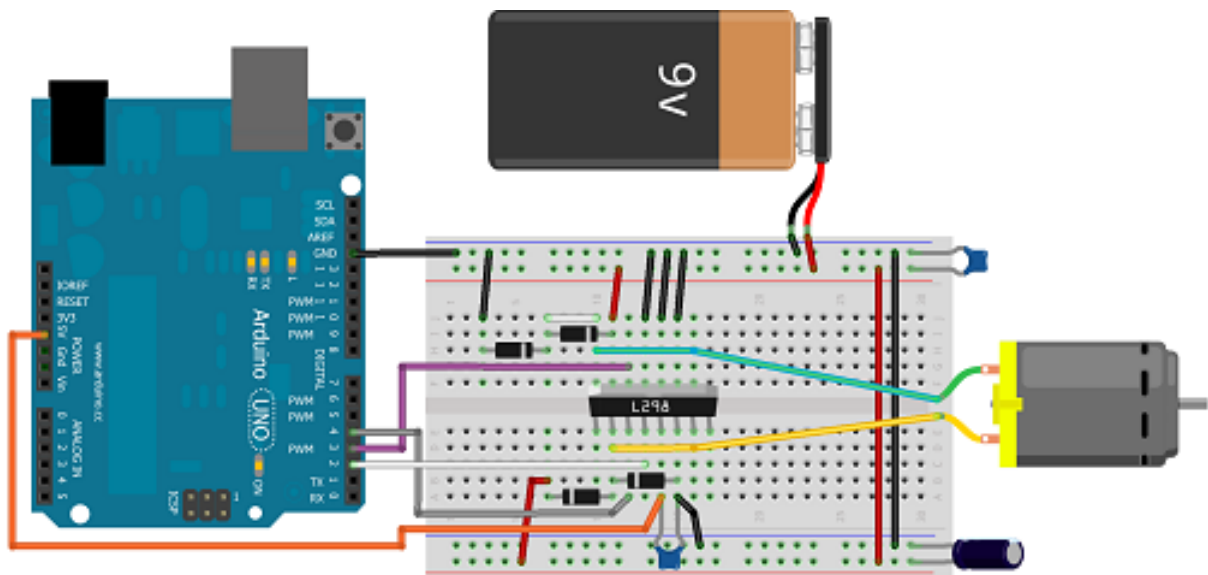


Figure 7.47 – Montage du L298 avec un moteur et ses diodes

7.1.4 Et Arduino dans tout ça ?

7.1.4.1 Bref rappel sur les PWM

Si vous avez bien lu la partie précédente, vous avez dû apprendre que pour pouvoir modifier la vitesse de rotation du moteur il faut utiliser un signal PWM. Mais vous souvenez-vous comment on s'en sert avec Arduino ? Allez, zou, petite piqûre de rappel ! Commençons par redire où sont situées les broches utilisables avec la PWM. Elles sont au nombre de 6 et ont les numéros 3, 5, 6, 9, 10 et 11. Pour les utiliser, vous devrez les configurer en sortie dans le `setup()` de votre programme :

```
const int brochePWM = 3;

void setup()
{
  // configuration en sortie de la broche 3
  pinMode(brochePWM, OUTPUT);
}
```

Ensuite, vous pourrez agir sur le *rapport cyclique* du signal PWM (le ratio entre temps à l'état HAUT et temps à l'état BAS) en utilisant la fonction `analogWrite(broche, ratio)`. L'argument *broche* désigne... la broche à utiliser et l'argument *ratio* indique la portion de temps à l'état haut du signal.

```
/* Le signal PWM est généré sur la broche 3 de la carte Arduino
   avec un rapport cyclique de 50%
   (état HAUT égal en temps à celui de l'état BAS */

analogWrite(brochePWM, 127);
```

Code : La fonction `analogWrite`

Le rapport cyclique est défini par un nombre allant de 0 à 255. Cela signifie qu'à 0, le signal de sortie sera nul et à 255, le signal de sortie sera à l'état HAUT. Toutes les valeurs comprises entre ces deux extrêmes donneront un rapport cyclique plus ou moins grand. Dans notre cas, le moteur tourne plus ou moins vite selon si le rapport cyclique est grand ou petit. Pour savoir quel rapport cyclique correspond avec quelle valeur, il faut faire une règle de trois :

->

Valeur argument	Rapport cyclique (%)
0	0
127	50
255	100

Table 7.2 – Quelques exemples de rapport cyclique

<-

Le calcul donnant la valeur pour chaque portion est défini par cette relation :

$$\text{argument} = \frac{x \times 100}{255}$$

Le résultat de ce calcul donne la valeur de l'argument pour le rapport cyclique recherché. x est la valeur du rapport cyclique que vous souhaitez donner au signal.

7.1.4.2 Utiliser un shield moteur

Comme nous l'avons vu précédemment, réaliser un pont en H demande quelques efforts (surtout si vous désirez tout faire vous mêmes :D). Afin de rendre ces derniers plus accessibles aux personnes ayant moins de moyens techniques (tout le monde ne dispose pas du matériel pour réaliser ses propres cartes électroniques!), l'équipe Arduino a développé et mis en productions un shield (une carte d'extension) pour pouvoir utiliser facilement des moteurs. Cette extension possède ainsi tout ce qu'il faut pour mettre en œuvre rapidement un ou des moteurs. La seule contrepartie est que les broches à utiliser sont imposées. Cependant, il existe une multitude de shields moteurs *non officiels* pouvant faire les mêmes choses ou presque. L'avantage de ces derniers est indéniablement leur prix souvent moins cher. En revanche, il n'est pas toujours facile de trouver leur documentation et le format de la carte ne se soucie pas forcément du "standard" Arduino (et n'est donc pas forcément adaptable en "s'ajoutant par dessus" comme un shield officiel le ferait). Je vais donc maintenant vous présenter le shield officiel, son fonctionnement et son utilisation, puis ensuite un shield non-officiel (acheté pas cher sur le net) que je possède et avec lequel je ferai mes photos/vidéos. Vous devriez alors avoir assez de connaissances pour utiliser n'importe quel shield non-officiel que vous pourrez trouver. Les deux shields présentés ont un point commun : ils utilisent tous les deux le L298 comme composant pour les ponts en H.

7.1.4.2.1 Le shield officiel d'Arduino Tout d'abord, voici l'adresse de description de ce shield : [le shield moteur](#). Comme vous avez bien lu la partie précédente à propos du L298, vous connaissez déjà la majeure partie des choses à savoir. Parmi elles, vous savez que le L298 nécessite trois broches de "pilotage" (par pont intégré) et envoie la puissance sur deux broches (par moteur). Éventuellement nous disposons aussi des deux "sondes de courant" mais nous y reviendrons plus tard. Voici un petit synoptique de résumé que je vous ai concocté pour l'occasion : :)

Voici comment il fonctionne et les quelques précautions d'utilisation.

- L'alimentation de puissance sur les borniers à visser à gauche est reliée à l'Arduino et peut donc lui servir de source d'alimentation. Si vous voulez dédier cette alimentation à la carte moteur, il faut donner un coup de cutter sur le strap marqué Vin en dessous de la carte
- Les entrées/sorties du shield sont reliées à l'Arduino de la manière suivante :

->

Fonction	Broches mot. A	Broches mot. B
Direction	12	13
PWM	3	11
Frein	9	8
Mesure de courant	A0	A1

Table 7.3 – Entrées/sorties du shield

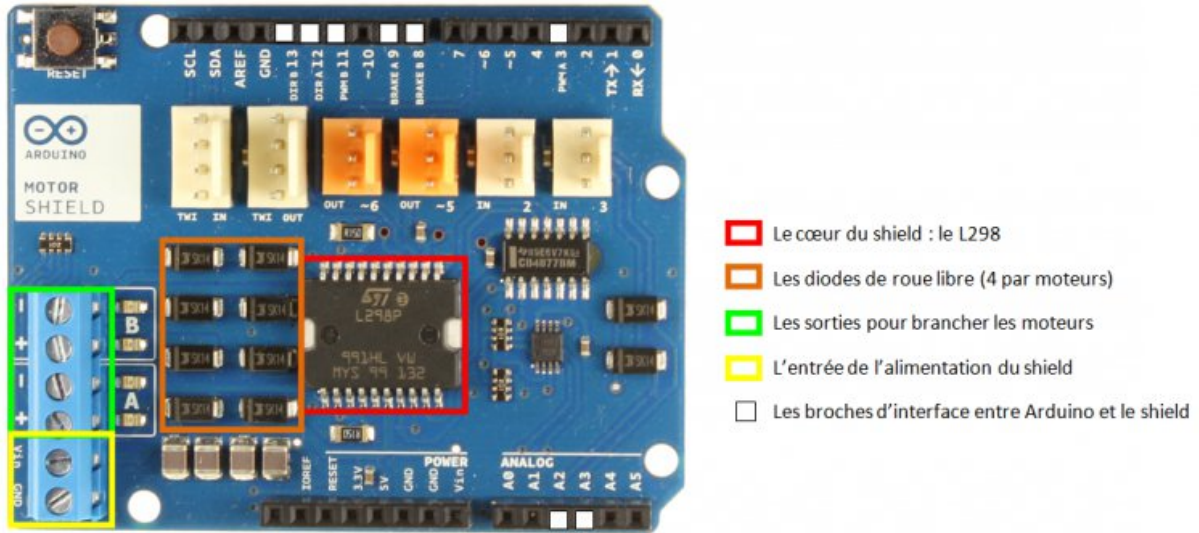


Figure 7.48 – Le shield moteur officiel en image

<-

La mesure de courant se fait sur les broches A0 et A1. Si vous avez besoin de ces broches pour d’autres applications, vous pouvez là encore désactiver la fonction en coupant le strap en dessous de la carte. Sinon, la mesure se fera simplement avec la fonction porte logique OU Exclusif, on peut désactiver la fonction de “frein” tout en gardant celle du sens. Grâce à cela, on peut se limiter à seulement deux broches pour commander chaque moteur : celle du sens et celle de la vitesse. Voici comment ils ont fait : Tout d’abord, regardons la table de vérité du OU EXCLUSIF. Cette dernière s’interprète comme suit : “La sortie est à 1 si une des deux entrées **uniquement** est à 1”. Sous forme de tableau on obtient ça :

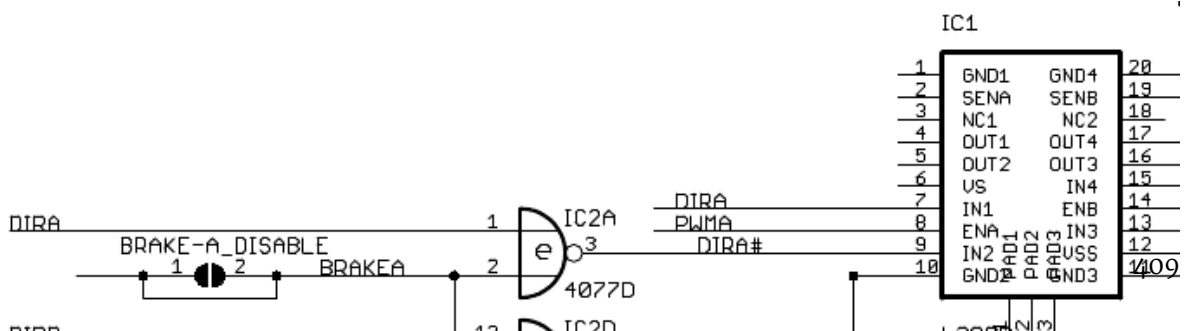
->

Entrée A	Entrée B	Sortie
0	0	0
1	0	1
0	1	1
1	1	0

Table 7.4 – Le OU Exclusif (XOR)

<-

Maintenant rappelez-vous, les conditions de freinage étaient justement représentées lorsque les deux entrées du pont étaient au même niveau. En couplant intelligemment le résultat de cette porte logique et les entrées de pilotage, on peut décider oui ou non d’avoir la fonction de frein. Afin de mieux comprendre, je vous invite à consulter cet extrait du schéma technique du shield :



préférez avoir deux broches disponibles et ne pas avoir de frein (juste une roue libre lorsque la PWM est à 0), alors il vous suffira une fois de plus de couper les straps en dessous de la carte.

[[information]] |N'ayez pas peur d'avoir des regrets ! Si vous coupez un strap, vous pourrez toujours le remettre en ajoutant un petit point de soudure pour relier les deux pastilles prévues à cet effet. ;) Le mieux aurait été d'avoir la possibilité de mettre des cavaliers que l'on enlève à la main, mais bon, c'est comme ça.

Vous savez maintenant tout à propos de ce shield. Je vais maintenant vous en présenter un non-officiel et ensuite nous passerons à un petit montage/code d'exemple pour finir ce chapitre.

7.1.4.2.2 Mon shield non-officiel Maintenant que vous connaissez le fonctionnement global du shield officiel, vous allez pouvoir utiliser sans problème la plupart des shields moteurs. Afin de ne pas faire de publicité pour un site ou un autre, je vais vous présenter mon shield qui vaut aussi bien qu'un autre (mais pas forcément mieux). Il n'y a aucun parti pris, j'ai acheté ce dernier afin de profiter de tarif intéressant lors d'une commande avec d'autres composants. Si j'avais été uniquement à la recherche d'un shield moteur, j'en aurais peut-être pris un autre qui sait ! Bref, assez de ma vie, passons à l'étude du module ! Afin de bien commencer les choses, je vais d'abord vous montrer une photo d'identité de ce dernier. Ensuite je vous expliquerai où sont les broches qui nous intéressent et ferai un parallèle avec le shield officiel. Les deux étant basés sur un L298 l'explication sera assez rapide car je n'ai pas envie de me répéter. Je ferai néanmoins un petit aparté sur les différences (avantages et inconvénients) entre les deux.

Voici une petite liste des points importants :

- À gauche en **jaune** : les entrées de commande. **EnA**, **In1**, **In2** pour le moteur A ; **EnB**, **In3**, **In4** pour le moteur B. On trouve aussi une broche de masse et une sortie 5V sur laquelle je reviendrai.
- En bas en **vert** différents *jumpers* (des cavaliers si vous préférez ;)) pour activer des résistances de pull-down (force une entrée/sortie à l'état bas) et câbler la mesure de courant de sortie des ponts
- À droite en **bleu**, les bornes pour brancher les moteurs A et B (respectivement en haut et en bas) et au milieu le bornier pour amener l'alimentation de puissance (et une entrée ou sortie) de 5V

Au milieu on retrouve le L298 avec de chaque côté (en haut et en bas) les diodes de roue libre pour chaque moteur. Une petite précision s'impose par rapport à ce shield. La carte embarque un régulateur 5V (le petit bloc noir en haut à gauche marqué 78M05). Ce dernier peut être utilisé ou non (Activez-le avec le jumper vert juste à coté). Si vous le laissez activé, c'est lui qui fournira l'alimentation pour la logique du L298. Si vous le désactivez, vous devrez fournir vous-même le 5V pour la logique. Dans tous les cas, il vous faut relier les masses puissances et logiques entre Arduino et le shield afin d'avoir un référentiel commun. Si vous l'activez, alors vous obtiendrez une sortie de 5V sur le bornier bleu à droite (utile pour alimenter l'Arduino par exemple). Si vous le désactivez, alors vous devrez fournir le 5V (et donc le bornier bleu devra être utilisé comme une entrée). Ce shield n'est en fait qu'une simple carte électronique disposant du L298 et facilitant l'accès à ses broches. Le fonctionnement se fait exactement comme nous l'avons vu dans le chapitre précédent, lorsque je vous présentais le L293 et L298 pour la première fois. Pas de facétie avec des portes logiques pour gagner des broches. Ici, tout est brut de décoffrage, on commande directement le pont en H. Il vous faudra donc trois broches par moteur, deux pour gérer la direction et le frein et une (PWM) pour la vitesse.

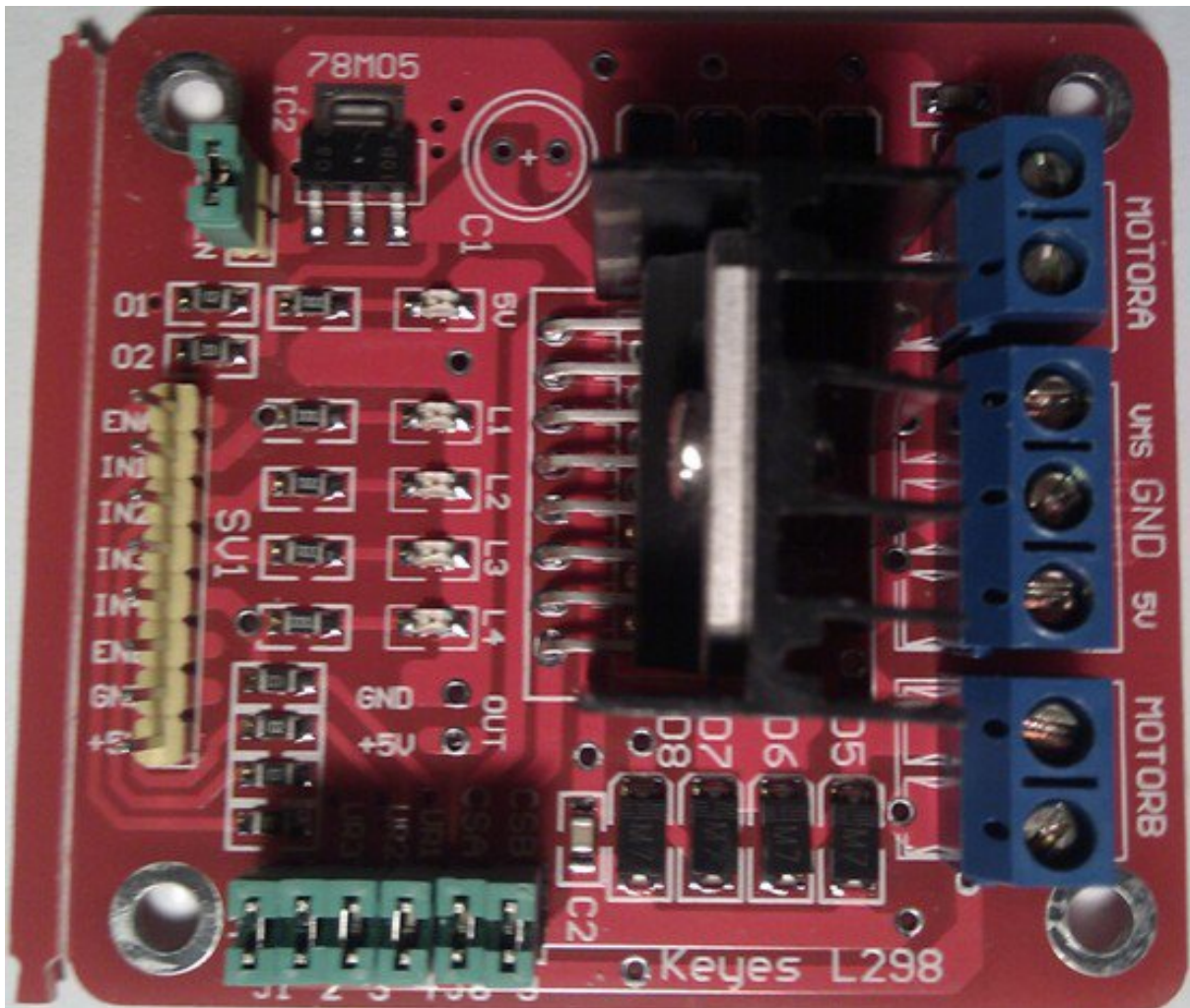


Figure 7.50 – Le shield moteur étudié

7.1.4.3 Petit programme de test

Nous allons maintenant pouvoir passer aux choses sérieuses : l'utilisation du moteur avec l'Arduino !

7.1.4.3.1 L'électronique Pour cela, nous allons commencer par câbler le shield. En ayant la partie précédente concernant le vôtre sous les yeux, vous devriez pouvoir vous en sortir sans trop de difficulté. (Désolé, pas de schéma ce coup-ci car le logiciel que j'utilise ne possède pas encore le shield moteur dans sa base de données, faites donc preuve d'imagination. ;)) Personnellement, je n'utiliserai qu'un seul moteur (car dans l'immédiat j'en ai qu'un sous la main :P). Je vais donc le brancher sur les bornes bleues "Moteur A". Ensuite, je vais relier les différentes broches de commande à mon Arduino. La broche EnA sera reliée à une sortie de PWM (dans mon cas la broche 3) et les broches In1 et In2 seront reliées à n'importe quelles broches numériques (2 et 4 pour moi). Il ne nous reste plus qu'à nous occuper de l'alimentation. Tout d'abord, je mets un fil entre la masse du shield et celle de l'Arduino (pour avoir un référentiel commun). Comme ma carte possède son propre régulateur de tension 5V, pas besoin de l'amener depuis Arduino. Enfin, je relie les deux fils pour la puissance. Dans mon cas ce sera une alimentation 12V (400 mA max, wouhou) qui vient d'un adaptateur allume-cigare (censé fournir du 5V) que j'ai démonté pour obtenir une source de 12V. Je vous propose aussi de rajouter un potentiomètre sur une entrée analogique. De cette façon nous allons pouvoir faire varier la vitesse sans recharger le programme ;) . Et voilà, point de vue électronique tout est prêt. Voilà ce que ça donne chez moi (un beau bazar :D , mais j'ai oublié le potentiomètre) :

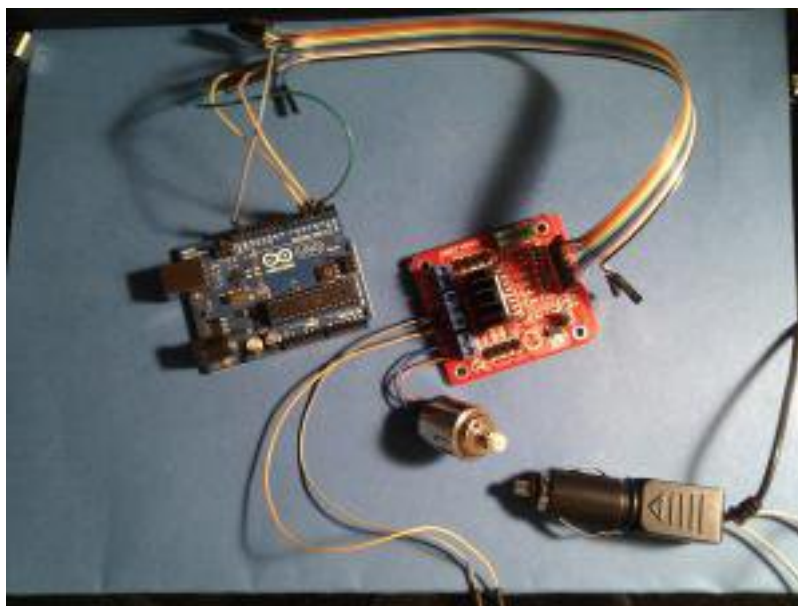


Figure 7.51 – Le montage avec le shield et Arduino

7.1.4.3.2 L'informatique Maintenant, nous allons devoir nous occuper du code et comme toujours, nous commençons par lister les variables concernant les broches utilisées :

```
// la PWM pour la vitesse
const int enable = 3;
// les broches de signal pour le sens de rotation
```

```

const int in1 = 2;
const int in2 = 4;

// une entrée analogique (A0) pour régler la vitesse manuellement
const int potar = 0;

```

Ces différentes broches seront bien entendu des broches de sortie (sauf l'analogique), donc nous les déclarons comme telles dans le setup() :

```

void setup()
{
  pinMode(enable, OUTPUT);
  pinMode(in1, OUTPUT);
  pinMode(in2, OUTPUT);
  // j'utilise la liaison série pour vérifier
  // la vitesse définie par le potentiomètre
  Serial.begin(115200);

  // on démarre moteur en avant et en roue libre
  analogWrite(enable, 0);
  digitalWrite(in1, LOW);
  digitalWrite(in2, HIGH);
}

```

Code : Un setup pour l'utilisation d'un moteur à courant continu

Et voilà, si vous exécutez le code maintenant votre moteur sera... arrêté! Eh oui, j'ai volontairement mis une vitesse nulle à la fin du setup() pour éviter que le moteur ne s'emballe au démarrage du programme. Mais si vous changez cette dernière (mettez 50 pour voir) vous verrez votre moteur se mettre à tourner. Nous allons donc rajouter un peu d'interactivité, pour que vous puissiez vous-même augmenter/diminuer la vitesse en fonction de la valeur lue sur le potentiomètre :

```

void loop()
{
  // on lit la valeur du potentiomètre
  int vitesse = analogRead(potar);

  // division de la valeur lue par 4
  vitesse /= 4;

  // envoie la nouvelle vitesse sur le moteur
  analogWrite(enable, vitesse);

  // on affiche la vitesse sur le moniteur série
  Serial.println(vitesse);

  delay(50);
}

```

Code : Utilisation basique d'un moteur à courant continu

[[question]] |Mais pourquoi tu divises la vitesse par 4 à la ligne 5? Je veux aller à fond moi!

C'est très simple. La lecture analogique nous renvoie une valeur entre 0 et 1023 (soit 1024 valeurs possibles). Or la fonction analogWrite ne peut aller qu'entre 0 et 255 (total de 256 valeurs). Je divise donc par 4 pour rester dans le bon intervalle ! Car : $4 \times 256 = 1024$.

7.1.4.4 Programme plus élaboré

Maintenant, je vous fais cadeau d'un code vous permettant d'aller dans les deux sens et à vitesse variable. Mais, je vous conseille d'essayer de le faire par vous-même avant de regarder ce qu'il y a dans la balise secret. Le potentiomètre est utilisé comme régulateur de vitesse, mais on va virtuellement décaler l'origine. Autrement dit, entre 0 et 511 nous irons dans un sens, et entre 512 et 1023 nous irons dans l'autre sens. Nous ferons aussi en sorte que la vitesse soit de plus en plus élevée lorsque l'on "s'éloigne" du 0 virtuel (de la valeur 512 donc). Je vous donne le code tel quel (avec des commentaires bien sûr). Libre à vous de le traiter comme un exercice. À sa suite, une petite vidéo du résultat.

```
[[secret]] |cpp |const int enable = 3; // la PWM |const int in1 = 2; // les
broches de signal |const int in2 = 4; |const int potar = 0; // la broche
pour régler la vitesse | |void setup() |{ | pinMode(enable, OUTPUT); |
pinMode(in1, OUTPUT); | pinMode(in2, OUTPUT); | Serial.begin(115200); | |
// on démarre moteur en avant et en roue libre | analogWrite(enable, 0); |
digitalWrite(in1, LOW); | digitalWrite(in2, HIGH); |} | |void loop() |{ |
int vitesse = analogRead(potar); | | // dans le sens positif | if(vitesse
> 512) | { | // on décale l'origine de 512 | vitesse -= 512; | // le moteur
va dans un sens | digitalWrite(in1, LOW); | digitalWrite(in2, HIGH); |
Serial.print("+"); | } | else // dans l'autre sens | { | // de même on
décale pour que la vitesse augmente en s'éloignant de 512 | vitesse = 512-vitesse;
| // le moteur va dans l'autre sens | digitalWrite(in1, HIGH); | digitalWrite(in2,
LOW); | Serial.print("-"); | } | | // pour rester dans l'intervalle [0;255]
(sinon on est dans [0;512]) | vitesse /= 2; | // envoie la vitesse | analogWrite(enable,
vitesse); | | // et l'affiche | Serial.println(vitesse); | delay(50); |} |
|Code : Exercice de rotation du moteur à courant continu
```

Bravo à ceux qui ont essayé de faire ce programme, même s'ils n'y sont pas arrivé ! Dans ce dernier cas, vous pouvez aller voir sur les forums et poser vos éventuelles questions après avoir vérifié que vous avez bien tout essayé de comprendre. ;) Voilà la vidéo qui montre le fonctionnement du programme :

->!(https://www.youtube.com/watch?v=K1NRLzt_zSI)<-

Désolé pour la qualité de la vidéo, il faut vraiment que je change d'appareil...

Vous savez désormais comment fonctionne un moteur à courant continu et quels sont les moyens de le piloter. Il va dorénavant être possible de vous montrer l'existence de moteurs un peu particuliers qui se basent sur le moteur à courant continu pour fonctionner. Et vous allez voir que l'on va pouvoir faire plein de choses avec ! ;)

7.2 Un moteur qui a de la tête : le Servomoteur

Dans ce chapitre, nous allons parler d'un moteur que nos amis modélistes connaissent bien : le **Servomoteur** (abrégé : "servo"). C'est un moteur un peu particulier, puisqu'il confond un ensemble de mécanique et d'électronique, mais son principe de fonctionnement reste assez simple. Les parties seront donc assez courtes dans l'ensemble car les servomoteurs contiennent dans leur "ventre" des moteurs à courant continu que vous connaissez à présent. Cela m'évitera des explications supplémentaires. :-°

7.2.1 Principe du servomoteur

Un servomoteur... Étrange comme nom, n'est-ce pas ? Cela dit, il semblerait qu'il le porte bien puisque ces moteurs, un peu particuliers je le disais, emportent avec eux une électronique de commande (faisant office de "cerveau"). Le nom vient en fait du latin *servus* qui signifie esclave. Mais avant de nous atteler à l'exploration interne de ce cher ami, façon de parler, nous allons avant tout voir à quoi il sert.

7.2.1.1 Vue générale

7.2.1.1.1 Le servo, un drôle de moteur Commençons en image, avec la photographie d'un servomoteur :



Figure 7.52 – Un servomoteur

C'est, en règle générale, à quoi ils ressemblent, variant selon leur taille.

[[q]] | Pfiouuu, c'est quoi ce moteur, ça n'y ressemble même pas! :o

J'veus l'avais dit que c'était des moteurs particuliers ! En détail, voyons à quoi ils servent. De manière semblable aux moteurs à courant continu, les servomoteurs disposent d'un axe de rotation.

Sur la photo, il se trouve au centre de la roue blanche. Cet axe de rotation est en revanche entravé par un système de bridage. Cela ne veut pas dire qu'il ne tourne pas, mais cela signifie qu'il ne peut pas tourner au-delà d'une certaine limite. Par exemple, certains servomoteurs ne peuvent même pas faire tourner leur axe de rotation en leur faisant faire un tour complet ! D'autres en sont capables, mais pas plus d'un tour. Enfin, un cas à part que nous ne ferons qu'évoquer, ceux qui tournent sans avoir de limite (autant de tours qu'ils le veulent). Et là, c'est le moment où je vous dis : "*détrompez-vous !*" en répondant à la question critique que vous avez en tête : "*Un moteur qui ne peut même pas faire un tour avec son axe de rotation, ça ne sert à rien ? o_0*" En effet, s'il ne peut pas faire avancer votre robot, il peut cependant le guider. Prenons l'exemple d'une petite voiture de modélisme à quatre roues. Les roues arrière servent à faire avancer la voiture, elles sont mises en rotation par un moteur à courant continu, tandis que les roues avant, qui servent à la direction de la voiture pour ne pas qu'elle se prenne les murs, sont pilotées par un servomoteur. Comment ? Eh bien nous allons vous l'expliquer.

7.2.1.1.2 L'exemple de la voiture radiocommandée Regardons l'image que je vous ai préparée pour comprendre à quoi sert un servomoteur :

Chaque roue est positionnée sur un axe de rotation (partie bleue) lui-même monté sur un pivot sur le châssis de la voiture (en vert). La baguette (rouge) permet de garder le parallélisme entre les roues. Si l'une pivote vers la gauche, l'autre en fait de même (ben ouais, sinon la voiture devrait se couper en deux pour aller dans les deux directions opposées :lol :). Cette baguette est fixée, par un pivot encore, au bras de sortie du servomoteur. Ce bras est à son tour fixé à l'axe de rotation du servomoteur. Ainsi, lorsque le servomoteur fait tourner son axe, il entraîne le bras qui entraîne la baguette et fait pivoter les roues pour permettre à la voiture de prendre une direction dans son élan (tourner à gauche, à droite, ou aller tout droit). Il n'y a rien de bien compliqué. Ce qu'il faut retenir est que le servomoteur va entraîner la baguette pour orienter les roues dans un sens ou dans l'autre. Elles auront donc un angle d'orientation par rapport au châssis de la voiture. Voyez plutôt :

Alors, vous allez me dire : "*mais pourquoi on ne met pas un moteur à courant continu avec un bras sur son axe, ce serait plus simple, non ?*" Eh bien non, car cela ne conviendrait pas. Je vous explique pourquoi. Nous l'avons vu, un moteur à courant continu tourne sans s'arrêter, sauf si on lui coupe l'alimentation. Le problème c'est que, dans notre cas, si on laisse le moteur tourner, il pourrait faire pivoter les roues plus loin que leur angle maximal et casser le système de guidage car il ne saura pas quand il faut s'arrêter (à savoir, quand les roues sont arrivées à leur angle maximal). Bon, on pourrait très bien faire un système qui coupe l'alimentation quand les roues arrivent sur leur butée. En plus, les moteurs à courant continu sont de bien piètres athlètes, il leur faudrait nécessairement un réducteur pour arriver à avoir une vitesse faible et un couple plus élevé. Mais pourquoi s'embêter avec ça plutôt que d'utiliser quelque chose de déjà tout prêt ? C'est le servomoteur qui va faire tout ça ! Pour être précis, le servomoteur est commandé de telle sorte qu'au lieu de donner une vitesse de rotation de son axe, il donne une position angulaire de l'arbre relié à son axe. Donc, on lui demande de faire tourner son axe de 10° vers la gauche et il s'exécute !

7.2.1.2 Composition d'un servomoteur

Les servomoteurs ont donc l'avantage d'être *asservis* en **position angulaire**. Cela signifie, je vous l'expliquais, que l'axe de sortie du servomoteur respectera une consigne d'orientation que vous lui envoyez en son entrée. En plus, tenez-vous bien, si par malheur les roues venaient à changer d'orientation en passant sur un caillou par exemple, l'électronique interne du servomoteur

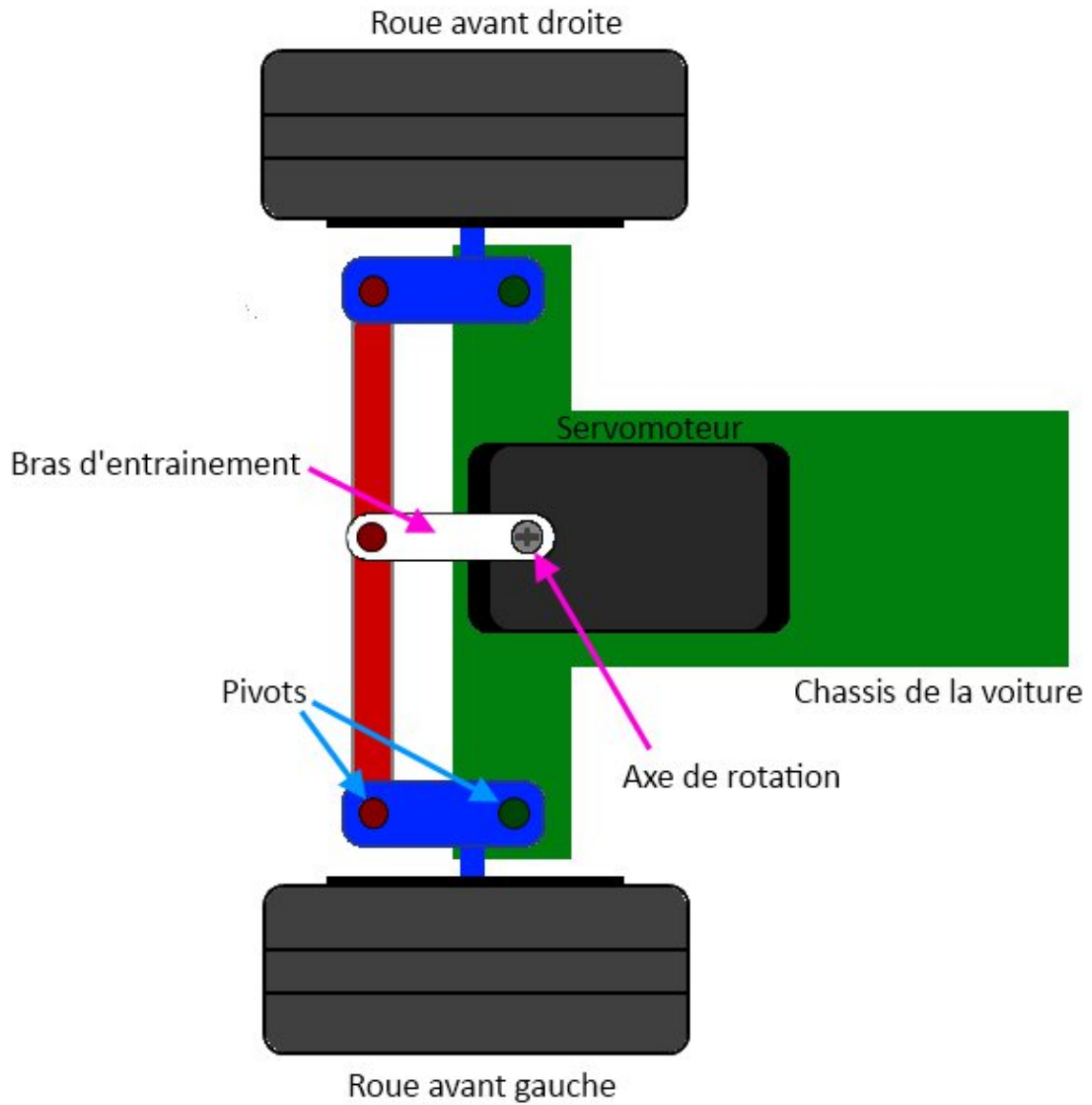


Figure 7.53 – Vue de dessus Représentation schématique du système de guidage des roues d'une

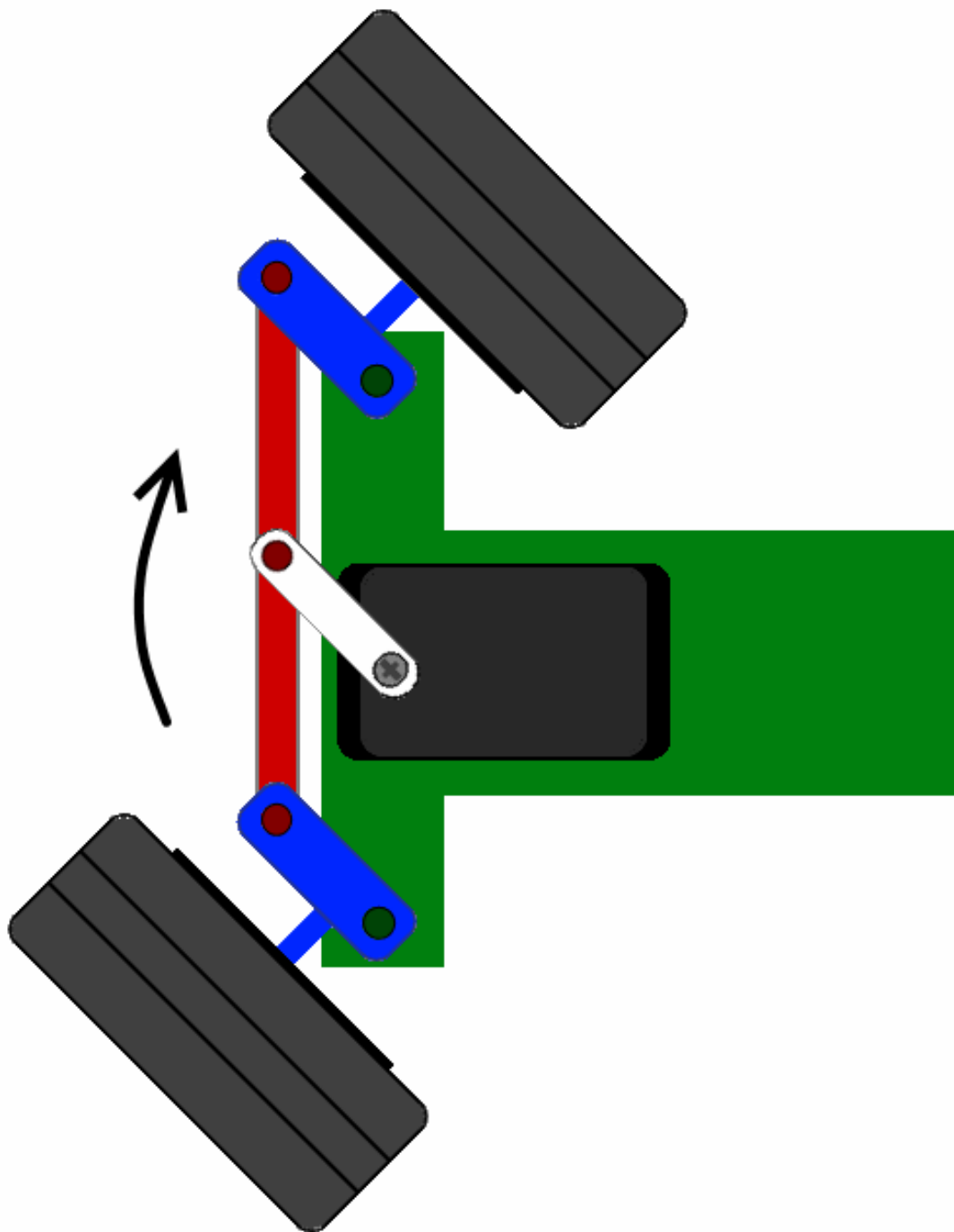


Figure 7.54 – Animation de la direction

essaiera tant bien que mal de conserver cette position ! Et quelle que soit la force que l'on exerce sur le bras du servomoteur, il essaiera de toujours garder le même angle (dans les limites du raisonnable évidemment). En quelque sorte vous ne pilotez pas directement le moteur, mais plutôt **vous imposez le résultat que vous voulez avoir en sortie.**

7.2.1.2.1 Apparence On en trouve de toutes les tailles et de toutes les puissances. La plupart du temps la sortie peut se positionner entre 0 et 180°. Cela dit, il en existe également dont la sortie peut se débattre sur seulement 90° et d'autres, ayant un plus grand débattement, sur 360°. Ceux qui ont la possibilité de faire plusieurs tours sont souvent appelés **servo-treuil**s. Enfin, les derniers, qui peuvent faire tourner leur axe sans jamais se buter, sont appelés **servomoteurs à rotation continue**. Les servomoteurs sont très fréquemment employés dans les applications de modélisme pour piloter le safran d'un bateau, le gouvernail d'un avion ou bien même les roues d'une voiture téléguidée dont on a parlé jusqu'à présent. Maintenant que les présentations sont faites, mettons-le à nu ! Il est composé de plusieurs éléments visibles ... :

- Les fils, qui sont au nombre de trois (nous y reviendrons)
- L'axe de rotation sur lequel est monté un accessoire en plastique ou en métal
- Le boîtier qui le protège

... mais aussi de plusieurs éléments que l'on ne voit pas :

- un moteur à courant continu
- des engrenages pour former un réducteur (en plastique ou en métal)
- un capteur de position de l'angle d'orientation de l'axe (un potentiomètre bien souvent)
- une carte électronique pour le contrôle de la position de l'axe et le pilotage du moteur à courant continu

Voilà une image 3D (extraite du [site internet suivant](#)) de vue de l'extérieur et de l'intérieur d'un servomoteur :

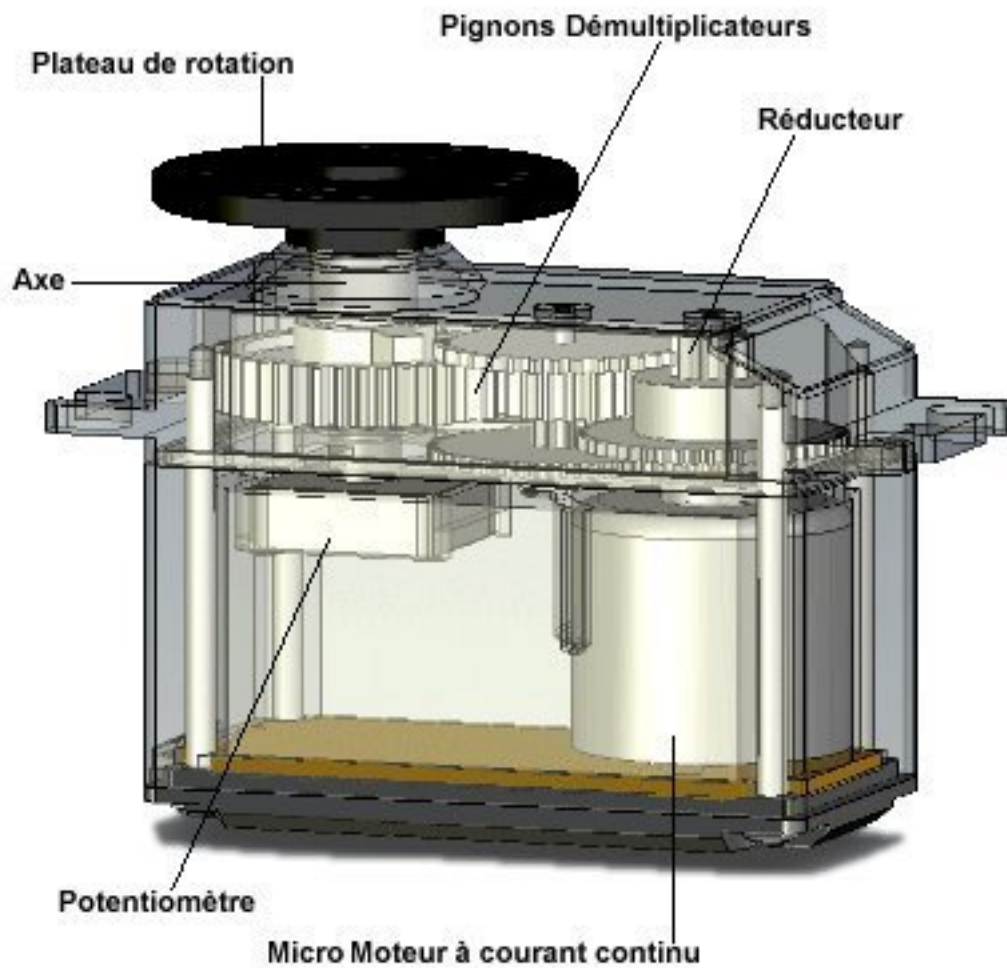
([Source de l'image](#))

7.2.1.2.2 Connectique Le servomoteur a besoin de trois fils de connexion pour fonctionner. Deux fils servent à son alimentation, le dernier étant celui qui reçoit le signal de commande :

- rouge : pour l'alimentation positive (4.5V à 6V en général)
- noir ou marron : pour la masse (0V)
- orange, jaune, blanc, ... : entrée du signal de commande

Nous verrons tout à l'heure ce que nous devons entrer sur le dernier fil.

7.2.1.2.3 La mécanique Comme on le voit dans l'image précédente, le servomoteur possède plusieurs pignons (engrenages) en sortie du petit moteur CC. Cet ensemble est ce qui constitue le **réducteur**. Ce réducteur fait deux choses : d'une part il réduit la vitesse de rotation en sortie de l'axe du servomoteur (et non du moteur CC), d'autre part il permet d'augmenter le couple en sortie du servomoteur (là encore non en sortie du moteur CC). Alors, à quoi ça sert de réduire la vitesse et d'augmenter le couple ? Eh bien les moteurs CC se débrouillent très bien pour tourner très vite, mais lorsqu'ils font une si petite taille ils sont bien moins bons pour fournir du couple. On va donc utiliser ce réducteur qui va réduire la vitesse, car nous n'avons pas besoin d'avoir une vitesse trop élevée, et augmenter le couple pour ainsi pouvoir déplacer une charge plus lourde.



Eric G

Figure 7.55 – Vue interne d'un servomoteur (sans l'électronique de commande)

Ceci est prouvé par la formule que je vous ai donnée dans le chapitre précédent : $R = \frac{\omega_{entree}}{\omega_{sortie}} = \frac{C_{sortie}}{C_{entree}}$.

Le rapport de réduction (R) du réducteur définit le couple et la vitesse de sortie (en sortie du réducteur) selon la vitesse et le couple d'entrée (en sortie du moteur CC). Ces données sont souvent transparentes lorsque l'on achète un servomoteur. Dans la quasi-totalité des cas, nous n'avons que la vitesse angulaire (en degré par seconde /s), le couple de sortie du servomoteur et le débattement maximal (s'il s'agit d'un servomoteur ayant un débattement de 0 à 90°, 180, 360 ou autre). Et c'est largement suffisant étant donné que c'est que ce qui nous intéresse dans le choix d'un servomoteur. Il y a cependant une unité qui pourra peut-être vous donner quelques doutes ou une certaine incompréhension. Cette caractéristique est celle du couple du servomoteur et a pour unité le *kg.cm* (kilogramme-centimètre). Nous allons tout de suite rappeler ce que cela signifie. Avant tout, rappelons la formule suivante : $C = F \times r$ qui donne la relation entre le couple C du servomoteur (en kilogramme mètre), F la force exercée sur le bras du servomoteur (en kilos) et r la distance (en m) à laquelle s'exerce cette force par rapport à l'axe de rotation du servomoteur. Disséquons dans notre langage la signification de cette formule : le couple (C) exercé sur un axe est égal à la force (F) appliquée au bout du levier accroché à ce même axe.

À force identique, plus le levier est long et plus le couple exercé sur cet axe est important. En d'autres termes, si votre servomoteur dispose d'un bras d'un mètre de long (oui c'est très long) eh bien il aura beaucoup plus de difficultés à soulever une charge de, disons 10g, que son homologue qui supporte la même charge avec un bras nettement raccourci à 10 centimètres. Prenons l'exemple d'un servomoteur assez commun, le [Futaba s3003](#). Sa documentation nous indique que lorsqu'il est alimenté sous 4.8V (on reviendra dessus plus tard), il peut fournir un couple (*torque* en anglais) de 3, 2kg.cm. C'est à dire, qu'au bout de son bras, s'il fait 1 centimètre, il pourra soulever une charge de 3,2kg. Simple, n'est-ce pas ? ; Si le bras fait 10 centimètres, vous aurez compris que l'on perd 10 fois la capacité à soulever une masse, on se retrouve alors avec un poids de 320g au maximum (sans compter le poids du bras lui-même, certes négligeable ici, mais parfois non).

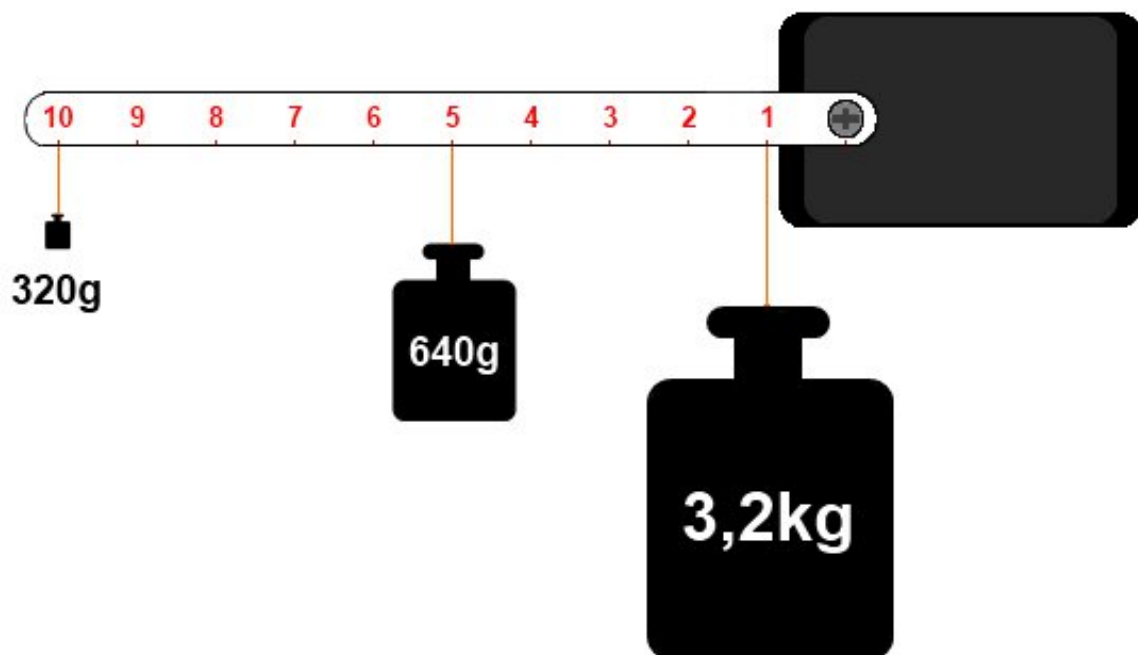


Figure 7.56 – Principe du couple mécanique

Voilà une image qui permet d'illustrer un peu ce que je vous raconte depuis tout à l'heure (ça commençait à être ennuyeux, non ?). Bref. Ici, chaque poids représenté est celui maximum que peut soulever le servomoteur selon la distance à laquelle il est situé. Et ne vous avisez pas de les mettre tous car votre pauvre servo serait bien dans l'incapacité de les soulever en même temps. Et oui, malgré le fait qu'il n'y ait que 320g au bout du bras, le servo voit comme s'il y avait un poids de 3,2kg ! Dans cette situation on aurait trois fois 3,2kg, ce qui ferait un poids total de 9,6kg ! Impossible pour le servo de ne bouger ne serait-ce que d'un millimètre (vous risqueriez fort de le détruire d'ailleurs).

[[q]] | Bon, d'accord, je comprends, mais et le zéro il y est pas sur ton dessin. Comment je sais quel poids je peux mettre sur l'axe du moteur ? o_O

Eh bien tout dépend du diamètre de cet axe. Voilà une question pertinente ! Alors, oui, répondons à la question. Mais avant, vous devriez avoir une idée de la réponse que je vais vous donner. Non ? Ben si, voyons ! Plus on éloigne le poids le l'axe et plus celui-ci diminue, et cela fonctionne dans l'autre sens : plus on le rapproche, plus sa valeur maximale augmente. En théorie, si on se met à 0cm, on pourrait mettre un poids infini. Admettons, plus rigoureusement, que l'on mette le poids à 1mm de l'axe (soit un axe de diamètre 2mm). Le poids que le servo pourrait soulever serait de... 10 fois plus ! Soit 32kg !! En conclusion, on peut admettre la formule suivante qui définit le poids maximal à mettre à la distance voulue :

$$P_{max} = \frac{C}{d}$$

Avec :

- P_{max} : poids maximal de charge en kilogramme (kg)
- C : couple du servomoteur, en kilogramme centimètre (kg.cm)
- d : distance à laquelle le poids est placé en centimètre (cm)

Et si on se concentrait sur le pourquoi du servomoteur, car son objectif principal est avant tout de donner une position angulaire à son bras. Allez, voyons ça tout de suite !

7.2.1.2.4 L'électronique d'asservissement “*Qu'est-ce que l'asservissement ?*”, vous demandez-vous sans doute en ce moment. Malgré la signification peu intuitive que ce terme porte, il se cache derrière quelque chose de simple à comprendre, mais parfois très compliqué à mettre en œuvre. Heureusement, ce n'est pas le cas pour le servomoteur. Toutefois, nous n'entrerons pas dans le détail et nous nous contenterons de présenter le fonctionnement. L'asservissement n'est ni plus ni moins qu'un moyen de gérer une consigne de régulation selon une commande d'entrée. Euuuh, vous me suivez ? :euh :

Prenons l'exemple du servomoteur : on l'alimente et on lui envoie un signal de commande qui permet de définir à quel angle va se positionner le bras du servomoteur. Ce dernier va s'exécuter. Essayez de forcer sur le bras du servomoteur... vous avez vu ? Quelle que soit la force que vous exercez (dans les limites du raisonnable), le servo va faire en sorte de toujours garder la position de son bras à l'angle voulu. Même si le poids est largement supérieur à ce qu'il peut supporter, il va essayer de remettre le bras dans la position à laquelle il se trouvait (à éviter cependant).

Ainsi, si vous changez l'angle du bras en forçant dessus, lorsque vous relâcherez le bras, il va immédiatement reprendre sa position initiale (celle définie grâce au signal de commande). Pour pouvoir réaliser le maintien de la position du bras de manière correcte, le servo utilise une **électronique de commande**. On peut la nommer **électronique d'asservissement**, car c'est elle qui va

gérer la position du bras du servomoteur. Cette électronique est constituée d'une zone de comparaison qui compare (étonnamment ^^) la position du bras du servo au signal de commande. Le deuxième élément qui constitue cette électronique, c'est le capteur de position du bras. Ce capteur n'est autre qu'un potentiomètre couplé à l'axe du moteur. La mesure de la tension au point milieu de ce potentiomètre permet d'obtenir une tension image de l'angle d'orientation du bras.

Cette position est ensuite comparée, je le disais, à la consigne (le signal de commande) qui est transmise au servomoteur. Après une rapide comparaison entre la consigne et valeur réelle de position du bras, le servomoteur (du moins son électronique de commande) va appliquer une correction si le bras n'est pas orienté à l'angle imposé par la consigne.

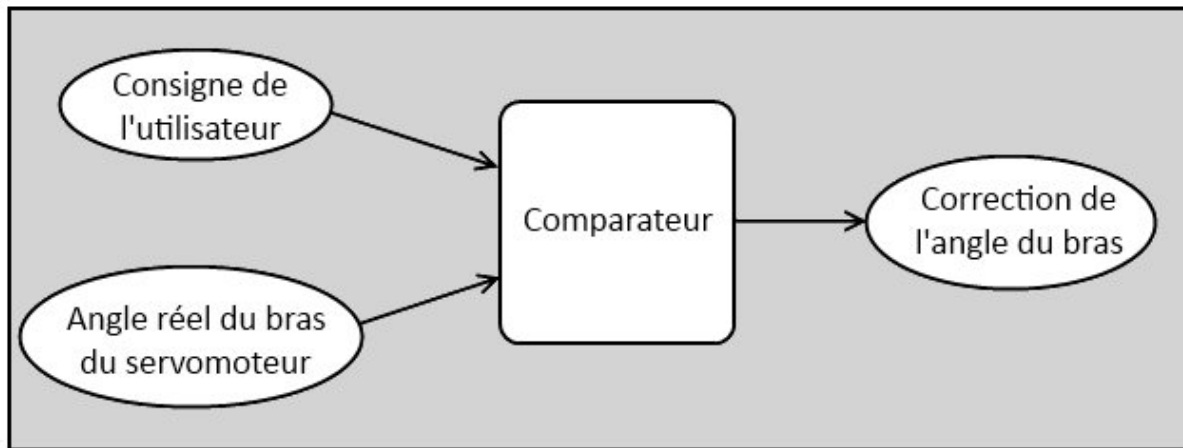


Figure 7.57 – Synoptique de fonctionnement de l'asservissement du servomoteur

Afin de garder la position de son bras stable, il est donc important de savoir quelle est la charge maximale applicable sur le bras du servomoteur. En somme, bien vérifier que le poids de la charge que vous comptez mettre sur votre servomoteur ne dépasse pas celui maximal qu'il peut supporter. Avant de passer à la suite, je vous propose de regarder cette superbe vidéo que j'ai trouvée par hasard sur [ce site web](#). Vous allez pouvoir comprendre au mieux le fonctionnement de la mécanique du servomoteur :

->!(<https://www.youtube.com/watch?v=-XSXfqd1N58>)<-

[[q]] | Mais au fait, comment est transmise la consigne de commande de position du bras ? On lui dit par la liaison série ?

C'est ce que nous allons voir tout de suite dans la partie suivante. En avant !

7.2.2 La commande d'un servomoteur

Ce qu'il est intéressant de découvrir à présent, c'est de savoir comment piloter un moteur de ce type. Eh oui, car cela n'a pas beaucoup de ressemblances avec le moteur à courant continu. Il ne va pas être question de pont en H ou autres bizarreries de ce type, non, vous allez voir, ça va être très simple.

[[i]] | Sachez toutefois qu'il existe deux types de servomoteur : ceux qui possèdent une électronique de commande de type analogique, qui sont les plus courants et les moins chers et ceux qui sont asservis par une électronique de commande numérique, très fiables et très performants, mais bien plus onéreux que leurs homologues analogiques. Vous comprendrez pourquoi notre

choix s'oriente sur le premier type. :P De plus, leur contrôle est bien plus simple que les servomoteurs à régulation numérique qui utilisent parfois des protocoles bien particuliers.

7.2.2.1 Le signal de commande

La consigne envoyée au servomoteur n'est autre qu'un signal électronique de type PWM. Il dispose cependant de deux caractéristiques indispensables pour que le servo puisse comprendre ce qu'on lui demande. À savoir : une fréquence fixe de valeur 50Hz (comme celle du réseau électrique EDF) et d'une durée d'état HAUT elle aussi fixée à certaines limites. Nous allons étudier l'affaire.

[[i]] | Certains sites de modélisme font état d'un nom pour ce signal : une PPM pour *Pulse Position Modulation*. J'utiliserais également ce terme de temps en temps, n'en soyez pas surpris !

7.2.2.1.1 La fréquence fixe Le signal que nous allons devoir générer doit avoir une fréquence de 50 Hz. Autrement dit, le temps séparant deux fronts montants est de 20 ms. Je rappelle la formule qui donne la relation entre la fréquence (F) et le temps de la période du signal (T) : $F = \frac{1}{T}$

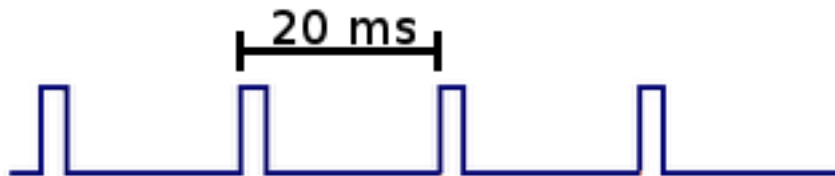


Figure 7.58 – Signal de fréquence 50 Hz

Malheureusement ,la fonction analogWrite() de Arduino ne possède pas une fréquence de 50Hz, mais dix fois plus élevée, de 500Hz environ. On ne pourra donc pas utiliser cette fonction.

[[q]] | Haaaaaaaaa! Mais comment on va faire !!! :'(

Ola, ne vous affolez pas! Il existe une alternative, ne vous pressez pas, on va voir ça dans un moment. ^^

7.2.2.1.2 La durée de l'état HAUT Pourquoi est-ce si important ? Qu'avons-nous à savoir sur la durée de l'état HAUT du signal PWM ? À quoi cela sert-il, finalement ? Eh bien ces questions trouvent leurs réponses dans ce qui va suivre, alors tendez bien l'oreille et ne perdez pas une miette de ce que je vais vous expliquer. ~ (Eh ! Entre nous, c'est pas mal cette petite intro, non ? Elle captive votre attention tout en faisant durer le suspense. Perso j'aime bien, pas vous ? Bon, je continue. ^^) ~ Cette durée, chers petits zéros, est ce qui compose l'essentiel du signal. Car c'est selon elle que le servomoteur va savoir comment positionner son bras à un angle précis. Vous connaissez comment fonctionne un signal PWM, qui sert également à piloter la vitesse d'un moteur à courant continu. Eh bien, pour le servomoteur, c'est quelque peu semblable. En fait, un signal ayant une durée d'état HAUT très faible donnera un angle à 0°, le même signal avec une durée d'état HAUT plus grande donnera un angle au maximum de ce que peut admettre le servomoteur. Mais, soyons rigoureux ! Précisément, je vous parlais de valeurs limites pour cet état HAUT et ce n'est pas pour rien, car ce dernier est limité entre une valeur de 1ms au minimum et au maximum de 2ms (ce sont bien des millisecondes puisque l'on parle de durée en temps) pour les servos standards. Comme un schéma vaut mieux qu'un long discours :

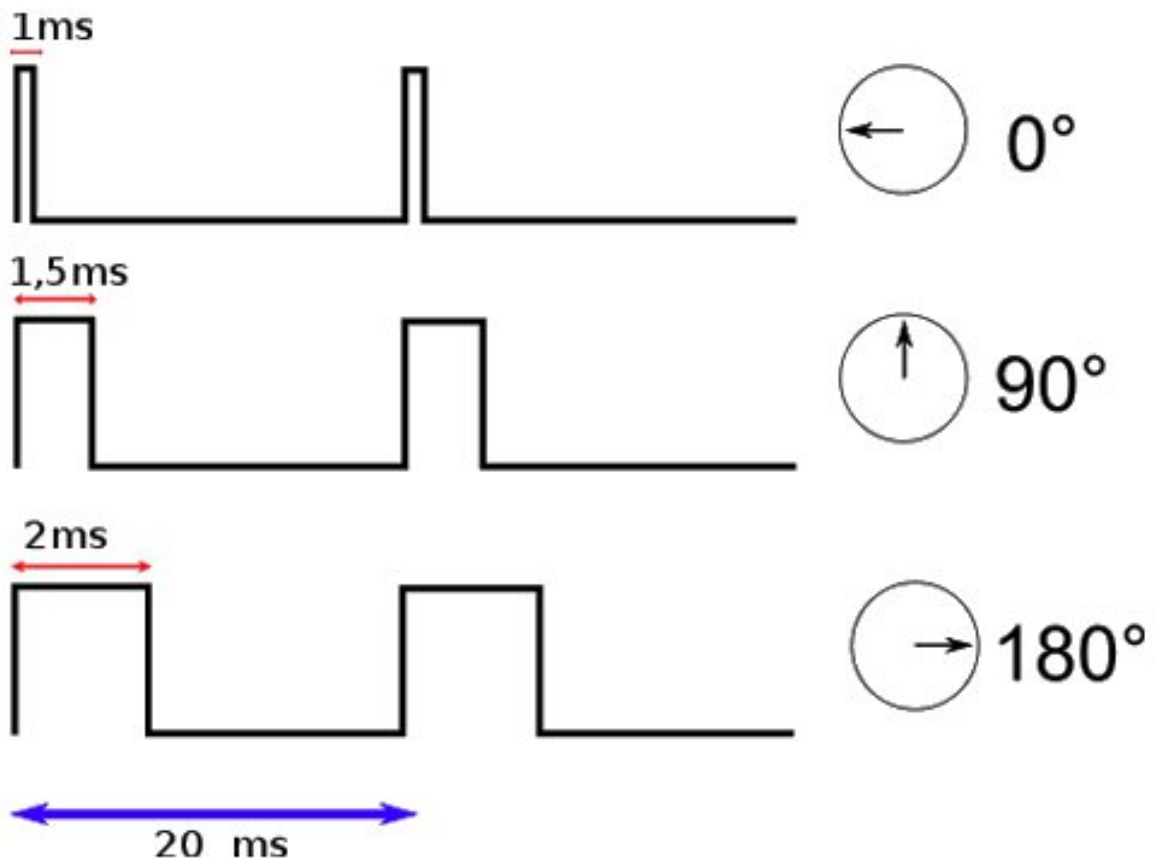


Figure 7.59 – Position en fonction de la pulsation

Vous aurez deviné, à travers cette illustration, que la durée de l'état HAUT fixe la position du bras du servomoteur à un angle déterminé.

[[q]] | Et comment je fais si je veux que mon servomoteur fasse un angle de 45° ? Ça ne marche pas? o_O

Si, bien sûr. En fait, il va falloir faire jouer le temps de l'état HAUT. Pour un angle de 45° , il va être compris entre 1ms et 1,5ms. À 1,25ms précisément. Après, c'est un rapport qui utilise une relation très simple, le calcul ne vous posera donc aucun problème. Tous les angles compris dans la limite de débattement du bras du servomoteur sont possibles et configurables grâce à ce fameux état HAUT.

[[q]] | Et si mon servomoteur n'a pas l'angle 0° pour origine, mais 90° , comment on fait?

C'est pareil! Disons que 90° est l'origine, donc on peut dire qu'il est à l'angle 0° , ce qui lui donne un débattement de -90° à $+90^\circ$:

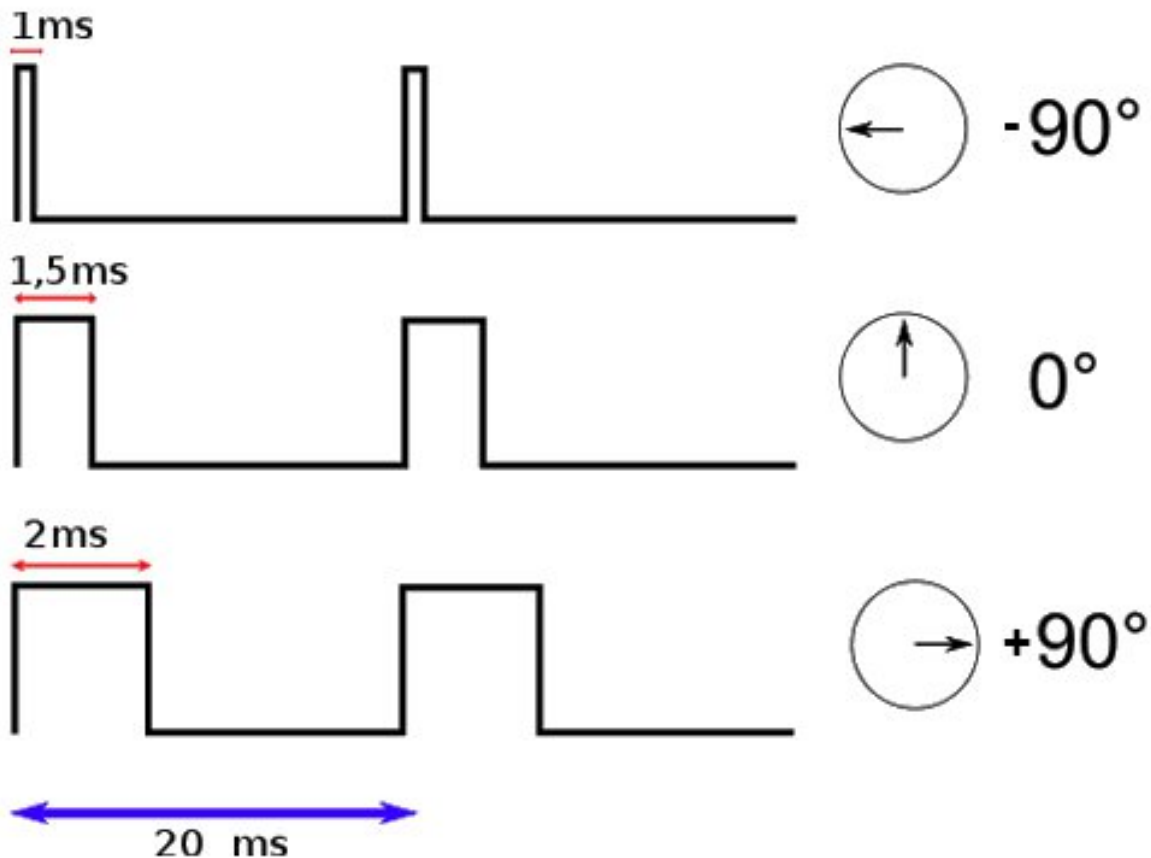


Figure 7.60 – Position en fonction de la pulsation avec décalage

Et dans le cas où le servo peut faire un tour complet (donc 360°), c'est aussi la même chose. En fait c'est toujours pareil, quel que soit le débattement du moteur. En revanche, c'est légèrement différent pour les servomoteurs à rotation continue. Le signal ayant un état HAUT de 1ms donnera l'ordre "vitesse maximale dans un sens", la même ayant 2ms sera l'ordre pour "vitesse maximale dans l'autre sens" et 1.5ms sera la consigne pour "moteur arrêté". Entre chaque temps (par exemple entre 1ms et 1,5ms), le moteur tournera à une vitesse proportionnelle à la durée de l'état HAUT. On peut donc commander la vitesse de rotation du servo.

7.2.3 Arduino et les servomoteurs

Bon, eh bien à présent, voyons un peu comment utiliser ces moteurs dont je vous vente les intérêts depuis tout à l'heure. Vous allez le voir, et ça ne vous surprendra même plus, la facilité d'utilisation est encore améliorée grâce à une bibliothèque intégrée à l'environnement Arduino. Ils nous mâchent vraiment tout le travail ces développeurs ! ^^

7.2.3.1 Câblage

Nous l'avons vu plus haut, la connectique d'un servomoteur se résume à trois fils : deux pour l'alimentation positive et la masse et le dernier pour le signal de commande. Rappelons qu'un servomoteur accepte généralement une plage d'alimentation comprise entre 4.5V et 6V (à 6V il aura plus de couple et sera un peu plus rapide qu'à 4.5V). Si vous n'avez besoin d'utiliser qu'un ou deux servomoteurs, vous pouvez les brancher sur la sortie 5V de la carte Arduino. Si vous voulez en utiliser plus, il serait bon d'envisager une alimentation externe car le régulateur de l'Arduino n'est pas fait pour délivrer trop de courant, vous risqueriez de le cramer. Dans ce cas, n'oubliez pas de relier la masse de l'alimentation externe et celle de l'Arduino afin de garder un référentiel électrique commun. Le câble permettant le transit du signal de commande du servo peut-être branché sur n'importe quelle broche de l'Arduino. Sachez cependant que lorsque nous utiliserons ces derniers, les sorties 9 et 10 ne pourront plus fournir un signal PWM (elles pourront cependant être utilisées comme de simples entrées/sorties numériques). C'est une des contraintes de la bibliothèque que nous allons utiliser.

[[i]] | Ces dernières contraintes s'appliquent différemment sur les cartes MEGA. [Cette page](#) vous dira tout !

Voici maintenant un petit exemple de montage d'un servo sur l'Arduino :

7.2.3.2 La librairie *Servo*

Pour utiliser le servo avec Arduino, il va nous falloir générer le signal PPM vu précédemment. C'est-à-dire créer un signal d'une fréquence de 50Hz et modifier l'état haut d'une durée comprise entre 1 et 2ms. Contraignant n'est-ce pas ? Surtout si l'on a plusieurs servos et tout un programme à gérer derrière... C'est pourquoi l'équipe d'Arduino a été sympa en implémentant une classe très bien nommée : *Servo*. Tout comme l'objet *Serial* vous permettait de faire abstraction du protocole de la voie série, l'objet *Servo* va vous permettre d'utiliser les servomoteurs. Et comme elle est développée par une équipe de personnes compétentes, on peut leur faire totalement confiance pour qu'elle soit optimisée et sans bugs ! ;) Voyons maintenant comment s'en servir !

7.2.3.2.1 Préparer le terrain

Tout d'abord, il nous faut inclure la librairie dans notre sketch. Pour cela, vous pouvez au choix écrire vous-même au début du code `#include <Servo.h>` ou alors cliquer sur *library* dans la barre de menu puis sur "Servo" pour que s'écrive automatiquement et sans faute la ligne précédente. Ensuite, il vous faudra créer un objet de type *Servo* pour chaque servomoteur que vous allez utiliser. Nous allons ici n'en créer qu'un seul que j'appellerai "monServo" de la manière suivante : `Servo monServo;`. Nous devons lui indiquer la broche sur laquelle est connecté le fil de commande du servo en utilisant la fonction `attach()` de l'objet *Servo* créé. Cette fonction prend 3 arguments :

- Le numéro de la broche sur laquelle est relié le fil de signal

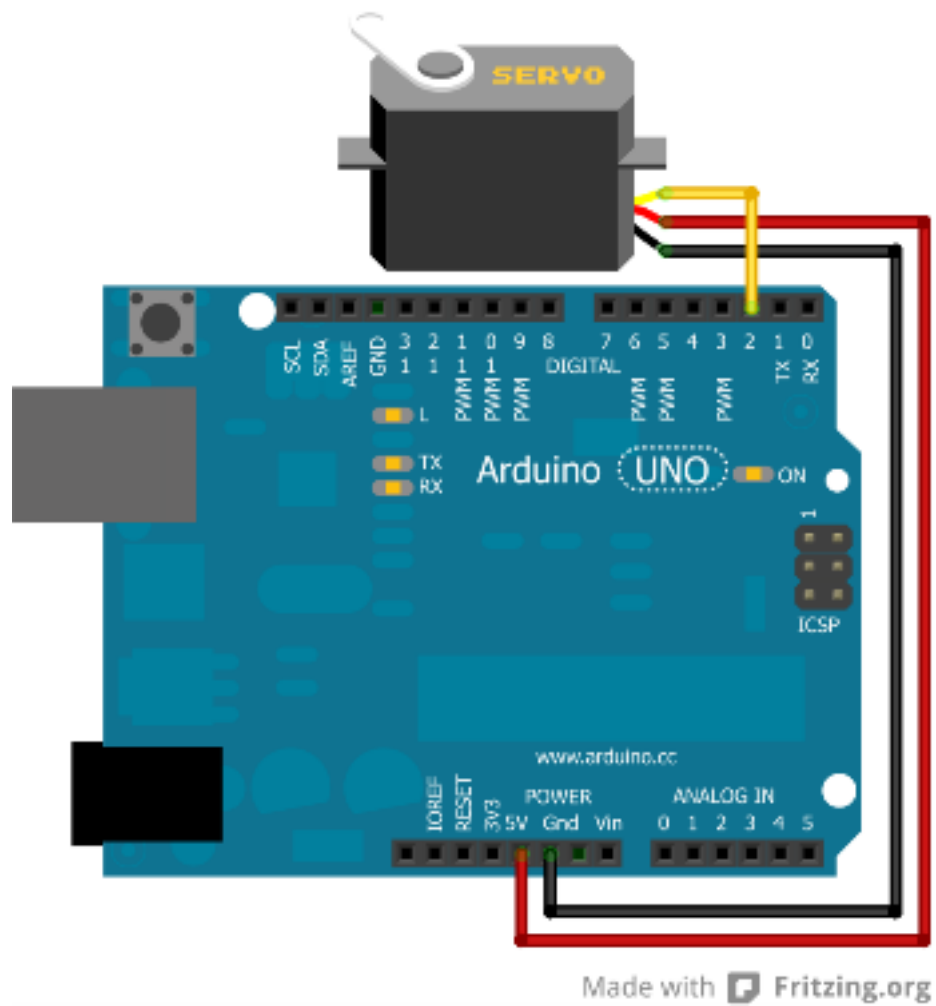


Figure 7.61 – Montage simple d'un servomoteur avec l'Arduino

- La valeur basse (angle à 0°) de la durée de l'état haut du signal de PPM en microsecondes (optionnel, défaut à 544 µs)
- La valeur haute (angle à 90°, 180°, 360°, etc.) de la durée de l'état haut du signal de PPM en microsecondes (optionnel, défaut à 2400 µs)

Par exemple, si mon servo possède comme caractéristique des durées de 1ms pour 0° et 2ms pour 180° et que je l'ai branché sur la broche 2, j'obtiendrais le code suivant :

```
####include <Servo.h>

Servo monServo;

void setup()
{
  monServo.attach(2, 1000, 2000);
}
```

Code : Initialisation d'un servo moteur

7.2.3.2 Utiliser le servo Une fois ces quelques étapes terminées, notre servo est fin prêt à être mis en route. Nous allons donc lui donner une consigne d'angle à laquelle il doit s'exécuter. Pour cela, nous allons utiliser la fonction prévue à cet effet : `w r i t e ()`. Tiens, c'est la même que lorsque l'on utilisait la liaison série ! Eh oui. ;) Comme son nom l'indique, elle va *écrire* quelque chose au servo. Ce quelque chose est l'angle qu'il doit donner à son axe. Cette fonction prend pour argument un nombre, de type `i n t`, qui donne la valeur en degré de l'angle à suivre. Si par exemple je veux placer le bras du servo à mi-chemin entre 0 et 180°, j'écrirais :

```
monServo.w r i t e ( 9 0 );
```

Pour terminer, voilà le code complet qui vous permettra de mettre l'angle du bras de votre servomoteur à 90° :

```
####include <Servo.h>

Servo monServo;

void setup()
{
  monServo.attach(2, 1000, 2000);
  monServo.w r i t e ( 9 0 );
}

void loop()
{
}
```

Code : Initialisation et déplacement d'un servo

J'ai mis l'ordre de l'angle dans la fonction `setup()` mais j'aurais tout autant pu la mettre dans la `loop()`. En effet, lorsque vous utilisez `w r i t e ()`, la valeur est enregistrée par Arduino et est ensuite envoyée 50 fois par seconde (rappelez-vous du 50Hz du signal ;)) au servo moteur afin qu'il garde toujours la position demandée.

7.2.4 L'électronique d'asservissement

Je le disais donc, on va voir un peu comment se profile le fonctionnement de l'électronique interne des servomoteurs analogiques. Je précise bien *analogiques* car je rappelle qu'il y a aussi des servomoteurs numériques, beaucoup plus complexes au niveau de l'électronique.

7.2.4.1 Principe de fonctionnement

Commençons par un simple synoptique de fonctionnement. Référez-vous à la vidéo et aux explications que je vous ai données jusqu'à présent pour comprendre ce synoptique :

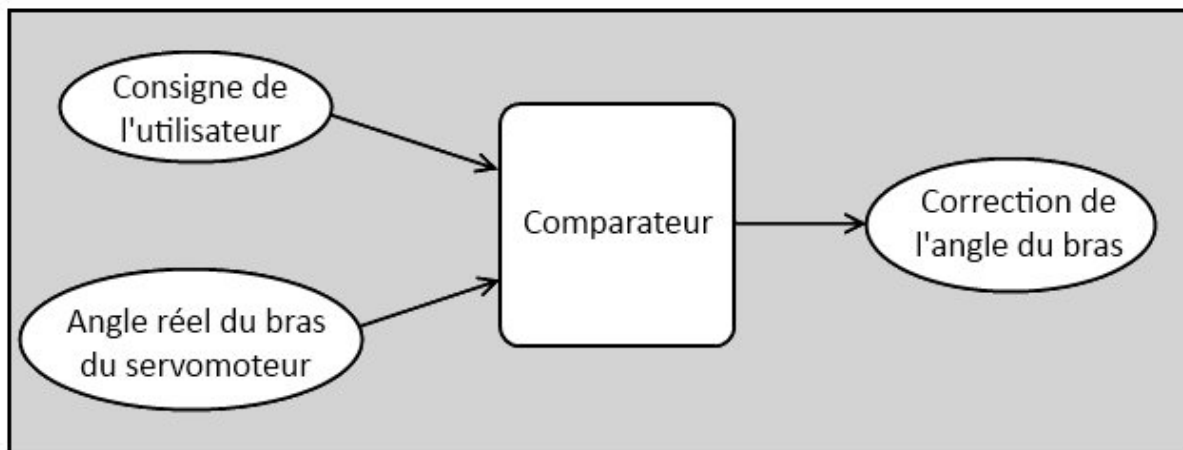


Figure 7.62 – Principe de fonctionnement de l'électronique de commande d'un servomoteur

Rapidement : la consigne donnée par l'utilisateur (dans notre cas, il va s'agir du signal envoyé par la carte Arduino) est comparée par rapport à la position réelle de l'axe du moteur. Ainsi, s'il y a une différence d'angle entre la consigne et l'angle mesuré par le capteur (le potentiomètre qui est fixé sur l'axe du servomoteur) eh bien le comparateur va commander le moteur et le faire tourner jusqu'à ce que cette différence s'annule.

[[a]] | Avant d'aller plus loin, il faut savoir que les servomoteurs analogiques du commerce emploient en fait, dans leur électronique de commande, un microcontrôleur. Je ne vais donc pas vous expliquer comment ceux-là fonctionnent, mais je vais prendre le montage le plus basique qui soit. D'ailleurs, à l'issue de mes explications, vous serez capable de mettre en œuvre le montage que je vais donner et créer votre propre servomoteur avec un moteur CC anodin. ;)

7.2.4.2 Électronique à consigne manuelle

On va commencer par un montage dont la simplicité est extrême, mais dont vous ne connaissez pas encore le fonctionnement d'un composant essentiel : le **comparateur**. Allez, c'est parti pour un bon petit cours d'électronique pure ! :euh : Alors, déjà, pourquoi "manuelle" ? Simplement parce que la consigne envoyée à l'électronique de commande est une tension continue et qu'elle sera réglable par un potentiomètre. En gros vous aurez simplement à faire tourner l'axe d'un potentiomètre pour régler l'angle du bras du servomoteur.

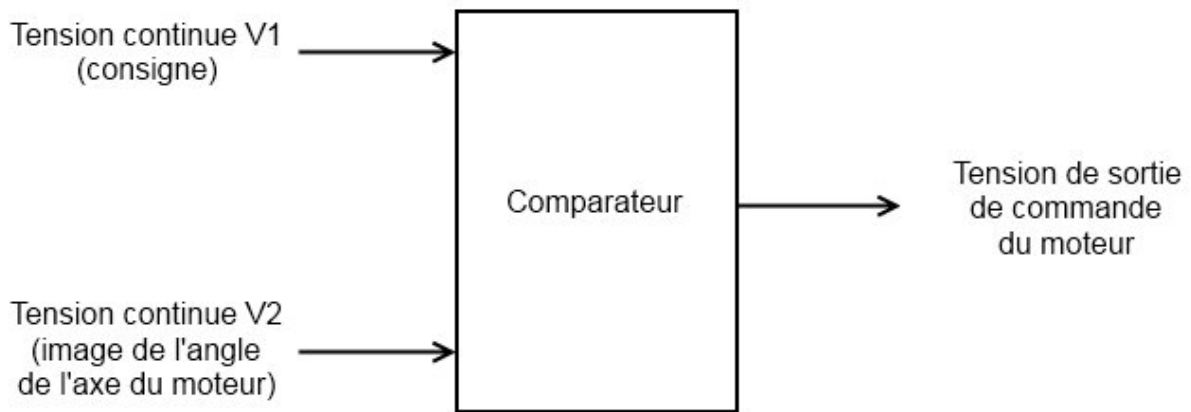


Figure 7.63 – Fonctionnement de l'électronique de contrôle

7.2.4.2.1 Synoptique de l'électronique interne Commençons par un synoptique qui établit le fonctionnement de l'électronique de contrôle :

Il y a donc en entrée les deux paramètres : la consigne et l'angle réel de l'axe du moteur ; Et en sortie, la tension qui va commander le moteur. On l'a vu, un moteur à courant continu doit être commandé par une tension continue, si cette tension est positive, le moteur tournera dans un sens, si elle est négative, le moteur tournera dans l'autre sens. C'est pourquoi le comparateur délivrera une tension positive ou négative selon la correction d'angle à effectuer.

7.2.4.2.2 Schéma de principe À présent, voici le schéma de principe qui a pour fonctionnement celui expliqué par le synoptique précédent :

De gauche à droite on a : les alimentations qui fournissent la tension positive et négative ; les potentiomètres P1 et P2 ; le comparateur (oui c'est ce gros triangle avec un plus et un moins) ; enfin le moteur à courant continu.

7.2.4.2.3 Fonctionnement du comparateur Un comparateur est un composant électronique de la famille des circuits intégrés car, il contient en vérité d'autres composants, essentiellement des semi-conducteurs (diodes, transistors) et des résistances. **Ce composant a toujours besoin d'une alimentation externe pour fonctionner, c'est-à-dire qu'on ne peut lui mettre des signaux à son entrée que s'il est alimenté.** Autrement, il pourrait être endommagé (ce n'est pas souvent le cas, mais mieux vaut être prudent). Vous le constatez par vous-même, le comparateur est un composant qui possède deux entrées et une sortie. Et, de la manière la plus simple qui soit, en fait il n'y a rien de plus simple qui puisse exister, son fonctionnement réside sur le principe suivant :

- Si la tension (je me base par rapport au schéma) $V1$ qui arrive sur l'entrée $E1$ du comparateur est supérieure à la tension $V2$ qui entre sur l'entrée $E2$ du comparateur, alors la tension en sortie S du comparateur est égale à $+V_{cc}$ (l'alimentation du comparateur).
- Tandis que dans le cas opposé où la tension $V2$ va être supérieure à $V1$, la sortie S du comparateur aura une tension égale à $-V_{cc}$.

En transposant mes dires sous une forme mathématique, cela donnerait ceci :

- Si $V1 > V2$, alors $V_s = +V_{cc}$
- Si $V1 < V2$, alors $V_s = -V_{cc}$

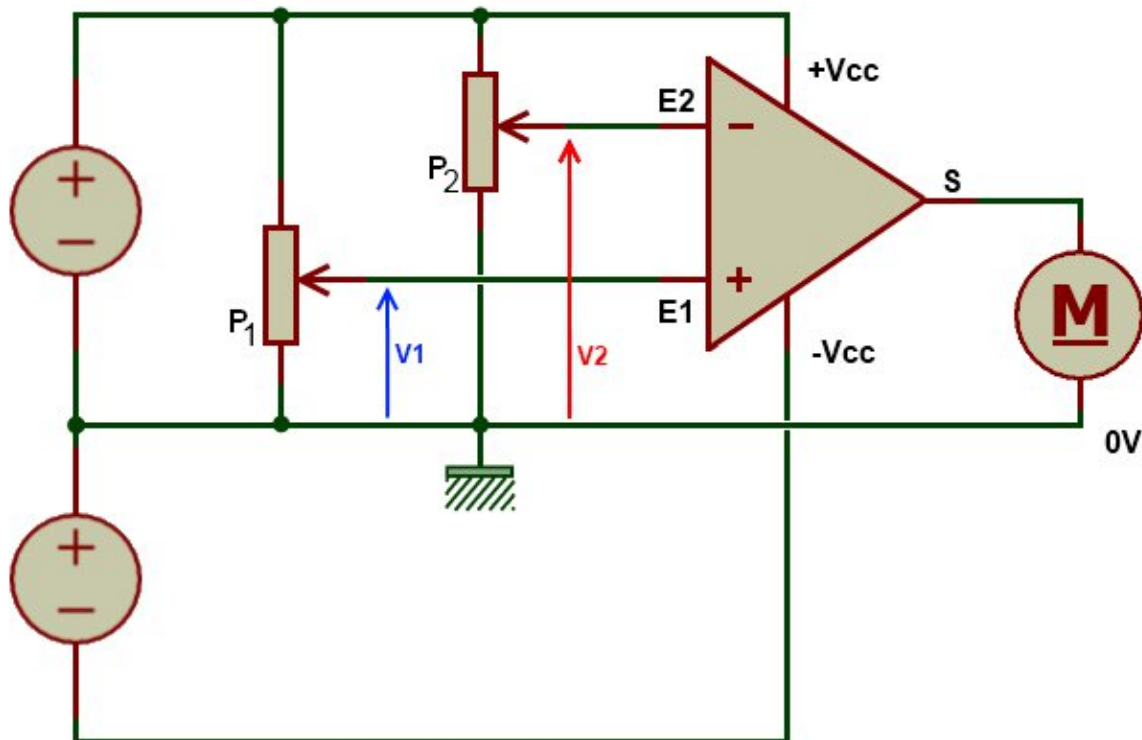


Figure 7.64 – Schéma de principe

Comment s'en rappeler ? Eh bien grâce aux petits symboles “+” et “-” présents dans le triangle représentant le comparateur. La sortie du comparateur prendra +Vcc si la tension sur l'entrée “+” du comparateur est supérieure à celle sur l'entrée “-” et inversement. Voyez, j'avais dit que c'était ultra simple. ;) Il y a encore quelque chose à savoir : il est impossible que les tensions V_1 et V_2 soient égales ! Oui, car le comparateur ne peut pas fournir une tension positive ET une tension négative en sa sortie, c'est pourquoi, même si vous reliez E1 et E2 avec un fil, la tension en sortie du comparateur sera toujours OU +Vcc OU -Vcc.

7.2.4.3 Électronique à consigne PWM

7.2.4.3.1 Synoptique de principe Prenons l'exemple d'un servomoteur qui utilise une PWM, oui j'ai bien dit... euh écrit PWM. Je prends cet exemple fictif car comme je le disais il y a quelques instants, c'est bien souvent un microcontrôleur qui gère l'asservissement du servomoteur. Et puis, avec l'exemple que je vais vous donner, vous pourrez vous-même créer un servomoteur. ;) En fait, on ne peut pas utiliser directement ce signal PWM avec le schéma précédent. Il va falloir que l'on fasse une extraction de la composante continue de ce signal pour obtenir une consigne dont la tension varie et non la durée de l'état HAUT du signal. Et ceci, nous l'avons déjà vu dans [un chapitre dédié à la PWM](#) justement. Le synoptique ne change guère, il y a simplement ajout de ce montage intermédiaire qui va extraire cette tension continue du signal :

Le schéma électrique ne change pas non plus de beaucoup, on retire le potentiomètre qui permettait de régler la consigne manuellement en le remplaçant par le montage qui fait l'extraction de la composante continue :

À la place du potentiomètre de commande manuelle on retrouve un couple résistance/condensateur

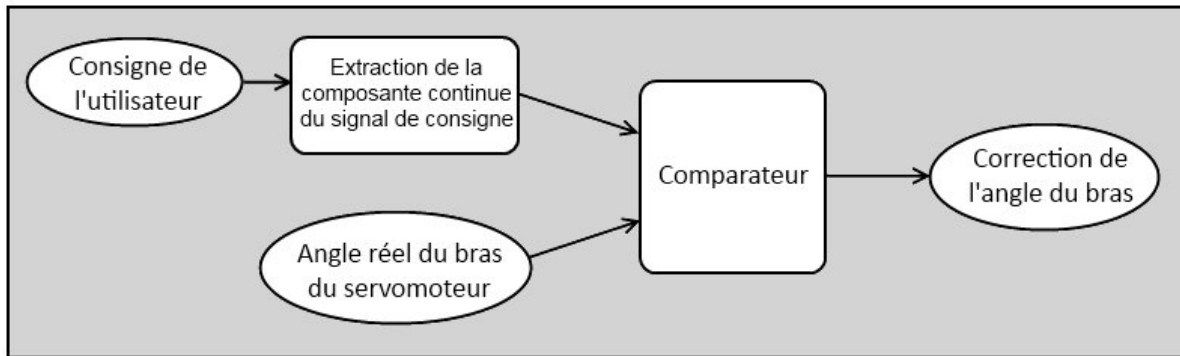


Figure 7.65 – Principe d'extraction de la tension continue du signal

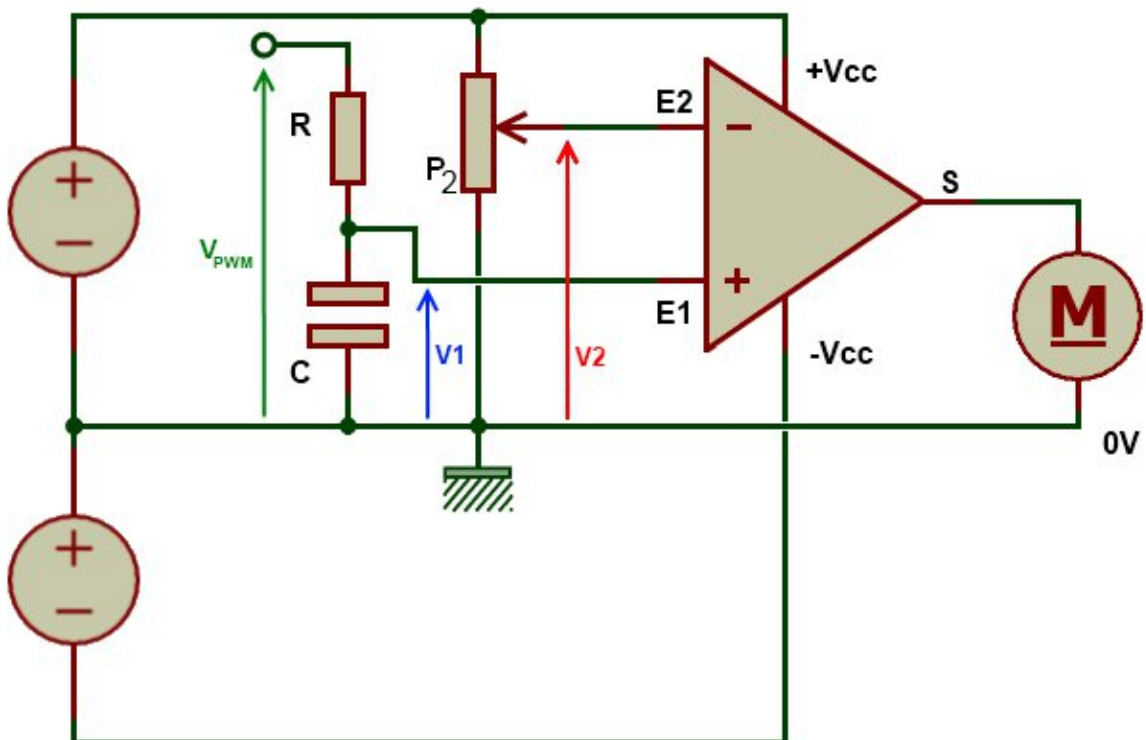


Figure 7.66 – Montage d'extraction de la tension continue du signal

avec R et C , qui permet d'extraire la tension continue du signal V_{PWM} qui est donc un signal de type PWM dont le rapport cyclique varie de 0 à 100%. Et là, tenez-vous bien, on en arrive au point où je voulais vous amener ! :D Que remarquez-vous ? Rien ? Alors je vous le dis : que se passe-t-il si on arrête d'envoyer le signal V_{PWM} ? Le moteur garde son bras au même angle ? Ou bien il reprend sa position initiale ? Réponse : il reprend sa position initiale. Eh oui, car la tension continue $V1$ n'existe plus puisqu'elle est créée à partir du signal V_{PWM} . Quand il y avait le potentiomètre, la tension $V1$ gardait la même valeur tant que vous ne tourniez pas l'axe du potentiomètre, hors là, si on enlève le signal V_{PWM} , eh bien la tension $V1$ perd sa valeur et retombe à 0V. Par conséquent, le moteur redonne à son bras sa position initiale.

[[q]] | Et si je veux que mon servomoteur continue de garder l'angle de la consigne qui lui a été transmise sans que je continue à lui envoyer cette consigne, est-ce possible ?

Oui, c'est tout à fait possible. En fait, cela va peut-être paraître un peu "barbare", mais c'est la seule solution envisageable avec les servomoteurs analogiques : il suffit de le positionner à l'angle voulu et de couper son alimentation. L'angle du bras du servomoteur sera alors conservé. **Mais attention, cet angle ne sera conservé que s'il n'y a pas de contrainte mécanique exercée sur le bras du servo !** C'est-à-dire qu'il n'y ait pas un poids accroché à l'axe du moteur, ou alors il faut qu'il soit bien inférieur à la force de maintien de la position du bras du servo lorsque celui-ci n'est plus alimenté.

[[q]] | Et pour l'électronique à consigne PPM alors ? o_O

Pour ce type d'électronique de commande (présent dans tous les servos du commerce), je vous l'ai dit : il y a utilisation d'un microcontrôleur. Donc tout se fait par un programme qui scrute la position réelle du bras du moteur par rapport à la consigne PPM qu'il reçoit. Je n'ai donc rien d'intéressant à vous raconter. :-°

7.2.5 Un peu d'exercice !

Bon allez, il est temps de faire un peu d'entraînement ! Je vous laisse découvrir le sujet...

7.2.5.0.1 Consigne Nous allons utiliser trois éléments dans cet exercice :

- un servomoteur (évidemment)
- un potentiomètre (valeur de votre choix)
- la liaison série

7.2.5.0.2 Objectif Le servo doit "suivre" le potentiomètre. C'est-à-dire que lorsque vous faites tourner l'axe du potentiomètre, le bras du servomoteur doit tourner à son tour et dans le même sens. Pour ce qui est de l'utilisation de la liaison série, je veux simplement que l'on ait un retour de la valeur donnée par le potentiomètre pour faire une supervision. Je ne vous en dis pas plus, vous savez déjà tout faire. Bon courage et à plus tard ! ;)

->!(<https://www.youtube.com/watch?v=LQt19KkunVE>)<-

7.2.5.1 Correction

J'espère que vous avez réussi ! Tout d'abord le schéma, même si je sais que vous avez été capable de faire les branchements par vous-même. C'est toujours bon de l'avoir sous les yeux. ;)

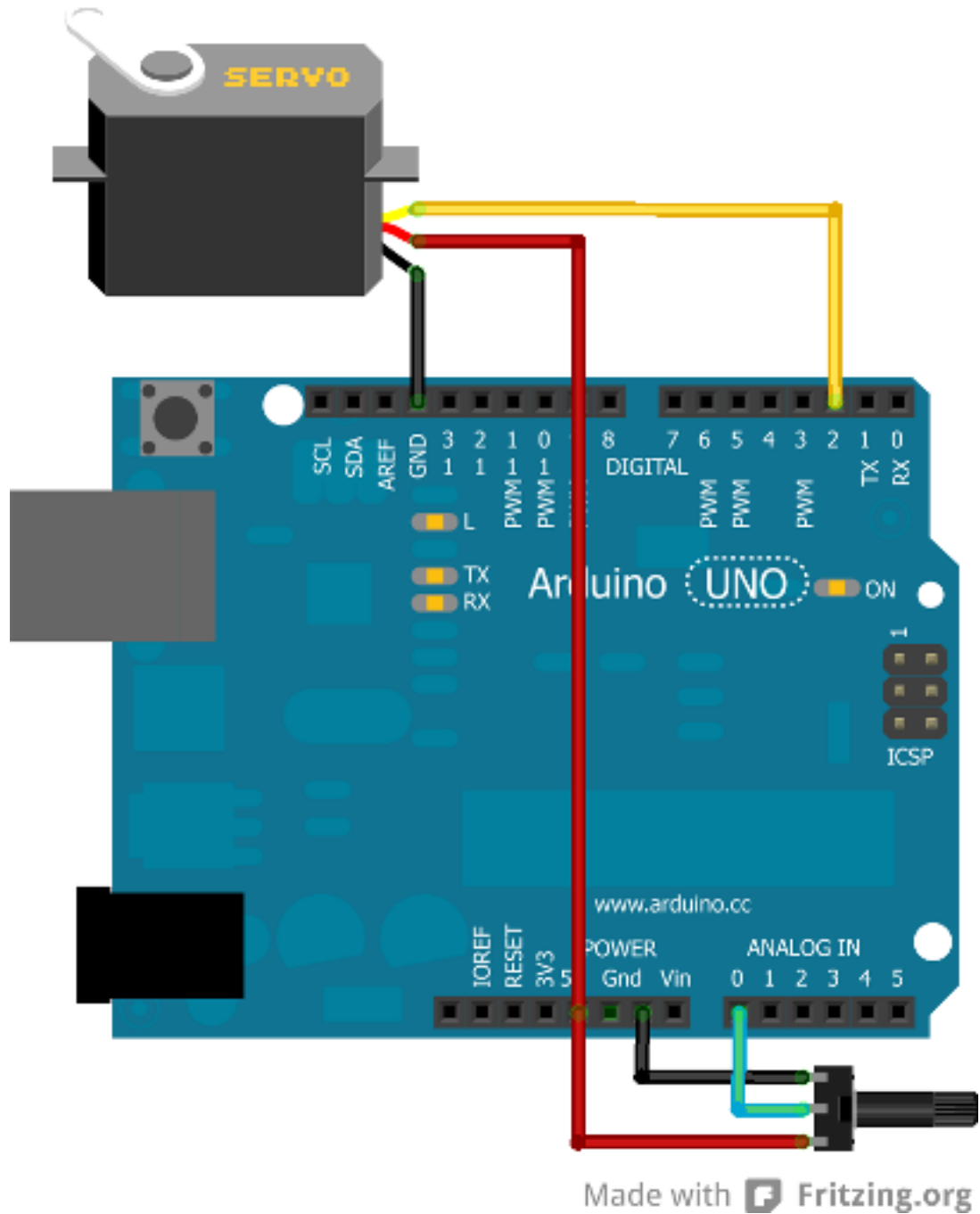


Figure 7.67 – Arduino branché avec un servomoteur et un potentiomètre

Pour ma part, j'ai branché le servo sur la broche numérique 2 et le potentiomètre sur la broche analogique 0. J'ai donc le code suivant pour préparer l'ensemble :

```
####include <Servo.h> // On n'oublie pas d'ajouter la bibliothèque!

// notre potentiomètre
const int potar = 0;

// création d'un nouveau servomoteur
Servo monServo;

void setup()
{
    // on déclare l'entrée du servo connectée sur la broche 2
    monServo.attach(2);
    // on n'oublie pas de démarrer la liaison série;-)
    Serial.begin(9600);
}
```

Voilà qui est fait pour les préparatifs, il n'y a plus qu'à travailler un tout petit peu pour faire la logique du code. Commençons par la lecture analogique que nous allons renvoyer sur le servo ensuite. Le potentiomètre délivre une tension variable de 0 à 5V selon sa position. La carte Arduino, elle, lit une valeur comprise entre 0 et 1023. Ce nombre est stocké au format `int`. Il faut ensuite que l'on donne à la fonction qui permet d'envoyer la consigne au servo une valeur comprise entre 0 et 180°. On va donc utiliser une fonction dédiée à cela. Cette fonction permet de faire le rapport entre deux gammes de valeurs ayant chacune des extremums différents. Il s'agit de la fonction `map()` (nous en avons parlé dans le chapitre sur les lectures analogiques) :

```
map(value, fromLow, fromHigh, toLow, toHigh)
```

Avec pour correspondance :

- **value** : valeur à convertir pour la changer de gamme
- **fromLow** : valeur minimale de la gamme à convertir
- **fromHigh** : valeur maximale de la gamme à convertir
- **toLow** : valeur minimale de la gamme vers laquelle est convertie la valeur initiale
- **toHigh** : valeur maximale de la gamme vers laquelle est convertie la valeur initiale

Nous utiliserons cette fonction de la manière suivante :

```
map(valeur_potentiometre, 0, 1023, 0, 180);
```

[[q]] | On aurait pu faire un simple produit en croix, non ?

Tout à fait. Mais les programmeurs sont de véritables fainéants et aiment utiliser des outils déjà prêts. :P Cela dit, ils les ont créés. Et pour créer de nouveaux outils, il est plus facile de prendre des outils déjà existants. Mais si vous voulez, on peut recréer la fonction `map()` par nous-mêmes :

```
int conversion(int mesure)
{
    return mesure*180/1023;
}
```

7.2.5.1.1 Fonction loop() Dans la fonction loop() on a donc la récupération et l'envoi de la consigne au servomoteur :

```
void loop()
{
    // on lit la valeur du potentiomètre
    int val = analogRead(potar);
    // mise à l'échelle de la valeur lue vers la plage [0;180]
    int angle = map(val, 0, 1023, 0, 180);
    // on met à jour l'angle sur le servo
    monServo.write(angle);
}
```

Code : Rotation du servo en fonction du potentiomètre

Avez-vous remarqué que ces trois lignes de code auraient pu être réduites en une seule ? Comme ceci :

```
monServo.write(map(analogRead(potar), 0, 1023, 0, 180));
```

Ou bien la version utilisant le produit en croix :

```
[[s]] | cpp | void loop() | { | // on lit la valeur du potentiomètre | int
val = analogRead(potar); | // on converti la valeur lue en angle compris
dans l'intervalle [0;180] | int angle = val / 5.7; | // 5,7 provient de la
division de 1023/180 | // pour la mise à l'échelle de la valeur lue | | //
on met à jour l'angle sur le servo | monServo.write(angle); | } || Et à nou-
veau une condensation de ces trois lignes en une: | cpp | monServo.write(analogRead(potar)/5.7);
|
```

[[a]] Mais comme il nous faut renvoyer la valeur convertie vers l'ordinateur, il est mieux de stocker cette valeur dans une variable. Autrement dit, préférez garder le code à trois lignes.

7.2.5.1.2 Et la liaison série Pour renvoyer la valeur, rien de bien sorcier :

```
Serial.println(angle);
```

7.2.5.1.3 Code final Au final, on se retrouve avec un code tel que celui-ci :

```
[[s]]|cpp | #include <Servo.h> // On n'oublie pas d'ajouter la bibliothèque !
| | const int potar = 0; // notre potentiomètre | Servo monServo; | | void
setup() | { | // on déclare le servo sur la broche 2 (éventuellement régler
les bornes) | monServo.attach(2); | // on n'oublie pas de démarrer la voie
série | Serial.begin(9600); | } | | void loop() | { | // on lit la valeur
du potentiomètre | int val = analogRead(potar); | // on convertit la valeur
lue en angle compris dans l'intervalle [0;180] | int angle = val / 5.7;
| // on met à jour l'angle sur le servo | monServo.write(angle); | // on
renvoie l'angle par la voie série pour superviser | Serial.println(angle);
| // un petit temps de pause | delay(100); | } || Code : Code final de l'exercice
de rotation d'un servo en fonction d'un potentiomètre
```

Je vous laisse mixer avec les différents codes que l'on vous a donnés pour que vous fassiez celui qui vous convient le mieux (avec la fonction map(), ou bien celui qui est tout condensé, etc.). Dorénavant, vous allez pouvoir vous amuser avec les servomoteurs ! ;)

7.2.6 Tester un servomoteur “non-standard”

[[q]] | C’est déjà la fin ? o_O

Eh oui, je n’ai plus grand-chose à vous dire, car ce n’est pas très compliqué puisqu’il suffit d’utiliser un outil déjà tout prêt qui est la bibliothèque *Servo*. Je vais cependant vous montrer deux autres fonctions bien utiles.

7.2.6.0.1 `writeMicroSeconds()` En premier, la fonction `writeMicroSeconds()`. Cette fonction permet de définir un temps à l’état HAUT du signal PPM autre que celui compris entre 1 et 2 ms. Elle est très pratique pour tester un servo dont vous ne connaissez pas les caractéristiques (servo 0 à 90° ou autre). De plus, il arrive que certains constructeurs ne se soucient pas trop des standards [1ms-2ms] et dépassent un peu ces valeurs. De par ce fait, si vous utilisez un servo avec les valeurs originales vous n’obtiendrez pas le comportement escompté. En utilisant cette fonction, vous pourrez ainsi tester le servo petit à petit en envoyant différentes valeurs une à une (par la voie série par exemple).

[[i]] | Une valeur incorrecte se repère assez facilement. Si vous voyez votre servo “trembler” aux alentours des 0° ou 180° ou bien encore s’il fait des allers-retours étranges sans que vous n’ayez changé la consigne alors c’est que la valeur utilisée est probablement fausse.

7.2.6.0.2 `read()` Une deuxième fonction pouvant être utile est la fonction `read()`. Tout l’intérêt de cette fonction est perdu si elle est utilisée pour le code que l’on a vu dans l’exercice précédent. En revanche, elle a très bien sa place dans un système où le servomoteur est géré automatiquement par le programme de la carte Arduino et où l’utilisateur ne peut y accéder.

7.2.6.1 Programme de test

En préparant ce chapitre, j’ai pu commencer à jouer avec un servomoteur issu de mes fonds de tiroirs. N’ayant bien entendu aucune documentation sur place ou sur internet, j’ai commencé à jouer avec en assumant qu’il utiliserait des valeurs “standards”, donc entre 1000 et 2000µs pour l’état haut de la PPM. J’ai ainsi pu constater que mon servo fonctionnait, mais on était loin de parcourir les 180° attendus. J’ai donc fait un petit code utilisant une des fonctions précédentes pour tester le moteur en mode “pas à pas” et ainsi trouver les vrais timings de ces bornes. Pour cela, j’ai utilisé la liaison série. Elle m’a servi pour envoyer une commande simple (‘a’ pour augmenter la consigne, ‘d’ pour la diminuer). Ainsi, en recherchant à tâtons et en observant le comportement du moteur, j’ai pu déterminer qu’il était borné entre 560 et 2130 µs. Pas super proche des 1 et 2ms attendues! :P Comme je suis sympa (:euh :), je vous donne le code que j’ai réalisé pour le tester. Les symptômes à observer sont : aucune réaction du servo (pour ma part en dessous de 560 il ne se passe plus rien) ou au contraire, du mouvement sans changement de la consigne (de mon côté, si l’on augmente au-dessus de 2130 le servo va continuer à tourner sans s’arrêter).

```
####include <Servo.h> // On oublie pas d'ajouter la bibliothèque!
```

```
int temps = 1500; // censée être à mi-chemin entre 1000 et 2000, un bon point de départ
```

```
Servo monServo;
```

```
void setup()
```

```

{
  Serial.begin(115200);
  Serial.println("Hello World");

  monServo.attach(2);
  // on démarre à une valeur censé être la moitié de
  // l'excursion totale de l'angle réalisé par le servomoteur
  monServo.writeMicroseconds(temps);
}

void loop()
{
  // des données sur la liaison série? (lorsque l'on appuie sur 'a' ou 'd')
  if(Serial.available())
  {
    char commande = Serial.read(); // on lit

    // on modifie la consigne si c'est un caractère qui nous intéresse
    if(commande == 'a')
      temps += 10; // ajout de 10µs au temps HAUT
    else if(commande == 'd')
      temps -= 10; // retrait de 10µs au temps HAUT

    // on modifie la consigne du servo
    monServo.writeMicroseconds(temps);

    // et on fait un retour sur la console pour savoir où on est rendu
    Serial.println(temps, DEC);
  }
}

```

Code : Programme de test d'un servomoteur

Ce programme est très simple d'utilisation et vous pouvez d'ailleurs le modifier comme bon vous semble pour qu'il corresponde à ce que vous voulez faire avec. Il suffit en fait de brancher la carte Arduino à un ordinateur et ouvrir un terminal série (par exemple le moniteur intégré dans l'environnement Arduino). Ensuite, appuyez sur 'a' ou 'd' pour faire augmenter ou diminuer le temps de l'état HAUT du signal PPM. Vous pourrez ainsi avoir un retour des temps extrêmes qu'utilise votre servomoteur.

On en termine avec les servomoteurs. Vous avez sans doute plein de nouvelles idées avec lesquelles vous emploieriez les servomoteurs qui vous permettront de faire beaucoup de choses très utiles, voire inutiles mais indispensables. :P

7.3 A petits pas, le moteur pas-à-pas

Pour en terminer avec les différents types de moteurs qui existent, nous allons parler d'un moteur un peu particulier (encore plus que le servomoteur !) et qui est cependant très utilisé dans le domaine de la robotique et tout ce qui touche à la précision d'un mouvement. Comme à l'habitude, nous allons d'abord voir le fonctionnement de ces moteurs, pour ensuite apprendre à les utiliser.

[[i]] | Ce moteur utilise des éléments que nous avons vus dans des chapitres précédents (sur les moteurs à courant continu). Si vous ne vous rappelez pas du L298 je vous conseille de retourner prendre quelques informations à ce sujet. ;)

7.3.1 Les différents moteurs pas-à-pas et leur fonctionnement

Les moteurs pas-à-pas... encore un nouveau type de moteur. Une question vous taraude sûrement l'esprit :

[[q]] | Pourquoi il existe tant de moteurs différents !?

Et bien je vous répondrais par une autre question : pourquoi existe-t'il autant de langages de programmation différents !? La réponse est pourtant simple : car ils ont tous leurs avantages et leurs inconvénients. Par exemple, un servomoteur pourra facilement maintenir la position de son axe, tandis que le moteur à courant continu sera plus facile à faire tourner à différentes vitesses. Eh bien, le but du moteur pas-à-pas (que j'abrégerais moteur pàp) est un peu une réunion de ces deux avantages. Vous pourrez le faire tourner à des vitesses variables et la position parcourue sera aussi facile à déterminer. En contrepartie, ces moteurs ne peuvent pas tourner à des vitesses hallucinantes et sont plus délicats à mettre en œuvre que les moteurs CC par exemple (mais rien d'insurmontable je vous rassure).

En parlant de précision, savez-vous dans quel objet du quotidien on retrouve beaucoup de moteurs pàp ? Dans l'imprimante (éventuellement scanner aussi) qui traîne sur votre bureau ! En effet, l'aspect "précision" du moteur est utilisé dans cette situation sans avoir besoin d'aller vraiment vite. Vous pourrez donc en trouver un pour faire avancer les feuilles et un autre pour déplacer le chariot avec les cartouches d'encre (et encore un autre pour déplacer le capteur du scanner). Donc si vous avez une vieille imprimante destinée à la poubelle, vous savez ce qu'il vous reste à faire ;)! Les moteurs que vous pourrez trouver posséderont 4, 5 voire 6 fils. Le premier (4 fils) est appelé **moteur bipolaire**, les deux autres sont des moteurs **unipolaires** ou à **réluctance variable**. Tout cela doit-être encore un peu confus. Voyons donc plus clairement comment cela marche !

7.3.1.1 Fonctionnement des moteurs

Comme précédemment avancé, ce moteur possède une certaine complexité pour être mis en œuvre. Et ce, plus que les précédents. Vous souvenez-vous du moteur CC (j'espère bien ! :P). Il était composé d'un ensemble d'aimants sur l'extérieur (le stator) et d'une partie bobinée où le champ magnétique était créée dynamiquement avec un ensemble collecteur/balais qui transmettait l'électricité aux bobines au centre (rotor).

Dans le cas du moteur pàp, c'est sur le rotor (au centre) que l'on retrouve l'aimant permanent, et les bobines sont sur le stator (autour du rotor). Comme pour les moteurs à courant continu, le but du jeu, en quelque sorte, est de "faire tourner un champ magnétique" (à prendre avec des pinces) pour faire tourner l'aimant fixé au rotor. Il existe cependant différents types de moteurs

pâp dont le placement des bobinages diffère les uns des autres et la façon de les alimenter n'est pas non plus identique (d'où une complexité supplémentaire lorsque l'on veut changer le type de moteur pâp à utiliser...). Nous allons maintenant les étudier l'un après l'autre en commençant par celui qui semble avoir le fonctionnement le plus simple à assimiler.

7.3.1.1.1 Moteur pâp bipolaire à aimants permanents Ce moteur possède quatre fils d'alimentation pour piloter des bobines par paire. Comme un schéma vaut mieux qu'un long discours, voici comment il est constitué :

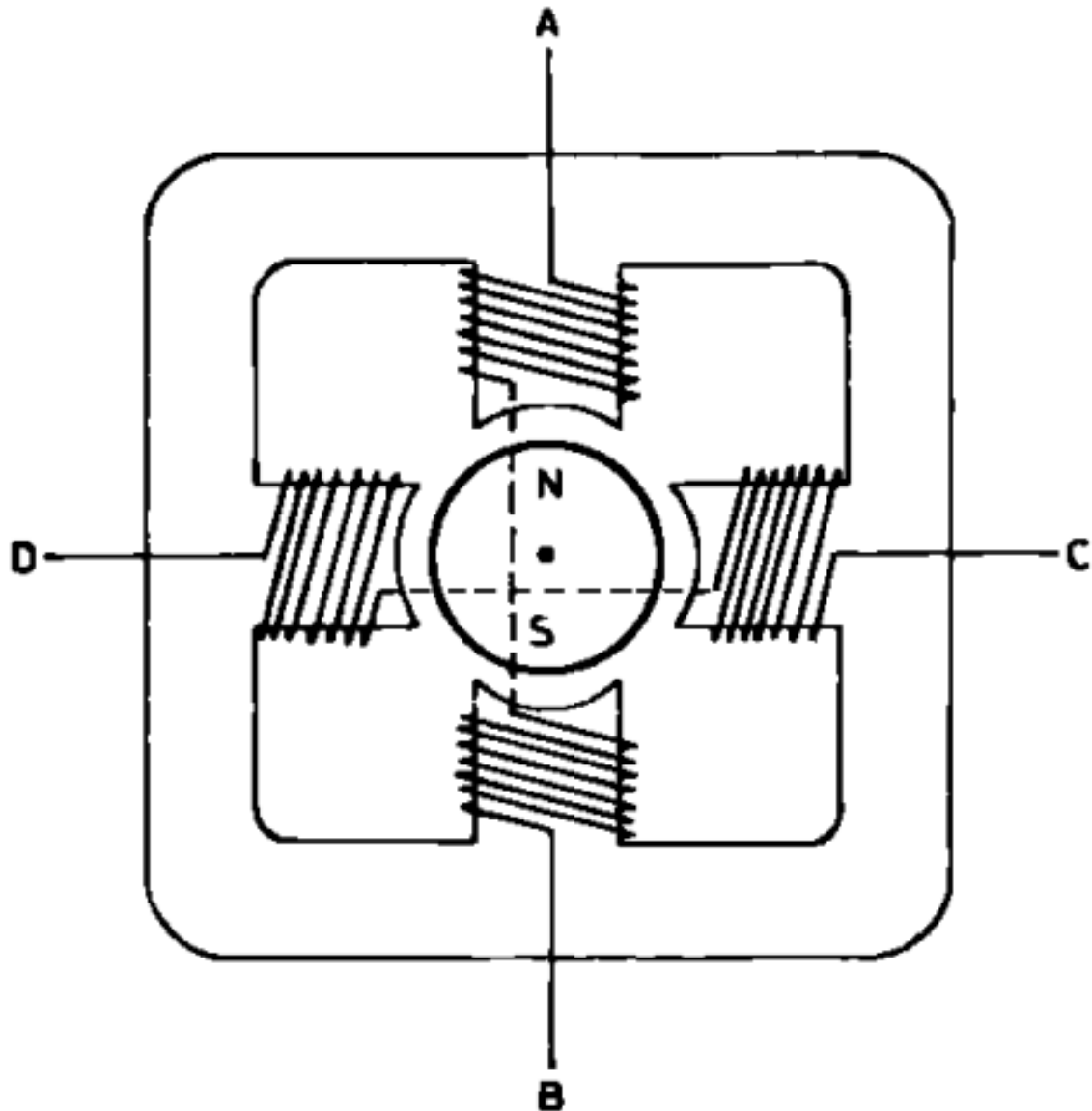


Figure 7.68 – Vue schématisée et simplifiée d'un moteur pas-à-pas bipolaire

Vous l'aurez compris, les bobines sont reliées deux à deux en série et sont donc pilotées ensemble. Il n'y a donc finalement que deux enroulements à commander puisque deux bobines montées en série n'en font plus qu'une. Leur placement de part et d'autre de l'aimant permanent du rotor permet de piloter ce dernier. Voyons comment.

- Lorsqu'il n'y a aucun courant traversant les bobines, le rotor (où l'axe de sortie est lié) est libre de tourner, rien ne cherche à le retenir dans sa course.
- Maintenant, si nous décidons de faire passer du courant entre les points C et D pour alimenter la bobine de gauche et celle de droite. Un courant va s'établir et deux champs électromagnétiques vont apparaître de part et d'autre du rotor. Que va-t-il alors se passer ? L'aimant du rotor va tourner sur lui-même pour se placer de façon à ce que son pôle Nord soit en face du pôle Sud du champ magnétique créé dans la première bobine et que son pôle Sud soit en face du pôle Nord créé dans la deuxième bobine.
- Si ensuite on alimente non plus les bobines entre C et D mais plutôt celles entre A et B, le rotor va alors tourner pour s'aligner à nouveau vers les pôles qui l'intéressent (Nord/Sud, Sud/Nord).
- Et c'est reparti, on va alors alimenter de nouveau les bobines entre D et C, donc avec un courant de signe opposé à la fois où l'on les a alimenter entre C et D (par exemple C était relié au "+" de l'alimentation tout à l'heure et là on le fait passer au "-", idem pour D que l'on fait passer du "-" au "+") et le moteur va encore faire un quart de tour.
- On peut continuer ainsi de suite pour faire tourner le moteur en faisant attention de ne pas se tromper dans les phases d'alimentation.

À chaque phase on va donc faire tourner le moteur d'un quart de tour :

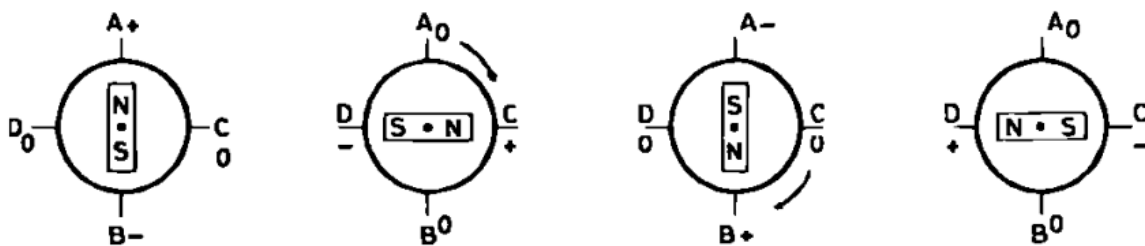


Figure 7.69 – Rotation pas-à-pas du moteur

[[i]] | Ce quart de rotation s'appelle un **pas**. Et comme il faut plusieurs pas pour faire tourner le moteur sur 360° , on l'a donc appelé ainsi, le moteur *pas-à-pas*.

Dans le cas illustré ci-dessus, on dit que le moteur fait 4 pas par tour. Il existe bien des moteurs qui font ce nombre de pas, mais il en existe qui ont un nombre de pas plus conséquent (24, 48, etc.). Leur constitution mécanique est différente, ce qui leur confère ce pouvoir, bien que le fonctionnement reste identique, puisque l'on cherche toujours à attirer un aimant grâce à des champs magnétiques créés par des bobines parcourues par un courant. Pour avoir plus de pas, on multiplie les aimants au centre. Sur l'image ci-dessous, on peut bien voir les bobines (en cuivre à l'extérieur) et tous les aimants au centre (les petites dents). Il existe aussi deux autres modes de fonctionnement que nous verrons dans la partie suivante : **le pilotage avec couple maximal** et **le pilotage par demi-pas**.

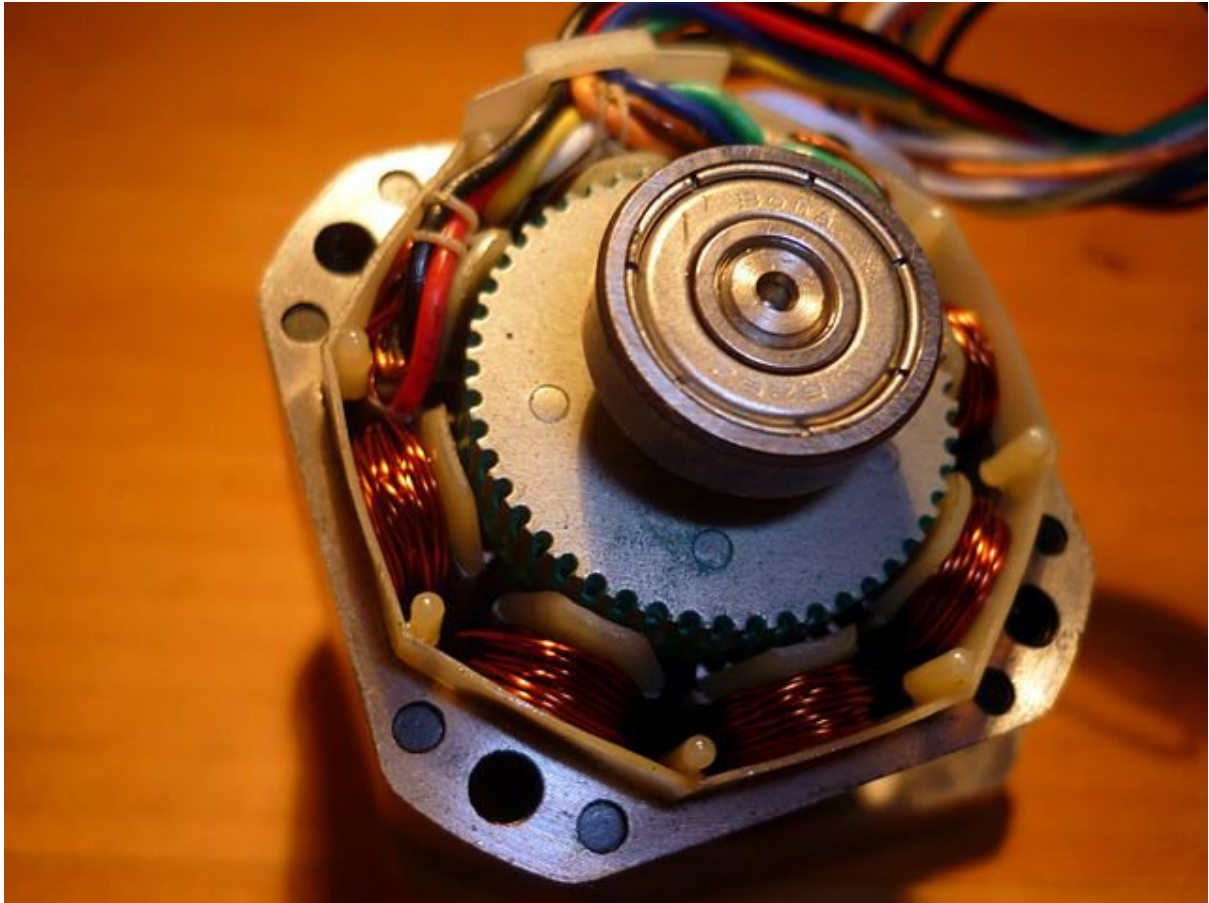


Figure : Vue en coupe d'un moteur pas-à-pas bipolaire - (CC-BY-SA, vincenzov.net)

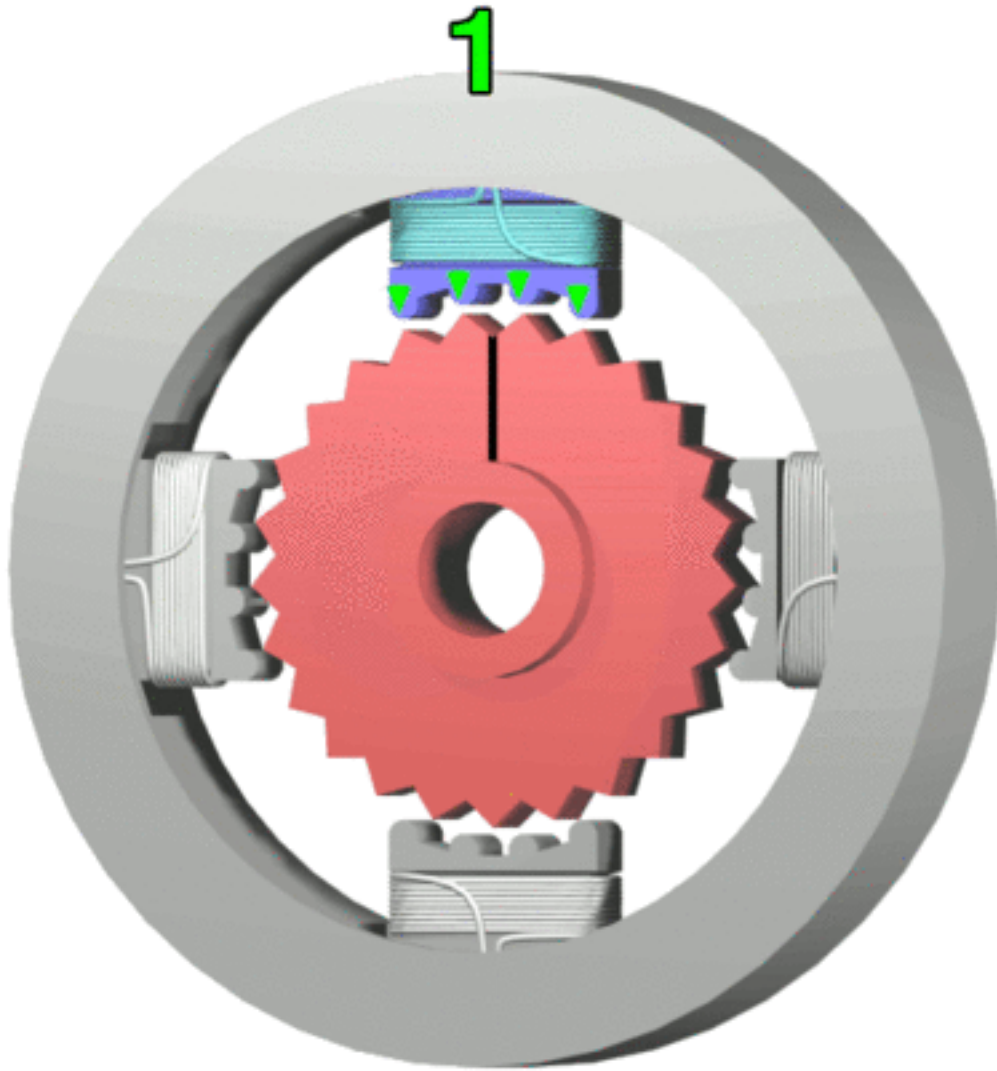
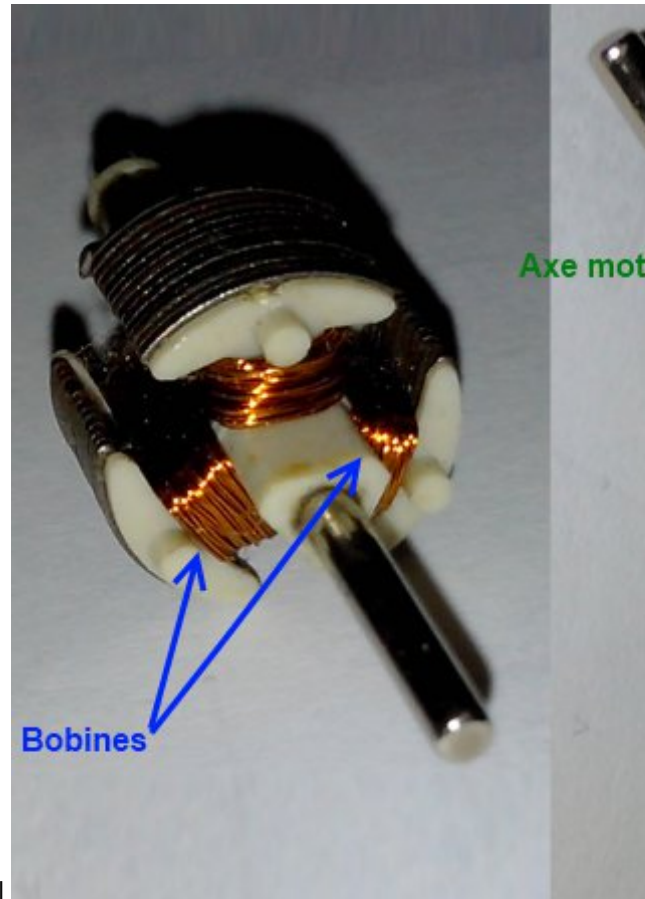


Figure : Animation de la rotation d'un moteur pas à pas - (CC-BY-SA, [Teravolt](#))



Pour rappel, voici la vue d'un moteur à courant continu : [[s]] |

7.3.1.1.2 Le moteur unipolaire Le moment est enfin venu de vous révéler la véritable signification des noms du moteur vu précédemment et de celui-ci même... non il ne faut pas lire ça sur un ton tragique. ^^ Le moteur bipolaire est nommé ainsi car il présente la faculté d'être commandé en inversant simplement la polarité de ces bobinages. Quant au moteur unipolaire, pas besoin de faire cette inversion, chaque bobinage est commandé séparément et ne requiert qu'une alimentation présente ou absente selon que l'on veuille ou non créer un champ magnétique en son sein. La commande sera donc plus simple à mettre en place qu'avec le moteur bipolaire. Cependant, le nombre de bobines étant plus important, la quantité de cuivre également et le prix s'en ressent ! En effet, il s'agit bien de 4 bobines bien distinctes, alors que le moteur bipolaire à aimant permanent en possède finalement quatre moitiés de bobines, donc deux bobines complètes.

On retrouve nos quatre fils A, B, C et D ainsi qu'un fil de masse commun (bon ben imaginez-le puisqu'il n'est pas dessiné comme tel sur le schéma). Soit 5 fils (v'là pas qu'on sait compter maintenant! ^^). Le fonctionnement est rigoureusement identique que le précédent moteur. On cherche à créer un champ magnétique "rotatif" pour faire passer l'aimant alternativement devant chacune des bobines. On va donc alimenter la bobine A, puis la C, puis la B, puis la D selon le schéma ci-dessus. Et voilà, le moteur aura fait tout un tour assez simplement. Il suffit d'avoir quatre transistors (un par enroulement) sans avoir besoin de les disposer en H et de les piloter deux à deux. Ici il suffit de les alimenter un par un, chacun leur tour. Facile, non ? ;)

7.3.1.1.3 Le moteur à réluctance variable [[q]] | Gné :o ? c'est quoi ce nom barbare ?

Ce moteur est un peu plus compliqué, mais c'est aussi le dernier que nous verrons et le plus fascinant ! Contrairement aux deux précédents, ce moteur ne possède pas d'aimants permanents,

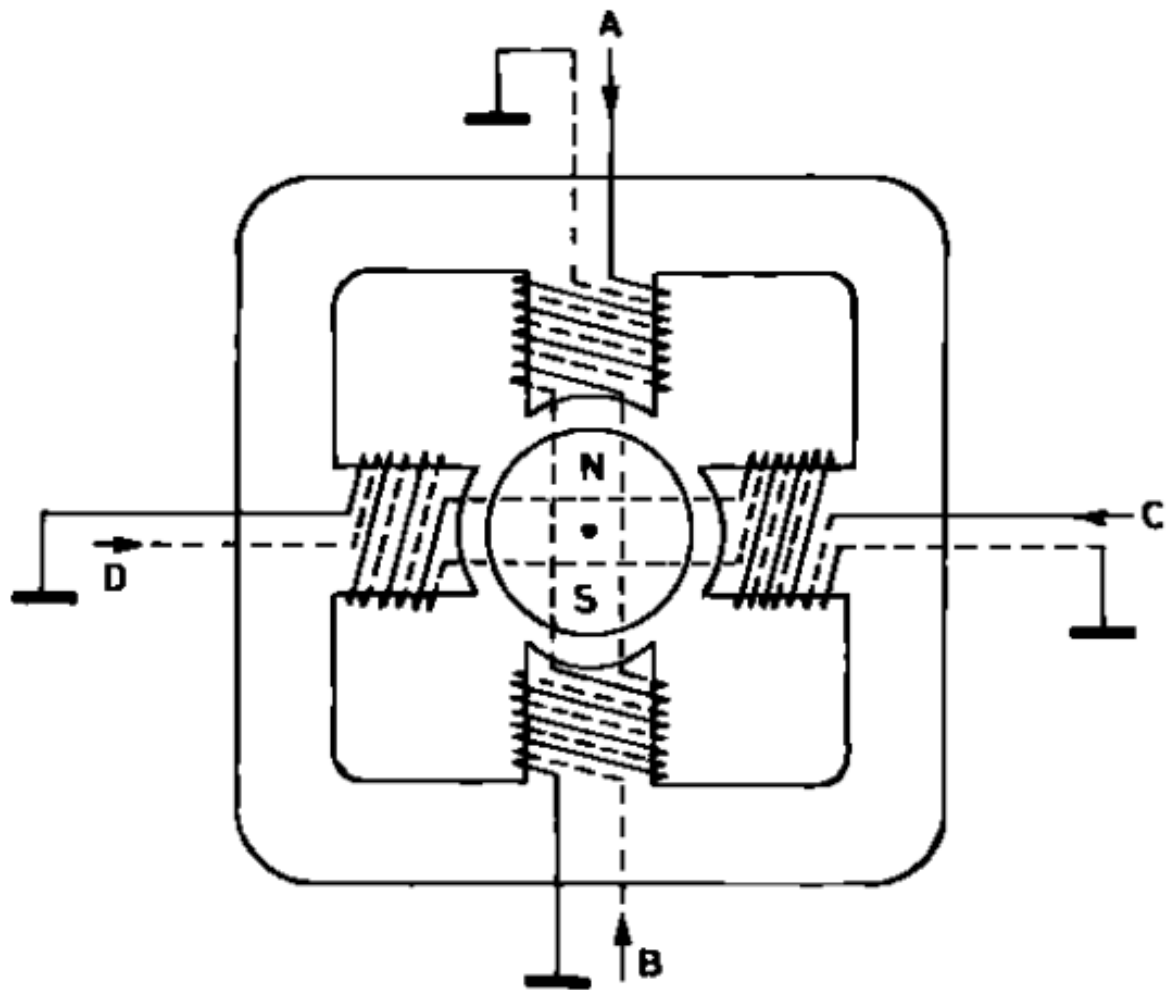


Figure 7.70 - Vue en coupe d'un moteur pas-à-pas unipolaire

ni même d'aimant tout court ! Non, en son centre on trouve un simple morceau de **fer doux**. Ce dernier a la particularité de très bien conduire les champs magnétiques. Du coup, si un champ magnétique le traverse, il voudra absolument s'aligner dans son sens. C'est cette propriété qui est exploitée. Commençons par voir tout de suite le schéma de ce moteur :

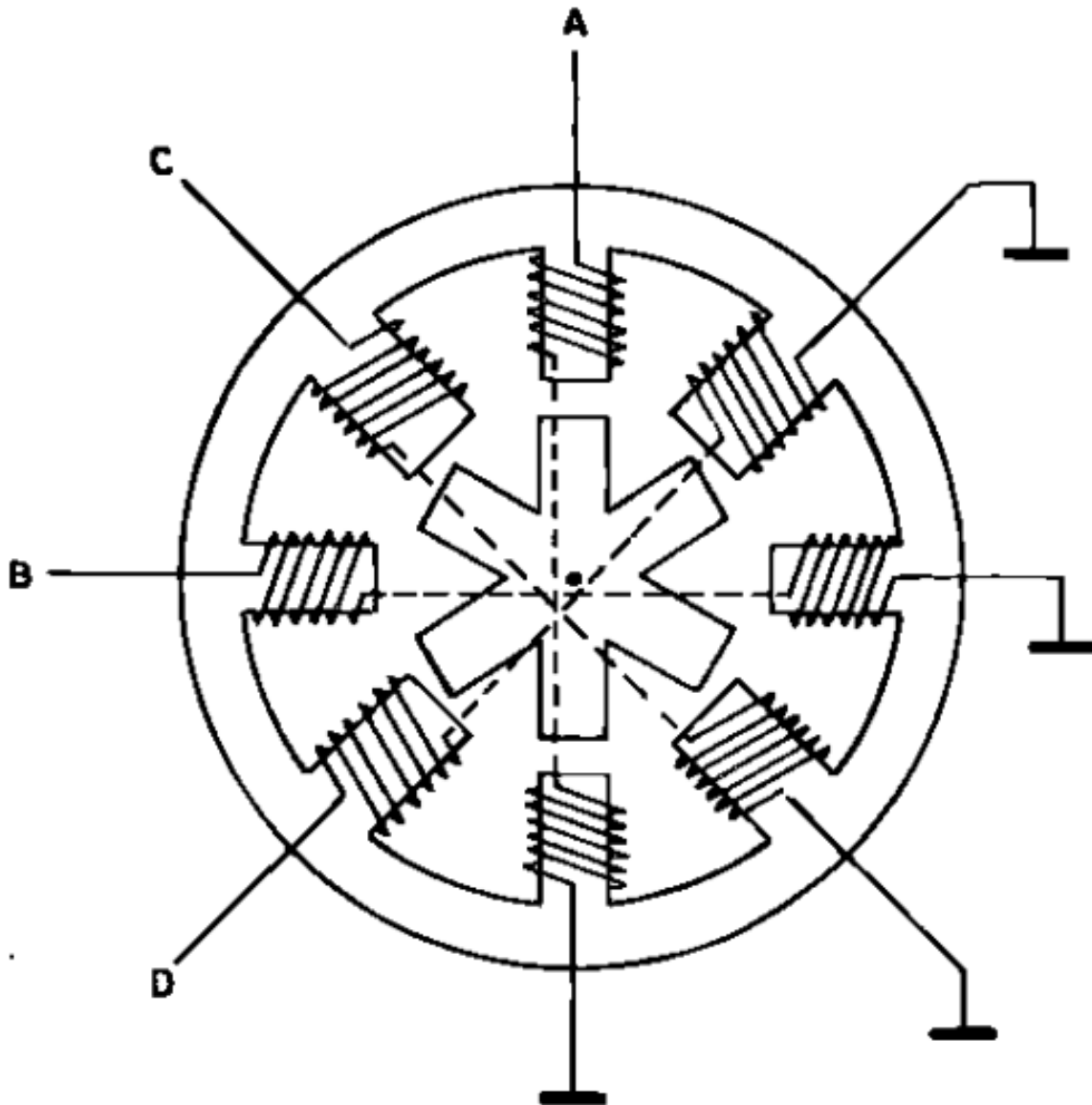


Figure 7.71 - Vue en coupe d'un moteur pas-à-pas à réluctance variable

Comme vous pouvez le voir, il possède 4 enroulements (formant 4 paires) et le morceau de fer doux au milieu à une forme d'étoile à 6 branches. Et ce n'est pas un hasard ! Le ratio de 6 pour 8 a une raison très précise car cela introduit un léger décalage (15°) entre une branche et une bobine. En effet, si l'on a 8 bobines (4 paires) on a un décalage entre chaque bobine de : $\frac{360^\circ}{8} = 45^\circ$ Donc tous les 45° le long du cylindre qu'est le moteur, on trouve un bobinage. En revanche il n'y a que 60° entre chaque extrémité de l'étoile du rotor : $\frac{360^\circ}{6} = 60^\circ$ Mais pourquoi exactement ? Eh bien, c'est simple avec un peu d'imagination (et quelques dessins). Si l'on commence par alimenter le premier enroulement, le A, le rotor va s'aligner avec. Maintenant que se passera-t-il si l'on alimente B ? Le rotor, qui était alors positionné avec une de ses branches bien en face de A, va

bouger pour s'aligner correctement vers B. Ensuite, si l'on alimente C il va se passer de même, le rotor va encore tourner de 15° pour s'aligner, etc. Si l'on effectue cette opération 24 fois, on fera un tour complet car : $24 \times 15^\circ = 360^\circ$

[[i]] | Vous remarquerez que dans cet exemple le rotor tourne dans le sens horaire alors que l'alimentation des bobines se fera dans le sens antihoraire.

Ces moteurs ont certains avantages. Parmi ces derniers, il n'y a pas besoin de polariser les bobines (peu importe le sens du champ magnétique, l'entrefer n'étant pas polarisé essaiera de s'aligner sans chipoter). Le fait que l'on utilise un simple entrefer en fer doux le rend aussi moins cher qu'un modèle avec des aimants permanents. Vous savez maintenant tout sur les trois types de moteurs pas-à-pas que l'on peut trouver, place maintenant à la pratique !

[[a]] | De manière générale, n'essayez pas d'ouvrir vos moteurs pas-à-pas pour regarder comment c'est fait et espérer les remonter après. Le simple démontage a tendance à faire diminuer la qualité des aimants permanents à l'intérieur et donc votre moteur ne sera plus aussi bon après remontage.

7.3.2 Se servir du moteur

Continuons à parler de notre super moteur. Si vous avez suivi ce que j'ai dit plus tôt, j'ai expliqué qu'il y avait des bobines qui génèrent un champ magnétique. Lorsqu'elles sont alimentées, ces bobines ont besoin de courant pour pouvoir générer un champ magnétique suffisant. Vous ne serez donc pas surpris si je vous dis qu'il faudra utiliser un composant pour faire passer la puissance dans ces dernières. Et là, comme les choses sont bien faites, nous allons retrouver le pont en H et le L298. :)

[[i]] | Afin de limiter la redondance d'informations, je vais me contenter de vous expliquer le pilotage du moteur unipolaire et bipolaire. Si vous comprenez correctement ces derniers, vous n'aurez aucun problème avec le moteur restant ;)

[[e]] | Les schémas qui vont suivre ont pour source d'énergie une pile +9V. Cependant il est déconseillé de les faire avec car la consommation des moteurs est assez importante et la pile risque de se fatiguer très vite. Utilisez plutôt une source d'alimentation prévue à cet effet (une alimentation de laboratoire).

7.3.2.1 Le moteur unipolaire

7.3.2.1.1 Connecter un moteur unipolaire Pour rappel, voici la structure du moteur unipolaire :

Si vous avez bien lu la partie précédente, vous devez avoir retenu que ce moteur est assez simple à piloter. En effet, il suffit d'alimenter une à une les bobines pour que le moteur tourne. Et c'est tout ! Il nous faut juste utiliser le bon composant pour alimenter les bobines et en avant ! A priori, le bon composant serait un bon transistor qui pourrait supporter 50V et 500mA pour débiter. À cela il faudrait ensuite rajouter une diode de roue libre pour ne pas l'abîmer lors des phases de roue libre (tout cela multiplié par 4 puisqu'on a 4 bobines). Plutôt que de s'embêter à câbler tout ça, je vous propose l'intervention d'un nouveau composant : le ULN2003A. Ce dernier regroupe les transistors pour faire passer la puissance et aussi la diode. Il est très simple à câbler puisqu'il faut simplement amener l'alimentation et les broches de commandes. Chacune de ces dernières possède respectivement une sortie où la tension sera celle de l'alimentation. Voici une illustration de ce câblage :

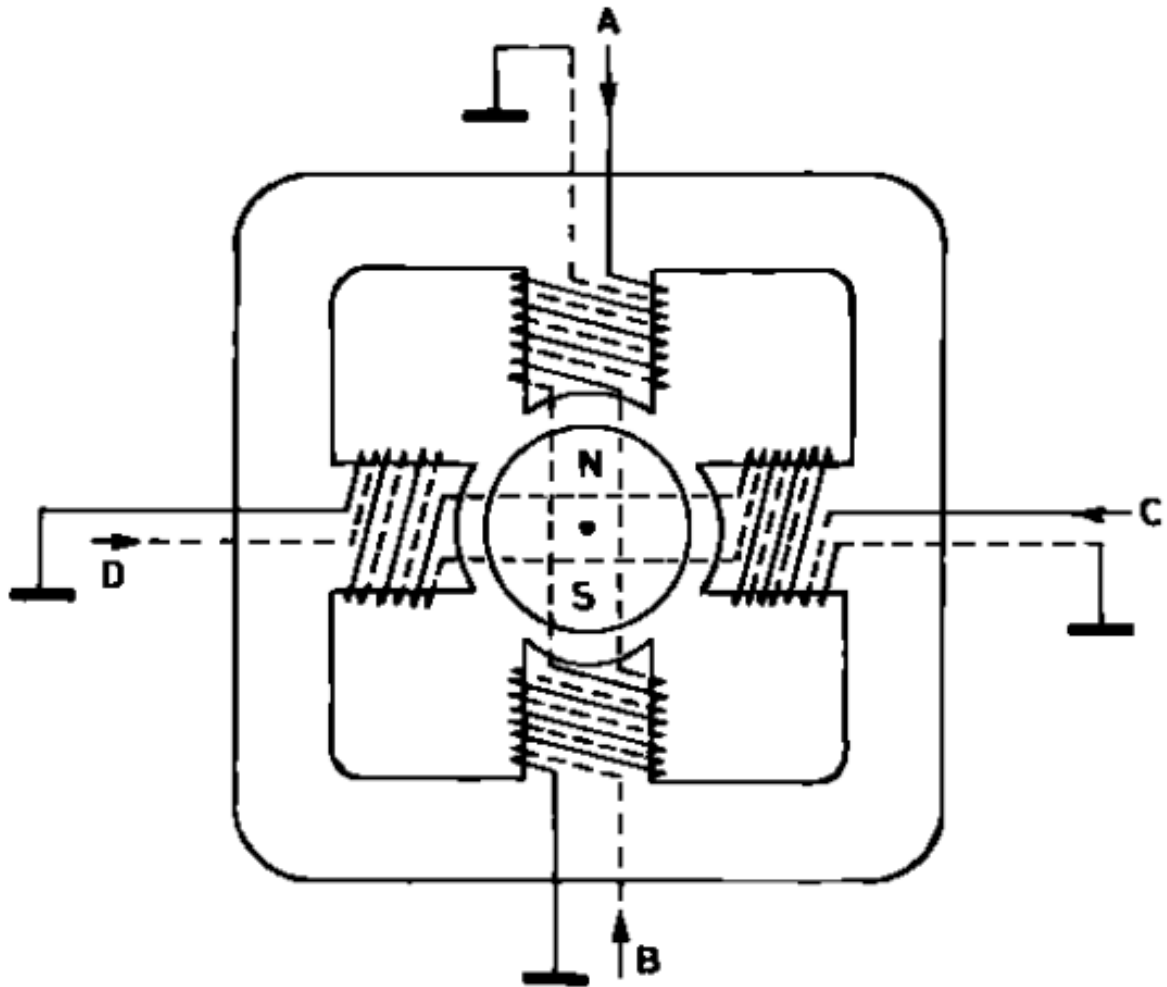


Figure 7.72 - Vue en coupe d'un moteur pas-à-pas unipolaire

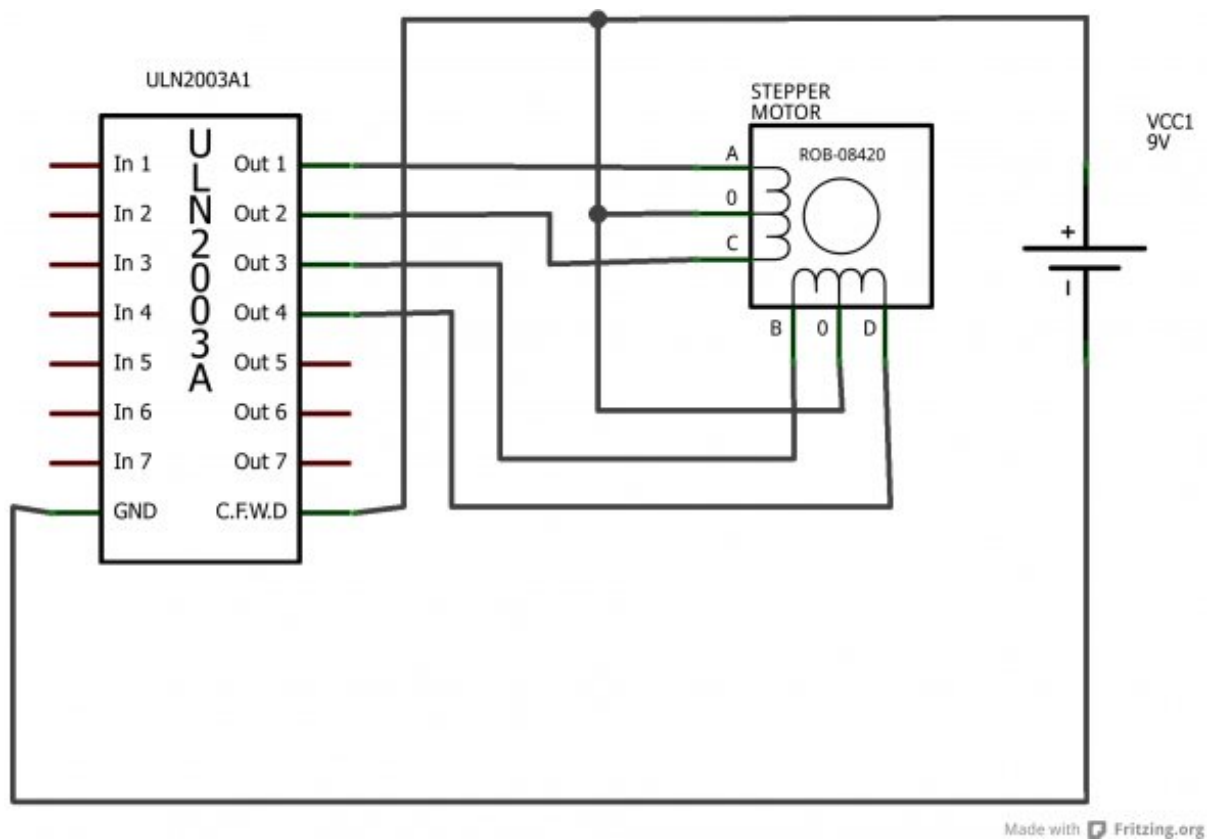


Figure 7.73 – Câblage simple du moteur unipolaire – Schéma

7.3.2.1.2 Utiliser un moteur unipolaire Je l’ai maintenant dit et répété plusieurs fois, pour ce moteur il suffit de piloter chaque bobine une à une, chacune leur tour. Je vais donc vous résumer tout cela de manière plus schématique et on sera bon pour ce moteur. :D A la fin, si vous avez bien compris vous devriez être capable de le faire tourner tout doucement en plaçant alternativement les fils In1 à 4 (abrégé In1..4) au 5V ou à la masse.

[[i]] | Les différentes illustrations de séquences peuvent être trouvées sous licence CC-BY-SA sur [wikipedia](https://fr.wikipedia.org/wiki/Wikipédia:Modèle:M4RCO) (M4RCO)

->

Etape	In 1	In 2	In 3	In 4
Pas n°1	HIGH	LOW	LOW	LOW
Pas n°2	LOW	HIGH	LOW	LOW
Pas n°3	LOW	LOW	HIGH	LOW
Pas n°4	LOW	LOW	LOW	HIGH

Table 7.5 – Séquence du moteur unipolaire

<-

Si vous placez les fils de commande à la masse ou au 5V convenablement, votre moteur devrait tourner :) Vous n’avez plus qu’à créer le programme qui va bien pour piloter tout ce bazar... ben vous vous attendiez à quoi ? Une solution peut-être ? Non. :diable : Bon bon, d’accord... vous vous souvenez de vos premiers pas avec le chenillard ? Tout est dit. ;) (et si vous ne voulez pas vous fatiguer, attendez la suite du tuto :D)

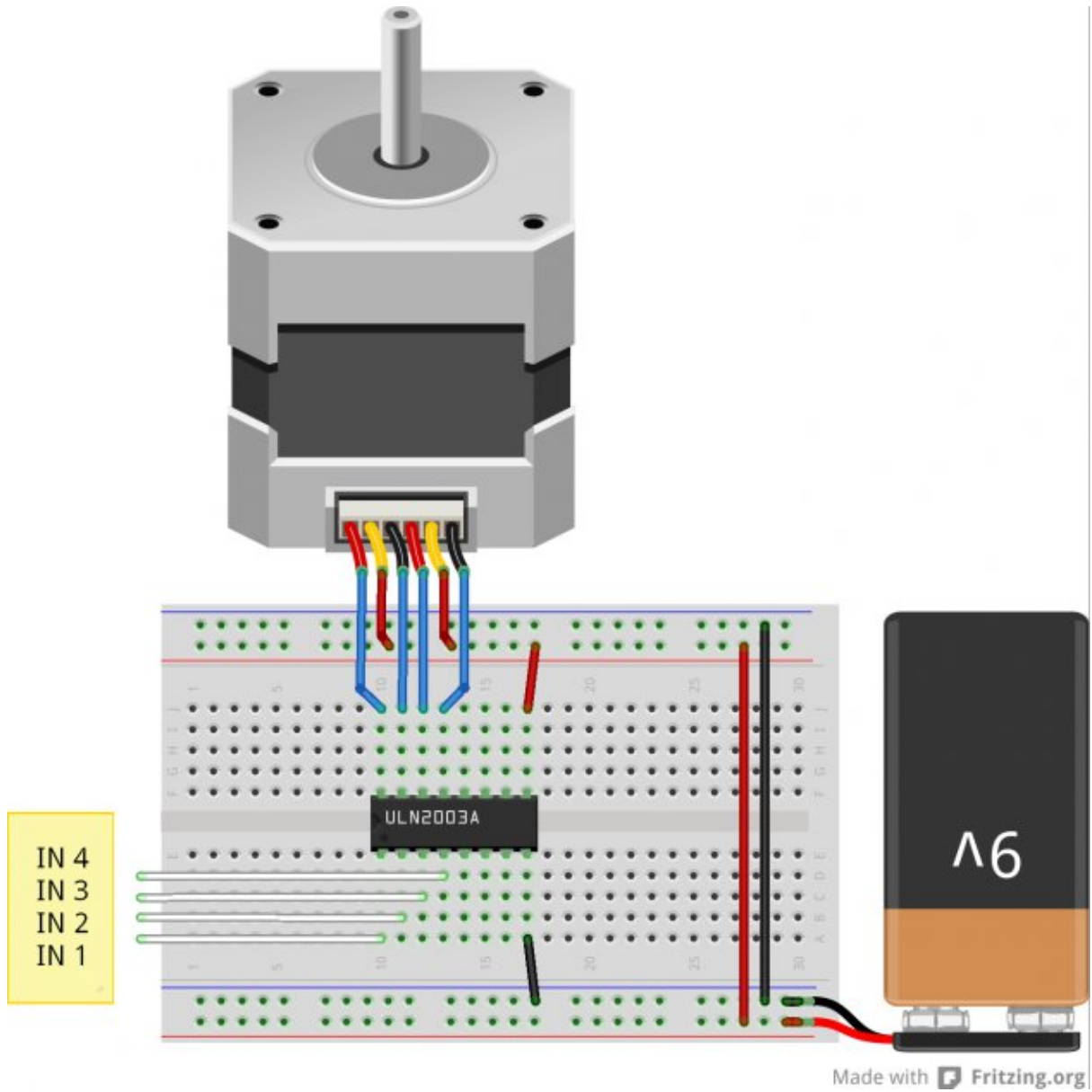


Figure 7.74 – Câblage simple du moteur unipolaire - Montage

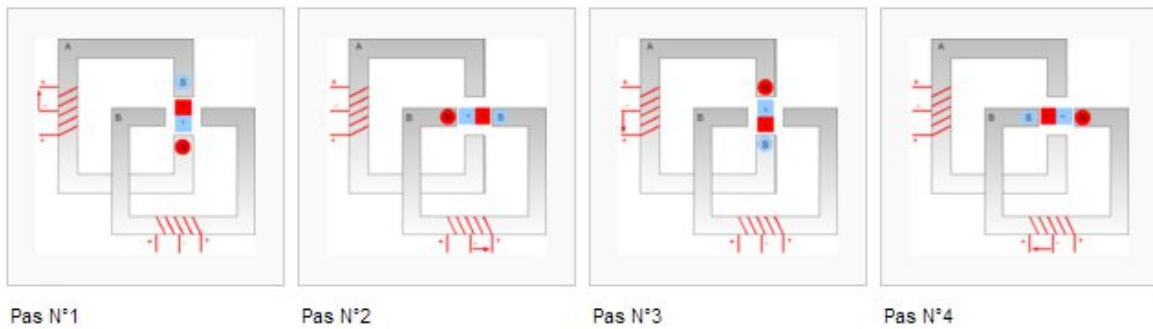


Figure 7.75 – Séquence du moteur unipolaire

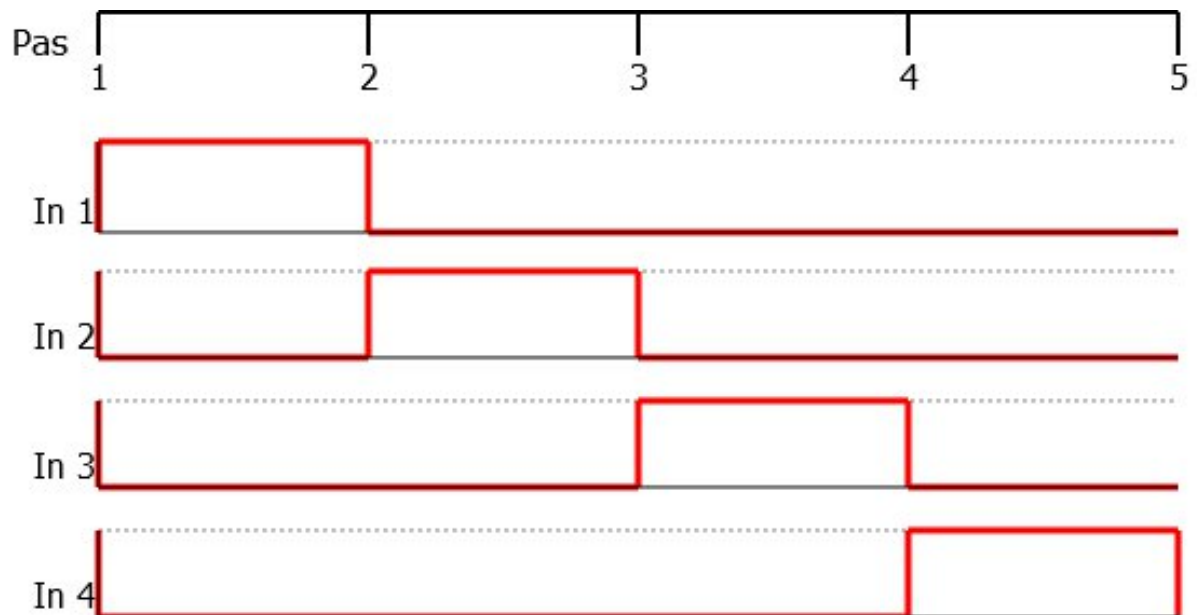


Figure 7.76 - Chronogramme du moteur unipolaire

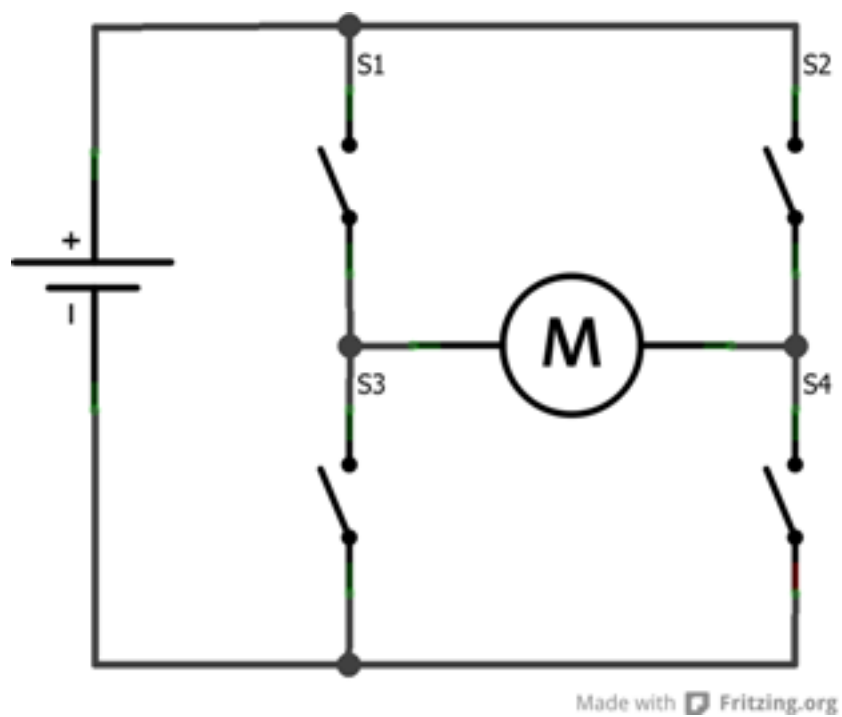


Figure 7.77 - Le pont en H - vue simplifiée

nous voulons activer les deux ponts, mettez les deux entrées “enable” au +5V. Nous verrons dans la prochaine partie comment l’utiliser avec Arduino ;). Voici un petit schéma récapitulatif :

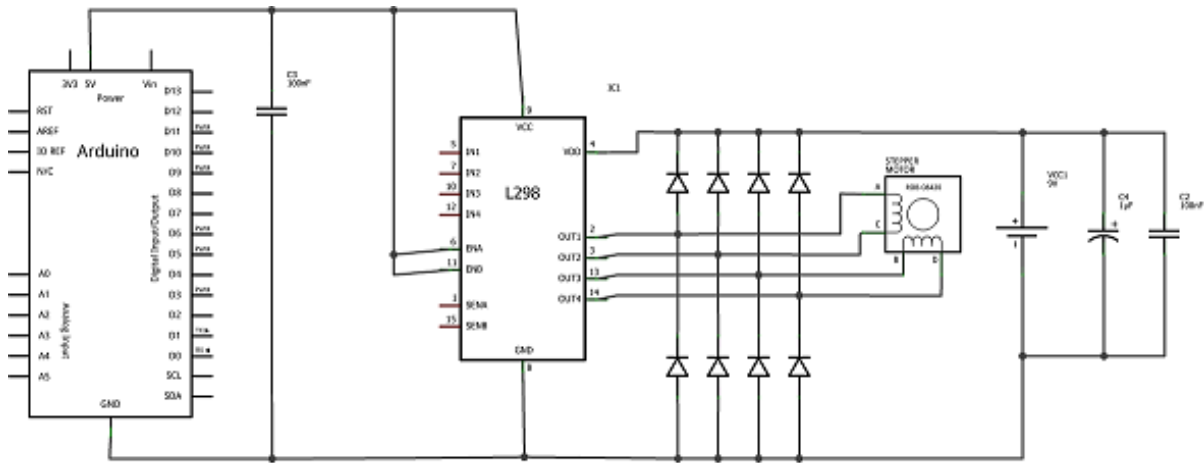


Figure 7.78 – Câblage simple – Schéma

Le schéma de montage avec quelques condensateurs de filtrage qui viennent aider l’alimentation en cas de forte demande de courant et des condensateurs qui filtre les parasites engendrés par les bobines du moteur.

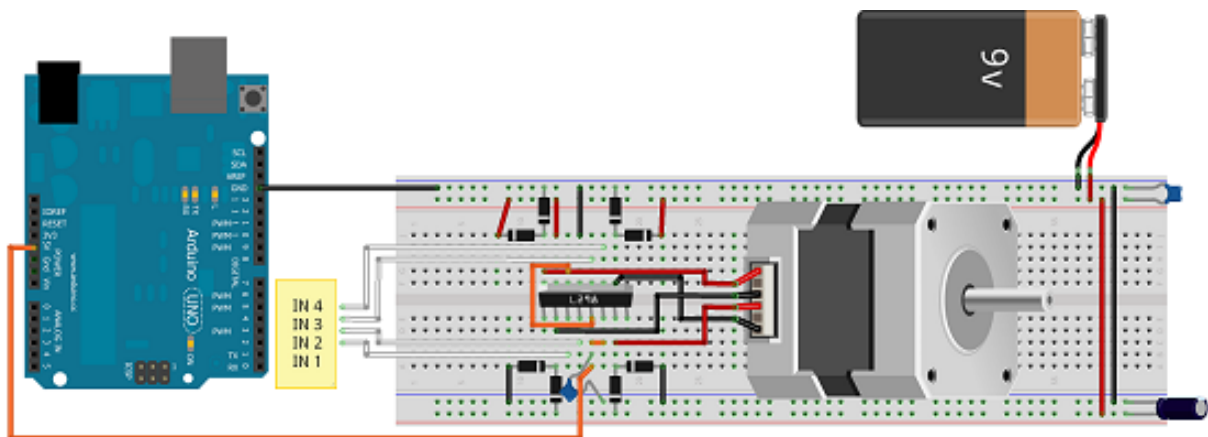


Figure 7.79 – Câblage simple – Montage

7.3.2.3 Piloter le moteur bipolaire

Une fois que le moteur est branché, nous allons pouvoir le faire tourner. Comme son nom l’indique, il s’appelle pas à pas et donc nous allons le faire pivoter étape par étape et non de manière continue comme le moteur à courant continu. Cependant, en répétant les étapes de rotation rapidement et successivement, le moteur donnera l’impression de tourner sans s’arrêter entre chaque étape. Il existe trois méthodes distinctes pour piloter les moteurs bipolaires. Nous allons maintenant les voir une par une. Dans les cas qui vont suivre, on va considérer un moteur de 4 pas par tour (ce qui est ridiculement faible). Voici ce à quoi il va ressembler (comment sont placées ses bobines) :

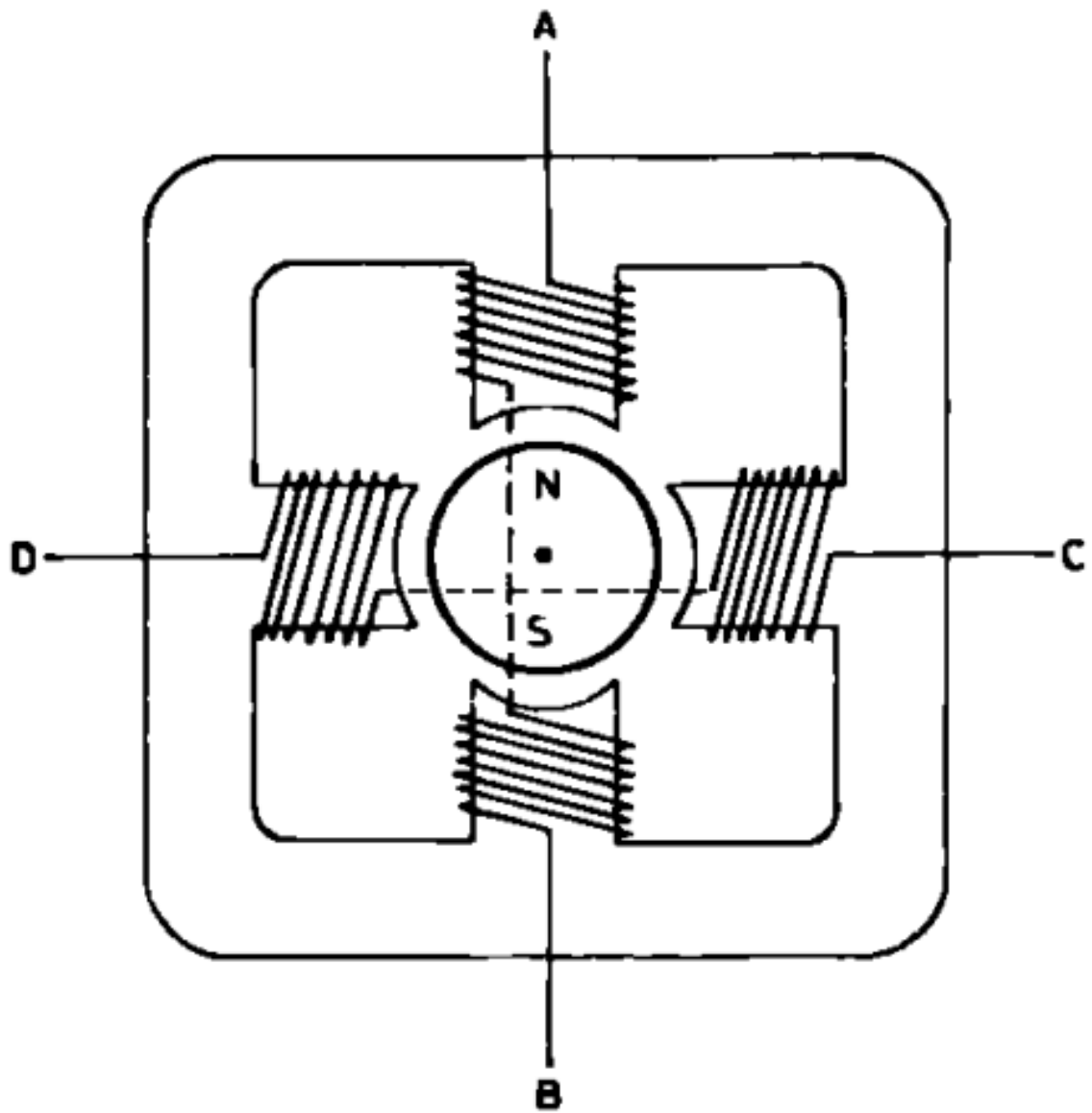


Figure 7.80 – Vue schématisé d'un moteur pas-à-pas

7.3.2.3.1 Rotation par pas complet Ce mode de fonctionnement est le plus simple, c'est pourquoi nous allons commencer par lui. Grâce à ce dernier, vous allez pouvoir faire des rotations "pas par pas". Pour cela, nous allons alternativement alimenter les bobines de droites et de gauche et inverser le sens du courant pour pouvoir faire une rotation complète du champ magnétique. Voici l'illustration Wikipédia très bien faite à ce sujet :

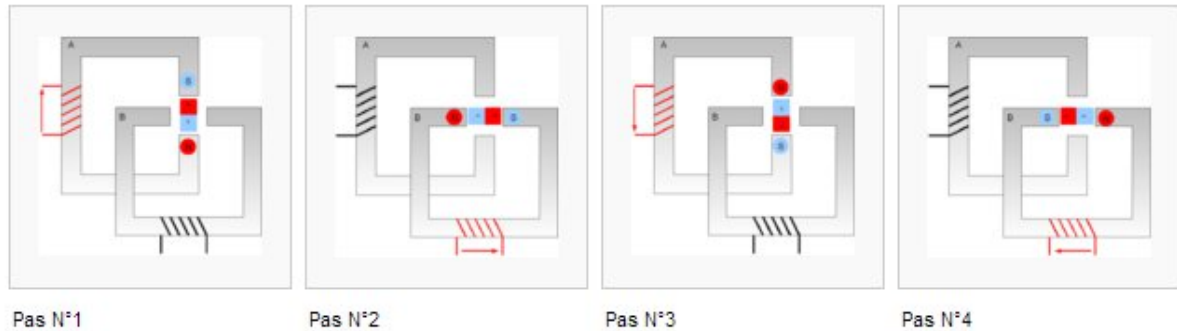


Figure 7.81 – Séquence à pas complet

[[i]] | En rouge la bobine alimentée ainsi que le sens du courant symbolisé par une flèche et en noir la bobine qui n'est pas alimentée.

En considérant que la bobine A est connectée aux entrées IN1 et IN2 et que la bobine B est connectée aux commandes IN3 et IN4, on peut donc écrire la séquence de commande suivante :

->

Etape	In 1	In 2	In 3	In 4
Pas n°1	HIGH	LOW	-	-
Pas n°2	-	-	HIGH	LOW
Pas n°3	LOW	HIGH	-	-
Pas n°4	-	-	LOW	HIGH

Table 7.6 – Séquence à pas complet

<-

(un état '-' signifie "non nécessaire", placez-le à 0V pour que la bobine soit bien inactive).

On peut traduire cette activité par le chronogramme suivant :

Comme vous pouvez le voir à travers ces différents moyens d'explication, c'est somme toute assez simple. On va chercher à déplacer l'aimant central en le faisant tourner petit à petit. Pour cela on cherchera à l'attirer dans différentes positions.

7.3.2.3.2 Rotation à couple maximal Un autre mode de fonctionnement est celui dit à **couple maximal**. Cette méthode de pilotage utilise toutes les bobines à la fois pour pouvoir immobiliser au maximum l'aimant central. En effet, en utilisant plus de champs magnétiques on obtient une force supplémentaire. Par contre, on consomme évidemment d'avantage de courant. Pour comprendre ce fonctionnement, voyons les différentes étapes par un dessin puis par un

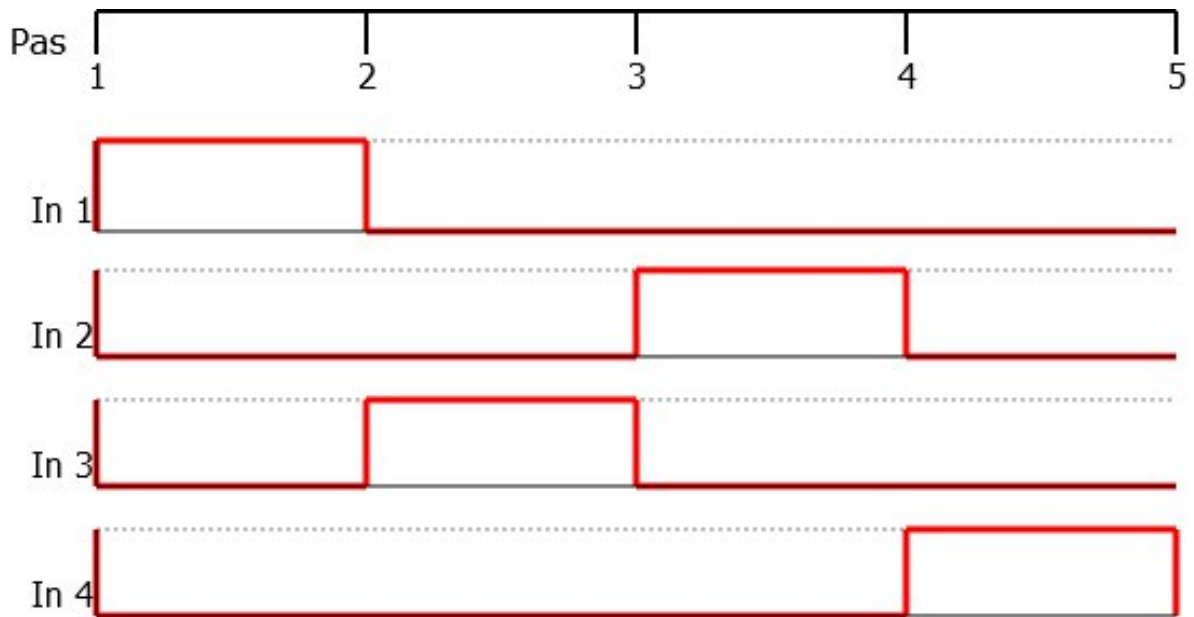


Figure 7.82 – Chronogramme du pilotage à pas entier

chronogramme. Vous verrez, ce n'est pas très compliqué, le fonctionnement est très similaire, seules les activations de bobines changent un peu :

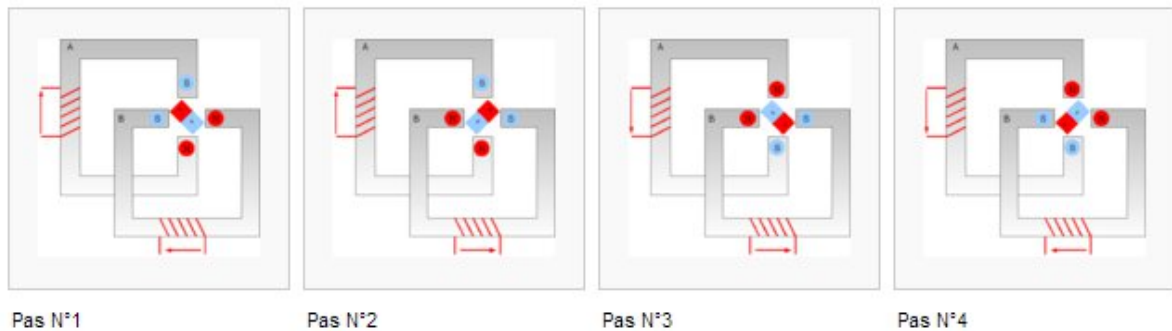


Figure 7.83 – Séquence à couple maximal

[[i]] | Avez-vous remarqué quelque chose de particulier ? Dans cette utilisation, l'aimant ne fait plus face aux bobines mais se place *entre* les deux. Par contre, il effectue toujours des pas entiers, ces derniers ont juste un décalage constant par rapport à avant.

Voici le tableau correspondant au pilotage des bobines :

->

Etape	In 1	In 2	In 3	In 4
Pas n°1	HIGH	LOW	HIGH	LOW
Pas n°2	HIGH	LOW	LOW	HIGH
Pas n°3	LOW	HIGH	LOW	HIGH
Pas n°4	LOW	HIGH	HIGH	LOW

Etape	In 1	In 2	In 3	In 4
-------	------	------	------	------

Table 7.7 – Séquence à couple maximal

<-

(un état ‘-’ signifie “non nécessaire”, placez-le à 0V pour que la bobine soit bien inactive).

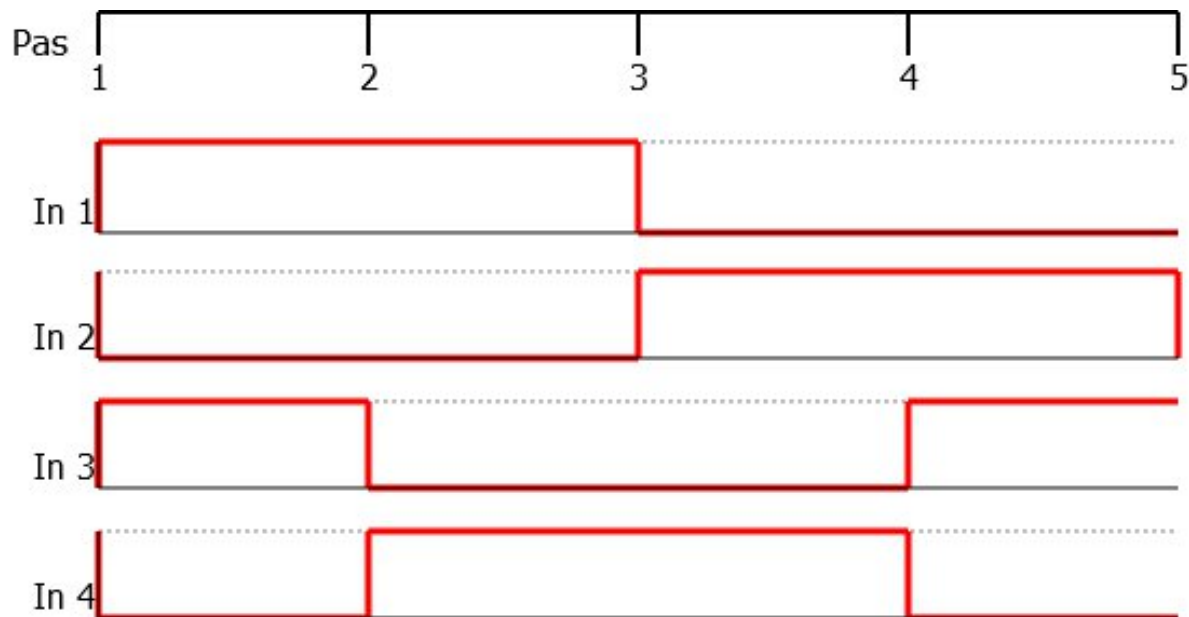


Figure 7.84 – Chronogramme pour un couple max.

7.3.2.3.3 Rotation par demi-pas Enfin, le dernier mode de fonctionnement est celui dit à **demi-pas**. Ce mode mélange les deux précédents puisqu'on va alterner les étapes du mode à pas complet et les étapes du mode à couple maximal. En effet, comme nous avons pu le voir dans les explications précédentes, les deux modes placent l'aimant central de manière différente. L'un est “en face des bobines” alors qu'avec l'autre est plutôt “entre les bobines”.

Ainsi, en se mettant alternativement “en face” puis “entre” les bobines on va effectuer deux fois plus de pas que précédemment puisqu'on intercalera des étapes supplémentaires. Attention, lorsque je dis “deux fois plus de pas” je veux surtout dire que l'on aura des étapes intermédiaires qui augmentent la précision du déplacement. Ce mode de pilotage est un peu plus compliqué que les précédents puisqu'il est “plus long” (8 étapes au lieu de 4) mais rien d'insurmontable vous allez voir !

->

Etape	In 1	In 2	In 3	In 4
Pas n°1	HIGH	LOW	-	-
Pas n°1 1/2	HIGH	LOW	HIGH	LOW
Pas n°2	-	-	HIGH	LOW
Pas n°2 1/2	LOW	HIGH	HIGH	LOW

Etape	In 1	In 2	In 3	In 4
Pas n°3	LOW	HIGH	-	-
Pas n°3 1/2	LOW	HIGH	LOW	HIGH
Pas n°4	-	-	LOW	HIGH
Pas n°4 1/2	HIGH	LOW	LOW	HIGH

Table 7.8 – Séquence à demi-pas

<-

(un état ‘-’ signifie “non nécessaire”, placez-le à 0V pour que la bobine soit bien inactive).

Maintenant que vous connaissez les différents modes de fonctionnement, vous pouvez essayer de faire tourner le moteur en branchant les entrées IN1..4 à la masse ou au 5V. Si vous le faites dans le bon ordre, votre moteur devrait tourner tout doucement, en allant d’une étape vers l’autre. :)

[[a]] | Si vous avez une charge qui demande trop de couple (par exemple un poids à faire monter), il peut arriver que le moteur “saute” un/des pas. Cette donnée est à prendre en compte si vous vous servez du nombre de pas effectué logiquement comme moyen de calcul de distance.

7.3.3 Utilisation avec Arduino

7.3.3.1 Câbler les moteurs

Avec les chapitres précédents, vous avez vu comment on devait utiliser les moteurs avec les composants gérant la puissance. Cependant, nous n’avons pas vu à quoi relier les broches de commande... Et c’est très simple! En effet, tous les signaux sont tout ou rien et n’ont même pas besoin d’être des PWM! Ce qui veut dire que les 4 broches de pilotage ont juste besoin d’être reliées à 4 broches numériques de la carte Arduino (2, 3, 4, 5 par exemple). Voyons ce que cela donne en schéma (qui sont exactement les mêmes que ceux de la partie précédente, mais avec une carte Arduino en plus :D)

7.3.3.1.1 Le moteur unipolaire

7.3.3.1.2 Le moteur bipolaire Jusqu’à là rien de vraiment compliqué, on passe à la suite!

7.3.3.2 Piloter les moteurs avec Arduino

7.3.3.2.1 Le principe L’idée est toute simple, il suffit de générer la bonne séquence pour piloter les moteurs à la bonne vitesse, vous vous en doutez sûrement. La principale difficulté réside dans la génération des signaux dans le bon ordre afin que le moteur se déplace correctement. Bien entendu, c’est plus facile à dire qu’à faire. En effet, pour que le mouvement soit fluide, il faut que les changements dans la séquence soient faits de manière régulière et pour cela il faut une gestion du temps correcte. Ça peut sembler simple au premier abord, mais quand il s’agira de mixer le comportement du moteur avec celui du programme principal (qui devra peut-être faire des traitements assez lourds) cela deviendra probablement beaucoup moins trivial. Une bonne

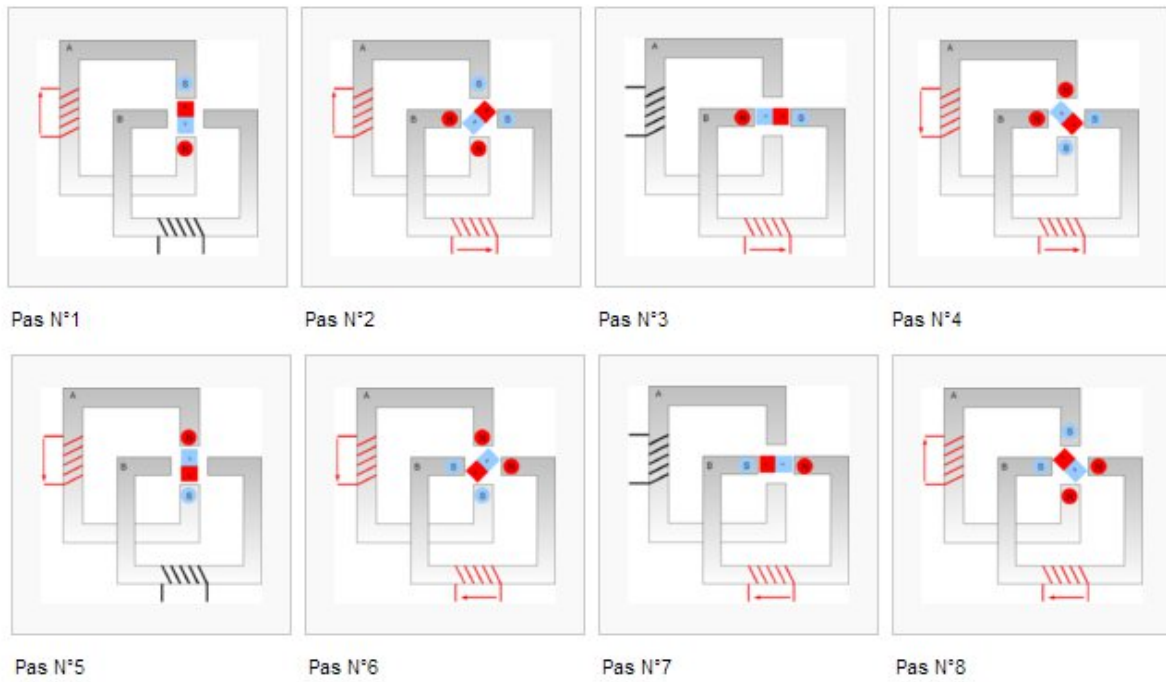


Figure 7.85 – Séquence à demi-pas

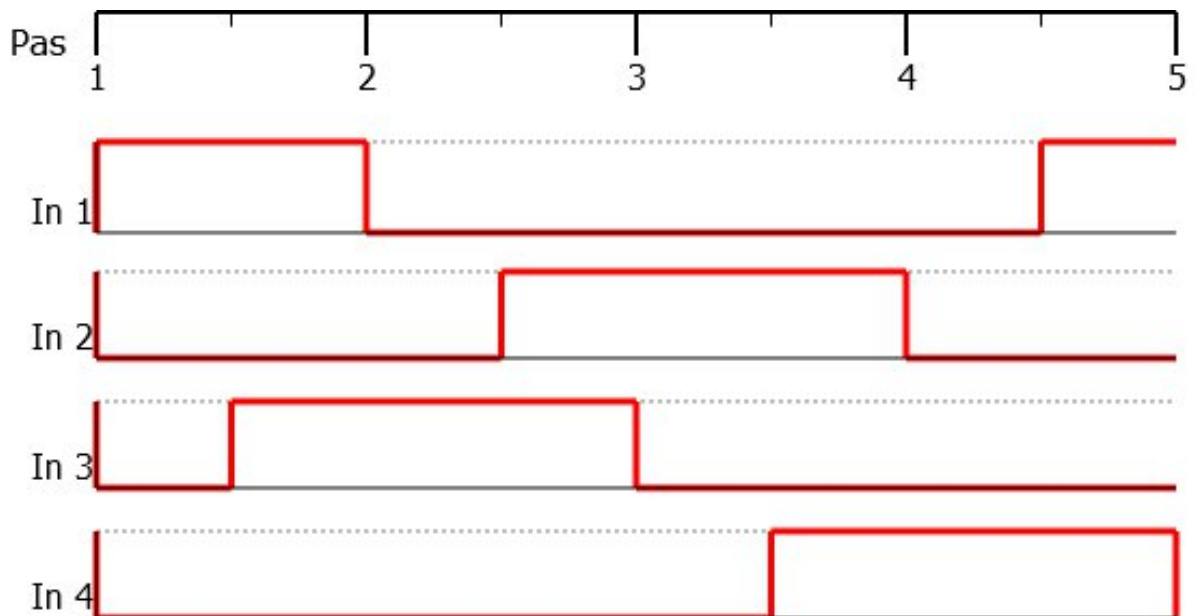


Figure 7.86 – Chronogramme du pilotage à demi-pas

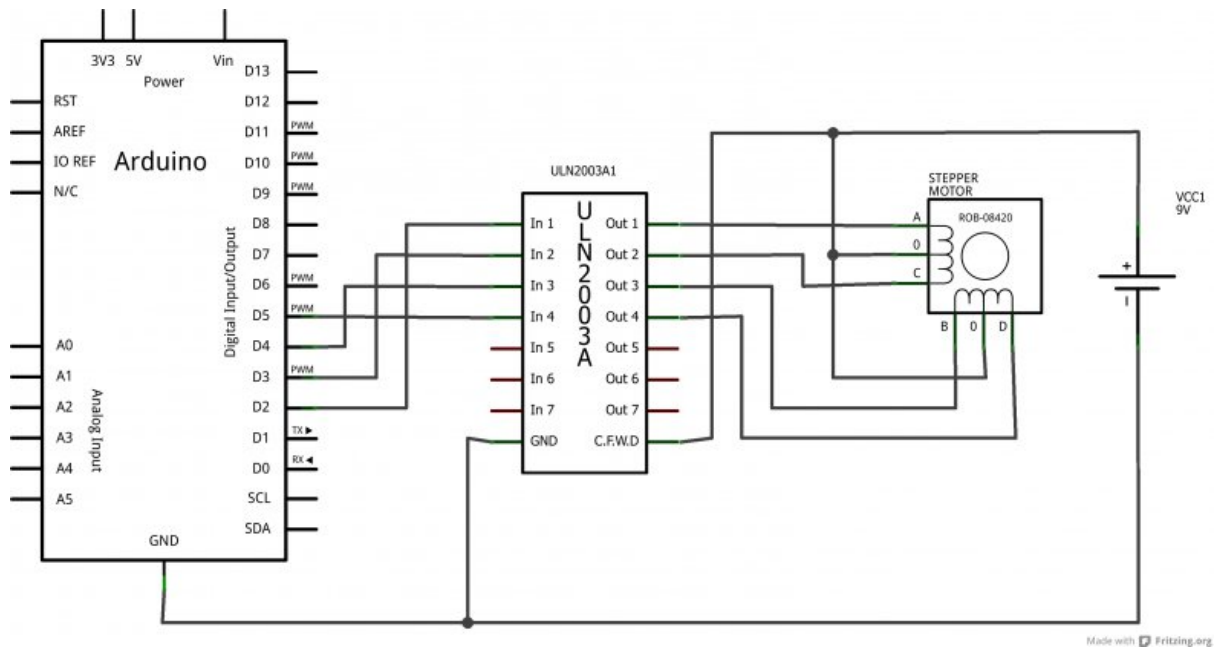


Figure 7.87 – Câblage du moteur unipolaire - Schéma

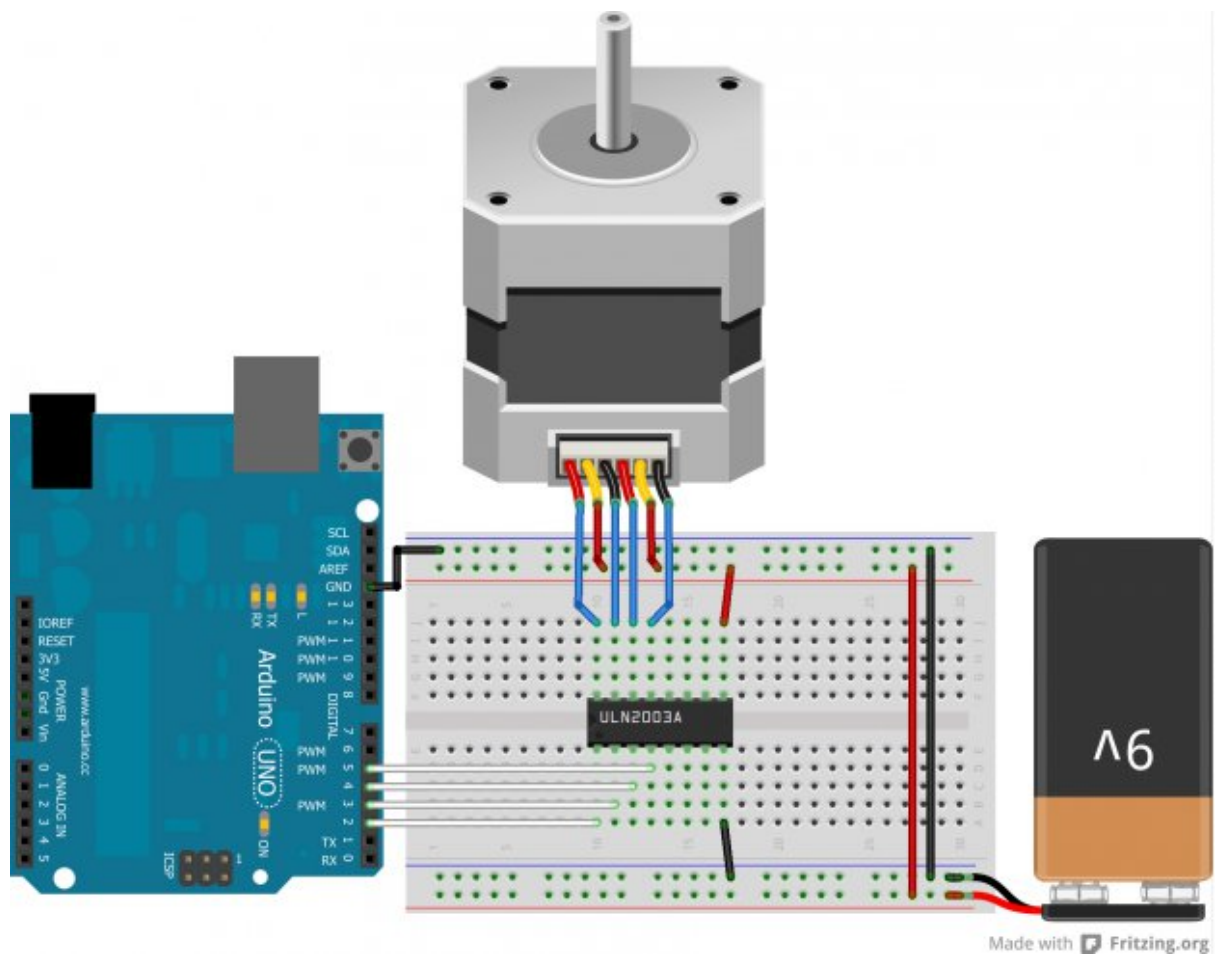


Figure 7.88 – Câblage du moteur unipolaire - Montage

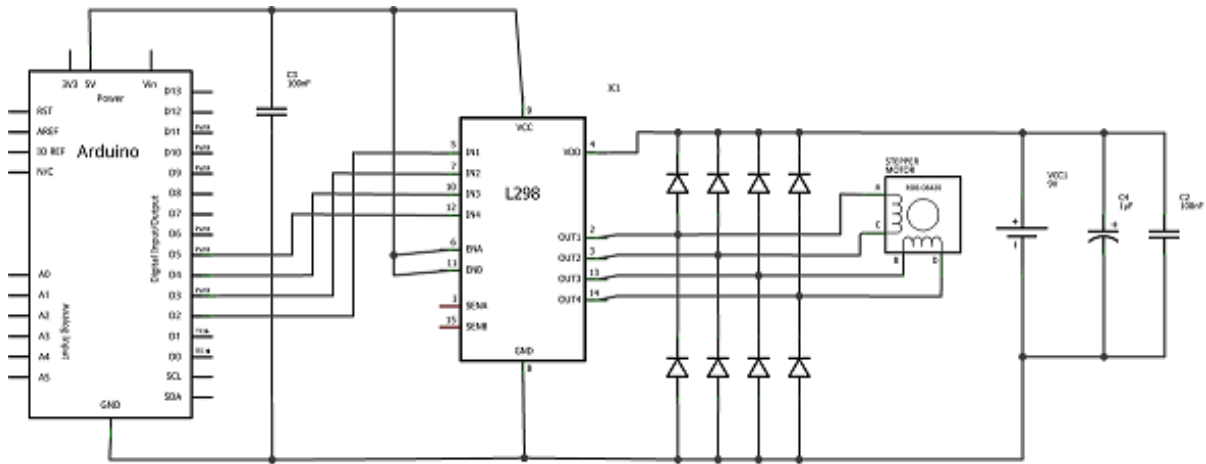


Figure 7.89 – Câblage du moteur bipolaire - Schéma

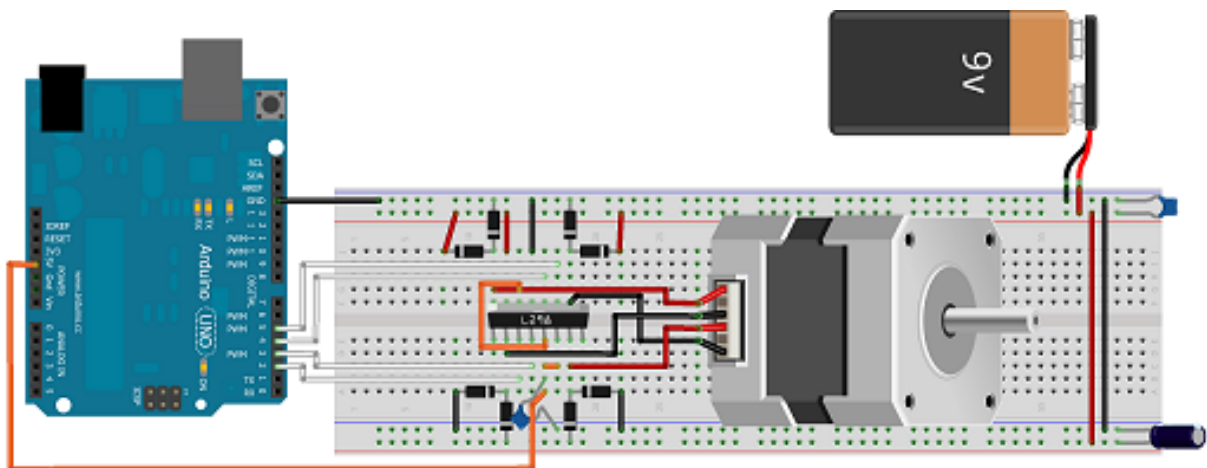


Figure 7.90 – Câblage du moteur bipolaire - Montage

méthode consisterait à utiliser un **timer** sur lequel on réglerait la période à avoir qui refléterait ainsi la vitesse à obtenir. Mais avec Arduino vous allez voir que tout devient plus simple...

7.3.3.2 L'objet Stepper Sur Arduino les choses sont bien faites pour rester simples et accessibles, un objet a déjà été créé pour vous aider à piloter un moteur pas à pas. Attention cependant, il ne fonctionne que pour les moteurs unipolaires et bipolaires. Il tire parti du fait que ces deux types de moteur peuvent fonctionner avec une séquence commune. Ainsi, tout est généralisé et utilisable le plus simplement possible ! Ce nouveau composant s'appelle "**Stepper**". À sa création, il prend en argument le nombre de pas total que fait le moteur pour faire un tour (information trouvable dans la documentation constructeur ou empiriquement). Cette information sert à déterminer la vitesse de rotation par minute que vous pourrez ensuite régler à loisir pour faire des déplacements lents ou rapides. Il prend aussi en arguments les quatre broches servant à contrôler l'engin. Son constructeur est donc : `Stepper(steps, pin1, pin2, pin3, pin4)`. Pour initialiser le moteur, nous pouvons donc écrire la ligne suivante :

```
// pour un moteur de 200 pas par tour et brancher sur les broches 2, 3, 4, 5
Stepper moteur(200, 2, 3, 4, 5);
```

Code : Initialisation d'un moteur pas à pas

Pour l'utiliser, deux fonctions sont utilisables. La première sert à définir la vitesse de rotation, exprimée en tours par minute (**trs/min**). Pour cela, on utilise la fonction `step(steps)` qui prend en paramètre le nombre de pas à effectuer. Si ce nombre est négatif, le moteur tournera en sens inverse du nombre de pas spécifié. Voici un petit exemple qui va faire faire un aller-retour de 200 pas toutes les 2 secondes à votre moteur :

```
####include <Stepper.h>
```

```
// pour un moteur de 200 pas par tour et brancher sur les broches 2, 3, 4, 5
Stepper moteur(200, 2, 3, 4, 5);
```

```
void setup()
{
  moteur.setSpeed(30); // 30 tours par minute
  // (rappel : ici le moteur fait 200 pas par tour,
  // on fera donc 6000 pas par minute)
}

void loop()
{
  moteur.step(1000);
  delay(100);
  moteur.step(-1000);
  delay(2000);
}
```

Code : Utilisation simple d'un moteur pas à pas

[[a]] | La fonction `step(x)` est bloquante. Cela signifie qu'elle agit comme un délai. Tant que le moteur n'a pas fait les `x` pas demandés, le reste du programme est en attente.

7.3.3.3 Aller plus loin

Vous êtes devenus incollables sur les moteurs pas à pas ? Vous en voulez encore plus ? Suffit de demander, voilà du bonus d'informations rien que pour toi cher lecteur !

7.3.3.3.1 2 fils au lieu de 4 ! On a toujours utilisé 4 fils pour commander les moteurs. C'est bien, mais que diriez-vous de sauver deux broches et de passer à seulement 2 fils au lieu de 4 ? Pas mal comme amélioration non ? Petite anecdote : un jour, un utilisateur des moteurs pàp s'est rendu compte d'un truc, dans une paire de fils pour piloter un moteur (dans la séquence utilisée par Arduino), l'information est toujours antagoniste. Si un fil est à HIGH, sa paire sera à LOW et vice versa. Du coup il suffit d'un peu d'électronique pour pouvoir inverser un des deux signaux et se retrouver ainsi avec seulement deux fils au lieu de 4 sortants d'Arduino. :)

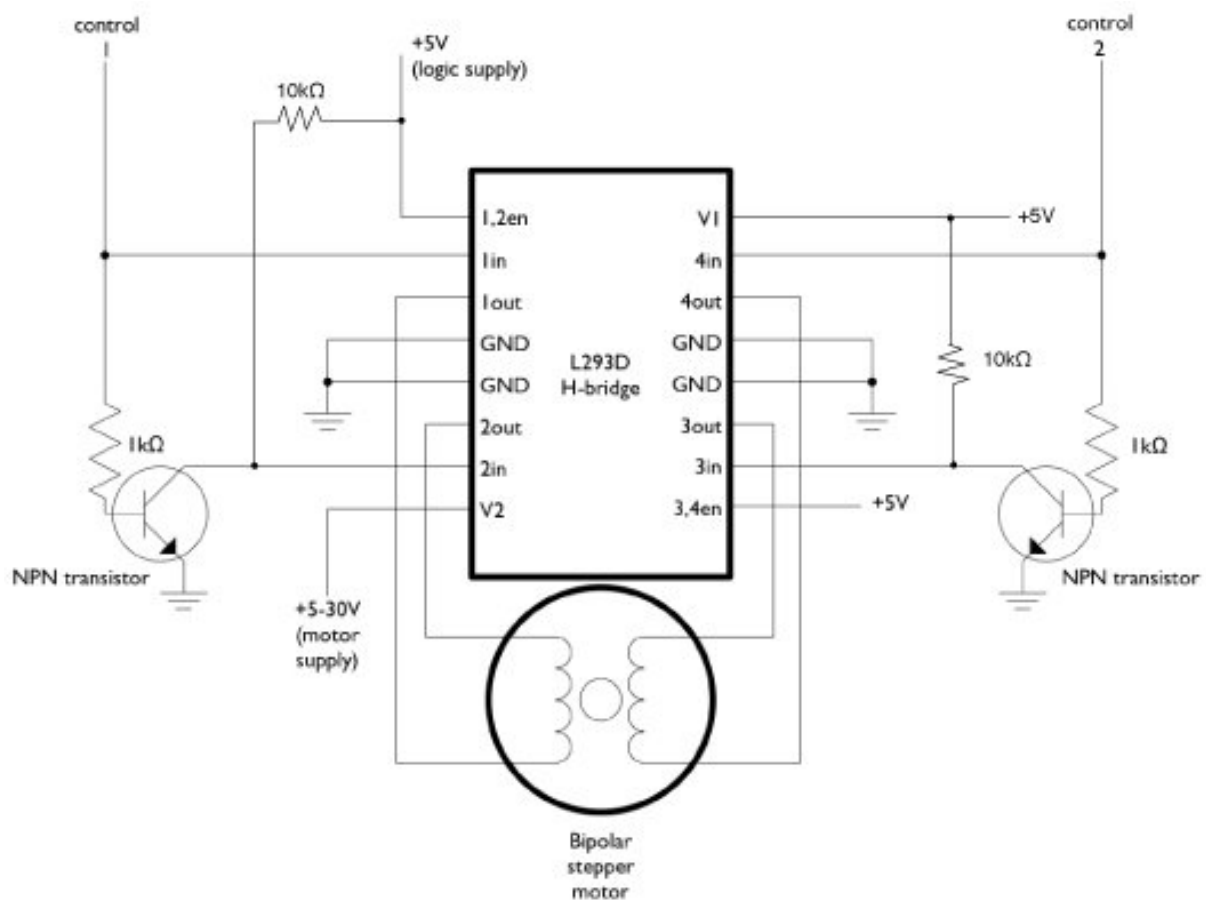


Figure 7.91 – Moteur bipolaire avec 2 fils

(source des images : tigoe.net) Cette solution est plutôt intéressante du fait que les entrées/sorties sont parfois une denrée rare sur Arduino ! ^^

7.3.3.3.2 Le L297 Lorsque vous utilisez votre Arduino, vous ne pouvez utiliser qu'une seule séquence. Par exemple pour un moteur bipolaire vous n'avez pas le choix entre le mode pas entier, demi-pas ou couple max. Une des solutions serait de générer vous-même les séquences. Mais c'est assez fastidieux. Une autre solution est électronique et compensera le développement informatique à faire. Un composant, nommé L297 (de la famille du L298 vous vous en doutez) est

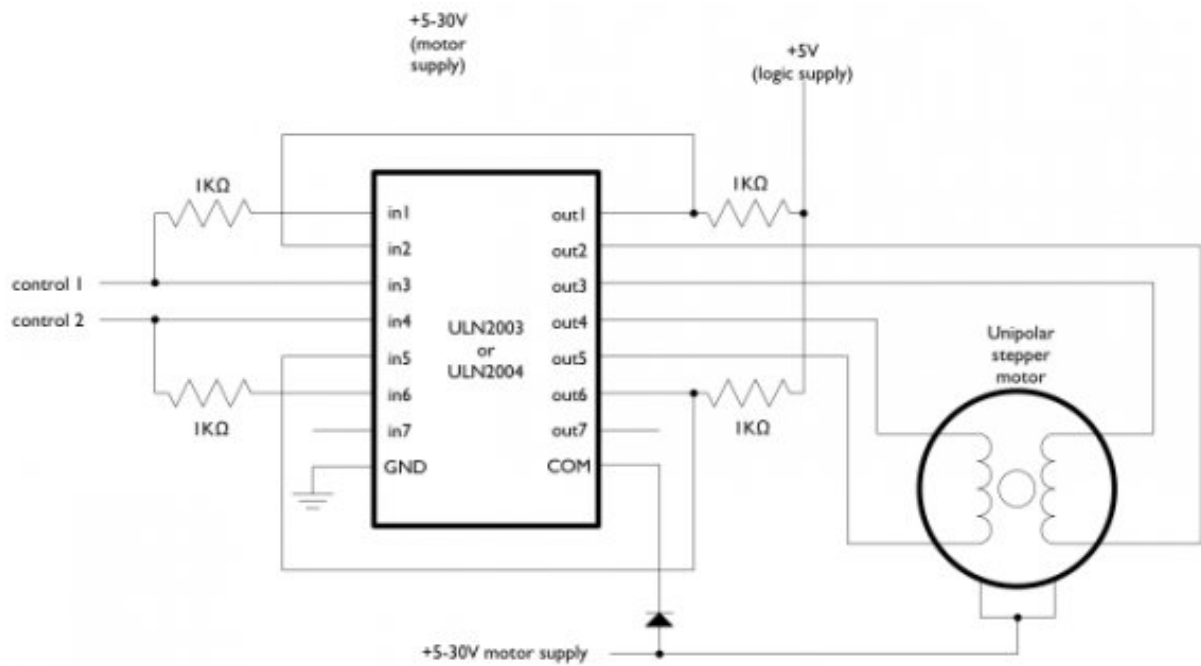


Figure 7.92 – Moteur unipolaire avec 2 fils

justement fait pour générer les séquences de moteur pas à pas. Il possède 4 broches de sorties pour générer la séquence et plusieurs en entrée pour “paramétrer” le fonctionnement voulu. Parmi elles on en retrouve trois principales :

- CW/CCW : (ClockWise ou Counter ClockWise) qui décidera du sens de rotation du moteur (horaire ou antihoraire).
- Half/Full : Qui décide si on est en mode pas entier ou demi-pas.
- Clk : (Clock) qui est l’horloge pour la vitesse. À chaque front descendant, le moteur fera un pas. Je vous laisse un peu chercher sur le net, vous trouverez de plus amples informations à ce sujet. Avant même de regarder sur le net, en fait, regardez plutôt sa datasheet !! ;) Un des avantages de déléster le travail des séquences au L297 est que vous n’aurez plus besoin de l’objet Stepper et de sa fonction step() bloquante. Il faudra cependant toujours utiliser un composant de puissance pour laisser passer les forts courants nécessaires au moteur (comme le L298 par exemple).

8 L'affichage, une autre manière d'interagir

Vous souhaitez rendre votre projet un peu plus autonome, en le disloquant de son attachement à votre ordinateur parce que vous voulez afficher du texte ? Eh bien grâce aux afficheurs LCD, cela va devenir possible !

Vous allez apprendre à utiliser ces afficheurs pour pouvoir réaliser vos projets les plus fous. Il est courant d'utiliser ces écrans permettant l'affichage du texte en domotique, robotique, et même pour déboguer un programme ! Avec eux, vos projet n'auront plus la même allure !

8.1 Les écrans LCD

Vous avez appris plus tôt comment interagir avec l'ordinateur, lui envoyer de l'information. Mais maintenant, vous voudrez sûrement pouvoir afficher de l'information sans avoir besoin d'un ordinateur. Avec les écrans LCD, nous allons pouvoir afficher du texte sur un écran qui n'est pas très coûteux et ainsi faire des projets sensationnels !

*[LCD] : Liquid Crystal Display

8.1.1 Un écran LCD c'est quoi ?

Mettons tout de suite au clair les termes : LCD signifie "Liquid Crystal Display" et se traduit, en français, par "Écran à Cristaux Liquides" (mais on n'a pas d'acronymes classe en français donc on parlera toujours de LCD). Ces écrans sont PARTOUT ! Vous en trouverez dans plein d'appareils électroniques disposant d'afficheur : les montres, le tableau de bord de votre voiture, les calculatrices, etc. Cette utilisation intensive est due à leur faible consommation et coût. Mais ce n'est pas tout ! En effet, les écrans LCD sont aussi sous des formes plus complexes telles que la plupart des écrans d'ordinateur ainsi que les téléviseurs à écran plat. Cette technologie est bien maîtrisée et donc le coût de production est assez bas. Dans les années à venir, ils vont avoir tendance à être remplacés par les écrans à affichage LED qui sont pour le moment trop chers.

[[e]] | J'en profite pour mettre l'alerte sur la différence des écrans à LED. Il en existe deux types : | - les écrans à rétro-éclairage LED : ce sont des écrans LCD tout à fait ordinaires qui ont simplement la particularité d'avoir un rétro-éclairage à LED à la place des tubes néon. Leur prix est du même ordre de grandeur que les LCD "normaux". En revanche, la qualité d'affichage des couleurs semble meilleure comparés aux LCD "normaux". | - les écrans à affichage LED : ceux-ci ne disposent pas de rétro-éclairage et ne sont ni des écrans LCD, ni des écrans plasma. Ce sont des écrans qui, en lieu et place des pixels, se trouvent des LED de très très petite taille. Leur coût est prohibitif pour le moment, mais la qualité de contraste et de couleur inégale tous les écrans existants !

Les deux catégories précédentes (écran LCD d'une montre par exemple et celui d'un moniteur d'ordinateur) peuvent être différenciées assez rapidement par une caractéristique simple : *la couleur*. En effet, les premiers sont monochromes (une seule couleur) tandis que les seconds sont colorés (rouge, vert et bleu). Dans cette partie, nous utiliserons uniquement le premier type pour des raisons de simplicité et de coût.

8.1.1.0.1 Fonctionnement de l'écran N'étant pas un spécialiste de l'optique ni de l'électronique "bas-niveau" (jonction et tout le tralala) je ne vais pas vous faire un cours détaillé sur le "comment ça marche ?" mais plutôt aller à l'essentiel, vers le "pourquoi ça s'allume ?". Comme son nom l'indique, un écran LCD possède des cristaux liquides. Mais ce n'est pas tout ! En effet, pour fonctionner il faut plusieurs choses. Si vous regardez de très près votre écran (éteint pour ne pas vous bousiller les yeux) vous pouvez voir une grille de carré. Ces carrés sont appelés des pixels (de l'anglais "Picture Element", soit "Élément d'image" en français, encore une fois c'est moins classe). :P Chaque pixel est un cristal liquide. Lorsqu'aucun courant ne le traverse, ses molécules sont orientées dans un sens (admettons, 0°). En revanche lorsqu'un courant le traverse, ses molécules vont se tourner dans la même direction (90°). Voilà pour la base.

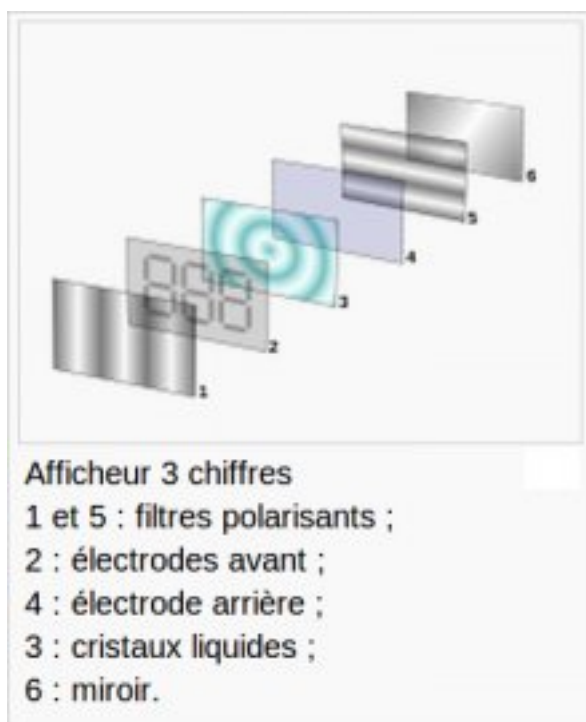


Figure : Composition d'un écran LCD - (CC-BY, [ed](#)

[g2s](#))

[[q]] | Mais pourquoi il y a de la lumière dans un cas et pas dans l'autre ?

Tout simplement parce que cette lumière est **polarisée**. Cela signifie que la lumière est orientée dans une direction (c'est un peu compliqué à démontrer, je vous demanderais donc de l'admettre). En effet, entre les cristaux liquides et la source lumineuse se trouve un filtre polariseur de lumière. Ce filtre va orienter la lumière dans une direction précise. Entre vos yeux et les cristaux se trouve un autre écran polariseur, qui est perpendiculaire au premier. Ainsi, il faut que les cristaux liquides soient dans la bonne direction pour que la lumière passe de bout en bout et revienne à vos yeux. Un schéma vaut souvent mieux qu'un long discours, je vous conseille donc de regarder celui sur la droite de l'explication pour mieux comprendre (source : Wikipédia). Enfin, vient le rétro-éclairage (fait avec des LED) qui vous permettra de lire l'écran même en pleine nuit (sinon il vous faudrait l'éclairer pour voir le contraste).

[[i]] | Si vous voulez plus d'informations sur les écrans LCD, j'invite votre curiosité à se diriger vers ce lien [Wikipédia](#) ou d'autres sources. :)

8.1.1.1 Commande du LCD

Normalement, pour pouvoir afficher des caractères sur l'écran il nous faudrait activer individuellement chaque pixel de l'écran. Un caractère est représenté par un bloc de 75 pixels. *Ce qui fait qu'un écran de 16 colonnes et 2 lignes représente un total de $16 \times 2 \times 75 = 1120$ pixels!* :P Heureusement pour nous, des ingénieurs sont passés par là et nous ont simplifié la tâche.

8.1.1.1.1 Le décodeur de caractères Tout comme il existe un driver vidéo pour votre carte graphique d'ordinateur, il existe un driver "LCD" pour votre afficheur. Rassurez-vous, aucun composant ne s'ajoute à votre liste d'achats puisqu'il est intégré dans votre écran. Ce composant va servir à décoder un ensemble "simple" de bits pour afficher un caractère à une position précise ou exécuter des commandes comme déplacer le curseur par exemple. Ce composant est fabriqué principalement par *Hitachi* et se nomme le **HC44780**. Il sert de **décodeur de caractères**. Ainsi, plutôt que de devoir multiplier les signaux pour commander les pixels un à un, il nous suffira d'envoyer des octets de commandes pour lui dire "écris moi 'zeste' à partir de la colonne 3 sur la ligne 1". Ce composant possède 16 broches que je vais brièvement décrire :

->

N°	Nom	Rôle
1	VSS	Masse
2	Vdd	+5V
3	Vo	Réglage du contraste
4	RS	Sélection du registre (commande ou donnée)
5	R/W	Lecture ou écriture
6	E	Entrée de validation
7 à 14	Do à D7	Bits de données
15	A	Anode du rétroéclairage (+5V)
16	K	Cathode du rétroéclairage (masse)

Table 8.1 – Liste des broches du LCD et leur rôle

<- [[i]] | Normalement, pour tous les écrans LCD (non graphiques) ce brochage est le même. Donc pas d'inquiétude lors des branchements, il vous suffira de vous rendre sur cette page pour consulter le tableau. ;)

Par la suite, les broches utiles qu'il faudra relier à l'Arduino sont les broches 4, 5 (facultatives), 6 et les données (7 à 14 pouvant être réduite à 8 à 14) en n'oubliant pas l'alimentation et la broche de réglage du contraste. Ce composant possède tout le système de traitement pour afficher les caractères. Il contient dans sa mémoire le schéma d'allumage des pixels pour afficher chacun d'entre eux. Voici la table des caractères affichables :

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
0	0	[NULL]	32	20	[SPACE]	64	40	@	96	60	`
1	1	[START OF HEADING]	33	21	!	65	41	A	97	61	a
2	2	[START OF TEXT]	34	22	"	66	42	B	98	62	b
3	3	[END OF TEXT]	35	23	#	67	43	C	99	63	c
4	4	[END OF TRANSMISSION]	36	24	\$	68	44	D	100	64	d
5	5	[ENQUIRY]	37	25	%	69	45	E	101	65	e
6	6	[ACKNOWLEDGE]	38	26	&	70	46	F	102	66	f
7	7	[BELL]	39	27	'	71	47	G	103	67	g
8	8	[BACKSPACE]	40	28	(72	48	H	104	68	h
9	9	[HORIZONTAL TAB]	41	29)	73	49	I	105	69	i
10	A	[LINE FEED]	42	2A	*	74	4A	J	106	6A	j
11	B	[VERTICAL TAB]	43	2B	+	75	4B	K	107	6B	k
12	C	[FORM FEED]	44	2C	,	76	4C	L	108	6C	l
13	D	[CARRIAGE RETURN]	45	2D	-	77	4D	M	109	6D	m
14	E	[SHIFT OUT]	46	2E	.	78	4E	N	110	6E	n
15	F	[SHIFT IN]	47	2F	/	79	4F	O	111	6F	o
16	10	[DATA LINK ESCAPE]	48	30	0	80	50	P	112	70	p
17	11	[DEVICE CONTROL 1]	49	31	1	81	51	Q	113	71	q
18	12	[DEVICE CONTROL 2]	50	32	2	82	52	R	114	72	r
19	13	[DEVICE CONTROL 3]	51	33	3	83	53	S	115	73	s
20	14	[DEVICE CONTROL 4]	52	34	4	84	54	T	116	74	t
21	15	[NEGATIVE ACKNOWLEDGE]	53	35	5	85	55	U	117	75	u
22	16	[SYNCHRONOUS IDLE]	54	36	6	86	56	V	118	76	v
23	17	[ENG OF TRANS. BLOCK]	55	37	7	87	57	W	119	77	w
24	18	[CANCEL]	56	38	8	88	58	X	120	78	x
25	19	[END OF MEDIUM]	57	39	9	89	59	Y	121	79	y
26	1A	[SUBSTITUTE]	58	3A	:	90	5A	Z	122	7A	z
27	1B	[ESCAPE]	59	3B	;	91	5B	[123	7B	{
28	1C	[FILE SEPARATOR]	60	3C	<	92	5C	\	124	7C	
29	1D	[GROUP SEPARATOR]	61	3D	=	93	5D]	125	7D	}
30	1E	[RECORD SEPARATOR]	62	3E	>	94	5E	^	126	7E	~
31	1F	[UNIT SEPARATOR]	63	3F	?	95	5F	_	127	7F	[DEL]

Figure : Une table ASCII - (Domaine public - LanoxthShadow)

8.1.2 Quel écran choisir ?

8.1.2.1 Les caractéristiques

8.1.2.1.1 Texte ou Graphique ? Dans la grande famille afficheur LCD, on distingue plusieurs catégories :

- Les afficheurs alphanumériques
- Les afficheurs graphiques monochromes
- Les afficheurs graphiques couleur

Les premiers sont les plus courants. Ils permettent d'afficher des lettres, des chiffres et quelques caractères spéciaux. Les caractères sont prédéfinis (voir table juste au-dessus) et on n'a donc aucunement besoin de gérer chaque pixel de l'écran. Les seconds sont déjà plus avancés. On a accès à chacun des pixels et on peut donc produire des dessins beaucoup plus évolués. Ils sont cependant légèrement plus onéreux que les premiers. Les derniers sont l'évolution des précédents, la couleur en plus (soit 3 fois plus de pixels à gérer : un sous-pixel pour le rouge, un autre pour le bleu et un dernier pour le vert, le tout forme la couleur d'un seul pixel). Pour le TP on se servira d'afficheur de la première catégorie, car ils suffisent à faire de nombreux montages et restent accessibles pour des zesteurs. ;)

8.1.2.1.2 Ce n'est pas la taille qui compte ! Les afficheurs existent dans de nombreuses tailles. Pour les afficheurs de type textes, on retrouve le plus fréquemment le format 2 lignes par 16 colonnes. Il en existe cependant de nombreux autres avec une seule ligne, ou 4 (ou plus) et 8 colonnes, ou 16, ou 20 ou encore plus ! Libre à vous de choisir la taille qui vous plaît le plus, sachant que le TP devrait s'adapter sans souci à toute taille d'écran (pour ma part, ce sera un 2 lignes 16 colonnes) !



Figure 8.1 – Un écran LCD alphanumérique

8.1.2.1.3 La couleur, c'est important Nan je blague ! Prenez la couleur qui vous plait ! Vert, blanc, bleu, jaune, amusez-vous ! (moi c'est écriture blanche sur fond bleu, mais je rêve d'un afficheur à la matrix, noir avec des écritures vertes !)

8.1.2.2 Communication avec l'écran

8.1.2.2.1 La communication parallèle De manière classique, on communique avec l'écran de manière **parallèle**. Cela signifie que l'on envoie des bits **par blocs**, en utilisant plusieurs broches en même temps (opposée à une transmission série où les bits sont envoyés un par un sur une seule broche). Comme expliqué plus tôt dans ce chapitre, nous utilisons 10 broches différentes, 8 pour les données (en parallèle donc) et 2 pour de la commande (E : Enable et RS : Register Selector). La ligne R/W peut être connecté à la masse si l'on souhaite uniquement faire de l'écriture.

Pour envoyer des données sur l'écran, c'est en fait assez simple. Il suffit de suivre un ordre logique et un certain timing pour que tout se passe bien. Tout d'abord, il nous faut placer la broche RS à 1 ou 0 selon que l'on veut envoyer une commande, par exemple "déplacer le curseur à la position (1;1)" ou que l'on veut envoyer une donnée : "écris le caractère 'a' ". Ensuite, on place sur les 8 broches de données (D0 à D7) la valeur de la donnée à afficher. Enfin, il suffit de faire une impulsion d'au moins 450 ns pour indiquer à l'écran que les données sont prêtes. C'est aussi simple que ça !

Cependant, comme les ingénieurs d'écrans sont conscients que la communication parallèle prend beaucoup de broches, ils ont inventé un autre mode que j'appellerai "semi-parallèle". Ce dernier se contente de travailler avec seulement les broches de données D4 à D7 (en plus de RS et E) et il faudra mettre les quatre autres (D0 à D3) à la masse. Il libère donc quatre broches. Dans ce mode, on fera donc deux fois le cycle "envoi des données puis impulsion sur E" pour envoyer un octet complet.

[[i]] | Ne vous inquiétez pas à l'idée de tout cela. Pour la suite du chapitre, nous utiliserons une librairie nommée **LiquidCrystal** qui se chargera de gérer les timings et l'ensemble du protocole.

Pour continuer ce chapitre, le mode “semi-parallèle” sera choisi. Il nous permettra de garder plus de broches disponibles pour de futurs montages et est souvent câblé par défaut dans de nombreux shields (dont le mien). La partie suivante vous montrera ce type de branchement. Et pas de panique, je vous indiquerai également la modification à faire pour connecter un écran en mode “parallèle complet”.

8.1.2.2.2 La communication série Lorsque l'on ne possède que très peu de broches disponibles sur notre Arduino, il peut être intéressant de faire appel à un composant permettant de communiquer par voie série avec l'écran. Un tel composant se chargera de faire la conversion entre les données envoyées sur la voie série et ce qu'il faut afficher sur l'écran. Le gros avantage de cette solution est qu'elle nécessite seulement un seul fil de donnée (avec une masse et le VCC) pour fonctionner là où les autres méthodes ont besoin de presque une dizaine de broches.

Toujours dans le cadre du prochain TP, nous resterons dans le classique en utilisant une connexion parallèle. En effet, elle nous permet de garder l'approche “standard” de l'écran et nous permet de garder la liaison série pour autre chose (encore que l'on pourrait en émuler une sans trop de difficulté). Ce composant de conversion “Série -> parallèle” peut-être réalisé simplement avec un 74h595 :) (je vous laisse coder le driver comme exercice si vous voulez :P)

8.1.2.2.3 Et par liaison I²C Un dernier point à voir, c'est la communication de la carte Arduino vers l'écran par la liaison I²C. Cette liaison est utilisable avec seulement 2 broches (une broche de donnée et une broche d'horloge) et nécessite l'utilisation de deux broches analogiques de l'Arduino (broche 4 et 5).

8.1.3 Comment on s'en sert ?

Comme expliqué précédemment, je vous propose de travailler avec un écran dont seulement quatre broches de données sont utilisées. Pour le bien de tous je vais présenter ici les deux montages, mais ne soyez pas surpris si dans les autres montages ou les vidéos vous voyez seulement un des deux. ;)

8.1.3.1 Le branchement

L'afficheur LCD utilise 6 à 10 broches de données ((D0 à D7) ou (D4 à D7) + RS + E) et deux d'alimentations (+5V et masse). La plupart des écrans possèdent aussi une entrée analogique pour régler le contraste des caractères. Nous brancherons dessus un potentiomètre de 10 kOhms. Les 10 broches de données peuvent être placées sur n'importe quelles entrées/sorties numériques de l'Arduino. En effet, nous indiquerons ensuite à la librairie LiquidCrystal qui est branché où.

8.1.3.1.1 Le montage à 8 broches de données

8.1.3.1.2 Le montage à 4 broches de données

8.1.3.2 Le démarrage de l'écran avec Arduino

Comme écrit plus tôt, nous allons utiliser la librairie “LiquidCrystal”. Pour l'intégrer, c'est très simple, il suffit de cliquer sur le menu “Import Library” et d'aller chercher la bonne. Une ligne

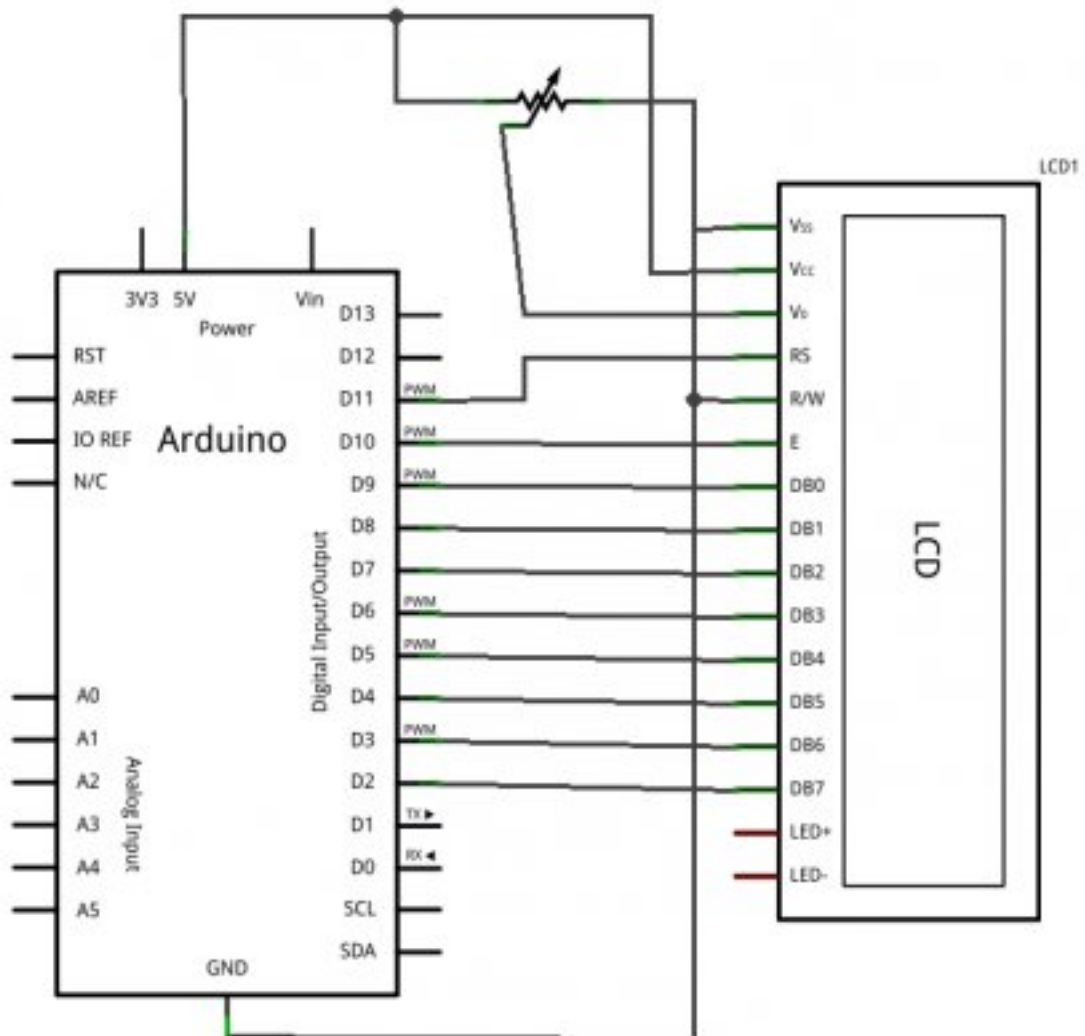


Figure 8.2 – Branchement du LCD avec fils de données

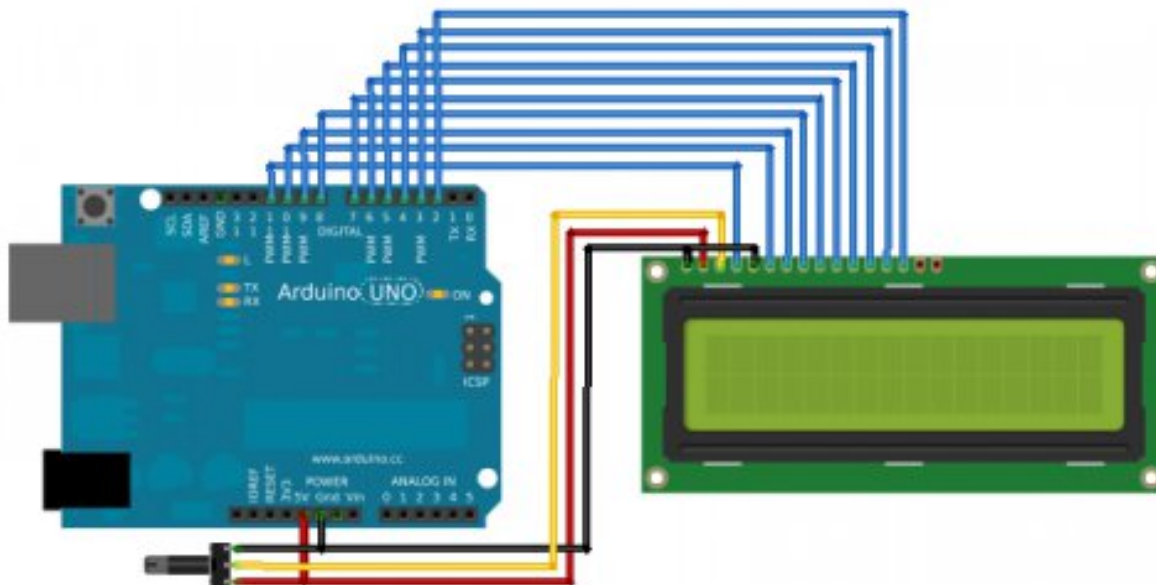


Figure 8.3 – Branchement du LCD avec 8 fils de données – montage

#include "LiquidCrystal.h" doit apparaître en haut de la page de code (les prochaines fois vous pourrez aussi taper cette ligne à la main directement, ça aura le même effet). Ensuite, il ne nous reste plus qu'à dire à notre carte Arduino où est branché l'écran (sur quelles broches) et quelle est la taille de ce dernier (nombre de lignes et de colonnes). Nous allons donc commencer par déclarer un objet (c'est en fait une variable évoluée, plus de détails dans la prochaine partie) lcd, de type LiquidCrystal et qui sera global à notre projet. La déclaration de cette variable possède plusieurs formes ([lien vers la doc.](#)) :

- LiquidCrystal(rs, enable, d0, d1, d2, d3, d4, d5, d6, d7) où rs est le numéro de la broche où est branché "RS", "enable" est la broche "E" et ainsi de suite pour les données.
- LiquidCrystal(rs, enable, d4, d5, d6, d7) (même commentaire que précédemment)

Ensuite, dans le setup() il nous faut démarrer l'écran en spécifiant son nombre de **colonnes** puis de **lignes**. Cela se fait grâce à la fonction begin(cols, rows). Voici un exemple complet de code correspondant aux deux branchements précédents (commentez la ligne qui ne vous concerne pas) :

```
####include "LiquidCrystal.h" // ajout de la librairie

// Vérifiez les broches!
LiquidCrystal lcd(11,10,9,8,7,6,5,4,3,2); // liaison 8 bits de données
LiquidCrystal lcd(11,10,5,4,3,2); // liaison 4 bits de données

void setup()
{
  lcd.begin(16,2); // utilisation d'un écran 16 colonnes et 2 lignes
  lcd.write("Salut ca zeste?"); // petit test pour vérifier que tout marche
}
```

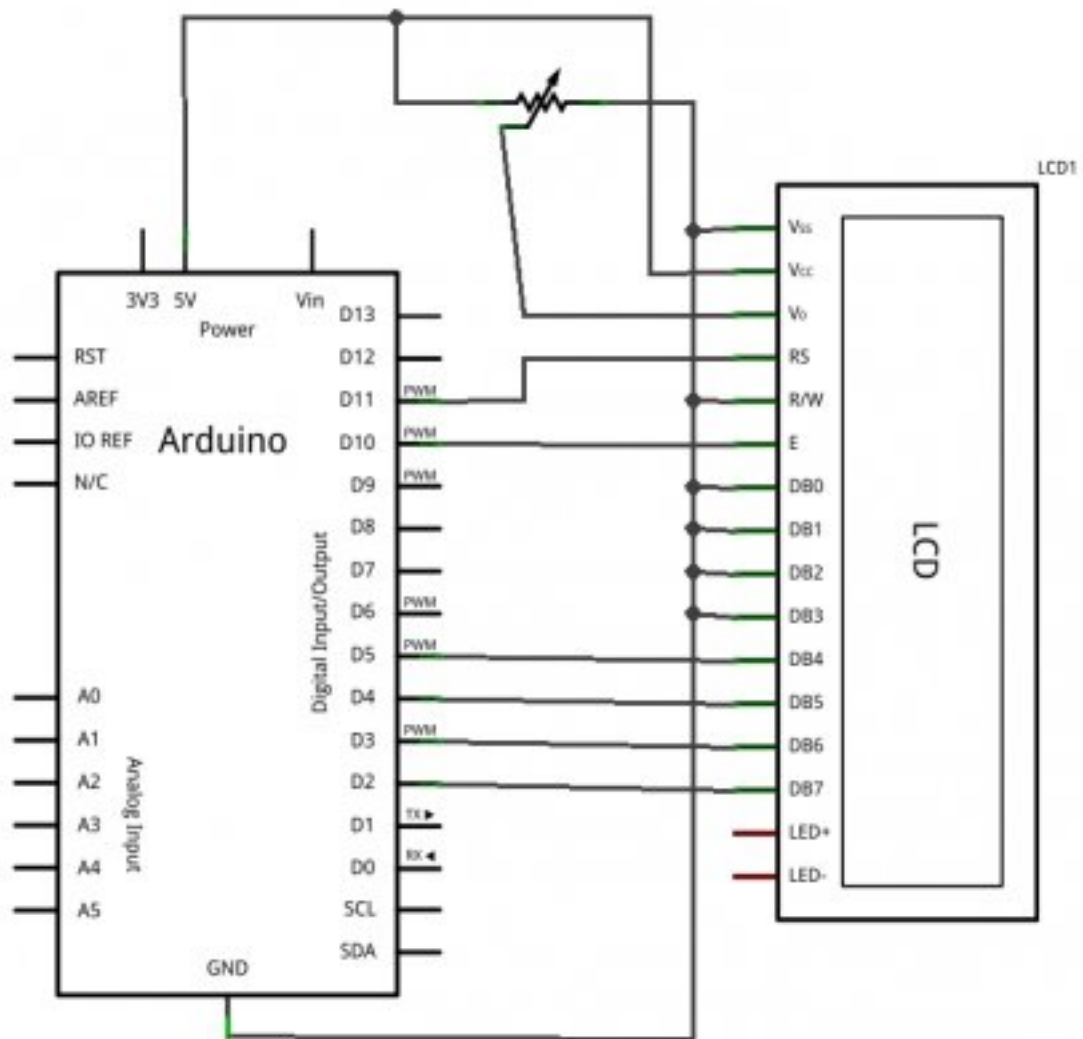



Figure 8.4 – Branchement du LCD avec 4 fils de données

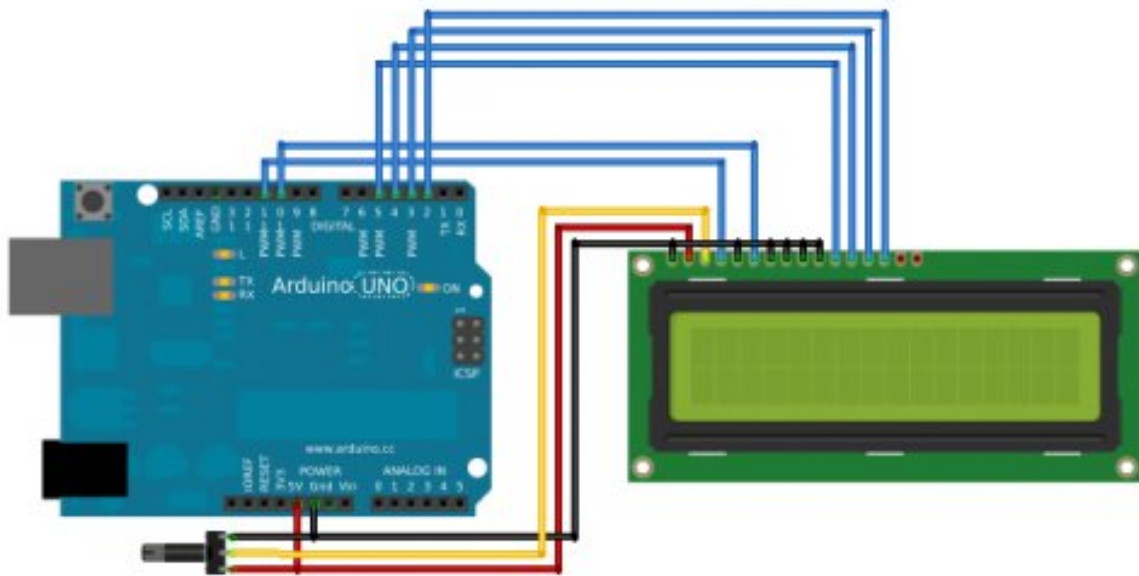


Figure 8.5 – Branchement du LCD avec 4 fils de données – montage

```
void loop() {}
```

Code : Setup minimal pour un écran LCD

[[e]] | Surtout ne mettez **pas d'accents** ! L'afficheur ne les accepte pas par défaut et affichera du grand n'importe quoi à la place.

Vous remarquez que j'ai rajouté une ligne dont je n'ai pas parlé encore. Je l'ai juste mise pour vérifier que tout fonctionne bien avec votre écran, nous reviendrons dessus plus tard. Si tout se passe bien, vous devriez obtenir l'écran suivant :



[[a]] | Si jamais rien ne s'affiche, essayez de tourner votre potentiomètre de contraste. Si cela ne marche toujours pas, vérifiez les bonnes attributions des broches (surtout si vous utilisez un shield).

Maintenant que nous maîtrisons les subtilités concernant l'écran, nous allons pouvoir commencer à jouer avec... En avant!

8.2 Votre premier texte sur le LCD!

Ça y est, on va pouvoir commencer à apprendre des trucs avec notre écran LCD! Alors, au programme : afficher des variables, des tableaux, déplacer le curseur, etc. Après toutes ces explications, vous serez devenu un pro du LCD, du moins du LCD alphanumérique. :lol : Aller, en route! Après ça vous ferez un petit TP plutôt intéressant, notamment au niveau de l'utilisation pour l'affichage des mesures sans avoir besoin d'un ordinateur. De plus, pensez au fait que vous pouvez vous aider des afficheurs pour déboguer votre programme!

8.2.1 Ecrire du texte sur le LCD

8.2.1.1 Afficher du texte

Vous vous rappelez quand je vous disais il y a longtemps "Les développeurs Arduino sont des gens sympas, ils font les choses clairement et logiquement!" ? Eh bien, ce constat se reproduit (encore) pour la bibliothèque `LiquidCrystal` ! En effet, une fois que votre écran LCD est bien paramétré, il nous suffira d'utiliser qu'une seule fonction pour afficher du texte ! Allez je vous laisse 10 secondes pour deviner le nom de la fonction que nous allons utiliser. Un indice, ça a un lien avec la voie série... C'est trouvé ? Félicitations à tous ceux qui auraient dit `print()`. En effet, une fois de plus nous retrouvons une fonction `print()`, comme pour l'objet `Serial`, pour envoyer du texte. Ainsi, pour saluer tous les zesteurs de la terre nous aurons juste à écrire :

```
lcd.print("Salut ca zeste?");
```

et pour code complet avec les déclarations, on obtient :

```
#####include "LiquidCrystal.h" // on inclut la librairie

// initialise l'écran avec les bonnes broches

// ATTENTION, REMPLACER LES NOMBRES PAR VOS BRANCHEMENTS À VOUS !

LiquidCrystal lcd(8,9,4,5,6,7);

void setup() {
    lcd.begin(16, 2);
    lcd.print("Salut ca zeste?");
}

void loop() {
}
```

[[q]] | Mais c'est nul ton truc on affiche toujours au même endroit, en haut à gauche !

Oui je sais, mais chaque chose en son temps, on s'occupera du positionnement du texte bientôt, promis !

8.2.1.2 Afficher une variable

Afficher du texte c'est bien, mais afficher du contenu dynamique c'est mieux ! Nous allons maintenant voir comment afficher une variable sur l'écran. Là encore, rien de difficile. Je ne vais donc pas faire un long discours pour vous dire qu'il n'y a qu'une seule fonction à retenir... le suspense est terrible... OUI évidemment cette fonction c'est `print()` ! Décidément elle est vraiment tout-terrain (et rédacteur du tutoriel Arduino devient un vrai boulot de feignant, je vais finir par me copier-coller à chaque fois !) Allez zou, un petit code, une petite photo et en avant Guingamp !

```
int mavariable = 42;
lcd.print(mavariable);
```

Code : `Print` pour afficher une variable

8.2.1.3 Combo! Afficher du texte ET une variable

Bon vous aurez remarqué que notre code possède une certaine faiblesse... On n'affiche au choix qu'un texte ou qu'un nombre, mais pas les deux en même temps! Nous allons donc voir maintenant une manière d'y remédier.

8.2.1.3.1 La fonction solution La solution se trouve dans les bases du langage C, grâce à une fonction qui s'appelle `sprintf()` (aussi appelé "string printf"). Les personnes qui ont fait du C doivent la connaître, ou connaître sa cousine "printf". Cette fonction est un peu particulière, car elle ne prend pas un nombre d'arguments fini. En effet, si vous voulez afficher 2 variables vous ne lui donnerez pas autant d'arguments que pour en afficher 4 (ce qui paraît logique d'une certaine manière). Pour utiliser cette dernière, il va falloir utiliser un tableau de char qui nous servira de *buffer*. Ce tableau sera celui dans lequel nous allons écrire notre chaîne de caractère. Une fois que nous aurons écrit dedans, il nous suffira de l'envoyer sur l'écran en utilisant... `print()`!

*[buffer] : Zone tampon, permet de garder en mémoire un certain nombre de données

8.2.1.3.2 Son fonctionnement Comme dit rapidement plus tôt, `sprintf()` n'a pas un nombre d'arguments fini. Cependant, elle en aura au minimum deux qui sont le tableau de la chaîne de caractère et une chaîne à écrire. Un exemple simple serait d'écrire :

```
char message[16] = "";
sprintf(message, "J'ai 42 ans");
```

Code : `sprintf` pour mettre une chaîne dans un tableau de char

Au début, le tableau `message` ne contient rien. Après la fonction `sprintf()`, il possédera le texte "J'ai 42 ans". Simple non ? [[i]] | J'utilise un tableau de 16 cases car mon écran fait 16 caractères de large au maximum, et donc inutile de gaspiller de la mémoire en prenant un tableau plus grand que nécessaire.

Nous allons maintenant voir comment changer mon âge en le mettant en dynamique dans la chaîne grâce à une variable. Pour cela, nous allons utiliser des **marqueurs de format**. Le plus connu est `%d` pour indiquer un nombre entier (nous verrons les autres ensuite). Dans le contenu à écrire (le deuxième argument), nous placerons ces marqueurs à chaque endroit où l'on voudra mettre une variable. Nous pouvons en placer autant que nous voulons. Ensuite, il nous suffira de mettre dans le même ordre que les marqueurs les différentes variables en argument de `sprintf()`. Tout va être plus clair avec un exemple!

```
char message[16] = "";
int nbA = 3;
int nbB = 5;
sprintf(message, "%d + %d = %d", nbA, nbB, nbA+nbB);
```

Code : `sprintf` pour enregistrer une (des) variables dans une chaîne

Cela affichera :

```
3 + 5 = 8
```

8.2.1.3.3 Les marqueurs Comme je vous le disais, il existe plusieurs marqueurs. Je vais vous présenter ceux qui vous serviront le plus, et différentes astuces pour les utiliser à bon escient :

- `%d` qui sera remplacé par un `int` (signé)
- `%s` sera remplacé par une chaîne (un tableau de `char`)
- `%u` pour un entier non signé (similaire à `%d`)
- `%%` pour afficher le symbole `'%'` ;)

Malheureusement, Arduino ne les supporte pas tous. En effet, le `%f` des `float` ne fonctionne pas. :(Il vous faudra donc bricoler si vous désirez l'afficher en entier (je vous laisse deviner comment). Si jamais vous désirez forcer l'affichage d'un marqueur sur un certain nombre de caractères, vous pouvez utiliser un indicateur de taille de ce nombre entre le `'%'` et la lettre du marqueur. Par exemple, utiliser `"%3d"` forcera l'affichage du nombre en paramètre (quel qu'il soit) **sur trois caractères au minimum**. La variable ne sera pas tronquée s'il est plus grand que l'emplacement prévu. Ce paramètre prendra donc toujours autant de place *au minimum* sur l'écran (utile pour maîtriser la disposition des caractères). Exemple :

```
int age1 = 42 ;
int age2 = 5 ;
char prenom1[10] = "Ben" ;
char prenom2[10] = "Luc" ;
char message[16] = "" ;
sprintf(message, "%s :%2d,%s :%2d", prenom1, age1, prenom2, age2) ;
```

Code : Différents marqueurs de variable pour `sprintf`

À l'écran, on aura un texte tel que :

```
Ben :42,Luc: 5
```

On note l'espace avant le 5 grâce au forçage de l'écriture de la variable sur 2 caractères induits par `%2d`.

8.2.1.4 Exercice, faire une horloge

8.2.1.4.1 Consigne Afin de conclure cette partie, je vous propose un petit exercice. Comme le titre l'indique, je vous propose de réaliser une petite horloge. Bien entendu elle ne sera pas fiable du tout car nous n'avons aucun repère réel dans le temps, mais ça reste un bon exercice. L'objectif sera donc d'afficher le message suivant : "Il est hh:mm:ss" avec 'hh' pour les heures, 'mm' pour les minutes et 'ss' pour les secondes. Ça vous ira ? Ouais, enfin je ne vois pas pourquoi je pose la question puisque de toute manière vous n'avez pas le choix ! :diable : Une dernière chose avant de commencer. Si vous tentez de faire plusieurs affichages successifs, le curseur ne se replacera pas et votre écriture sera vite chaotique. Je vous donne donc rapidement une fonction qui vous permet de revenir à la position en haut à gauche de l'écran : `home()`. Il vous suffira de faire un `lcd.home()` pour replacer le curseur en haut à gauche. Nous reparlerons de la position curseur dans le chapitre suivant !

8.2.1.4.2 Solution Je vais directement vous parachuter le code, sans vraiment d'explications car je pense l'avoir suffisamment commenté (et entre nous, l'exercice est sympa et pas trop dur). ;)

```

[[s]] |  cpp | #include "LiquidCrystal.h" // on inclut la librairie | | //
initialise l'écran avec les bonnes broches | // ATTENTION, REMPLACER LES
NOMBRES PAR VOS BRANCHEMENTS À VOUS! | LiquidCrystal lcd(8,9,4,5,6,7); |
| int heures,minutes,secondes; | char message[16] = ""; | | void setup() |
{ | lcd.begin(16, 2); // règle la taille du LCD : 16 colonnes et 2 lignes
| | // changer les valeurs pour démarrer à l'heure souhaitée! | heures =
0; | minutes = 0; | secondes = 0; | } | | void loop() | { | // on commence
par gérer le temps qui passe... | if(secondes == 60) // une minute est
atteinte? | { | secondes = 0; // on recompte à partir de 0 | minutes++;
| } | if(minutes == 60) // une heure est atteinte? | { | minutes = 0; |
heures++; | } | if(heures == 24) // une journée est atteinte? | { | heures
= 0; | } | | // met le message dans la chaine à transmettre | sprintf(message,"Il
est %2d:%2d:%2d",heures,minutes,secondes); | | lcd.home(); // met le curseur
en position (0;0) sur l'écran | | lcd.write(message); // envoi le message
sur l'écran | | delay(1000); // attend une seconde | // une seconde s'écoule...
| secondes++; | } |

```

8.2.2 Se déplacer sur l'écran

Bon, autant vous prévenir, ce morceau de chapitre ne sera pas digne du nom de “tutoriel”. Malheureusement, pour se déplacer sur l'écran (que ce soit le curseur ou du texte) il n'y a pas 36 solutions, juste quelques appels relativement simples à des fonctions. Désolé d'avance pour le “pseudo-listing” de fonctions que je vais faire tout en essayant de le garder intéressant...

8.2.2.1 Gérer l'affichage

Les premières fonctions que nous allons voir concernent l'écran dans son ensemble. Nous allons apprendre à enlever le texte de l'écran mais le garder dans la mémoire pour le réafficher ensuite. En d'autres termes, vous allez pouvoir faire un mode “invisible” où le texte est bien stocké en mémoire, mais pas affiché sur l'écran. Les deux fonctions permettant ce genre d'action sont les suivantes :

- `noDisplay()` : fait disparaître le texte
- `display()` : fait apparaître le texte (s'il y en a évidemment)

Si vous tapez le code suivant, vous verrez le texte clignoter toutes les secondes :

```

####include "LiquidCrystal.h" // on inclut la librairie

// initialise l'écran avec les bonnes broches
// ATTENTION, REMPLACER LES NOMBRES PAR VOS BRANCHEMENTS À VOUS !
LiquidCrystal lcd(8,9,4,5,6,7);

void setup() {
    // règle la taille du LCD
    lcd.begin(16, 2);
    lcd.print("Salut ca zeste?");
}

```

```
void loop() {  
    lcd.noDisplay();  
    delay(500);  
    lcd.display();  
    delay(500);  
}
```

Code : Afficher ou non un texte sur le LCD : `display` et `noDisplay`

Utile si vous voulez attirer l'attention de l'utilisateur ! Une autre fonction utile est celle vous permettant de nettoyer l'écran. Contrairement à la précédente, cette fonction va supprimer le texte de manière permanente. Pour le réafficher, il faudra le renvoyer à l'afficheur. Cette fonction au nom évident est : `clear()` (bien sûr, il faut être un peu anglophone pour s'en douter ^^). Le code suivant vous permettra ainsi d'afficher un texte puis, au bout de 2 secondes, il disparaîtra (pas de `loop()`, pas nécessaire) :

```
#####include "LiquidCrystal.h" // on inclut la librairie  
  
// initialise l'écran avec les bonnes broches  
// ATTENTION, REMPLACER LES NOMBRES PAR VOS BRANCHEMENTS À VOUS !  
LiquidCrystal lcd(8,9,4,5,6,7);  
  
void setup() {  
    // règle la taille du LCD  
    lcd.begin(16, 2);  
    lcd.print("Salut ca zeste?");  
    delay(2000);  
    lcd.clear();  
}
```

Code : Effacer l'écran LCD : `clear`

Cette fonction est très utile lorsque l'on fait des menus sur l'écran, pour pouvoir changer de page. Si on ne fait pas un `clear()`, il risque d'ailleurs de subsister des caractères de la page précédente. Ce n'est pas très joli.

[[a]] | Attention à ne pas appeler cette fonction plusieurs fois de suite, par exemple en la mettant dans la fonction `loop()`, vous verrez le texte ne s'affichera que très rapidement puis disparaîtra et ainsi de suite.

8.2.2.2 Gérer le curseur

8.2.2.2.1 Se déplacer sur l'écran Voici maintenant d'autres fonctions que vous attendez certainement, celles permettant de déplacer le curseur sur l'écran. En déplaçant le curseur, vous pourrez écrire à n'importe quel endroit sur l'écran (attention cependant à ce qu'il y ait suffisamment de place pour votre texte). :P Nous allons commencer par quelque chose de facile que nous avons vu très rapidement dans le chapitre précédent. Je parle bien sûr de la fonction `home()` ! Souvenez-vous, cette fonction permet de replacer le curseur au début de l'écran. [[q]] | Mais au fait, savez-vous comment est organisé le repère de l'écran ?

C'est assez simple, mais il faut être vigilant quand même. Tout d'abord, sachez que les coordonnées s'expriment de la manière suivante (x, y) . x représente les abscisses, donc les pixels hori-

zontaux et y les ordonnées, les pixels verticaux. L'origine du repère sera logiquement le pixel le plus en haut à gauche (comme la lecture classique d'un livre, on commence en haut à gauche) et à pour coordonnées ... (0,0)! Eh oui, on ne commence pas aux pixels (1,1) mais bien (0,0). Quand on y réfléchit, c'est assez logique. Les caractères sont rangés dans des chaînes de caractères, donc des tableaux, qui eux sont adressés à partir de la case 0. Il paraît donc au final logique que les développeurs aient gardé une cohérence entre les deux.

Puisque nous commençons à 0, un écran de 16x2 caractères pourra donc avoir comme coordonnées de 0 à 15 pour x et 0 ou 1 pour y . Ceci étant dit, nous pouvons passer à la suite. La prochaine fonction que nous allons voir prend directement en compte ce que je viens de vous dire. Cette fonction nommée `setCursor()` vous permet de positionner le curseur sur l'écran. On pourra donc faire `setCursor(0,0)` pour se placer en haut à gauche (équivalent à la fonction "home()") et en faisant `setCursor(15,1)` on se placera tout en bas à droite (toujours pour un écran de 16x2 caractères). Un exemple :

```
#####include "LiquidCrystal.h" // on inclut la librairie

// initialise l'écran avec les bonnes broches
// ATTENTION, REMPLACER LES NOMBRES PAR VOS BRANCHEMENTS À VOUS !
LiquidCrystal lcd(8,9,4,5,6,7);

void setup()
{
  lcd.begin(16, 2);
  lcd.setCursor(2,1);          // place le curseur aux coordonnées (2,1)
  lcd.print("Texte centré"); // texte centré sur la ligne 2
}
```

Code : Positionner le curseur sur l'écran LCD : `setCursor`

8.2.2.2.2 Animer le curseur Tout comme nous pouvons faire disparaître le texte, nous pouvons aussi faire disparaître le curseur (comportement par défaut). La fonction `noCursor()` va donc l'effacer. La fonction antagoniste `cursor()` de son côté permettra de l'afficher (vous verrez alors un petit trait en bas du carré (5*8 pixels) où il est placé, comme lorsque vous appuyez sur la touche Insér. de votre clavier). Une dernière chose sympa à faire avec le curseur est de le faire clignoter. En anglais clignoter se dit "blink" et donc tout logiquement la fonction à appeler pour activer le clignotement est `blink()`. Vous verrez alors le curseur remplir le carré concerné en blanc puis s'effacer (juste le trait) et revenir. S'il y a un caractère en dessous, vous verrez alternativement un carré tout blanc puis le caractère. Pour désactiver le clignotement, il suffit de faire appel à la fonction `noBlink()`.

```
#####include "LiquidCrystal.h" // on inclut la librairie

// ATTENTION, REMPLACER LES NOMBRES PAR VOS BRANCHEMENTS À VOUS !
LiquidCrystal lcd(8,9,4,5,6,7);

void setup()
{
  lcd.begin(16, 2);
  lcd.home();          // place le curseur aux coordonnées (0,0)
```

```
    lcd.setCursor(); // affiche le curseur
    lcd.blink();    // et le fait clignoter
    lcd.print("Curseur clignotant"); // texte centré sur la ligne 2
}
```

Code : Faire clignoter le curseur : blink

[[i]] | Si vous faites appel à blink() puis à noCursor() le carré blanc continuera de clignoter. En revanche, quand le curseur est dans sa phase “éteinte” vous ne verrez plus le trait du bas.

8.2.2.3 Jouer avec le texte

Nous allons maintenant nous amuser avec le texte. Ne vous attendez pas non plus à des miracles, il s'agira juste de déplacer le texte automatiquement ou non.

8.2.2.3.1 Déplacer le texte à la main Pour commencer, nous allons déplacer le texte manuellement, vers la droite ou vers la gauche. N'essayez pas de produire l'expérience avec votre main, ce n'est pas un écran tactile, hein!;) Le comportement est simple à comprendre. Après avoir écrit du texte sur l'écran, on peut faire appel aux fonctions scrollDisplayRight() et scrollDisplayLeft() vous pourrez déplacer le texte d'un carré vers la droite ou vers la gauche. S'il y a du texte sur chacune des lignes avant de faire appel aux fonctions, c'est le texte de chaque ligne qui sera déplacé par la fonction. Utilisez deux petits boutons poussoirs pour utiliser le code suivant. Vous pourrez déplacer le texte en appuyant sur chacun des poussoirs!

```
#####include "LiquidCrystal.h" // on inclut la librairie

// les branchements
const int boutonGauche = 2; // le bouton de gauche
const int boutonDroite = 3; // le bouton de droite

// initialise l'écran avec les bonnes broches
// ATTENTION, REMPLACER LES NOMBRES PAR VOS BRANCHEMENTS À VOUS !
LiquidCrystal lcd(8,9,4,5,6,7);

// -----

void setup() {
    // réglage des entrées/sorties
    pinMode(boutonGauche, INPUT);
    pinMode(boutonDroite, INPUT);

    // on attache des fonctions aux deux interruptions externes (les boutons)
    attachInterrupt(0, aDroite, RISING);
    attachInterrupt(1, aGauche, RISING);

    // Réglage du LCD
    lcd.begin(16, 2); // règle la taille du LCD
    lcd.print("Salut ca zeste?");
}
```

```

void loop() {
    // pas besoin de loop pour le moment
}

// fonction appelée par l'interruption du premier bouton
void aGauche() {
    lcd.scrollDisplayLeft(); // on va à gauche!
}

// fonction appelée par l'interruption du deuxième bouton
void aDroite() {
    lcd.scrollDisplayRight(); // on va à droite!
}

```

Code : Déplacer le texte : scrollDisplay

->!(<https://www.youtube.com/watch?v=G-uCcoqYdWg>)<-

8.2.2.3.2 Déplacer le texte automatiquement De temps en temps, il peut être utile d'écrire toujours sur le même pixel et de faire en sorte que le texte se décale tout seul (pour faire des effets zolis par exemple). ^^ Un couple de fonctions va nous aider dans cette tâche. La première sert à définir la direction du défilement. Elle s'appelle `leftToRight()` pour aller de la gauche vers la droite et `rightToLeft()` pour l'autre sens. Ensuite, il suffit d'activer (ou pas si vous voulez arrêter l'effet) avec la fonction `autoScroll()` (et `noAutoScroll()` pour l'arrêter). Pour mieux voir cet effet, je vous propose d'essayer le code qui suit. Vous verrez ainsi les chiffres de 0 à 9 apparaître et se "pousser" les uns après les autres :

```

####include "LiquidCrystal.h" // on inclut la librairie

// ATTENTION, REMPLACER LES NOMBRES PAR VOS BRANCHEMENTS À VOUS !
LiquidCrystal lcd(8,9,4,5,6,7);

void setup()
{
    lcd.begin(16, 2);
    lcd.setCursor(14,0);
    lcd.leftToRight(); // indique que le texte doit être déplacé vers la gauche
    lcd.autoscroll(); // rend automatique ce déplacement
    lcd.print("{");
    int i=0;
    for(i=0; i<10; i++)
    {
        lcd.print(i);
        delay(1000);
    }
    lcd.print("}");
}

```

Code : Déplace le texte automatiquement : autoScroll

8.2.3 Créer un caractère

Dernière partie avant la pratique, on s'accroche vous serez bientôt incollable sur les écrans LCD ! En plus, réjouissez-vous ! je vous ai gardé un petit truc sympa pour la fin. En effet, dans ce dernier morceau, toute votre âme créatrice va pouvoir s'exprimer ! Nous allons créer des caractères !

8.2.3.0.1 Principe de la création Créer un caractère n'est pas très difficile, il suffit d'avoir un peu d'imagination. Sur l'écran, les pixels sont en réalité divisés en grille de 5x8 (5 en largeur et 8 en hauteur). C'est parce que le contrôleur de l'écran connaît l'alphabet qu'il peut dessiner sur ces petites grilles les caractères et les chiffres. Comme je viens de le dire, les caractères sont une grille de 5x8. Cette grille sera symbolisée en mémoire par un tableau de huit octets (type `byte`). Les 5 bits de poids faible de chaque octet représenteront une ligne du nouveau caractère. Pour faire simple, prenons un exemple. Nous allons dessiner un smiley, avec ses deux yeux et sa bouche pour avoir le rendu suivant :

```
0 0 0 0 0
X 0 0 0 X
0 0 0 0 0
0 0 0 0 0
X 0 0 0 X
0 X X X 0
0 0 0 0 0
0 0 0 0 0
```

Ce dessin se traduira en mémoire par un tableau d'octet que l'on pourra coder de la manière suivante :

```
byte smiley[8] = {
    B00000,
    B10001,
    B00000,
    B00000,
    B10001,
    B01110,
    B00000,
    B00000
};
```

Code : Tableau de représentation d'un smiley

La lettre 'B' avant l'écriture des octets veut dire "Je t'écris la valeur en binaire". Cela nous permet d'avoir un rendu plus facile et rapide.

8.2.3.0.2 L'envoyer à l'écran et l'utiliser Une fois que votre caractère est créé, il faut l'envoyer à l'écran, pour que ce dernier puisse le connaître, avant toute communication avec l'écran (oui oui avant le `begin()`). La fonction pour apprendre notre caractère à l'écran se nomme `createChar()` signifiant "créer caractère". Cette fonction prend deux paramètres : "l'adresse" du caractère dans la mémoire de l'écran (de 0 à 7) et le tableau de byte représentant le caractère. Ensuite, l'étape de départ de communication avec l'écran peut-être faite (le `begin`). Ensuite,



Figure 8.6 – Oh le joli smiley!

si vous voulez écrire ce nouveau caractère sur votre bel écran, nous allons utiliser une nouvelle (la dernière fonction) qui s'appelle `write()`. En paramètre sera passé un `int` représentant le numéro (adresse) du caractère que l'on veut afficher. Cependant, il y a là une faille dans le code Arduino. En effet, la fonction `write()` existe aussi dans une librairie standard d'Arduino et prend un pointeur sur un `char`. Le seul moyen de les différencier pour le compilateur sera donc de regarder le paramètre de la fonction pour savoir ce que vous voulez faire. Dans notre cas, il faut passer un `int`. On va donc forcer (on dit "caster") le paramètre dans le type "`uint8_t`" en écrivant la fonction de la manière suivante : `write(uint8_t param)`. Le code complet sera ainsi le suivant :

```
#####include "LiquidCrystal.h" // on inclut la librairie

// initialise l'écran avec les bonnes broches
// ATTENTION, REMPLACER LES NOMBRES PAR VOS BRANCHEMENTS À VOUS !
LiquidCrystal lcd(8,9,4,5,6,7);

// notre nouveau caractère
byte smiley[8] = {
    B00000,
    B10001,
    B00000,
    B00000,
    B10001,
    B01110,
    B00000,
```

```
};  
  
void setup()  
{  
  lcd.createChar(0, smiley); // apprend le caractère à l'écran LCD  
  lcd.begin(16, 2);  
  lcd.write((uint8_t) 0); // affiche le caractère de l'adresse 0  
}
```

Code : Insertion et utilisation d'un caractère dans la mémoire du LCD

Désormais, vous savez l'essentiel sur les LCD alphanumériques, vous êtes donc aptes pour passer au TP. ;)

8.3 [TP] Supervision avec un LCD

Chers lecteurs et lectrices, savez-vous qu'il est toujours aussi difficile de faire une introduction et une conclusion pour chaque chapitre ? C'est pourquoi je n'ai choisi ici que de dire ceci : *amusez-vous bien avec les LCD!* :D

8.3.1 Consigne

Dans ce TP, on se propose de mettre en place un système de supervision, comme on pourrait en retrouver dans un milieu industriel (en plus simple ici, bien sûr !) ou dans d'autres applications. Le but sera d'afficher des informations sur l'écran LCD en fonction d'évènements qui se passent dans le milieu extérieur. Ce monde extérieur sera représenté par les composants suivants :

- Deux boutons, qui pourraient représenter par exemple deux barrières infrarouges et dont le signal reçu passe de 1 à 0 lorsqu'un objet passe devant.
- Deux potentiomètres. Le premier sert de "consigne" et est réglé par l'utilisateur. Le second représentera un capteur. À titre d'exemple, sur la vidéo à la suite vous verrez un potentiomètre rotatif qui représentera la consigne et un autre sous forme de glissière qui sera le capteur.
- Enfin, une LED rouge nous permettra de faire une alarme visuelle. Elle sera normalement éteinte mais si la valeur du capteur dépasse celle de la consigne alors elle s'allumera.

8.3.1.0.1 Comportement de l'écran L'écran que j'utilise ne propose que 2 lignes et 16 colonnes. Il n'est donc pas possible d'afficher toutes les informations de manière lisible en même temps. On se propose donc de faire un affichage alterné entre deux interfaces. Chaque interface sera affichée pendant cinq secondes à tour de rôle. La première affichera l'état des boutons. On pourra par exemple lire :

Bouton G : ON

Bouton D : OFF

La seconde interface affichera la valeur de la consigne et celle du capteur. On aura par exemple :

Consigne : 287
Capteur : 115

(Sur la vidéo vous verrez “gauche / droite” pour symboliser les deux potentiomètres, chacun fait comme il veut/peut :P).

Enfin, bien que l’information “consigne/capteur” ne s’affiche que toutes les 5 secondes, l’alarme (la LED rouge), elle, doit être visible à tout moment si la valeur du capteur dépasse celle de la consigne. En effet, imaginez que cette alarme représentera une pression trop élevée, ce serait dommage que tout explose à cause d’un affichage 5 secondes sur 10 ! :P Je pense avoir fait le tour de mes attentes ! Je vous souhaite un bon courage, prenez votre temps, faites un beau schéma/montage/code et à bientôt pour la correction !

->!(<https://www.youtube.com/watch?v=rGdUo7j2ou8>)<-

8.3.2 Correction

8.3.2.1 Le montage

Vous en avez l’habitude maintenant, je vais vous présenter le schéma puis ensuite le code. Pour le schéma, je n’ai pas des milliers de commentaires à faire. Parmi les choses auxquelles il faut être attentif se trouvent :

- Des condensateurs de filtrage pour éviter les rebonds parasites créés par les boutons
- Mettre les potentiomètres sur des entrées analogiques
- Brancher la LED dans le bon sens et ne pas oublier sa résistance de limitation de courant

Et en cas de doute, voici le schéma (qui est un peu fouillis par endroit, j’en suis désolé)!

8.3.2.2 Le code

Là encore, je vais reprendre le même schéma de fonctionnement que d'habitude en vous présentant tout d'abord les variables globales utilisées, puis les initialisations pour continuer avec quelques fonctions utiles et la boucle principale.

8.3.2.2.1 Les variables utilisées Dans ce TP, beaucoup de variables vont être déclarées. En effet, il en faut déjà 5 pour les entrées/sorties (2 boutons, 2 potentiomètres, 1 LED), j'utilise aussi deux tableaux pour contenir et préparer les messages à afficher sur la première et deuxième ligne. Enfin, j'en utilise 4 pour contenir les mesures faites et 4 autres servant de mémoire pour ces mesures. Ah et j'oubliais, il me faut aussi une variable contenant le temps écoulé et une servant à savoir sur quelle "interface" nous sommes en train d'écrire. Voici un petit tableau résumant tout cela ainsi que le type des variables.

```
[[s]] | -> | | Nom | Type | Description | --|---|----- | boutonGauche | const int | Broche du bouton de gauche | boutonDroite | const int | Broche du bouton de droite | potentiometreGauche | const int | Broche du potar "consigne" | potentiometreDroite | const int | Broche du potar "alarme" | ledAlarme | const int | Broche de la LED d'alarme | messageHaut[16] | char | Tableau représentant la ligne du haut | messageBas[16] | char | Tableau représentant la ligne du bas | etatGauche | int | État du bouton de gauche | etatDroite | int | État du bouton de droite | niveauGauche | int | Conversion du potar de gauche | niveauDroite | int | Conversion du potar de droite | etatGauche_old | int | Mémoire de l'état du bouton de gauche | etatDroite_old | int | Mémoire de l'état du bouton de droite | niveauGauche_old | int | Mémoire de la conversion du potar de gauche | niveauDroite_old | int | Mémoire de la conversion du potar de droite | temps | unsigned long | Pour mémoriser le temps écoulé | ecran | boolean | Pour savoir sur quelle interface on écrit | | Table : Liste des broches utilisées | | <-
```

8.3.2.2.2 Le setup Maintenant que les présentations sont faites, nous allons passer à toutes les initialisations. Le setup n'aura que peu de choses à faire puisqu'il suffira de régler les broches en entrées/sorties et de mettre en marche l'écran LCD.

```
[[s]] | cpp | void setup() { | // réglage des entrées/sorties | pinMode(boutonGauche, INPUT); | pinMode(boutonDroite, INPUT); | pinMode(ledAlarme, OUTPUT); | | // réglage du LCD | lcd.begin(16, 2); // règle la taille du LCD | lcd.noBlink(); // pas de clignotement | lcd.noCursor(); // pas de curseur | lcd.noAutoscroll(); // pas de défilement | } | | Code : Correction de l'exercice : setup
```

8.3.2.2.3 Quelques fonctions utiles Afin de bien séparer notre code en morceaux logiques, nous allons écrire plusieurs fonctions, qui ont toutes un rôle particulier. La première d'entre elles sera celle chargée de faire le relevé des valeurs. Son objectif sera de faire les conversions analogiques et de regarder l'état des entrées numériques. Elle stockera bien entendu chacune des mesures dans la variable concernée.

```
[[s]] | cpp | void recupererDonnees() | { | // efface les anciens avec les "nouveaux anciens" | etatGauche_old = etatGauche; | etatDroite_old = etatDroite; | niveauGauche_old = niveauGauche; | niveauDroite_old = niveauDroite; | | // effectue les mesures | etatGauche = digitalRead(boutonGauche); | etatDroite = digitalRead(boutonDroite); | niveauGauche = analogRead(potentiometreGauche); | niveauDroite = analogRead(potentiometreDroite); | | // pour s'assurer
```

que les conversions analogiques sont terminées | // avant de passer à la suite on fait une petite pause | delay(2); | } | | Code : Correction de l'exercice : fonction recupererDonnees

Ensuite, deux fonctions vont nous permettre de déterminer si oui ou non il faut mettre à jour l'écran. En effet, afin d'éviter un phénomène de scintillement qui se produit si on envoie des données sans arrêt, on préfère écrire sur l'écran que si nécessaire. Pour décider si l'on doit mettre à jour les "phrases" concernant les boutons, il suffit de vérifier l'état "ancien" et l'état courant de chaque bouton. Si l'état est différent, notre fonction renvoie `true`, sinon elle renvoie `false`. Une même fonction sera codée pour les valeurs analogiques. Cependant, comme les valeurs lues par le convertisseur de la carte Arduino ne sont pas toujours très stables (je rappelle que le convertisseur offre plus ou moins deux bits de précision, soit 20mV de précision totale), on va faire une petite opération. Cette opération consiste à regarder si la valeur absolue de la différence entre la valeur courante et la valeur ancienne est supérieure à deux unités. Si c'est le cas, on renvoie `true`, sinon `false`.

```
[[s]] | cpp | boolean boutonsChanged() | { | // si un bouton à changé d'état
| if(etatGauche_old!= etatGauche || etatDroite_old!= etatDroite) | return
true; | else | return false; | } | | boolean potarChanged() | { | // si
un potentiomètre affiche une différence de plus de 2 unités | // entre ces
deux valeurs, alors on met à jour | if(abs(niveauGauche_old-niveauGauche)
> 2 || | abs(niveauDroite_old-niveauDroite) > 2) | { | return true; | }
| else | { | return false; | } | } | | Code : Correction de l'exercice : fonction
boutonsChanged et potarChanged
```

Une dernière fonction nous servira à faire la mise à jour de l'écran. Elle va préparer les deux chaînes de caractères (celle du haut et celle du bas) et va ensuite les envoyer successivement sur l'écran. Pour écrire dans les chaînes, on vérifiera la valeur de la variable `ecran` pour savoir si on doit écrire les valeurs des potentiomètres ou celles des boutons. L'envoi à l'écran se fait simplement avec `print()` comme vu antérieurement. On notera le `clear()` de l'écran avant de faire les mises à jour. En effet, sans cela les valeurs pourraient se chevaucher (essayez d'écrire un OFF puis un ON, sans `clear()`, cela vous fera un "ONF" à la fin. ;)

```
[[s]] | cpp | void updateEcran() | { | if(ecran) | { | // prépare les chaînes
à mettre sur l'écran : boutons | if(etatGauche) | sprintf(messageHaut,"Bouton
G : ON"); | else | sprintf(messageHaut,"Bouton G : OFF"); | if(etatDroite)
| sprintf(messageBas,"Bouton D : ON"); | else | sprintf(messageBas,"Bouton
D : OFF"); | } | else | { | // prépare les chaînes à mettre sur l'écran :
potentiomètres | sprintf(messageHaut,"gauche = %4d", niveauGauche); | sprintf(messageB
= %4d", niveauDroite); | } | | // on envoie le texte | lcd.clear(); | lcd.setCursor(0,
| lcd.print(messageHaut); | lcd.setCursor(0,1); | lcd.print(messageBas); |
} | | Code : Correction de l'exercice : fonction updateEcran
```

8.3.2.2.4 La boucle principale Nous voici enfin au cœur du programme, la boucle principale. Cette dernière est relativement légère, grâce aux fonctions permettant de répartir le code en unité logique. La boucle principale n'a plus qu'à les utiliser à bon escient et dans le bon ordre (:P) pour faire son travail. Dans l'ordre il nous faudra donc :

- Récupérer toutes les données (faire les conversions, etc.).
- Selon l'interface courante, afficher soit les états des boutons soit les valeurs des potentiomètres si ils/elles ont changé(e)s.

- Tester les valeurs des potentiomètres pour déclencher l'alarme ou non.
- Enfin, si 5 secondes se sont écoulées, changer d'interface et mettre à jour l'écran.

Simple non ? On ne le dira jamais assez, un code bien séparé est toujours plus facile à comprendre et à retoucher si nécessaire ! :) Allez, comme vous êtes sages, voici le code de cette boucle (qui va de paire avec les fonctions expliquées précédemment) :

```
[[s]] | cpp | void loop() { | | // commence par récupérer les données des
boutons et capteurs | recupererDonnees(); | | if(ecran) // quel écran affiche
t'on? (bouton ou potentiomètre?) | { | if(boutonsChanged()) // si un bouton
a changé d'état | updateEcran(); | } | else | { | if(potarChanged()) // si
un potentiomètre a changé d'état | updateEcran(); | } | | if(niveauDroite
> niveauGauche) | // RAPPEL : piloté à l'état bas donc on allume! | digitalWrite(ledAl
LOW); | else | digitalWrite(ledAlarme, HIGH); | | // si ça fait 5s qu'on
affiche la même donnée | if(millis() - temps > 5000) | { | ecran = ~ecran;
| lcd.clear(); | updateEcran(); | temps = millis(); | } | } | | Code : Correc-
tion de l'exercice : loop
```

8.3.2.2.5 Programme complet Voici enfin le code complet. Vous pourrez le copier/coller et l'essayer pour comparer si vous voulez. **Attention cependant à déclarer les bonnes broches en fonction de votre montage (notamment pour le LCD).**

```
[[s]] | cpp | #include "LiquidCrystal.h" // on inclut la librairie | | // les
branchements | const int boutonGauche = 11; // le bouton de gauche | const
int boutonDroite = 12; // le bouton de droite | const int potentiometreGauche
= 0; // le potentiomètre de gauche (analogique 0) | const int potentiometreDroite
= 1; // le potentiomètre de droite (analogique 1) | const int ledAlarme
= 2; // la LED est branché sur la sortie 2 | | // initialise l'écran avec
les bonnes broches | // ATTENTION, REMPLACER LES NOMBRES PAR VOS BRANCHEMENTS
À VOUS! | LiquidCrystal lcd(8,9,4,5,6,7); | | char messageHaut[16] = "";
// Message sur la ligne du dessus | char messageBas[16] = ""; // Message
sur la ligne du dessous | | unsigned long temps = 0; // pour garder une
trace du temps qui s'écoule | boolean ecran = LOW; // savoir si on affiche
les boutons ou les conversions | | int etatGauche = LOW; // état du bouton
de gauche | int etatDroite = LOW; // état du bouton de droite | int niveauGauche
= 0; // conversion du potentiomètre de gauche | int niveauDroite = 0; //
conversion du potentiomètre de droite | | // les mêmes variables mais "old"
| // servant de mémoire pour constater un changement | int etatGauche_old
= LOW; // état du bouton de gauche | int etatDroite_old = LOW; // état du
bouton de droite | int niveauGauche_old = 0; // conversion du potentiomètre
de gauche | int niveauDroite_old = 0; // conversion du potentiomètre de
droite | | // -----
| | void setup() { | // réglage des entrées/sorties | pinMode(boutonGauche,
INPUT); | pinMode(boutonDroite, INPUT); | pinMode(ledAlarme, OUTPUT); |
| // paramétrage du LCD | lcd.begin(16, 2); // règle la taille du LCD |
lcd.noBlink(); // pas de clignotement | lcd.noCursor(); // pas de curseur
| lcd.noAutoscroll(); // pas de défilement | } | | void loop() { | | //
commence par récupérer les données des boutons et capteurs | recupererDonnees();
| | if(ecran) // quel écran affiche-t'on? (bouton ou potentiomètre?) | {
| if(boutonsChanged()) // si un bouton a changé d'état | updateEcran(); |
```

```
} | else | { | if(potarChanged()) // si un potentiomètre a changé d'état
| updateEcran(); | } | | if(niveauDroite > niveauGauche) | // RAPPEL :
piloté à l'état bas donc on allume! | digitalWrite(ledAlarme, LOW); | else
| digitalWrite(ledAlarme, HIGH); | | // si ça fait 5s qu'on affiche la
même donnée | if(millis() - temps > 5000) | { | ecran = ~ecran; | lcd.clear();
| updateEcran(); | temps = millis(); | } | } | | // -----
| | void recupererDonnees() | { | // efface les anciens avec les "nouveaux
anciens" | etatGauche_old = etatGauche; | etatDroite_old = etatDroite; |
niveauGauche_old = niveauGauche; | niveauDroite_old = niveauDroite; | |
etatGauche = digitalRead(boutonGauche); | etatDroite = digitalRead(boutonDroite);
| niveauGauche = analogRead(potentiometreGauche); | niveauDroite = analogRead(potentio
| | // pour s'assurer que les conversions analogiques sont terminées | //
on fait une petite pause avant de passer à la suite | delay(1); | } | |
boolean boutonsChanged() | { | if(etatGauche_old != etatGauche || etatDroite_old !=
etatDroite) | return true; | else | return false; | } | | boolean potarChanged()
| { | // si un potentiomètre affiche une différence de plus de 2 unités |
// entre ces deux valeurs, alors on met à jour | if(abs(niveauGauche_old-niveauGauche)
> 2 || | abs(niveauDroite_old-niveauDroite) > 2) | { | return true; | } |
else | { | return false; | } | } | | void updateEcran() | { | if(ecran) |
{ | // prépare les chaines à mettre sur l'écran | if(etatGauche) | sprintf(messageHaut,
G : ON"); | else | sprintf(messageHaut,"Bouton G : OFF"); | if(etatDroite)
| sprintf(messageBas,"Bouton D : ON"); | else | sprintf(messageBas,"Bouton
D : OFF"); | } | else | { | // prépare les chaines à mettre sur l'écran |
sprintf(messageHaut,"gauche = %4d", niveauGauche); | sprintf(messageBas,"droite
= %4d", niveauDroite); | } | | // on envoie le texte | lcd.clear(); | lcd.setCursor(0,
| lcd.print(messageHaut); | lcd.setCursor(0,1); | lcd.print(messageBas); |
} | | Code : Correction de l'exercice : programme complet
```

Et voilà! C'est la fin de ce tutoriel. Enfin... Pour le moment. ;)

N'hésitez pas à nous faire part de vos remarques et questions sur le forum [Systèmes et Matériels](#).

9 Internet of Things : Arduino sur Internet

9.1 Découverte de l'Ethernet sur Arduino

Pour accéder à internet et connecter notre Arduino au monde extérieur, nous allons utiliser le shield Ethernet. Ce dernier repose sur le protocole éponyme dont nous allons voir les fondements dans ce chapitre. Vous allez donc pouvoir découvrir comment les choses fonctionnent au niveau du réseau puis nous finirons par une présentation du shield Ethernet que nous allons utiliser.

Ce chapitre sera principalement composé de théorie et de culture générale, mais ça ne fait jamais de mal ! Je vous conseille de le lire pour que nous soyons bien sur la même longueur d'onde pour la suite. Les chapitres suivants permettront de rentrer un peu plus dans le vif du sujet avec la mise en pratique.

9.1.1 Un réseau informatique c'est quoi ?

Internet, qu'est-ce que c'est au juste ? On en entend parler tout le temps et vous vous en servez probablement tous les jours, mais seriez-vous capable de le définir pour autant ? Essayons d'y voir plus clair...

Internet est un réseau de réseaux. C'est un support de transfert d'informations dans un sens le plus large possible. Afin d'organiser ces échanges, on utilise des **protocoles** spécifiques pour chaque type de transfert. Ces protocoles sont ainsi adaptés à une tâche en particulier. Le plus connu est certainement *http*, soit *HyperText Transfert Protocol* qui sert à naviguer sur des pages en se promenant via des liens. Ceux d'entre vous qui font du développement web connaissent aussi sûrement le *ftp*, *File Transfert Protocol* (Protocole d'échanges de fichiers) qui permet de faire du transfert de fichiers uniquement.

Si l'on descend dans des considérations plus techniques, on peut s'intéresser au support de l'information, comment cette dernière voyage-t-elle. On découvrira alors les différents vecteurs de communication comme le WiFi ou l'Ethernet.

Chaque partie du réseau, que ce soit le medium de transfert de l'information ou le protocole final utilisé, peut être retrouvée dans ce que l'on appelle le modèle OSI (Open Systems Interconnection). Ce modèle possède 7 couches servant à définir le rôle de chacun des composants importants du réseau :

Modèle OSI			
	PDU	Couche	Fonction
Couches Hautes	Donnée	7. Application	Point d'accès aux services réseaux
		6. Présentation	Gère le chiffrement et le déchiffrement des données, convertit les données machine en données exploitables par n'importe quelle autre machine
		5. Session	Communication Interhost, gère les sessions entre les différentes applications
	Segments/Datagramme	4. Transport	Connexion bout à bout, connectabilité et contrôle de flux . Intervient la notion de port.
Couches Matérielles	Paquet	3. Réseau	Détermine le parcours des données et l'adressage logique (Adresse IP)
	Trame	2. Liaison	Adressage physique (Adresse MAC)
	Bit	1. Physique	Transmission des signaux sous forme binaire

Figure : Les couches OSI – (source [Wikipédia](#))

L'Ethernet, que nous allons voir dans ce cours agit sur les couches 1 et 2, physique et liaison de données puisqu'il transforme et transporte les données dans un "format" électronique qui lui est spécifique.

9.1.2 Le shield Ethernet

[[a]] | Toute la documentation officielle du shield Ethernet peut être trouvée ici : <http://arduino.cc/en/Main/ArduinoEthernetShield>

Maintenant que nous y voyons plus clair dans ce qu'est un réseau, voyons un peu les caractéristiques du matériel que nous allons utiliser pour relier notre carte Arduino.

Il faut tout d'abord savoir que l'Arduino seule n'est PAS DU TOUT faite pour utiliser une liaison réseau comme l'Ethernet. Nous venons de le découvrir, il y a de nombreuses couches à respecter, protocoles à utiliser et paquets à réaliser. Le pauvre petit microcontrôleur de l'Arduino en serait bien incapable dans de bons délais, ce qui rendrait l'utilisation de la carte impossible.

C'est pourquoi l'Arduino va être épaulée par un shield qui se nomme très justement "shield Ethernet". Il est relativement simple à trouver et coûte moins d'une trentaine d'euros, comme sur le site de Farnell. Ce shield permettra alors de décharger l'Arduino de tout le traitement des couches réseau pour ne donner que les informations utiles à cette dernière. Commençons l'autopsie...

9.1.2.1 Le cœur du traitement : Le Wiznet 510

Tout le traitement ou presque va être géré par la puce que vous pouvez voir sur le dessus de la carte. Ce circuit intégré possède la référence Wiznet 5100. C'est un composant qui est dédié au traitement par Ethernet. Une fois configuré, il se chargera de faire toute la communication. C'est-à-dire que vous n'aurez qu'à envoyer vos données au shield (via SPI, nous y reviendrons) et le shield se chargera de les encapsuler dans des trames et de les transmettre à l'adresse que vous souhaitez. De la même façon, si des données sont reçues il se chargera de les récupérer et les transmettre à l'Arduino pour que votre programme puisse les exploiter.

Ce composant possède une mémoire *buffer* de 16 KB. C'est-à-dire que dans le cas d'un échange de données (téléchargement d'une page web par exemple) les données seront stockées ici le temps que l'Arduino les lise et les traite.

Le shield possède aussi plusieurs leds reliées au Wiznet dont voici le nom et le rôle :

- PWR : indique que la carte est alimentée ;
- LINK : indique que la carte est connectée à un réseau. Cette led clignote lors de l'émission/réception de données ;

- FULLD : cette led est allumée dans le cas d’une connexion full-duplex (émission et réception simultanées) ;
- 100M : allumée si le réseau peut aller à 100 Mb/s (vitesse max. du composant), éteinte dans le cas d’un réseau à 10 Mb/s ;
- RX : clignote lors de la réception de données ;
- TX : clignote lors de l’envoi de données ;
- COLL : clignote si des collisions de données sont détectées.

9.1.2.2 De la communication à droite à gauche...

Le shield Ethernet, on s’en doute, communique principalement par ... Ethernet ! Mais ce n’est pas tout. Il lui faut aussi échanger avec l’Arduino pour pouvoir recevoir une configuration, savoir quelle page aller chercher ou encore transmettre des informations reçues. Pour toutes ces opérations, la transmission se fera par une liaison que nous ne connaissons pas encore : **SPI** (*Serial Protocol Interface*).

Cette transmission, nous allons découvrir comment l’utiliser via la librairie Ethernet dans cette partie. Nous n’allons cependant pas rentrer dans les détails puisque ce n’est pas le but de cette partie.

9.1.2.2.1 Carte SD Sur le shield vous avez sûrement vu l’emplacement pour la carte SD. Cette dernière servira à stocker/charger des pages ou des données quelconques. Cette carte se sert elle aussi de la connexion SPI que je viens d’évoquer. On devra donc utiliser une broche comme “Slave Select” (sélection de cible) pour déclarer au système à quel composant on s’adresse, soit l’Ethernet, soit la SD.

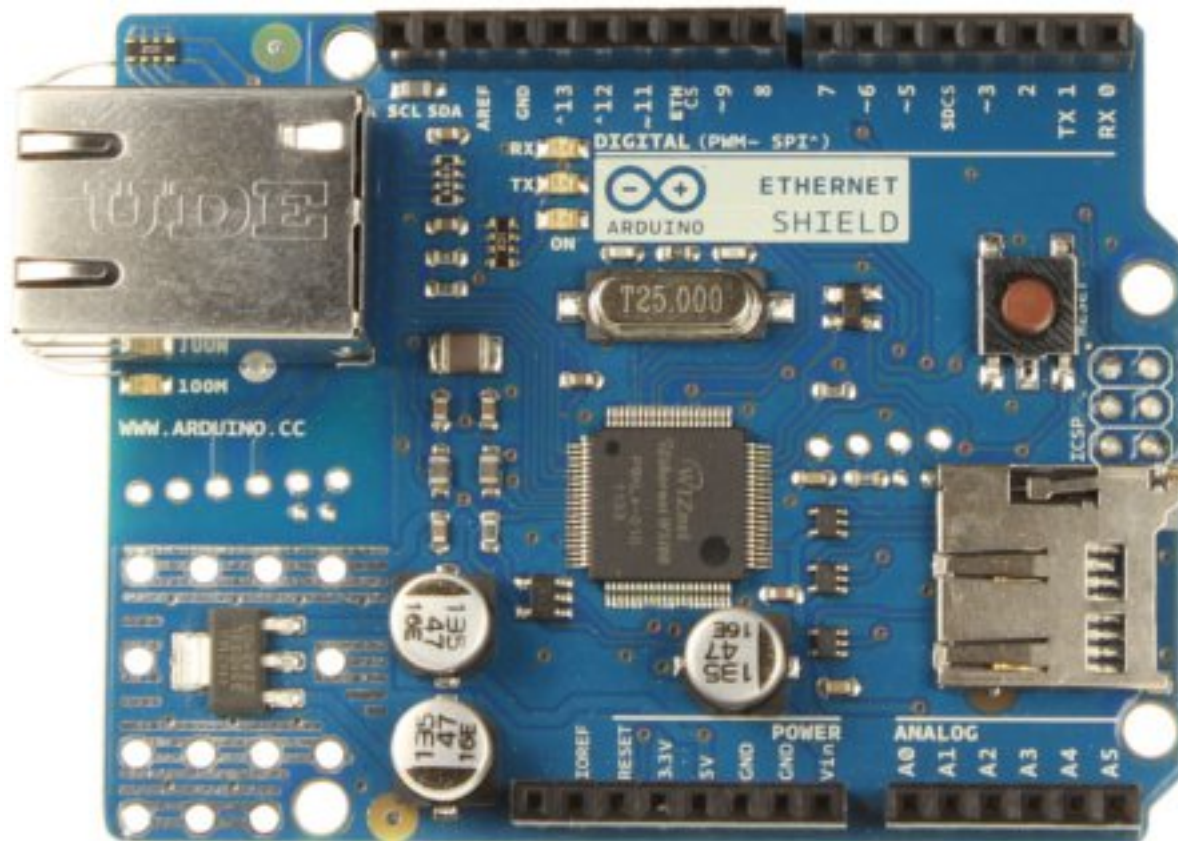


Figure : Le shield Ethernet Arduino - (CC-BY, arduino.cc)

9.1.3 Un peu de vocabulaire

9.1.3.1 Architecture Serveur/Client

Dans le monde magique d'internet, des ordinateurs se parlent entre eux. Cependant, on retrouve deux rôles particuliers : les clients et les serveurs. Bien que souvent exclusif, il peut cependant arriver qu'une machine serve aux deux. Mais voyons plus en détail ces rôles.

9.1.3.1.1 Le Serveur Le travail du serveur est de ... servir une information. Cette dernière peut être une page web ou un service quelconque. Par exemple, quand vous allez sur un site web, c'est un "serveur HTTP" (serveur web) qui sera chargé de vous répondre en vous renvoyant la bonne page. Autre exemple, dans votre établissement (scolaire ou professionnel), si une imprimante est branchée sur le réseau alors c'est un serveur d'impression qui sera utilisé pour convertir votre demande "imprime-moi la page 42 du pdf Arduino d'Eskimon" en une tâche dans le monde réel.

Vous vous en doutez sûrement, des serveurs il en existe pour plein de choses différentes! Web, e-mail, impression... Quand nous utiliserons notre Arduino pour *offrir/servir* une donnée, on dira alors qu'elle est en fonctionnement de "serveur". On aura donc un "Serveur Arduino" en quelque sorte (qui pourra être un simple serveur web par exemple).

9.1.3.1.2 Le Client Si vous avez compris les paragraphes précédents, alors vous avez sûrement deviné le rôle du client. Comme dans la vie réelle, le client est celui qui va demander une

information ou un service. Dans la vie de tous les jours, vous utilisez sûrement au quotidien un navigateur internet. Eh bien ce dernier est un client pour un serveur web. Il fera alors des demandes au serveur, ce dernier les renverra et le client les interprétera.

Il existe autant de types de clients qu'il y a de types de serveurs. À chaque service, sa tâche !

9.1.3.2 Des adresses pour tout !

Dans ce tutoriel, nous allons rencontrer deux types d'**adresses**, l'IP et la MAC. Voyons de quoi il s'agit.

9.1.3.2.1 IP L'adresse IP, Internet Protocol, représente l'adresse à l'échelle du réseau, qu'il soit global ou local. Dans le monde physique, vous possédez une adresse pour recevoir du courrier. Dans le monde de l'internet, c'est pareil. Quand des paquets de données sont échangés entre serveur et client, ces derniers possèdent une adresse pour savoir où les délivrer.

À l'échelle globale, votre connexion possède une adresse IP dite *publique*. C'est cette dernière qui sert à échanger avec le monde extérieur. En revanche, quand vous êtes connecté à votre box internet ou un routeur quelconque, vous possédez alors une adresse IP *locale*. Cette dernière est très souvent de la forme 192.168.0.xxx. C'est votre box/routeur qui servira d'aiguillage entre les paquets qui sont destinés à votre ordinateur ou votre Arduino (ou n'importe quel autre équipement sur le même réseau).

9.1.3.2.2 MAC L'adresse MAC, Media Access Control, représente une adresse physique, matérielle, de votre matériel. Elle est propre à votre ordinateur/carte réseau. Elle est normalement unique à n'importe quel matériel réseau. Mais elle est aussi falsifiable, donc l'unicité ne peut être garantie globalement.

9.1.3.3 D'autres notions utiles

9.1.3.3.1 Ports Comme on le voyait un peu plus tôt, différentes applications serveur peuvent fonctionner sur une même machine. MAIS, ils sont tous cachés derrière la même adresse IP. Comment donc séparer les paquets qui vont au serveur web de ceux qui vont au serveur e-mail par exemple ? Une solution pourrait être d'ajouter une information dans le paquet pour préciser l'application de destination. Et bien c'est presque ça. En effet, chaque paquet se verra attribuer un *port* de destination qui est celui sur lequel l'application fonctionne.

Pour imaginer, essayez d'imaginer la liaison Internet comme une autoroute avec de très nombreuses voies. Les paquets sont les voitures circulant sur cette autoroute. Chaque port sera alors une voie dédiée. Ainsi, le serveur web serait la voie de droite, le serveur mail celle du milieu et le serveur Arduino la voie de gauche. Et comme ça tout le monde arrive à bon port, le serveur au bout de la voie a juste à s'occuper de ses paquets et non pas ceux des autres applications !

Maintenant que les bases sont posées, nous allons pouvoir partir à l'aventure et la conquête du monde par l'Internet ! Place à la grande mode de "L'Internet des objets" !

9.2 Arduino et Ethernet : client

Commençons doucement à découvrir ce shield en utilisant notre Arduino comme **client**. Dans ce chapitre, nous allons découvrir comment faire en sorte que notre Arduino aille interroger un site internet pour obtenir une information de sa part.

[[information]] | Ici nous allons interroger un site internet, mais cela aurait très bien pu être un autre service, sur un autre port (vous allez comprendre)

Afin de bien comprendre, je vous propose tout d'abord quelques explications sur le protocole HTTP et comment se passe un échange de données (requête + réponse).

Ensuite, nous verrons comment mettre en œuvre le shield et faire en sorte qu'il puisse accéder au monde extérieur.

Enfin, nous mettrons tout en œuvre pour faire une simple requête sur un moteur de recherche.

Pour terminer, je vous propose un petit exercice qui permettra de voir un cas d'utilisation : le "monitoring" d'un serveur.

9.2.1 Client et requêtes HTTP

Faisons un petit retour étendu sur la notion de client et surtout découvrons plus en détail en quoi consiste exactement une requête HTTP.

9.2.1.1 Un client, ça fait quoi ?

Le rôle du client est finalement assez simple. Son but sera d'aller chercher des informations pour les afficher à un utilisateur ou effectuer une action particulière. D'une manière générale, le client sera celui qui traite l'information que le serveur envoie.

Prenons l'exemple du navigateur web. C'est un *client*. Lorsque vous l'utilisez, vous allez générer des *requêtes* vers des serveurs et ces derniers vont ensuite renvoyer un tas d'informations (la page web). Le navigateur va alors les traiter pour vous les afficher sous une forme agréable et prévue par le développeur web.

Finalement, c'est simple d'être client, ça consiste juste à demander des choses et attendre qu'elles arrivent :)!

Les termes "client" et "serveur" (et même "requête") sont d'ailleurs très bien choisis car ils illustrent très bien les mêmes rôles que leur équivalent "dans la vraie vie".

9.2.1.2 Une requête HTTP, c'est quoi ?

Des requêtes HTTP il en existe de plusieurs types. Les plus classiques sont sûrement les GET et les POST, mais on retrouve aussi les PUT, DELETE, HEAD... Pour faire simple, nous allons uniquement nous intéresser à la première car c'est celle qui est utilisée la plupart du temps!

9.2.1.2.1 Émission Dans la spécification du protocole HTTP, on apprend que GET nous permet de demander une ressource en lecture seule. On aura simplement besoin de spécifier une page cible et ensuite le serveur HTTP de cette page nous renverra la ressource ou un code d'erreur en

cas de problème. On peut passer des arguments/options à la fin de l'adresse que l'on interroge pour demander des choses plus particulières au serveur.

Par exemple, si vous êtes sur la page d'accueil de Google et que vous faites une recherche sur "Arduino", votre navigateur fera la requête suivante : `GET /search?q=arduino HTTP/1.0`. Il interroge donc la page principale "/" (racine) et envoie l'argument "search?q=arduino". Le reste définit le protocole utilisé.

9.2.1.2.2 Réception Une fois la requête faite, le serveur interrogé va vous renvoyer une réponse. Cette réponse peut être découpée en deux choses : l'en-tête (header) et le contenu. On pourrait comparer cela à un envoi de colis. L'en-tête posséderait les informations sur le colis (destinataires, etc.) et le contenu est ce qui est à l'intérieur du colis.

Typiquement, dans une réponse minimaliste, on lira les informations suivantes :

- Le code de réponse de la requête (200 si tout s'est bien passé) ;
- Le type MIME du contenu renvoyé (`text/html` dans le cas d'une page web) ;
- Une ligne blanche/vide ;
- Le contenu.

Les deux premières lignes font partie du header, puis après viendra le contenu.

Si l'on veut chercher des informations, en général on le fera dans le contenu.

9.2.2 Utilisation du shield comme client

Maintenant que l'on en sait un peu plus, on va pouvoir faire des requêtes et aller interroger le monde...

9.2.2.1 Préparation minimale

Pour commencer, il va falloir configurer notre module Ethernet pour qu'il puisse travailler correctement. Pour cela, il va falloir lui donner une adresse MAC et une adresse IP (qui peut être automatique, nous y reviendrons). On va utiliser 4 variables différentes :

- Un tableau de byte (ou char) pour les 6 octets de l'adresse MAC ;
- Un objet `IPAddress` avec l'adresse IP que l'on assigne au module ;
- Un objet `EthernetClient` qui nous servira à faire la communication ;
- Une chaîne de caractères représentant le nom du serveur à interroger.

[[i]] L'adresse MAC doit normalement être écrite sur un autocollant au dos de votre shield (sinon inventez-en une !)

De manière programmatrice, on aura les variables ci-dessous.

```
// Ces deux bibliothèques sont indispensables pour le shield
####include <SPI.h>
#include <Ethernet.h>

// L'adresse MAC du shield
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xA5, 0x7E };
// L'adresse IP que prendra le shield
```

```
IPAddress ip(192,168,0,143);  
// L'objet qui nous servira à la communication  
EthernetClient client;  
// Le serveur à interroger  
char serveur[] = "leserveur.com"
```

9.2.2.2 Démarrer le shield

Maintenant que nous avons nos variables, nous allons pouvoir démarrer notre shield dans le `setup()`. Pour cela, il suffira d'appeler une fonction bien nommée : `begin()`, eh oui, comme pour la voie série ! Cette fonction prendra deux paramètres, l'adresse MAC et l'adresse IP à utiliser. C'est aussi simple que cela !

```
// Ces deux bibliothèques sont indispensables pour le shield  
#####include <SPI.h>  
#include <Ethernet.h>  
  
// L'adresse MAC du shield  
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xA5, 0x7E };  
// L'adresse IP que prendra le shield  
IPAddress ip(192,168,0,143);  
// L'objet qui nous servira à la communication  
EthernetClient client;  
// Le serveur à interroger  
char serveur[] = "perdu.com"  
  
void setup() {  
  // On démarre la voie série pour déboguer  
  Serial.begin(9600);  
  
  // On démarre le shield Ethernet  
  Ethernet.begin(mac, ip);  
  // Donne une seconde au shield pour s'initialiser  
  delay(1000);  
}
```

Simple non ?

9.2.2.2.1 le DHCP Si vous bricolez à domicile, il est fort probable que vous soyez en train d'utiliser un routeur ou votre box internet. Bien souvent, ces derniers ont par défaut la fonction DHCP activée. Cette technologie permet de donner une adresse IP automatiquement à tous les équipements qui se connectent au réseau. Ainsi, plus besoin de le faire manuellement !

Du coup, on peut faire évoluer notre script d'initialisation pour prendre en compte cela.

```
// Ces deux bibliothèques sont indispensables pour le shield  
#####include <SPI.h>  
#include <Ethernet.h>
```

```

// L'adresse MAC du shield
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xA5, 0x7E };
// L'adresse IP que prendra le shield
IPAddress ip(192,168,0,143);
// L'objet qui nous servira a la communication
EthernetClient client;
// Le serveur à interroger
char serveur[] = "perdu.com";

void setup() {
  // On démarre la voie série pour déboguer
  Serial.begin(9600);

  char erreur = 0;
  // On démarre le shield Ethernet SANS adresse IP (donc donnée via DHCP)
  erreur = Ethernet.begin(mac);

  if (erreur == 0) {
    Serial.println("Parametrage avec ip fixe...");
    // si une erreur a eu lieu cela signifie que l'attribution DHCP
    // ne fonctionne pas. On initialise donc en forçant une IP
    Ethernet.begin(mac, ip);
  }
  Serial.println("Init...");
  // Donne une seconde au shield pour s'initialiser
  delay(1000);
  Serial.println("Pret!");
}

```

9.2.2.3 Envoyer une requête

Une fois que le shield est prêt, nous allons pouvoir commencer à l'utiliser ! Accrochez-vous, les choses amusantes commencent !

9.2.2.3.1 Requête simple Pour débiter, il va falloir générer une requête pour interroger un serveur dans l'espoir d'obtenir une réponse. Je vous propose de commencer par une chose simple, récupérer une page web très sommaire, celle de <http://perdu.com> !

C'est là que notre variable "client" de type `EthernetClient` entre enfin en jeu. C'est cette dernière qui va s'occuper des interactions avec la page. Nous allons utiliser sa méthode `connect()` pour aller nous connecter à un site puis ensuite nous ferons appel à sa méthode `println()` pour construire notre requête et l'envoyer.

```

void setup() {
  // ... Initialisation précédente ...

  // On connecte notre Arduino sur "perdu.com" et le port 80 (default pour l'http)
  erreur = client.connect(serveur, 80);

```

```
if(erreur == 1) {
  // Pas d'erreur? on continue!
  Serial.println("Connexion OK, envoi en cours...");

  // On construit l'en-tête de la requête
  client.println("GET / HTTP/1.1");
  client.println("Host: perdu.com");
  client.println("Connection: close");
  client.println();
} else {
  // La connexion a échoué :(
  Serial.println("Echec de la connexion");
  switch(erreur) {
    case(-1):
      Serial.println("Time out");
      break;
    case(-2):
      Serial.println("Serveur invalide");
      break;
    case(-3):
      Serial.println("Tronque");
      break;
    case(-4):
      Serial.println("Reponse invalide");
      break;
  }
  while(1); // Problème = on bloque
}
}
```

Étudions un peu ces quelques lignes.

Tout d’abord, on va essayer de connecter notre client au serveur de “perdu.com” sur son port 80 qui est le port par défaut du protocole HTTP :

```
client.connect("perdu.com", 80);
```

Ensuite, une fois que la connexion semble correcte, on va construire notre requête pas à pas. Tout d’abord, on explique vouloir faire un GET sur la racine (“/”) du site et le tout sous le protocole HTTP version 1.1.

```
client.println("GET / HTTP/1.1");
```

Ensuite, on redéclare le site qui devra faire (héberger, “host” en anglais) la réponse. En l’occurrence on reste sur perdu.com.

```
client.println("Host: perdu.com");
```

Puis, on signale au serveur que la connexion sera fermée lorsque les données sont transférées.

```
client.println("Connection: close");
```

Enfin, pour prévenir que l'on vient de finir d'écrire notre en-tête (*header*), on envoie une ligne blanche.

```
client.println();
```

J'ai ensuite rajouté un traitement des erreurs pour savoir ce qui se passe en cas de problème.

9.2.2.3.2 Requête avec paramètre Et si l'on voulait passer des informations complémentaires à la page ? Eh bien c'est simple, il suffira juste de modifier la requête GET en lui rajoutant les informations ! Par exemple, admettons que sur perdu.com il y ait une page de recherche "recherche.php" qui prenne un paramètre "m" comme mot-clé de recherche. On aurait alors :

```
client.println("GET /recherche.php?m=monmot HTTP/1.1");
```

Ce serait équivalent alors à aller demander la page "perdu.com/recherche.php?m=monmot", soit la page "recherche.php" avec comme argument de recherche "m" le mot "monmot".

9.2.2.4 Lire une réponse

Lorsque la requête est envoyée (après le saut de ligne), le serveur va nous répondre en nous renvoyant la ressource demandée. En l'occurrence se sera la page d'accueil de perdu.com. Eh bien vous savez quoi ? On fera exactement comme avec une voie série. On commencera par regarder si des données sont arrivées, et si c'est le cas, on les lira une à une pour en faire ensuite ce que l'on veut (recomposer des lignes, chercher des choses dedans...)

Dans l'immédiat, contentons-nous de tout afficher sur la voie série !

Première étape, vérifier que nous avons bien reçu quelque chose. Pour cela, comme avec Serial, on va utiliser la méthode `available()` qui nous retourne le nombre de caractères disponibles à la lecture.

```
client.available()
```

Si des choses sont disponibles, on les lit une par une (comment ? Avec `read()` bien sûr ! :D) et les envoie en même temps à la voie série :

```
char carlu = 0;
// on lit les caractères s'il y en a de disponibles
if(client.available()) {
    carlu = client.read();
    Serial.print(carlu);
}
}
```

Enfin, quand tout a été lu, on va vérifier l'état de notre connexion et fermer notre client si la connexion est terminée.

```
// Si on est bien déconnecté.
if (!client.connected()) {
    Serial.println();
    Serial.println("Deconnexion!");
    // On ferme le client
    client.stop();
    while(1); // On ne fait plus rien
}
```

Globalement, voici le code pour lire une réponse :

```
void loop()
{
  char carlu = 0;
  // on lit les caractères s'il y en a de disponibles
  if(client.available()) {
    carlu = client.read();
    Serial.print(carlu);
  }

  // Si on est bien déconnecté.
  if (!client.connected()) {
    Serial.println();
    Serial.println("Deconnexion!");
    // On ferme le client
    client.stop();
    while(1); // On ne fait plus rien
  }
}
```

[[i]] | Avez-vous remarqué, plutôt que de lire tous les caractères avec un `while`, on n'en lira qu'un seul à la fois par tour dans `loop()`. C'est un choix de design, avec un `while` cela marcherait aussi!

9.2.2.4.1 Code complet

En résumé, voici le code complet de la lecture de la page "perdu.com"

```
// Ces deux bibliothèques sont indispensables pour le shield
####include <SPI.h>
#include <Ethernet.h>

// L'adresse MAC du shield
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xA5, 0x7E };
// L'adresse IP que prendra le shield
IPAddress ip(192,168,0,143);
// L'objet qui nous servira à la communication
EthernetClient client;
// Le serveur à interroger
char serveur[] = "perdu.com";

void setup() {
  // On démarre la voie série pour déboguer
  Serial.begin(9600);

  char erreur = 0;
  // On démarre le shield Ethernet SANS adresse IP (donc donnée via DHCP)
  erreur = Ethernet.begin(mac);

  if (erreur == 0) {
    Serial.println("Parametrage avec ip fixe...");
  }
}
```



```

    // si une erreur a eu lieu cela signifie que l'attribution DHCP
    // ne fonctionne pas. On initialise donc en forçant une IP
    Ethernet.begin(mac, ip);
}
Serial.println("Init...");
// Donne une seconde au shield pour s'initialiser
delay(1000);
Serial.println("Pret!");

// On connecte notre Arduino sur "perdu.com" et le port 80 (default pour l'http)
erreur = client.connect(serveur, 80);

if(erreur == 1) {
    // Pas d'erreur? on continue!
    Serial.println("Connexion OK, envoi en cours...");

    // On construit l'en-tête de la requête
    client.println("GET / HTTP/1.1");
    client.println("Host: perdu.com");
    client.println("Connection: close");
    client.println();
} else {
    // La connexion a échoué :(
    Serial.println("Echec de la connexion");
    switch(erreur) {
        case(-1):
            Serial.println("Time out");
            break;
        case(-2):
            Serial.println("Serveur invalide");
            break;
        case(-3):
            Serial.println("Tronque");
            break;
        case(-4):
            Serial.println("Reponse invalide");
            break;
    }
    while(1); // On bloque la suite
}
}

void loop()
{
    char carlu = 0;
    // on lit les caractères s'il y en a de disponibles
    if(client.available()) {
        carlu = client.read();
    }
}

```

```

    Serial.print(carlu);
}

// Si on est bien déconnecté.
if (!client.connected()) {
    Serial.println();
    Serial.println("Deconnexion!");
    // On ferme le client
    client.stop();
    while(1); // On ne fait plus rien
}
}

```

Et voici le résultat que vous devez obtenir dans votre terminal série. Remarquez la présence du header de la réponse que l'on reçoit.

```

HTTP/1.1 200 OK
Date: Thu, 26 Mar 2015 16:14:15 GMT
Server: Apache
Last-Modified: Tue, 02 Mar 2010 18:52:21 GMT
ETag: "cc-480d5dd98a340"
Accept-Ranges: bytes
Content-Length: 204
Vary: Accept-Encoding
Connection: close
Content-Type: text/html

```

```
<html><head><title>Vous Etes Perdu?</title></head><body><h1>Perdu sur l'Internet?</h1>
```

9.2.2.5 Faire des requêtes en continu

Faire une requête en début de programme c'est bien, mais ce serait mieux de pouvoir la faire quand on veut. Faire évoluer notre programme ne va pourtant pas être si simple...

Pour commencer, on va ajouter quelques nouvelles variables. La première servira à se souvenir de quand date (via `millis()`) la dernière requête faite. Ce sera donc un `long()`. La seconde sera une constante servant à indiquer le temps minimal devant s'écouler entre deux requêtes. Enfin, la dernière variable sera un booléen servant de *flag* pour indiquer l'état de la connexion (ouverte ou fermée) entre deux tours de boucle `loop()`.

```

// moment de la dernière requête
long derniereRequete = 0;
// temps minimum entre deux requêtes
const long updateInterval = 10000;
// mémorise l'état de la connexion entre deux tours de loop
bool etaitConnecte = false;

```

Ensuite, on va créer une fonction sobrement nommée `requete()` qui se chargera de construire et envoyer la requête. Aucune nouveauté là-dedans, c'est le code que vous connaissez déjà.

```

void requete() {
  // On connecte notre Arduino sur "perdu.com" et le port 80 (default pour l'http)
  char erreur = client.connect(serveur, 80);

  if(erreur == 1) {
    // Pas d'erreur? on continue!
    Serial.println("Connexion OK, envoi en cours...");

    // On construit l'en-tête de la requête
    client.println("GET / HTTP/1.1");
    client.println("Host: perdu.com");
    client.println("Connection: close");
    client.println();

    // On enregistre le moment d'envoi de la dernière requête
    derniereRequete = millis();
  } else {
    // La connexion a échoué :(
    // On ferme notre client
    client.stop();
    // On avertit l'utilisateur
    Serial.println("Echec de la connexion");
    switch(erreur) {
      case(-1):
        Serial.println("Time out");
        break;
      case(-2):
        Serial.println("Serveur invalide");
        break;
      case(-3):
        Serial.println("Tronque");
        break;
      case(-4):
        Serial.println("Reponse invalide");
        break;
    }
  }
}

```

La seule différence ici est que l'on ferme le client si la connexion a un problème et aussi on enregistre le moment auquel a été faite la requête.

Bien. Maintenant, il faut revoir notre loop.

Le début ne change pas, on récupère/affiche les caractères s'ils sont disponibles.

Ensuite, on avait l'étape de fermeture du client si la connexion est fermée. Dans notre cas actuel, on ne doit pas l'arrêter n'importe quand, car sinon une connexion pourrait de nouveau avoir lieu sans que celle-là soit finie. On va donc la fermer QUE si elle était déjà fermée à la toute fin du tour précédent. Voici comme cela peut-être représenté :

```
void loop()
{
  // traite les caractères
  // ...

  // SI on était connecté au tour précédent
  // ET que maintenant on est plus connecté
  // ALORS on ferme la connexion
  if (etaitConnecte &&!client.connected()) {
    Serial.println();
    Serial.println("Deconnexion!");
    // On ferme le client
    client.stop();
  }

  // ...
  // bricole, traite, calcul, manigance, complote...
  // ...

  // enregistre l'état de la connexion (ouvert ou fermé)
  etaitConnecte = client.connected();
}
```

Maintenant, il ne nous reste plus qu'à redéclencher une requête si nos dix secondes se sont écoulées et que la précédente connexion a fini de travailler :

```
// Si on est déconnecté
// et que cela fait plus de xx secondes qu'on a pas fait de requête
if(!client.connected() && ((millis() - derniereRequete) > updateInterval)) {
  requete();
}
```

C'est tout clair? Ci-dessous le code complet pour vous aider à y voir un peu mieux dans l'ensemble;).

9.2.2.5.1 Le code complet

```
// Ces deux bibliothèques sont indispensables pour le shield
####include <SPI.h>
#include <Ethernet.h>

// L'adresse MAC du shield
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xA5, 0x7E };
// L'adresse IP que prendra le shield
IPAddress ip(192,168,0,143);
// L'objet qui nous servira a la communication
EthernetClient client;
// Le serveur à interroger
char serveur[] = "perdu.com";
```

```

// pour lire les caractères
char carlu = 0;
// moment de la dernière requête
long derniereRequete = 0;
// temps minimum entre deux requêtes
const long updateInterval = 10000;
// mémorise l'état de la connexion entre deux tours de loop
bool etaitConnecte = false;

void setup() {
  // On démarre la voie série pour déboguer
  Serial.begin(9600);

  char erreur = 0;
  // On démarre le shield Ethernet SANS adresse IP (donc donnée via DHCP)
  erreur = Ethernet.begin(mac);

  if (erreur == 0) {
    Serial.println("Parametrage avec ip fixe...");
    // si une erreur a eu lieu cela signifie que l'attribution DHCP
    // ne fonctionne pas. On initialise donc en forçant une IP
    Ethernet.begin(mac, ip);
  }
  Serial.println("Init...");
  // Donne une seconde au shield pour s'initialiser
  delay(1000);
  Serial.println("Pret!");
}

void loop()
{
  // on lit les caractères s'il y en a de disponibles
  if(client.available()) {
    carlu = client.read();
    Serial.print(carlu);
  }

  // SI on était connecté au tour précédent
  // ET que maintenant on est plus connecté
  // ALORS on ferme la connexion
  if (etaitConnecte &&!client.connected()) {
    Serial.println();
    Serial.println("Deconnexion!");
    // On ferme le client
    client.stop();
  }

  // Si on est déconnecté

```

```

// et que cela fait plus de xx secondes qu'on a pas fait de requête
if(!client.connected() && ((millis() - derniereRequete) > updateInterval)) {
  requete();
}

// enregistre l'état de la connexion (ouvert ou fermé)
etaitConnecte = client.connected();
}

void requete() {
  // On connecte notre Arduino sur "perdu.com" et le port 80 (default pour l'http)
  char erreur = client.connect(serveur, 80);

  if(erreur == 1) {
    // Pas d'erreur? on continue!
    Serial.println("Connexion OK, envoi en cours...");

    // On construit l'en-tête de la requête
    client.println("GET / HTTP/1.1");
    client.println("Host: perdu.com");
    client.println("Connection: close");
    client.println();

    // On enregistre le moment d'envoi de la dernière requête
    derniereRequete = millis();
  } else {
    // La connexion a échoué :(
    // On ferme notre client
    client.stop();
    // On avertit l'utilisateur
    Serial.println("Echec de la connexion");
    switch(erreur) {
      case(-1):
        Serial.println("Time out");
        break;
      case(-2):
        Serial.println("Serveur invalide");
        break;
      case(-3):
        Serial.println("Tronque");
        break;
      case(-4):
        Serial.println("Reponse invalide");
        break;
    }
  }
}
}

```

9.2.2.6 L'intérêt d'un client

En lisant tout ça, vous vous dites peut-être qu'utiliser le shield Ethernet en mode client est un peu inutile. C'est vrai, après tout on ne peut même pas afficher les pages reçues !

Cependant, avec un peu d'imagination on pourrait facilement voir des utilisations plus que pratiques !

9.2.2.6.1 Télécharger une configuration La première idée par exemple pourrait être de mettre à disposition des fichiers de paramètres à l'Arduino. Imaginons par exemple que je fais une application qui dépend des saisons. Mon application pourrait utiliser des moteurs et des capteurs pour faire certaines actions, mais ces dernières pourraient être différentes en fonction du moment de l'année. Plutôt que de stocker 4 ensembles de paramètres dans l'Arduino, cette dernière pourrait vérifier régulièrement en ligne (en interrogeant un petit script que nous aurions fait) pour savoir si quelque chose doit changer. Et voilà, configuration à distance !

9.2.2.6.2 Enregistrer des données en ligne J'ai souvent lu sur des forums que des gens aimeraient pouvoir sauvegarder des choses dans une base de données. Bien entendu, l'Arduino seule ne peut pas le faire, gérer une BDD est bien trop compliqué pour elle. En revanche, elle pourrait facilement interroger des pages en insérant des paramètres dans sa requête. La page qui reçoit alors la requête lirait ce paramètre et s'occuperait de sauvegarder les infos. Par exemple, on pourrait imaginer la requête `http://mapageweb.com/enregistrer.php?&analog1=123&analog2=28&millis=` Cette requête possède deux paramètres, **analog1** et **analog2** qui pourraient être les valeurs des entrées analogiques et un dernier "millis" qui pourrait être la valeur de `millis()` d'Arduino. Notre page "enregistrer.php" ferait alors la sauvegarde dans sa base de données ! Pour construire une telle requête, le code suivant devrait faire l'affaire :

```
// On construit l'en-tête de la requête
client.print("GET /enregistrer.php/?analog1="); //attention, pas de saut de ligne!
client.print(analogRead(A1));
client.print("&analog2=");
client.print(analogRead(A2));
client.print("&millis=");
client.print(millis());
// On finit par le protocole
client.println(" HTTP/1.1"); //ce coup-ci, saut de ligne pour valider!
// on aurait alors :
// "GET /enregistrer.php/?analog1=<valeur-de-A1>&analog2=<valeur-de-A2>&millis=<valeur
client.println("Host: mapageweb.com");
client.println("Connection: close");
client.println();
```

9.2.2.6.3 Sûrement plein d'autres choses ! Il existe sûrement un paquet d'autres idées auquel je n'ai pas pensé !

9.2.3 Exercice, lire l'uptime de Eskimon.fr

Pour finir ce chapitre je vous propose un petit exercice, allez lire l'uptime (temps écoulé depuis la mise en route du système) de mon blog, eskimon.fr.

9.2.3.1 Consigne

Pour cela, je vous ai concocté une petite page juste pour vous qui renvoie uniquement cette information, sans tout le bazar HTTP qui va avec une page classique. Vous obtiendrez ainsi simplement le header de la requête et la valeur brute de l'uptime.

La page à interroger pour votre requête est : `http://eskimon.fr/public/arduino.php`

Essayer de modifier votre code pour afficher cette information ;)

Vous êtes maintenant en mesure de vous balader sur Internet pour aller y chercher des informations. Bienvenue dans l'IoT, l'Internet of Things !

9.3 Arduino et Ethernet : serveur

Dans ce chapitre, l'Arduino sera maintenant responsable de l'envoi des données vers le monde extérieur. On dit qu'elle agit en serveur. Ce sera donc un outil externe (logiciel particulier, navigateur etc) qui viendra l'interroger et à ce moment-là, elle renverra les informations demandées. On pourra aussi, via un site externe, lui envoyer des ordres pour faire des actions.

9.3.1 Préparer l'Arduino

L'utilisation de l'Arduino en mode serveur est sûrement plus courante que celle en client. Cette partie devrait donc particulièrement vous intéresser. Deux grands rôles peuvent être accomplis :

- L'envoi de données *à la demande* (l'utilisateur vient demander les données quand il les veut) ;
- La réception d'ordre pour effectuer des actions.

Ces deux rôles ne sont pas exclusifs, ils peuvent tout à fait cohabiter. Mais dans un premier temps, reparlons un peu de ce qu'est un serveur.

Nous l'avons vu dans le premier chapitre, un serveur est chargé de réceptionner du trafic, l'interpréter puis agir en conséquence. Pour cela, il possède un **port** particulier qui lui est dédié. Chaque octet arrivant sur ce port lui est donc destiné. On dit que le serveur **écoute** sur un **port**.

C'est donc à partir de cela que nous allons pouvoir mettre en place notre serveur !

Comme pour le client, il va falloir commencer par les options du shield (MAC, IP...) afin que ce dernier puisse se connecter à votre box/routeur. On retrouve donc un setup similaire au chapitre précédent :


```

// Ces deux bibliothèques sont indispensables pour le shield
####include <SPI.h>
#include <Ethernet.h>

// L'adresse MAC du shield
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xA5, 0x7E };
// L'adresse IP que prendra le shield
IPAddress ip(192,168,0,143);

void setup()
{
  // On démarre la voie série pour déboguer
  Serial.begin(9600);

  char erreur = 0;
  // On démarre le shield Ethernet SANS adresse IP (donc donnée via DHCP)
  erreur = Ethernet.begin(mac);

  if (erreur == 0) {
    Serial.println("Parametrage avec ip fixe...");
    // si une erreur a eu lieu cela signifie que l'attribution DHCP
    // ne fonctionne pas. On initialise donc en forçant une IP
    Ethernet.begin(mac, ip);
  }
  Serial.println("Init...");
  // Donne une seconde au shield pour s'initialiser
  delay(1000);
  Serial.print("Pret!");
}

```

Bien. Au chapitre précédent cependant nous avons des variables concernant le *client*. Ici, de manière similaire nous aurons donc des variables concernant le **serveur**. La première et unique nouvelle chose sera donc une variable de type `EthernetServer` qui prendra un paramètre : le port d'écoute. J'ai choisi 4200 de manière un peu aléatoire, car je sais qu'il n'est pas utilisé sur mon réseau. Une liste des ports les plus souvent utilisés peut être [trouvée sur Wikipédia](#).

```

// Initialise notre serveur
// Ce dernier écoutera sur le port 4200
EthernetServer serveur(4200);

```

Puis, à la fin de notre setup, il faudra démarrer le serveur avec la commande suivante :

```

serveur.begin();

```

En résumé, on aura donc le code suivant pour l'initialisation :

```

// Ces deux bibliothèques sont indispensables pour le shield
####include <SPI.h>
#include <Ethernet.h>

// L'adresse MAC du shield

```

```
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xA5, 0x7E };
// L'adresse IP que prendra le shield
IPAddress ip(192,168,0,143);

// Initialise notre serveur
// Ce dernier écoutera sur le port 4200
EthernetServer serveur(4200);

void setup()
{
  // On démarre la voie série pour déboguer
  Serial.begin(9600);

  char erreur = 0;
  // On démarre le shield Ethernet SANS adresse IP (donc donnée via DHCP)
  erreur = Ethernet.begin(mac);

  if (erreur == 0) {
    Serial.println("Parametrage avec ip fixe...");
    // si une erreur a eu lieu cela signifie que l'attribution DHCP
    // ne fonctionne pas. On initialise donc en forçant une IP
    Ethernet.begin(mac, ip);
  }
  Serial.println("Init...");
  // Donne une seconde au shield pour s'initialiser
  delay(1000);
  // On lance le serveur
  serveur.begin();
  Serial.print("Pret!");
}
```

Et voilà, votre serveur Arduino est en train de surveiller ce qui se passe sur le réseau!

9.3.2 Répondre et servir des données

Maintenant que le serveur est prêt et attend qu'on lui parle, on va pouvoir coder la partie "communication avec le demandeur". La première étape va être de vérifier si un client attend une réponse de la part de notre serveur. On va donc retrouver notre objet `EthernetClient` vu au chapitre précédent et une fonction du serveur que l'on aurait presque pu deviner : `available()`

```
// Regarde si un client est connecté et attend une réponse
EthernetClient client = serveur.available();
```

Ensuite les choix sont simples. Soit un client (donc une application externe) est connecté avec l'Arduino et veut interagir, soit il n'y a personne et donc on ne fait... rien (ou autre chose). Pour cela, on va simplement regarder si `client` vaut autre chose que zéro. Si c'est le cas, alors on traite les données.

```
if (client) {
  // Quelqu'un est connecté!
```

```
}

```

Maintenant, on va faire au plus simple. On va considérer que l'on renvoie toujours les mêmes informations : la valeur de l'entrée analogique 0 et la variable `millis()`, quelle que soit la requête du client. On va alors se retrouver dans le cas similaire au chapitre précédent ou il faudra simplement construire une requête pour renvoyer des données. Comme j'aime les choses simples, j'ai décidé de ne pas renvoyer une page web (trop verbeux), mais juste du texte au **format JSON** qui est simple à lire et à fabriquer.

[[i]] | Le HTML demande **BEAUCOUP** trop de contenu texte à envoyer pour afficher une information utile. Soyons clairs, un système embarqué comme Arduino n'est **pas fait pour afficher des pages web**, il faut pouvoir renvoyer des informations de manière **simple et concise**.

Il faudra comme pour le client commencer par renvoyer un header. Le nôtre sera simple et possèdera seulement deux informations : le protocole utilisé avec le code de retour (encore http 1.1 et 200 pour dire que tout s'est bien passé) ainsi que le type mime du contenu renvoyé (en l'occurrence "application/json"). Si vous renvoyez de l'html, ce serait "text/html" par exemple.

```
// Tout d'abord le code de réponse 200 = réussite
client.println("HTTP/1.1 200 OK");
// Puis le type mime du contenu renvoyé, du json
client.println("Content-Type: application/json");
// Et c'est tout!
// On envoie une ligne vide pour signaler la fin du header
client.println();

```

Une fois le header envoyé, on construit et envoie notre réponse json. C'est assez simple, il suffit de bien former le texte en envoyant les données dans le bon ordre avec les bons marqueurs.

```
// Puis on commence notre JSON par une accolade ouvrante
client.println("{");
// On envoie la première clé : "uptime"
client.print("\t\"uptime (ms)\\" : ");
// Puis la valeur de l'uptime
client.print(millis());
//Une petite virgule pour séparer les deux clés
client.println(",");
// Et on envoie la seconde nommée "analog 0"
client.print("\t\"analog 0\\" : ");
client.println(analogRead(A0));
// Et enfin on termine notre JSON par une accolade fermante
client.println("}");

```

On a presque fini!

Une fois la réponse envoyée, on va faire une toute petite pause pour laisser le temps aux données de partir et enfin on fera le canal de communication avec le client.

```
// Donne le temps au client de prendre les données
delay(10);
// Ferme la connexion avec le client
client.stop();

```

Et voilà!

Il est maintenant temps de tester. Branchez votre Arduino, connectez-la au réseau et allez sur la page ip :port que vous avez paramétrée avec votre navigateur (en l'occurrence `http://192.168.0.143:4200/` pour moi). Si tout se passe bien, aux valeurs près vous obtiendrez quelque chose de similaire à ceci :

```
{
  "uptime (ms)" : 18155,
  "analog 0" : 386
}
```

Génial, non?

Voici le code complet pour faire tout cela :

```
// Ces deux bibliothèques sont indispensables pour le shield
####include <SPI.h>
#include <Ethernet.h>

// L'adresse MAC du shield
byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xA5, 0x7E };
// L'adresse IP que prendra le shield
IPAddress ip(192,168,0,143);

// Initialise notre serveur
// Ce dernier écoutera sur le port 4200
EthernetServer serveur(4200);

void setup()
{
  // On démarre la voie série pour déboguer
  Serial.begin(9600);

  char erreur = 0;
  // On démarre le shield Ethernet SANS adresse IP (donc donnée via DHCP)
  erreur = Ethernet.begin(mac);

  if (erreur == 0) {
    Serial.println("Parametrage avec ip fixe...");
    // si une erreur a eu lieu cela signifie que l'attribution DHCP
    // ne fonctionne pas. On initialise donc en forçant une IP
    Ethernet.begin(mac, ip);
  }
  Serial.println("Init...");
  // Donne une seconde au shield pour s'initialiser
  delay(1000);
  // On lance le serveur
  serveur.begin();
  Serial.print("Pret!");
}
```

```

void loop()
{
  // Regarde si un client est connecté et attend une réponse
  EthernetClient client = serveur.available();
  if (client) {
    // Quelqu'un est connecté!
    Serial.print("On envoi!");
    // On fait notre en-tête
    // Tout d'abord le code de réponse 200 = réussite
    client.println("HTTP/1.1 200 OK");
    // Puis le type mime du contenu renvoyé, du json
    client.println("Content-Type: application/json");
    // Et c'est tout!
    // On envoie une ligne vide pour signaler la fin du header
    client.println();

    // Puis on commence notre JSON par une accolade ouvrante
    client.println("{");
    // On envoie la première clé : "uptime"
    client.print("\t\"uptime (ms)\": ");
    // Puis la valeur de l'uptime
    client.print(millis());
    //Une petite virgule pour séparer les deux clés
    client.println(",");
    // Et on envoie la seconde nommée "analog 0"
    client.print("\t\"analog 0\": ");
    client.println(analogRead(A0));
    // Et enfin on termine notre JSON par une accolade fermante
    client.println("}");
    // Donne le temps au client de prendre les données
    delay(10);
    // Ferme la connexion avec le client
    client.stop();
  }
}

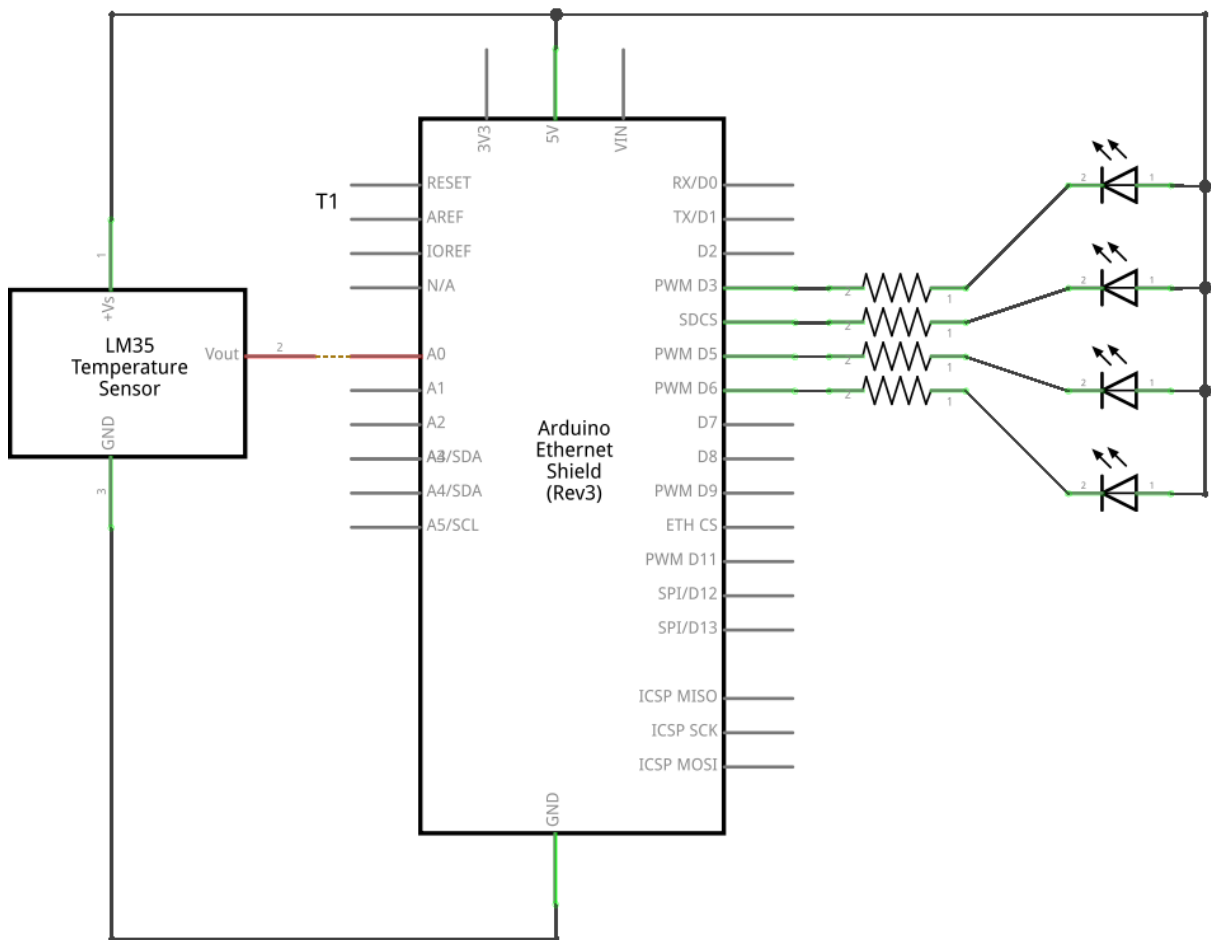
```

[[i]] | S'il faut encore vous convaincre, sachez que cet exemple affiche environ 50 caractères, donc 50 octets (plus le header) à envoyer. La même chose en HTML proprement formaté aurait demandé au grand minimum le double.

9.3.3 Agir sur une requête plus précise

Nous savons maintenant envoyer des données vers notre shield, mais il pourrait être sympa de pouvoir en interpréter et ainsi interagir avec le shield. Voyons voir comment, mais avant tout un petit avertissement :

[[i]] | On ne va pas se mentir, faire cela demande un peu de bidouille car il faut être à l'aise avec la manipulation de chaîne de caractères. Ce type de code est difficilement généralisable ce qui



fritzing

Figure 9.2 – Ethernet, schéma

```

Serial.println("\nRepondre"); // debug
// On fait notre en-tête
// Tout d'abord le code de réponse 200 = réussite
client.println("HTTP/1.1 200 OK");
// Puis le type mime du contenu renvoyé, du json
client.println("Content-Type: application/json");
// Autorise le cross origin
client.println("Access-Control-Allow-Origin: *");
// Et c'est tout!
// On envoi une ligne vide pour signaler la fin du header
client.println();

// Puis on commence notre JSON par une accolade ouvrante
client.println("{");
// On envoie la première clé : "uptime"
client.print("\t\"uptime\" : ");
// Puis la valeur de l'uptime
client.print(millis());
//Une petite virgule pour séparer les deux clés
client.println(",");
// Et on envoie la seconde nommée "analog 0"
client.print("\t\"A0\" : ");
client.print(analogRead(A0));
client.println(",");
// Puis la valeur de la PWM sur la broche 6
client.print("\t\"pwm\" : ");
client.print(pwm, DEC);
client.println(",");
// Dernières valeurs, les broches (elles-mêmes dans un tableau)
client.println("\t\"broches\" : {");
// La broche 3
client.print("\t\t\"3\" : ");
client.print(digitalRead(3));
client.println(",");
// La broche 4
client.print("\t\t\"4\" : ");
client.print(digitalRead(4));
client.println(",");
// La broche 5
client.print("\t\t\"5\" : ");
client.println(digitalRead(5));
client.println("\t}");
// Et enfin on termine notre JSON par une accolade fermante
client.println("}");
}

```


9.3.3.2 Lire la requête

Lorsque l'on reçoit une requête, il faut la traiter. Avez-vous essayé de faire un `Serial.print()` des caractères reçus lors des demandes? Vous pouvez remarquer que la première ligne est toujours "GET /... HTTP/1.1". Avec GET qui est "l'action", /... l'url demandée et HTTP/1.1 le protocole utilisé, on a tout ce qu'il faut pour (ré)agir! Par exemple, si on reçoit la requête GET /?b=3,4&p=42 HTTP/1.1, on aura "juste" à traiter la demande pour extraire le numéro des broches à allumer (3 et 4 mais pas 5) et la valeur du rapport cyclique à appliquer pour la PWM (42). Voyons comment faire.

9.3.3.2.1 Préparation Tout d'abord, on va réserver un tableau de caractères pour le traitement des données. Dans notre cas, 100 caractères devraient largement faire l'affaire. On va aussi déclarer quelques variables pour stocker l'état des broches à changer.

```
char *url = (char *)malloc(100); // L'url reçue à stocker
//char url[100]; // équivalent à la ligne du dessus mais qui ne semble pas vouloir fon
char index = 0; // index indiquant où l'on est rendu dans la chaîne
boolean etats[3] = {LOW, LOW, LOW}; // L'état des 3 sorties
unsigned char pwm = 0; // La valeur de la pwm
```

Une fois cela fait, nous allons devoir passer à la récupération de la première chaîne envoyée par le client.

9.3.3.2.2 Récupérer l'URL C'est parti, faisons notre loop! On va commencer comme avant, en allant lire la présence d'un client.

```
void loop() {
  // Regarde si un client est connecté et attend une réponse
  EthernetClient client = serveur.available();
  if (client) {
    url = ""; // on remet à zéro notre chaîne tampon
    index = 0;
    // traitement
  }
}
```

Si un client est présent, on va regarder s'il a des données à nous donner tant qu'il est connecté (car s'il ne se déconnecte pas, pas la peine de perdre du temps avec lui!).

```
void loop() {
  // Regarde si un client est connecté et attend une réponse
  EthernetClient client = serveur.available();
  if (client) { // Un client est là?
    url = ""; // on remet à zéro notre chaîne tampon
    index = 0;
    while(client.connected()) { // Tant que le client est connecté
      if(client.available()) { // A-t-il des choses à dire?
        // traitement des infos du client
      }
    }
  }
}
```

```

}
}

```

Maintenant que l'on sait que le client est là et nous parle, on va l'écouter en lisant les caractères reçus.

```

void loop() {
  // Regarde si un client est connecté et attend une réponse
  EthernetClient client = serveur.available();
  if (client) { // Un client est là?
    url = ""; // on remet à zéro notre chaîne tampon
    index = 0;
    while(client.connected()) { // Tant que le client est connecté
      if(client.available()) { // A-t-il des choses à dire?
        // traitement des infos du client
        char carlu = client.read(); //on lit ce qu'il raconte
        if(carlu != '\n') { // On est en fin de chaîne?
          // non! alors on stocke le caractère
          url[index] = carlu;
          index++;
        } else {
          // on a fini de lire ce qui nous intéresse
          // on marque la fin de l'url (caractère de fin de chaîne)
          url[index] = '\0';
          // + TRAITEMENT
          // on quitte le while
          break;
        }
      }
    }
    // Donne le temps au client de prendre les données
    delay(10);
    // Ferme la connexion avec le client
    client.stop();
  }
}

```

9.3.3.2.3 Interpréter l'URL Maintenant que nous avons la chaîne du client, il faut l'interpréter pour lire les valeurs des paramètres. Tout d'abord, on commencera par remettre les anciens paramètres à zéro. Ensuite, on va parcourir les caractères à la recherche de marqueur connu : b et p. Ce n'est pas ce qu'il y a de plus simple, mais vous allez voir avec un peu de méthode on y arrive! Rappel : Nous cherchons à interpréter GET /?b=3,4&p=42 HTTP/1.1

PS : le code est "volontairement" un peu plus lourd, car je fais des tests évitant les problèmes si quelqu'un écrit une URL un peu farfelue (sans le "b" ou le "p").

```

boolean interpreter() {
  // On commence par mettre à zéro tous les états
  etats[0] = LOW;
  etats[1] = LOW;

```

```

etats[2] = LOW;
pwm = 0;

// Puis maintenant on va chercher les caractères/marqueurs un par un.
index = 0; // Index pour se promener dans la chaîne (commence à 4 pour enlever "GET
while(url[index-1] != 'b' && url[index] != '=') { // On commence par chercher le "b="
  index++; // Passe au caractère suivant
  if(index == 100) {
    // On est rendu trop loin!
    Serial.println("Oups, probleme dans la recherche de 'b='");
    return false;
  }
}
// Puis on lit jusqu'à trouver le '&' séparant les broches de pwm
while(url[index] != '&') { // On cherche le '&'
  if(url[index] >= '3' && url[index] <= '5') {
    // On a trouvé un chiffre identifiant une broche
    char broche = url[index] - '0'; // On ramène ça au format décimal
    etats[broche-3] = HIGH; // Puis on met la broche dans un futur état haut
  }
  index++; // Passe au caractère suivant
  if(index == 100) {
    // On est rendu trop loin!
    Serial.println("Oups, probleme dans la lecture des broches");
    return false;
  }
  // NOTE : Les virgules séparatrices sont ignorées
}
// On a les broches, reste plus que la valeur de la PWM
// On cherche le "p="
while(url[index-1] != 'p' && url[index] != '=' && index < 100) {
  index++; // Passe au caractère suivant
  if(index == 100) {
    // On est rendu trop loin!
    Serial.println("Oups, probleme dans la recherche de 'p='");
    return false;
  }
}
// Maintenant, on va fouiller jusqu'a trouver un espace
while(url[index] != ' ') { // On cherche le ' ' final
  if(url[index] >= '0' && url[index] <= '9') {
    // On a trouve un chiffre!
    char val = url[index] - '0'; // On ramene ca au format decimal
    pwm = (pwm*10) + val; // On stocke dans la pwm
  }
  index++; // Passe au caractère suivant
  if(index == 100) {
    // On est rendu trop loin!

```

```
        Serial.println("Oups, probleme dans la lecture de la pwm");
        return false;
    }
    // NOTE : Les virgules séparatrices sont ignorées
}
// Rendu ici, on a trouvé toutes les informations utiles!
return true;
}
```

9.3.3.2.4 Agir sur les broches Lorsque toutes les valeurs sont reçues et interprétées, il ne reste plus qu'à les appliquer à nos broches. Vu ce que l'on vient de faire, c'est de loin le plus facile!

```
void action() {
    // On met à jour nos broches
    digitalWrite(3, etats[0]);
    digitalWrite(4, etats[1]);
    digitalWrite(5, etats[2]);
    // Et la PWM
    analogWrite(6, pwm);
}
```

9.3.3.2.5 On assemble !! Il ne reste plus qu'à enchaîner toutes nos fonctions pour avoir un code complet!!

```
void loop() {
    // Regarde si un client est connecté et attend une réponse
    EthernetClient client = serveur.available();
    if (client) { // Un client est là?
        Serial.println("Ping!");
        url = ""; // on remet à zéro notre chaîne tampon
        index = 0;
        while(client.connected()) { // Tant que le client est connecté
            if(client.available()) { // A-t-il des choses à dire?
                // traitement des infos du client
                char carlu = client.read(); //on lit ce qu'il raconte
                if(carlu!= '\n') { // On est en fin de chaîne?
                    // non! alors on stocke le caractère
                    Serial.print(carlu);
                    url[index] = carlu;
                    index++;
                } else {
                    // on a fini de lire ce qui nous intéresse
                    // on marque la fin de l'url (caractère de fin de chaîne)
                    url[index] = '\0';
                    boolean ok = interpreter(); // essaie d'interpréter la chaîne
                    if(ok) {
                        // tout s'est bien passé = on met à jour les broches
                        action();
                    }
                }
            }
        }
    }
}
```

```

    }
    // et dans tout les cas on répond au client
    répondre(client);
    // on quitte le while
    break;
  }
}
}
// Donne le temps au client de prendre les données
delay(10);
// Ferme la connexion avec le client
client.stop();
Serial.println("Pong!");
}
}

```

9.3.3.3 Code complet

```

[[secret]]|cpp | // Ces deux bibliothèques sont indispensables pour le shield
| #include <SPI.h> | #include <Ethernet.h> | | // L'adresse MAC du shield
| byte mac[] = { 0x90, 0xA2, 0xDA, 0x0E, 0xA5, 0x7E }; | // L'adresse IP
que prendra le shield | IPAddress ip(192,168,0,143); | | // Initialise
notre serveur | // Ce dernier écoutera sur le port 4200 | EthernetServer
serveur(4200); | | char *url = (char *)malloc(100); // L'url reçu à stocker
| //char url[100]; | char index = 0; // index indiquant où l'on est rendu
dans la chaîne | boolean etats[3] = {LOW, LOW, LOW}; // L'état des 3 sorties
| unsigned char pwm = 0; // La valeur de la pwm | | void setup() | { |
// On démarre la voie série pour déboguer | Serial.begin(9600); | | //
Configure et initialise les broches | pinMode(3, OUTPUT); digitalWrite(3,
LOW); | pinMode(4, OUTPUT); digitalWrite(4, LOW); | pinMode(5, OUTPUT);
digitalWrite(5, LOW); | pinMode(6, OUTPUT); analogWrite(6, 0); | | char
erreur = 0; | // On démarre le shield Ethernet SANS adresse ip (donc donnée
via DHCP) | erreur = Ethernet.begin(mac); | | if (erreur == 0) { | Serial.println("Par
avec ip fixe..."); | // si une erreur a eu lieu cela signifie que l'attribution
DHCP | // ne fonctionne pas. On initialise donc en forçant une IP | Ethernet.begin(mac,
ip); | } | Serial.println("Init..."); | // Donne une seconde au shield
pour s'initialiser | delay(1000); | // On lance le serveur | serveur.begin();
| Serial.println("Pret!"); | } | | void loop() { | // Regarde si un client
est connecté et attend une réponse | EthernetClient client = serveur.available();
| if (client) { // Un client est là? | Serial.println("Ping!"); | url =
""; // on remet à zéro notre chaîne tampon | index = 0; | while(client.connected())
{ // Tant que le client est connecté | if(client.available()) { // A-t-il
des choses à dire? | // traitement des infos du client | char carlu = client.read();
//on lit ce qu'il raconte | if(carlu!= '\n') { // On est en fin de chaîne?
| // non! alors on stocke le caractère | Serial.print(carlu); | url[index]
= carlu; | index++; | } else { | // on a fini de lire ce qui nous intéresse
| // on marque la fin de l'url (caractère de fin de chaîne) | url[index]
= '\0'; | boolean ok = interpreter(); // essaie d'interpréter la chaîne |

```

```

if(ok) { | // tout s'est bien passé = on met à jour les broches | action();
| } | // et dans tout les cas on répond au client | repondre(client); | //
on quitte le while | break; | } | } | } | // Donne le temps au client de
prendre les données | delay(10); | // Ferme la connexion avec le client |
client.stop(); | Serial.println("Pong!"); | } | } | | void rafraichir()
{ | // Rafraichit l'etat des broches / PWM | digitalWrite(3, etats[0]); |
digitalWrite(4, etats[1]); | digitalWrite(5, etats[2]); | analogWrite(6,
pwm); | } | | void repondre(EthernetClient client) { | // La fonction prend
un client en argument | | Serial.println("\nRepondre"); // debug | // On
fait notre en-tête | // Tout d'abord le code de réponse 200 = réussite
| client.println("HTTP/1.1 200 OK"); | // Puis le type mime du contenu
renvoyé, du json | client.println("Content-Type: application/json"); | //
Autorise le cross origin | client.println("Access-Control-Allow-Origin:
*"); | // Et c'est tout! | // On envoi une ligne vide pour signaler la
fin du header | client.println(); | | // Puis on commence notre JSON par
une accolade ouvrante | client.println("{}"); | // On envoie la première
clé : "uptime" | client.print("\t\"uptime\": "); | // Puis la valeur de
l'uptime | client.print(millis()); | //Une petite virgule pour séparer
les deux clés | client.println(","); | // Et on envoie la seconde nommée
"analog 0" | client.print("\t\"A0\": "); | client.print(analogRead(A0));
| client.println(","); | // Puis la valeur de la PWM sur la broche 6 |
client.print("\t\"pwm\": "); | client.print(pwm, DEC); | client.println(",");
| // Dernières valeurs, les broches (elles mêmes dans un tableau) | client.println("\t
{}"); | // La broche 3 | client.print("\t\t\"3\": "); | client.print(digitalRead(3));
| client.println(","); | // La broche 4 | client.print("\t\t\"4\": "); |
client.print(digitalRead(4)); | client.println(","); | // La broche 5 |
client.print("\t\t\"5\": "); | client.println(digitalRead(5)); | client.println("\t}");
| // Et enfin on termine notre JSON par une accolade fermante | client.println("}");
| } | | boolean interpreter() { | // On commence par mettre à zéro tous
les états | etats[0] = LOW; | etats[1] = LOW; | etats[2] = LOW; | pwm =
0; | | // Puis maintenant on va chercher les caractères/marqueurs un par
un. | index = 0; // Index pour se promener dans la chaîne (commence à 4
pour enlever "GET " | while(url[index-1] != 'b' && url[index] != '=') { //
On commence par chercher le "b=" | index++; // Passe au caractère suivant
| if(index == 100) { | // On est rendu trop loin! | Serial.println("Oups,
probleme dans la recherche de 'b='"); | return false; | } | } | // Puis on
lit jusqu'à trouver le '&' séparant les broches de pwm | while(url[index] !=
'&') { // On cherche le '&' | if(url[index] >= '3' && url[index] <= '5') {
| // On a trouvé un chiffre identifiant une broche | char broche = url[index]-'0';
// On ramène ça au format décimal | etats[broche-3] = HIGH; // Puis on met
la broche dans un futur état haut | } | index++; // Passe au caractère
suivant | if(index == 100) { | // On est rendu trop loin! | Serial.println("Oups,
probleme dans la lecture des broches"); | return false; | } | } | // NOTE :
Les virgules séparatrices sont ignorées | } | // On a les broches, reste
plus que la valeur de la PWM | // On cherche le "p=" | while(url[index-1] !=
'p' && url[index] != '=' && index<100) { | index++; // Passe au caractère
suivant | if(index == 100) { | // On est rendu trop loin! | Serial.println("Oups,
probleme dans la recherche de 'p='"); | return false; | } | } | // Maintenant,

```

```

on va fouiller jusqu'a trouver un espace | while(url[index] != ' ') { //
On cherche le ' ' final | if(url[index] >= '0' && url[index] <= '9') {
| // On a trouve un chiffre! | char val = url[index] - '0'; // On ramene
ca au format decimal | pwm = (pwm*10) + val; // On stocke dans la pwm |
} | index++; // Passe au caractère suivant | if(index == 100) { | // On
est rendu trop loin! | Serial.println("Oups, probleme dans la lecture de
la pwm"); | return false; | } | // NOTE : Les virgules séparatrices sont
ignorées | } | // Rendu ici, on a trouvé toutes les informations utiles!
| return true; | } | | void action() { | // On met à jour nos broches |
digitalWrite(3, etats[0]); | digitalWrite(4, etats[1]); | digitalWrite(5,
etats[2]); | // Et la PWM | analogWrite(6, pwm); | } |

```

9.3.4 Sortir de son réseau privé

À ce stade, nous arrivons à récupérer des informations et donner des ordres à notre Arduino. Cependant, on est bloqué dans notre réseau local. En effet, si vous essayez d'y accéder depuis un autre ordinateur à l'autre bout du monde, il est fort probable que cela ne marche pas...

Pour pallier cela, il va falloir faire un peu d'administration réseau. Je vais couvrir la démarche ici mais ne rentrerai pas trop dans les détails, car ce n'est pas non plus le but de ce tutoriel. Je partirai aussi du principe que vous êtes à votre domicile et utilisez une box ou un routeur que vous pouvez administrer.

L'opération que nous allons faire ici s'appelle une redirection NAT (Network address translation). Mais tout d'abord, essayons de comprendre le problème.

9.3.4.1 Le souci

Le problème dans l'état actuel c'est que votre box laisse passer le trafic vers l'extérieur, accepte les réponses, mais ne tolère pas trop qu'on accède directement à son adresse sur n'importe quel port. En effet, après tout c'est logique. Imaginez que vous avez plusieurs équipements connectés à votre routeur/box. Si vous essayez d'y accéder depuis l'extérieur, comment cette dernière saura à quel matériel vous souhaitez accéder ?

Dans votre réseau local, chaque appareil à sa propre adresse IP qui le représente **localement**. Cette adresse est très souvent de la forme 192.168.0.xyz. Vous pourriez avoir par exemple votre téléphone en 192.168.0.142, votre ordinateur en 192.168.0.158 et votre Arduino en 192.168.0.199. Votre box (qui gère ce réseau local) possède quant à elle une IP **publique**. C'est cette IP qui la représente aux yeux du monde. Admettons, pour l'exemple, que cette adresse soit 42.128.12.13 (trouvez la vôtre avec un service comme my-ip.com). Si vous cherchez à accéder à l'adresse publique avec le port 4200 en espérant atteindre votre Arduino vous serez déçus. En effet, vous allez bien atteindre votre box, mais cette dernière n'aura aucune idée de quel équipement doit être interrogé ! Est-ce votre ordinateur ? Votre smartphone ? Votre Arduino ?

Il va donc falloir lui expliquer...

9.3.4.2 La solution

[[a]] | Chaque constructeur de box/routeur possède sa propre interface. Je vais essayer d'être le plus générique possible pour que les explications parlent au plus grand nombre, mais ne m'en

voulez pas si toutes les dénominations ne sont pas exactement comme chez vous !

Et cette explication s'appelle une redirection **NAT**, ou redirection de port. En faisant cela, vous expliquerez à votre box que "tout le trafic qui arrive sur ce port particulier doit être envoyé sur cet équipement local" (et vous pouvez même rerouter le trafic pour le changer de port si vous voulez).

Mettons cela en place. Pour cela, commencez par aller dans l'interface d'administration de votre box (souvent c'est à l'adresse 192.168.0.1). Vous devez être dans le réseau local de la box pour le faire ! Ensuite, il vous faudra trouver la partie parlant de "NAT" ou de "redirection de port". Une fois dans cette dernière, il va falloir demander à ce que tout ce qui rentre dans le port 4200 (ou la plage 4200-4200) soit redirigé vers l'équipement "Arduino" (Wiznet) ou son adresse IP locale si vous la connaissez et que vous l'avez déjà imposée au routeur comme fixe.

Vous aurez alors quelque chose comme ça :

ADVANCE PORT FORWARDING RULES			
		Port	Traffic Type
Name	<< Application Name ▼	Public Port	Any ▼
Test Arduino		4200 ~ 4200	
IP Address	<< Computer Name ▼	Private Port	
192.168.0.143		4200 ~ 4200	

Figure 9.3 – Réglages NAT

Sauvegardez et éventuellement redémarrez votre box (normalement ce n'est pas nécessaire, mais sur du vieux matériel ça peut arriver...). Maintenant, démarrez votre Arduino avec un des programmes ci-dessus et essayez d'accéder à votre adresse publique et le port 4200 avec un navigateur internet. Normalement, l'Arduino devrait répondre comme si vous l'interrogiez avec son adresse locale !

9.3.5 Faire une interface pour dialoguer avec son Arduino

Pour terminer ce tutoriel, je vous propose de réaliser une petite interface de pilotage de l'Arduino via internet. Pour cela, on va garder le programme que nous avons dans l'Arduino pour le paragraphe concernant les requêtes avancées, et nous nous contenterons de simplement faire de l'HTML et du JavaScript. Le HTML servira à faire l'interface et le JavaScript fera les interactions via des requêtes ajax.

Avant toute chose, je vous ai menti ! Il faut en fait rajouter une petite ligne dans notre code de la fonction `repondre()` faite plus tôt. En effet, pour des raisons de sécurité les requêtes ajax ne peuvent pas aller d'un domaine à un autre (donc de "n'importe où sur le web" à "votre Arduino"). Il faut donc rajouter une ligne dans le header renvoyé pour signaler que l'on autorise le "cross-domain". Rajoutez donc cette ligne juste après le "content-type" :

```
// Autorise le cross origin
client.println("Access-Control-Allow-Origin: *");
```

Maintenant que cela est fait, nous allons créer une structure HTML toute simple pour avoir nos boutons.


```

<!DOCTYPE html>
<html lang="fr">
  <head>
    <meta charset="utf-8">
    <title>Interface de pilotage Arduino</title>
  </head>
  <body>
    <div class="main">
      <p>
        Adresse publique du shield : <br />
        http://<input type="text" id="ip" value="123.123.123.123" size="15"/>
        : <input type="text" id="port" value="4200" size="5"/>
      </p>
      <hr />
      <p>
        <input type="checkbox" id="broche3" name="broche3" />
        <label for="broche3">Activer Broche 3.</label>
        Etat : <input type="radio" id="etat3" disabled />
      </p>
      <p>
        <input type="checkbox" id="broche4" name="broche4" />
        <label for="broche4">Activer Broche 4.</label>
        Etat : <input type="radio" id="etat4" disabled />
      </p>
      <p>
        <input type="checkbox" id="broche5" name="broche5" />
        <label for="broche5">Activer Broche 5.</label>
        Etat : <input type="radio" id="etat5" disabled />
      </p>
      <p>
        PWM : 0<input type="range" min="0" max="255" id="pwm" />255
      </p>
      <p>
        A0 : <meter min="0" max="1023" id="a0" />
      </p>
      <button id="envoyer">Executer !</button>
      <p>
        Millis : <span id="millis">0</span> ms
      </p>
    </div>
  </body>
</html>

```

Ensuite, un peu de JavaScript nous permettra les interactions. L'ensemble est grosso modo divisé en deux fonctions importantes. `setup()` qui sera exécuté lorsque la page est prête puis `executer()` qui sera appelée à chaque fois que vous cliquez sur le bouton. Cette dernière fera alors une requête à votre Arduino et attendra la réponse json. La fonction `afficher()` utilisera alors les informations pour changer les composants html.

Activer Broche 3. Etat :
 Activer Broche 4. Etat :
 Activer Broche 5. Etat :
 PWM : 0 255
 A0 :

 Millis : 219760 ms

Figure 9.4 – Interface HTML

Comme vous pouvez le constater, toute la partie affichage est gérée de manière quasi complètement indépendante de l'Arduino. Cela va nous permettre de transmettre un minimum de données et garder une souplesse maximale sur l'affichage. Si demain vous décidez de changer l'interface voire carrément de faire une application dans un autre langage, vous n'aurez pas besoin de toucher à votre Arduino car les données sont envoyées dans un format générique.

```

var broches = []; // Tableau de broches
var etats = []; // Tableau d'etat des broches
var pwm;
var a0;
var millis;
var adresse = "http://82.143.160.118:4200/"; // L'url+port de votre shield

document.addEventListener('DOMContentLoaded', setup, false);

function setup() {
  // fonction qui va lier les variables à leur conteneur HTML
  broches[3] = document.getElementById("broche3");
  broches[4] = document.getElementById("broche4");
  broches[5] = document.getElementById("broche5");
  etats[3] = document.getElementById("etat3");
  etats[4] = document.getElementById("etat4");
  etats[5] = document.getElementById("etat5");
  pwm = document.getElementById("pwm");
  a0 = document.getElementById("a0");
  millis = document.getElementById("millis");

  // La fonction concernant le bouton

```

```

    var bouton = document.getElementById("envoyer");
    bouton.addEventListener('click', executer, false);
}

function executer() {
    // Fonction qui va créer l'url avec les paramètres puis
    // envoyer la requête
    var requete = new XMLHttpRequest(); // créer un objet de requête
    var url = adresse;
    url += "?b=";
    for(i=3; i <= 5; i++) { // Pour les broches 3 à 5 de notre tableau
        if(broches[i].checked) // si la case est cochée
            url += i + ",";
    }
    // enlève la dernière virgule si elle existe
    if(url[url.length-1] === ',')
        url = url.substring(0, url.length-1);
    // Puis on ajoute la pwm
    url += "&p=" + pwm.value;
    console.log(url) // Pour debugguer l'url formée
    requete.open("GET", url, true); // On construit la requête
    requete.send(null); // On envoie!
    requete.onreadystatechange = function() { // on attend le retour
        if (requete.readyState == 4) { // Revenu!
            if (requete.status == 200) { // Retour s'est bien passé!
                // fonction d'affichage (ci-dessous)
                afficher(requete.responseText);
            } else { // Retour s'est mal passé :(
                alert(requete.status, requete.statusText);
            }
        }
    }
};
}

function afficher(json) {
    // Affiche l'état des broches/pwm/millis revenu en json
    donnees = JSON.parse(json);
    console.log(donnees);

    for(i=3; i <= 5; i++) { // Pour les broches 3 à 5 de notre tableau
        etats[i].checked = donnees["broches"][i];
    }
    pwm.value = parseInt(donnees["pwm"]);
    a0.value = parseInt(donnees["A0"]);
    millis.textContent = donnees["uptime"];
}
}

```

En cadeau de fin, une version utilisable en ligne de cette interface :

->

<http://jsfiddle.net/f6c2kc11/7/>

<-

Pour l'utiliser, il suffit simplement de modifier l'url de base avec votre IP publique et le port utilisé par l'Arduino.

Magie de l'internet, votre Arduino fait maintenant partie de la grande sphère de l'IoT, le phénomène très à la mode de l'Internet of Things. Qu'allez-vous bien pouvoir envoyer comme informations dorénavant ?

->

!(https://www.youtube.com/watch?v=9j_j5Q-MZ_o)

<-

10 Conclusion

10.1 Aller plus loin

Ce tutoriel vous a plus ? Vous en voulez encore ? Voici quelques autres tutoriels qui pourrait sur-ement vous intéresser...

[[information]] | + [Ajouter des sorties numériques à l'Arduino, le 74HC595](#) : pour augmenter le nombre de sorties numériques possibles avec le 74HC595, un convertisseur série -> parallèle ; | + [Alimenter une Arduino sans USB](#) puis conquérir le monde ; | + [Gestion de la mémoire sur Arduino](#) : maîtrisez les différentes mémoires d'Arduino avec ce mini-tutoriel ; | + [Réaliser un télémètre à ultrasons](#) : mettez vos connaissances en pratique en réalisant cet outil de mesure ; | + [La Fabrication Numérique](#) : passer d'une idée à un prototype en utilisant les nouveaux outils de la Fabrication Numérique (big-tuto format MOOC sur ZdS)

Et quelques articles sur mon blog pas encore paru sur ZdS :

[[information]] | + [Utiliser un module bluetooth HC-05](#) | + [Utiliser Sublime Text comme IDE pour Arduino](#) | + [Découvrir les ports et les registres sur Arduino](#)

10.2 Remerciements

Un gros merci à plusieurs membres de Zeste de Savoir pour le coup de main à l'import de ce tutoriel, sa relecture et l'aide à sa mise en page. Un travail et un soin magnifique ont été apportés pour tenter de vous fournir un tuto le plus propre possible.

Plein de mercis en particulier à :

- [Arius](#) ;
- [artragis](#) ;
- [Flori@n.B](#) ;
- [ShigeruM](#) ;
- [simbilou](#)

Et pour leur travail sur la publication et l'export PDF, merci à :

- [gustavi](#) ;
- [pierre_24](#) ;
- [SpaceFox](#) ;
- Et tout ceux ayant apporté leur pierre à l'édifice de près ou de loin ! Ça a vraiment été un travail de longue haleine. :D