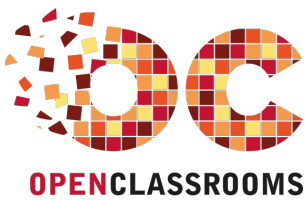
# Arduino pour bien commencer en électronique et en programmation

**Par Eskimon**
**et olyte**




www.openclassrooms.com

# Sommaire

# Arduino pour bien commencer en électronique et en programmation

Par  olyte et  Eskimon

Mise à jour : 02/12/2012
Difficulté : Intermédiaire Durée d'étude : 2 mois


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

Depuis que l'électronique existe, sa croissance est fulgurante et continue encore aujourd'hui. Si bien que faire de l'électronique est devenu accessible à toutes personnes en ayant l'envie. Mais, le manque de cours simples sur le net ou en libraire empêche la satisfaction des futurs électroniciens amateurs ou professionnels et parfois empêche certains génies à se révéler ( ). C'est pourquoi je souhaite intervenir contre cette insuffisance et écris ce cours sur l'électronique et la programmation.

Ce que nous allons apprendre aujourd'hui est un mélange d'électronique et de programmation. On va en effet parler d'**électronique 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 enchainerons sur la prise en main du système Arduino. Enfin, je vous ferais un cours très rapide sur le langage Arduino, mais il aura l'audace de poser les bases de la programmation. C'est une fois que ces étapes seront achevées que nous pourrons entamer notre premier programme et faire un pas dans l'électronique embarquée.

Avant de continuer, il est important que je vous informe d'une chose : dans ce cours, il est question d'utilisation de matériel. Ce matériel n'est pas fourni par le site du zéro, ni même par les auteurs. En outre, il faudra l'acheter. J'explique cette étape dans un des chapitres. Pour ceux qui ne voudraient pas dépenser un centime, vous pouvez suivre le cours et apprendre les bases de la programmation, mais ce sera plus difficile. Et puis, dites vous bien qu'il nous a fallu nous aussi acheter le matériel pour pouvoir tout vous expliquer en détail.

## Plan du cours

Je vais détailler un peu le plan du cours. Il est composé d'un certain nombre de parties qui ne se suivent pas forcément. Je m'explique.

### *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 après cela que vous pourrez suivre le cours selon les connaissances que vous aimeriez acquérir.

### *Notions en robotique et en domotique*

Là, ce sont les parties 5 et 6. Elles traitent de notions utilisées en **robotique** et en **domotique**. Elles vous permettrons d'acquérir des bases dans ces domaines. Si la lecture de ces parties ne vous emballe pas, vous pourrez toujours y revenir plus tard et

accéder aux parties suivantes, sans pour autant perdre le fil de la lecture.

### *Les écrans LCD*

Cette partie traite d'un sujet à part, à la fois utilisé en robotique et en domotique, mais tout aussi utilise dans d'autres domaines, tel que la mesure et l'affichage de données. On pourrait très bien imaginer l'utilisation d'écrans LCD pour déboguer vos programmes.

### *Interface Homme-Machine*

C'est le sujet de la partie 8 qui développe le fonctionnement d'un langage de programmation très proche d'Arduino et qui vous permettra de réaliser des **interfaces graphiques** (IG) sur votre ordinateur, dans le but de communiquer avec votre carte Arduino. En somme, vous pourrez créer des programmes (j'entends par là des IG) pour contrôler, depuis votre ordinateur, votre carte Arduino. Par exemple, vous pourrez ensuite réaliser une commande domotique qui éteint la lumière de votre salon ou allume la machine à café, juste en cliquant sur un bouton présent dans votre IG.

Ce n'est pas tout ! En effet, en plus de pouvoir faire des IG sur votre ordinateur, vous pourrez également les exporter pour les transférer sur un téléphone mobile qui supporte les applications Java !

### *Internet*

Cette dernière grande partie vous expliquera comment utiliser votre Arduino, avec un shield Ethernet, pour communiquer sur internet et créer votre propre mini-serveur web. Vous aurez même la possibilité de découvrir comment actionner des entrés/sorties à distance par l'interface d'une simple page Web !

### *Les annexes*

Pour finir, les annexes traiterons de sujets n'ayant pas une place conséquente dans le cours, mais tout aussi intéressant.

## Objectif du cours

Je l'ai déjà énoncé mais je préfère le re-préciser clairement.

Vous apprendrez tout au long de la lecture, les bases de l'électronique et de la programmation. Sauf que les notions électroniques abordées seront d'un bas niveau et ne vous permettrons que la mise en œuvre avec de la programmation. Vous ne pourrez donc pas créer tout seul des petits montages n'utilisant que des composants électroniques sans avoir à programmer un microcontrôleur. Cependant, il y aura deux grandes parties où l'on verra beaucoup d'électronique, il s'agit des moteurs et des capteurs. On utilisera des petits systèmes électroniques (par exemple la commande de pilotage d'un moteur à courant continu) associées à la programmation.

Pour ceux que l'électronique intéresserait beaucoup plus que ce qui ne sera abordé ici, je peut vous envoyer lire ce cours qui débute également sur le Site du Zéro.

En revanche, côté programmation, vous allez passer en revue tous les points essentiels, car c'est l'outil principal de la mise en œuvre des systèmes embarqués.

Paré pour commencer l'aventure ? Alors on y va ! 

**Citation : olyte et Eskimon**

> Les auteurs de ce tutoriel ont le plaisir de présenter Astalaseven qui est l'âme bienveillante du tutoriel. Nous le félicitons pour sa capacité à ne pas déprimer face aux fautes immondes que l'on peut écrire dans ce tutoriel. Et nous le remercions pour le travail qu'il effectue (corrections orthographiques, grammaticales, syntaxiques, etc.). Ainsi, nous avons décidé, en attendant un statut plus approprié de la part des administrateurs du site, de l'officialiser en tant que co-auteur spécialisé dans la correction de fautes.

Vous pouvez l'applaudir ! Si, si !!

# Partie 1 : [Théorie] Découverte de l'Arduino

Dans cette première partie, nous ferrons nos premiers pas avec l'Arduino. Soyez attentif car il s'agit de prendre en main le fonctionnement du système Arduino. Vous n'irez donc pas bien loin si vous ne savez pas l'utiliser.

## Présentation

Comment faire des montages électroniques simplement 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. Ces cartes polyvalentes sont donc parfaites pour nous, débutants, qui ne demandons qu'à apprendre et progresser.

Dans ce premier chapitre, nous allons donc parler du projet Arduino, de ses nombreux avantages, mais aussi du matériel dont nous aurons besoin durant tout le cours.

### Présentation d'Arduino

### Qu'est ce que c'est ?

Arduino est un projet créé par une équipe de développeurs, composée de six individus : *Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino, David Mellis et Nicholas Zambetti*. Cette équipe a créé le "système Arduino". C'est un outil qui va permettre aux débutants, amateurs ou professionnels de créer des systèmes électroniques plus ou moins complexes.

#### *Le but et l'utilité*

Le système Arduino, nous donne la possibilité d'allier les performances de la programmation à celles de l'électronique. Plus précisément, nous allons programmer des systèmes électroniques. Le gros avantage de l'électronique programmée c'est qu'elle simplifie grandement les schémas électroniques et par conséquent, le coût de la réalisation, mais aussi la charge de travail à la conception d'une carte électronique.

L'utilité est sans doute quelque chose que l'on perçoit mal lorsque l'on débute, mais une fois que vous serez rentré dans le monde de l'Arduino, vous serez fasciné par l'incroyable puissance dont il est question et des applications possibles !

#### *Applications*

Le système Arduino nous permet de réaliser un grand nombre de choses, qui ont une application dans tous les domaines ! Je vous l'ai dit, l'étendue de l'utilisation de l'Arduino est gigantesque. Pour vous donner quelques exemples, vous pouvez :

- contrôler les appareils domestiques
- fabriquer votre propre robot
- faire un jeu de lumières
- communiquer avec l'ordinateur
- télécommander un appareil mobile (modélisme)
- etc.

Avec Arduino, nous allons faire des systèmes électroniques tels qu'une bougie électronique, une calculatrice simplifiée, un synthétiseur, etc. Tous ces systèmes seront conçus avec pour base une carte Arduino et un panel assez large de composants électroniques.

### Les bonnes raisons de choisir Arduino

Il existe pourtant dans le commerce, une multitude de plateformes qui permettent de faire la même chose. Notamment les microcontrôleurs « PIC » du fabricant Microchip. Nous allons voir pourquoi choisir l'Arduino. (Je tiens à préciser que je n'ai aucun lien commercial avec eux ! )

### *Le prix*

En vue des performances qu'elles offrent, les cartes Arduino sont relativement peu couteuses, ce qui est un critère majeur pour le débutant. Celle que nous utiliserons pour la suite du cours a un prix qui tourne aux environs de 25 € TTC ce qui est un bon rapport qualité/prix.

### *La liberté*

C'est un bien grand mot, mais elle définit de façon assez concise l'esprit de l'Arduino. 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.

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 ».

Les cartes non officielles, on peut les trouver et les acheter sur Internet et sont pour la quasi-totalité compatibles avec les cartes officielles Arduino.

### *La compatibilité*

Le logiciel, tout comme la carte, est compatible sous les plateformes les plus courantes (Windows, Linux et Mac), contrairement aux autres outils de programmation du commerce qui ne sont, en général, compatibles qu'avec Windows.

### *La communauté*

La communauté Arduino est impressionnante et le nombre de ressources à son sujet est en constante évolution sur internet. De plus, on trouve les références du langage Arduino ainsi qu'une page complète de tutoriels sur le site arduino.cc (en anglais) et arduino.cc (en français).

Finalement, nous retiendrons ce projet pour toutes ses qualités !

## Les outils Arduino

À présent, rapprochons-nous de « l'utilisation » du système Arduino et voyons comment il se présente. Il est composé de deux choses principales, qui sont : le matériel et le logiciel. Ces deux outils réunis, il nous sera possible de faire n'importe quelle réalisation !

### *Le matériel*

Il s'agit d'une carte électronique basée autour d'un microcontrôleur Atmega du fabricant Atmel, dont le prix est relativement bas pour l'étendue possible des applications. Voilà à quoi ressemble la carte que nous allons utiliser :

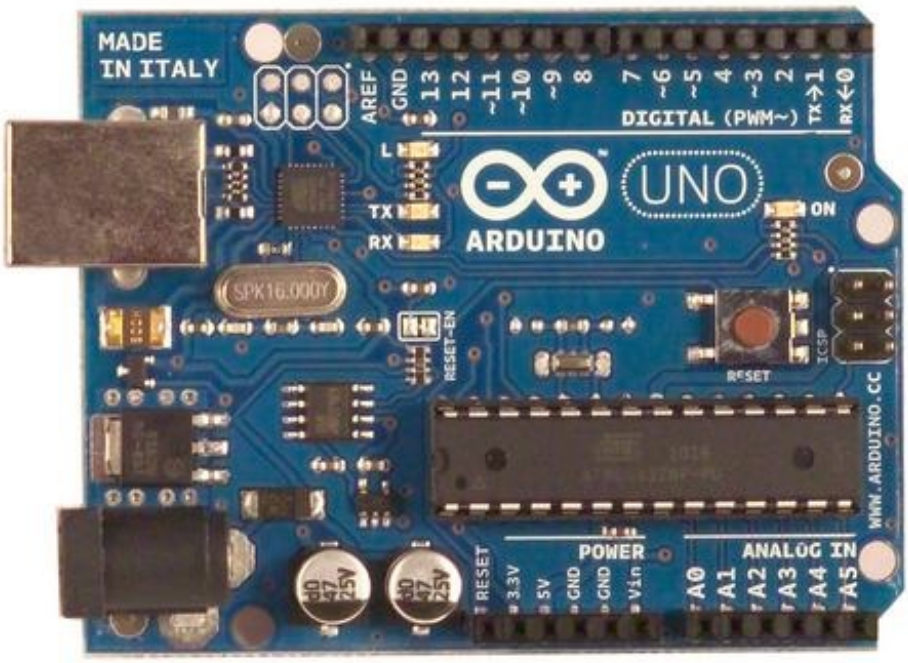



*Figure 1 : Carte Arduino "Uno"*

### *Le logiciel*

Le logiciel va nous permettre de programmer la carte Arduino. Il nous offre une multitude de fonctionnalités que nous verrons dans un chapitre dédié. Voilà à quoi il ressemble :

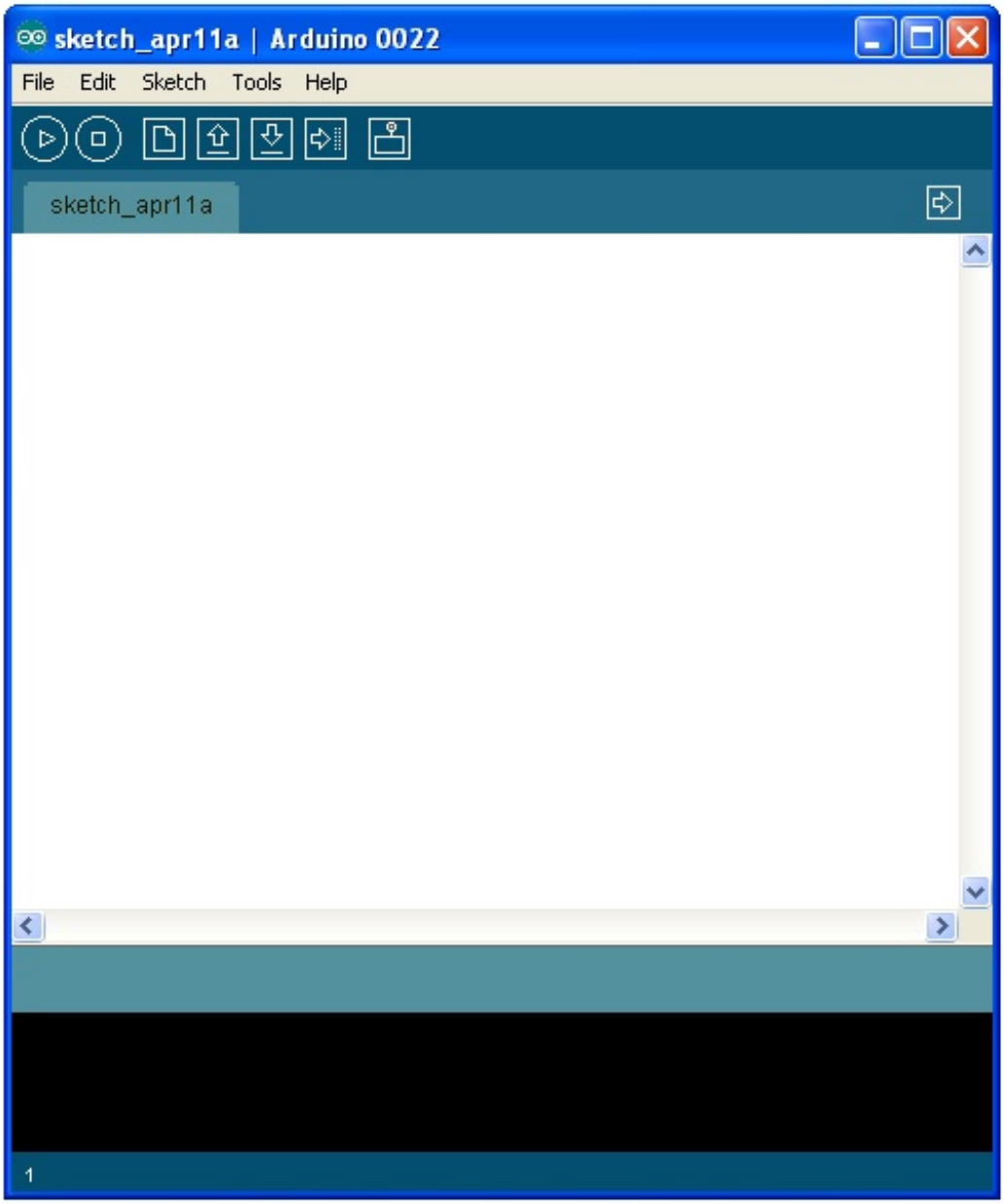



*Figure 2 : Logiciel Arduino*

## Acheter une carte

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 lignes et en boutiques qui vendent des cartes Arduino. Je vais vous en donner quelques-uns, mais avant, il va falloir différencier certaines choses.

### *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. Pour vous aider dans ce choix, je vous donnerai une liste de quelques fabricants à qui l'on peut faire confiance.

### *Les types de cartes*

Il y a trois types de cartes :

- Lesdites « officielles » qui sont fabriquées en Italie par le fabricant officiel : *Smart Projects*
- Lesdits « compatibles » qui ne sont pas fabriqués par *Smart Projects*, mais qui sont totalement compatibles avec les Arduino officielles.
- Les « autres » fabriquées par diverse entreprise et commercialisées sous un nom différent (Freeduino, Seeduino,

Femtoduino, ...).

### *Les différentes cartes*

Des cartes Arduino il en existe beaucoup ! Peut-être une centaine toutes différentes ! Je vais vous montrer lesquelles on peut utiliser et celle que j'utiliserai dans le cours.

**La carte *Uno* et *Duemilanove***

Nous choisirons d'utiliser la carte portant le nom de « Uno » ou « Duemilanove ». Ces deux versions sont presque identiques.


*Figure 3 : carte Arduino "Uno" sur laquelle nous allons travailler*

**La carte *Mega***
La carte Arduino Mega est une autre carte qui offre toutes les fonctionnalités des précédentes, mais avec des options en plus. On retrouve notamment un nombre d'entrées et de sorties plus importantes ainsi que plusieurs liaisons séries. En revanche, le prix est plus élevé : plus de 50 € !


*Figure 4 : carte Arduino "Mega"*

### *Où acheter ?*

Il existe sur le net une multitude de magasins qui proposent des cartes Arduino. Voici la liste des distributeurs de cartes Arduino en France. Elle se trouve également sur cette page.

- AlyaSoft
- Lextronic
- ZaRtronic
- Snootlab
- Jlectronique
- RobotShop
- Semageek

J'ai vu des cartes officielles "édition SMD/CMS". Ca 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".

## Listes 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

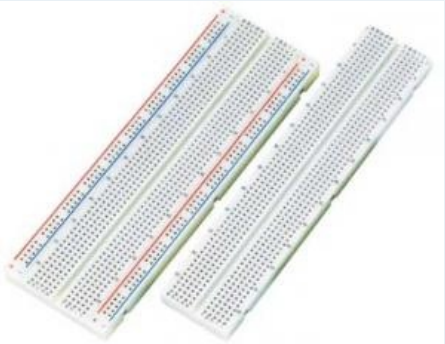
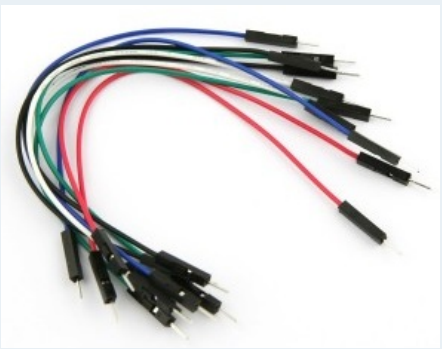
donne cette liste, cela vous évitera d'acheter en plusieurs fois. Vous allez devoir me croire sur parole sur leur intérêt. Nous découvrirons comment ils fonctionnent et comment les utiliser tout au long du tutoriel.

Afin que vous n'ayez pas à faire plusieurs commandes et donc subir plusieurs fois des frais de port si vous commandez par internet, nous vous avons préparé des listes de courses. Pourquoi "des" ? Car tout le monde n'a pas les mêmes ambitions et envies de travailler les mêmes choses. Vous aller donc trouver ci-dessous une liste de course par **partie**. Lorsque vous lirez le cours, à chaque début de partie sera rappelé ce dont vous avez besoin pour suivre le tutoriel (dans l'introduction dans une balise secret pour ne pas gêner la lecture).

Enfin, à la fin de tout cela vous trouverez une "Méga-Liste" qui regroupe tous les composants nécessaires pour suivre **tout** le tutoriel du début jusqu'à la fin (cependant les composants marqués d'une '*' sont là à titre indicatif puisqu'ils seront intégrés dans des chapitres prévus mais pas encore écrits. Leur présence est donc sujette à changement et nous ne pourrons pas assurer à 100% que nous les utiliserons). Cette liste vous montrera aussi des photos d'illustrations des composants.

Attention, ces listes ne contiennent 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 perdez ou détruisez...). Pour ce qui est des prix, j'ai regardé sur différents sites grands publics (donc pas Farnell par exemple), ils peuvent donc paraître plus élevé que la normale dans la mesure où ces sites amortissent moins sur des ventes à des clients fidèles qui prennent tout en grande quantité...

Avant que j'oublie, 3 éléments n'apparaitront pas dans les listes et sont indispensables :

| Une Arduino Uno | Une BreadBoard (plaque d'essai) | Un lot de fils pour brancher le tout ! |
|---|---|---|
|  |  |  |

Et maintenant, place aux listes !

### *Partie 1 : [Théorie] Découverte de l'Arduino*

Pas de liste de course pour cette partie !

### *Partie 2 : [Pratique] Gestion des entrées / sorties*

**Secret** (cliquez pour afficher)

| Désignation | Valeur - Caractéristique | Quantité | Prix unitaire indicatif (€) | Photo |
|---|---|---|---|---|
| LED | rouge | 6 | 0.10 |  |
| | verte | 2 | | |
| | jaune ou orange | 2 | | |
| Résistance | entre 220 et 470 Ohm | 9 | 0.10 |  |
| | entre 2.2 et 4.7 kOhm | 2 | | |

| | | | | |
|---|---|---|---|---|
| | 10 kOhm | 2 | | |
| Condensateur | 10 nF | 2 | 0.30 |  |
| Bouton Poussoir | - | 2 | 0.40 |  |
| Transistor | 2N2222 ou BC547 | 2 | 0.60 |  |
| Décodeur BCD | MC14543 | 1 | 1.00 |  |
| Afficheur 7 segments | anode commune | 2 | 1.00 |  |
| **Total €** | | | **7.9 €** | |

## *Partie 3 : [Pratique] Communication par la liaison série*

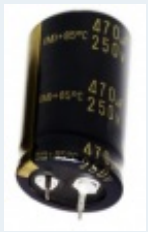
**Secret** (cliquez pour afficher)

| Désignation | Valeur - Caractéristique | Quantité | Prix unitaire indicatif (€) | Photo |
|---|---|---|---|---|
| LED | rouge | 1 | 0.10 |  |
| | jaune ou orange | 1 | | |
| | verte | 1 | | |
| Résistance | 10 kOhm | 2 | 0.10 |  |
| | entre 220 et 470 Ohm | 3 | 0.10 | |
| Condensateur | 10 nF | 2 | 0.30 |  |
| | | | |  |

| | | | | |
|---|---|---|---|---|
| Bouton Poussoir | - | 2 | 0.40 |  |
| **Total €** | | | **2.2 €** | |

### *Partie 4 : [Pratique] Les grandeurs analogiques*

**Secret** (cliquez pour afficher)

| Désignation | Valeur - Caractéristique | Quantité | Prix unitaire indicatif (€) | Photo |
|---|---|---|---|---|
| LED | rouge | 7 | 0.10 |  |
| | verte | 3 | | |
| | RVB | 1 | 3.00 |  |
| Résistance | entre 220 et 470 Ohm | 10 | 0.10 |  |
| | 1 kOhm | 2 | | |
| Potentiomètre linéaire | 10 kOhm | 1 | 0.40 |  |
| Condensateur électrochimique | 1000µF | 1 | 1 |  |
| **Total €** | | | **6.6 €** | |

### *Partie 5 : * [Pratique] Les capteurs*

**Secret** (cliquez pour afficher)

Attention, toute cette liste pourrait changer ! (d'ailleurs elle manque volontairement de précision sur les valeurs des composants)

| Prix unitaire |
|---|

| Désignation | Quantité | Prix unitaire indicatif (€) | Photo |
|---|---|---|---|
| Photorésistance | 1 | 1.00 |  |
| Thermistance (CTN) | 1 | 1.00 |  |
| Capteur de choc (tilt) | 1 | 3.00 |  |
| Capteur de distance Sharp GP2D120 | 1 | 20.00 |  |
| **Total €** | | **25 €** | |

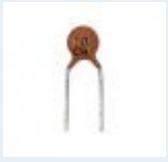
### *Partie 6 : * [Pratique] Les moteurs*

**Secret** (cliquez pour afficher)

Liste pas encore définie, désolé !

### *Partie 7 : [Pratique] L'affichage*

**Secret** (cliquez pour afficher)

| Désignation | Valeur - Caractéristique | Quantité | Prix unitaire indicatif (€) | Photo |
|---|---|---|---|---|
| LED | rouge | 1 | 0.10 |  |
| Résistance | 10 kOhm | 2 | 0.10 |  |
| | entre 220 et 470 Ohm | 1 | 0.10 | |
| Condensateur | 10 nF | 2 | 0.30 |  |
| Potentiomètre linéaire | 10 kOhm | 1 | 0.40 |  |

| | | | | |
|---|---|---|---|---|
| Écran LCD alphanumérique | 16*2<br>20*4<br>(valeur au choix) | 1 | 10 |  |
| Bouton Poussoir | - | 2 | 0.40 |  |
| **Total €** | | | **12.2 €** | |

### *Partie 8 : * [Théorie] Processing et Arduino*

**Secret** (cliquez pour afficher)

Liste pas encore définie, désolé !

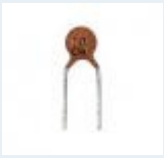
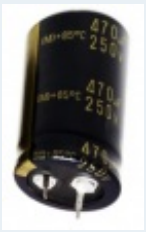
### *Partie 9 : * [Théorie] Arduino et internet*

**Secret** (cliquez pour afficher)

Liste pas encore définie, désolé !

## Liste Globale !

| Désignation | Quantité | Prix unitaire indicatif (€) | Photo | Description |
|---|---|---|---|---|
| LED rouge | 7 | 0.10 |  | Ce composant est une sorte de lampe un peu spécial. Nous nous en servirons principalement pour faire de la signalisation. |
| LED verte | 3 | | | |
| LED jaune (ou orange) | 2 | | | |
| Résistance (entre 220 et 470 Ohm) | 10 | 0.10 |  | 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 | 2 | | | |

| | | | | |
|---|---|---|---|---|
| kOhm) | | | | pour d'autres choses. |
| Résistance (10 kOhm) | 2 | | | |
| Bouton Poussoir | 2 | 0.40 |  | 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 | 0.60 |  | 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 | 1.00 |  | Un afficheur 7 segments est un ensemble de LEDs (cf. ci-dessus) disposées géométriquement pour afficher des chiffres. |
| Décodeur BCD (MC14543) | 1 | 1.00 |  | 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 | 0.30 |  | 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 | 1 |  | Celui-ci est un plus gros réservoir que le précédent |
| Potentiomètre linéaire (10 kOhm) | 1 | 0.40 |  | Le potentiomètre est une résistance que l'on peut faire varier manuellement. |
| LED RVB | 1 | 3.00 |  | Une LED RVB (Rouge Vert Bleu) est une LED permettant de mélanger les couleurs de bases pour en créer d'autres. |
| Écran LCD alphanumérique | 1 | 10 |  | L'écran LCD alphanumérique permet d'afficher des caractères tels que les chiffres et les lettres. Il va apporter de l'interactivité à vos projets les plus fous ! |
| *Module XBEE | 2 | - |  | Ce module permet de faire de la transmission sans fil, faible distance/consommation/débit/prix. |
| **Total €** | | **22.6 €** | | |

## Les revendeurs

Vous pourrez trouver ces composants chez :

- Selectronic
- Lextronic
- Electronique diffusion
- Radiospares
- Farnell
- Conrad

Ou dans un magasin électronique proche de chez vous (et pas de frais de port) !

Vous trouverez une liste non exhaustive des boutiques en ligne ou en magasin de matériel électronique sur ce forum dédié.

## Les kits

Enfin, il existe des kits tout prêts chez certains revendeurs. Nous n'en conseillerons 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 !

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 certainement toujours quelqu'un qui sera là pour vous guider.
Éventuellement nous ouvrirons un post fixe sur les différents kits pour les comparer (sans donner notre avis afin de rester objectif et car on a pas les moyens de les acheter et tester leur qualité !)

À vos achats, prêts ? Partez !

# Quelques bases élémentaires

En attendant que vous achetiez votre matériel, je vais vous présenter les bases de l'électronique et de la programmation en électronique. Nous allons voir un peu comment fonctionne l'électricité, pour ensuite nous pencher sur la programmation de l'électronique.

Étant un adepte de l'apprentissage par la pratique, ce chapitre aura de très pauvres notions, mais le cours sera enrichi de manipulations diverses qui vous feront apprendre à utiliser le système Arduino et l'électronique.

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. 

## L'électronique

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. Par exemple, la pression atmosphérique est une grandeur physique, ou bien la vitesse à laquelle circule une voiture en est aussi une. En électronique, nous ne mesurons pas ces grandeurs-là, nous avons nos propres grandeurs, qui sont : **le courant** et **la tension**.

### *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.

Retenez bien ce qui vient d'être dit, car c'est fondamental pour comprendre la suite.

## Le courant électrique

### *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 qui est **un déplacement ordonné de charges électriques.**

### *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

### *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. Ceci, c'est le **sens conventionnel** du courant.

On note le courant par une flèche qui indique le sens conventionnel de circulation du courant :

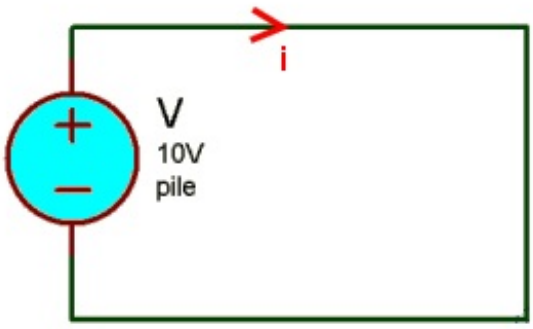



*Figure 1 : Indication du sens du courant*

### *Intensité du courant*

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.

## 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 et 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 créer 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'a 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 se serait vous embrouiller. 

### *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 orientée toujours dans le sens du courant aux bornes d'un générateur et toujours opposée au courant, aux bornes d'un récepteur :

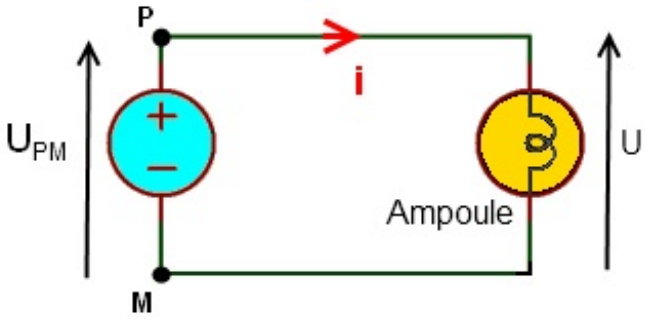



*Figure 2 : Fléchage de la tension*

### *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.

## La masse

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

### *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.

Et bien, ce chiffre zéro est la référence qui vous permet de vous mesurer. En électronique, cette référence existe, on l'appelle la **masse**.

### *Qu'est ce que c'est ?*

La masse, et bien 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.

### *Représentation et notation*

Elle se représente par ce symbole, sur un schéma électronique :

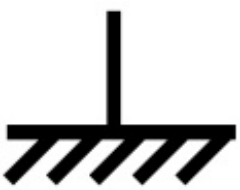


*Figure 3 : Symbole de la masse*

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 un diminutif 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.

#### *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.

## La résistance

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 essentiel en électronique.

#### *Présentation*

C'est le composant le plus utilisé en électronique. Sa principale fonction est de réduire l'intensité du courant.

Ce composant se présente sous la forme d'un petit boitier fait de divers matériaux et repéré par des anneaux de couleur indiquant la valeur de cette dernière. Photo de résistance :

*Figure 4 : Photo de résistance*

#### *Symbole*

Le symbole de la résistance ressemble étrangement à la forme de son boitier :

*Figure 5 : Symbole de la résistance*

#### *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 $\Omega$

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

#### *Unité*

L'unité de la résistance est l'**ohm**. On le note avec le symbole oméga majuscule : $\Omega$ .

### *Le code couleur*

La résistance possède une suite d'anneaux de couleurs différentes sur son boitier. 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 % |
| **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 % |

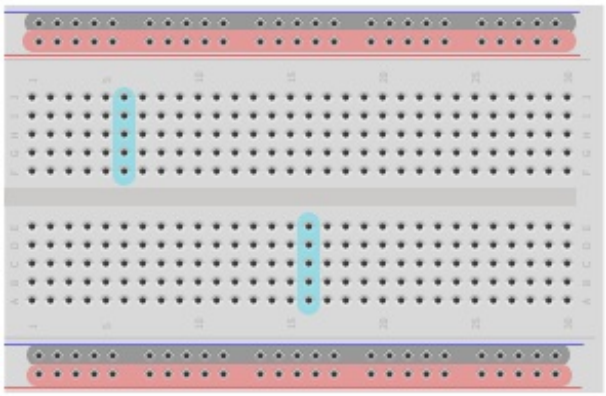
Bon, pour l'instant vous savez l'essentiel. On approfondira un peu dans la suite du cours. Parlons de programmation maintenant. 

## 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 !

### *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 vos 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. Cette espace coupe aussi la liaison des colonnes. Ainsi, sur le dessin ci-dessus on peut voir que chaque colonne possède 5 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.

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.

## La programmation

## Qu'est-ce qu'un programme

Il faut préciser que nous allons parler de **programme informatique** et non de programme télé !

En informatique, on utilise ce qu'on appelle des programmes informatiques. Pour répondre à la question, je dirai par analogie qu'un programme informatique est une « liste » d'informations (comme celle que vous avez pour préparer un diner) qui indique à l'ordinateur un certain nombre de tâches qu'il doit effectuer. Prenons votre lecteur multimédia qui est un programme informatique. Ce programme est donc une « liste d'informations » lue par votre ordinateur. Elle lui indique qu'il doit lire de la musique stockée sur votre disque dur.

Nous nous allons **créer des programmes**, ou bien **programmer**.

Voici quelques exemples de programmes informatiques :

- Votre navigateur Web (Internet Explorer, Firefox, Chrome, ...)
- Votre lecteur multimédia (VLC, Windows Media Player, ...)
- Votre antivirus (avast!, antivira, ...)

L'objectif de ce cours n'est pas de vous apprendre à faire votre propre navigateur web, ou votre propre système d'exploitation, non ce serait bien trop difficile et l'intérêt resterait plutôt restreint. Je vais vous apprendre à faire des programmes qui vont être exécutés par une carte électronique. Le but étant de vous former à la programmation de cette carte qui vous permettra par la suite de réaliser vos propres applications.

### *Créer un programme informatique*

Ecrire un programme informatique ne s'improvise pas comme ça ! Il faut d'abord savoir en quel langage il s'écrit et apprendre la syntaxe de ce langage.

Tiens ! Mais qu'est-ce qu'un langage informatique ?

Un langage informatique est un langage qui va vous permettre de « parler » à votre ordinateur. Reprenons notre analogie avec la liste de préparation au diner de ce soir. Sur cette liste, vous indiquez avec des mots ce que vous devez préparer pour ce diner. Ces mots sont écrits en langue française, mais pourraient très bien être en anglais ou en japonais. Cependant, ce n'est ni avec du français ni avec de l'anglais et encore moins avec du japonais que nous écrirons un programme informatique. Nous écrirons le programme avec un langage informatique, c'est-à-dire avec un langage que l'ordinateur peut comprendre.

Il existe, comme pour les langues, une diversité assez impressionnante de langage informatique. Heureusement, nous ne devrons en apprendre qu'un seul. Ouf ! Le langage que nous devrons apprendre s'appelle le **langage Arduino**.

### *Le compilateur*

Tout à l'heure, quand je vous disais que l'ordinateur comprenait le langage Arduino, je vous ai menti. Soyez sans crainte, ce n'est pas bien grave car j'ai seulement omis de préciser un détail !

En fait, l'ordinateur ne comprend pas directement le langage Arduino. En effet, l'ordinateur ne résonne qu'avec des états logiques. On parle d'états **binaires**, car ils ne peuvent prendre que deux valeurs : « 0 » ou « 1 ».

Voilà un exemple qui va vous effrayer : sachez que nous utiliserons des mots en provenance de la langue anglaise pour écrire un programme informatique (non ce n'est pas ça qui est effrayant ! ), mais comme l'ordinateur ne comprend pas les lettres et les chiffres (juste 0 et 1), nous devons écrire chaque mot en code binaire. Par exemple, la lettre « A » majuscule s'écrit en binaire : 1000001 ; et la lettre « m » minuscule : 1101101. Alors imaginez seulement si vous deviez transcrire le mot « Anticonstitutionnellement » en binaire !

Heureusement, des ~~fous~~ ingénieurs en informatiques ont créé ce qu'on appelle le **compilateur**. C'est en fait un programme informatique qui va transcrire à notre place les mots en langage binaire. C'est donc le traducteur qui se chargera de traduire le langage Arduino (que nous allons apprendre prochainement) en langage binaire (compréhensible par l'ordinateur). Ce traducteur est le logiciel Arduino, dont nous allons parler dans un prochain chapitre.

## La programmation en électronique

Au jour d'aujourd'hui, l'électronique est de plus en plus remplacée par de l'**électronique programmée**. On parle aussi d'**électronique embarquée** ou d'**informatique embarquée**. Son but est de simplifier les schémas électroniques et par conséquent réduire l'utilisation de composants électroniques, réduisant ainsi le cout de fabrication d'un produit. Il en résulte des systèmes plus complexes et performants pour un espace réduit.

### *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 ?

### *Le microcontrôleur*

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'entrerai pas dans l'utilisation poussée de ces derniers car le niveau est rudement élevé et la compréhension difficile.

Voici la photo d'un microcontrôleur :

*Figure 6 : Photo de microcontrôleur*

### *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 constitué des mêmes éléments que sur la carte mère d'un ordinateur. Si on veut, c'est un ordinateur (sans écran, sans disque dur, sans lecteur de disque) dans un espace très restreins.

Je vais vous présenter les différents éléments qui composent un microcontrôleur typique et uniquement ceux qui vont nous être utiles.

**La mémoire**
Il en possède 4 types :

- 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 qu'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. Cette mémoire ne s'efface pas lorsque l'on éteint le microcontrôleur ou lorsqu'on le reprogramme.
- Les registres : c'est un type 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.

**Le processeur**
C'est le composant principal du micro-contrôleur. 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 micro-contrôleur, mais pas tout de suite, c'est bien trop compliqué. Je ne voudrais pas perdre la moitié des visiteurs en un instant ! 

### *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.

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 tension 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.

## Les bases du comptage (2,10,16...)

## Les bases ~~du~~ de comptage

On va apprendre à compter ? 

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**.

### *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 :

$$1 \times 10^3 + 0 \times 10^2 + 2 \times 10^1 + 4 \times 10^0 = 1024$$

Et bien c'est ça, compter en base 10 ! Vous allez mieux comprendre avec la partie suivante.

### *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. Et 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 :

$$1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1$$

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.

### *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 : $(A2)_{16}$

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

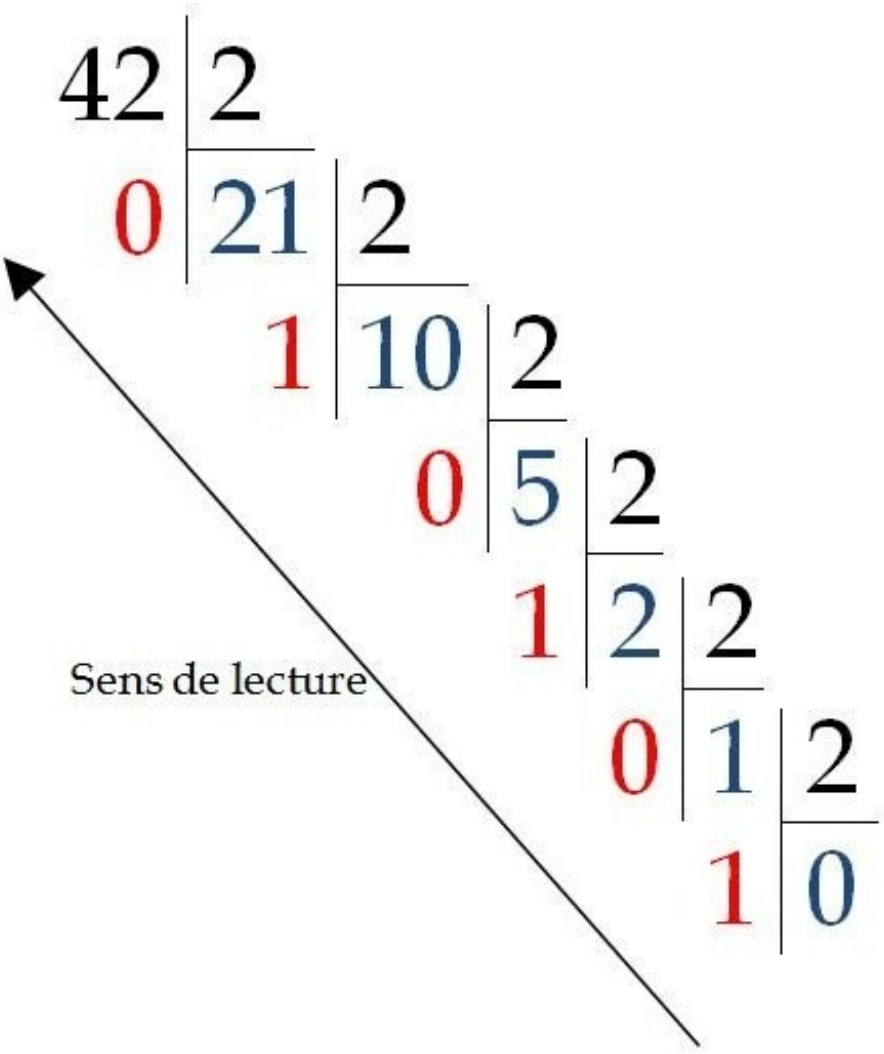
## Conversions

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.

### *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 :




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

### *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. Si 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 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)

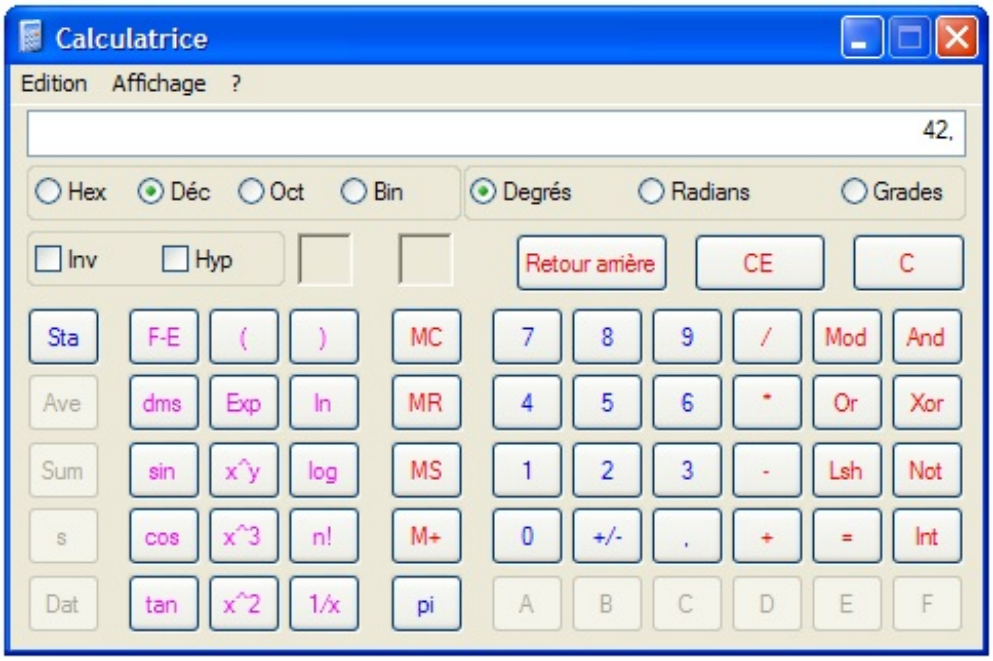
### *Décimal <-> Hexadécimal*

Ce cas est plus délicat à traiter, car il nécessite de bien connaître la table de multiplication par 16. 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 !

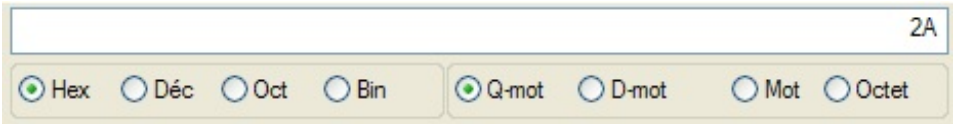
Pour en apprendre plus, vous pouvez suivre ce lien qui explique de façon plus complète ce qui vient d'être dit maintenant.

### *Méthode rapide*

Pour cela, je vais dans *Démarrer / Tous les programmes / Accessoires / Calculatrice* . Qui a dit que j'étais fainéant ?

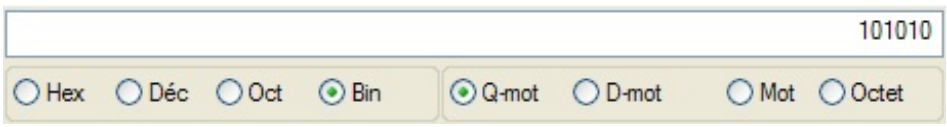



Vous voyez en haut qu'il y a des options à cocher pour afficher le nombre entré dans la base que l'on veut. Présentement, je suis en base 10 (décimale - bouton *Déc*). Si je clique sur *Hex* :




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**

Oui, c'est vrai ça. Pour quoi on a pas commencé par expliquer ça ? Qui sait.
Maintenant que vous avez acquis les bases essentielles pour continuer le cours, nous allons voir comment se présente le matériel que vous venez d'acheter et dont nous aurons besoin pour suivre ce cours.

# Le logiciel

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

## 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.

## Téléchargement

Pour télécharger le logiciel, il faut se rendre sur la page de téléchargement du site 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.

### *Sous Windows*

Pour moi ce sera sous Windows. Je clique sur le lien Windows et le fichier apparait :

Image utilisateur

*Figure 1 : Téléchargement du logiciel Arduino*

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

*Figure 2 : Exécutable du logiciel Arduino*

### *Mac os*

Cliquez sur le lien Mac OS. Un fichier **.dmg** apparait. Enregistrez-le.

*Figure 3 : Téléchargement sous Mac os*

Double-cliquez sur le fichier *.dmg* :

*Figure 4 : Contenu du téléchargement*

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.

#### *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:

**Code : Console**

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

Plusieurs dépendances seront installées en même temps.

Je rajoute un lien qui vous mènera vers la page officielle.

## Interface du logiciel

### Lancement du logiciel

Lançons le logiciel en double-cliquant sur l'icône avec le symbole "infinie" en vert. C'est l'exécutable du logiciel.

Après un léger temps de réflexion, une image s'affiche :

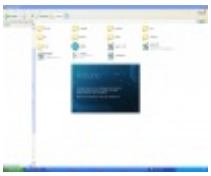


*Figure 5 : lancement du logiciel Arduino*

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

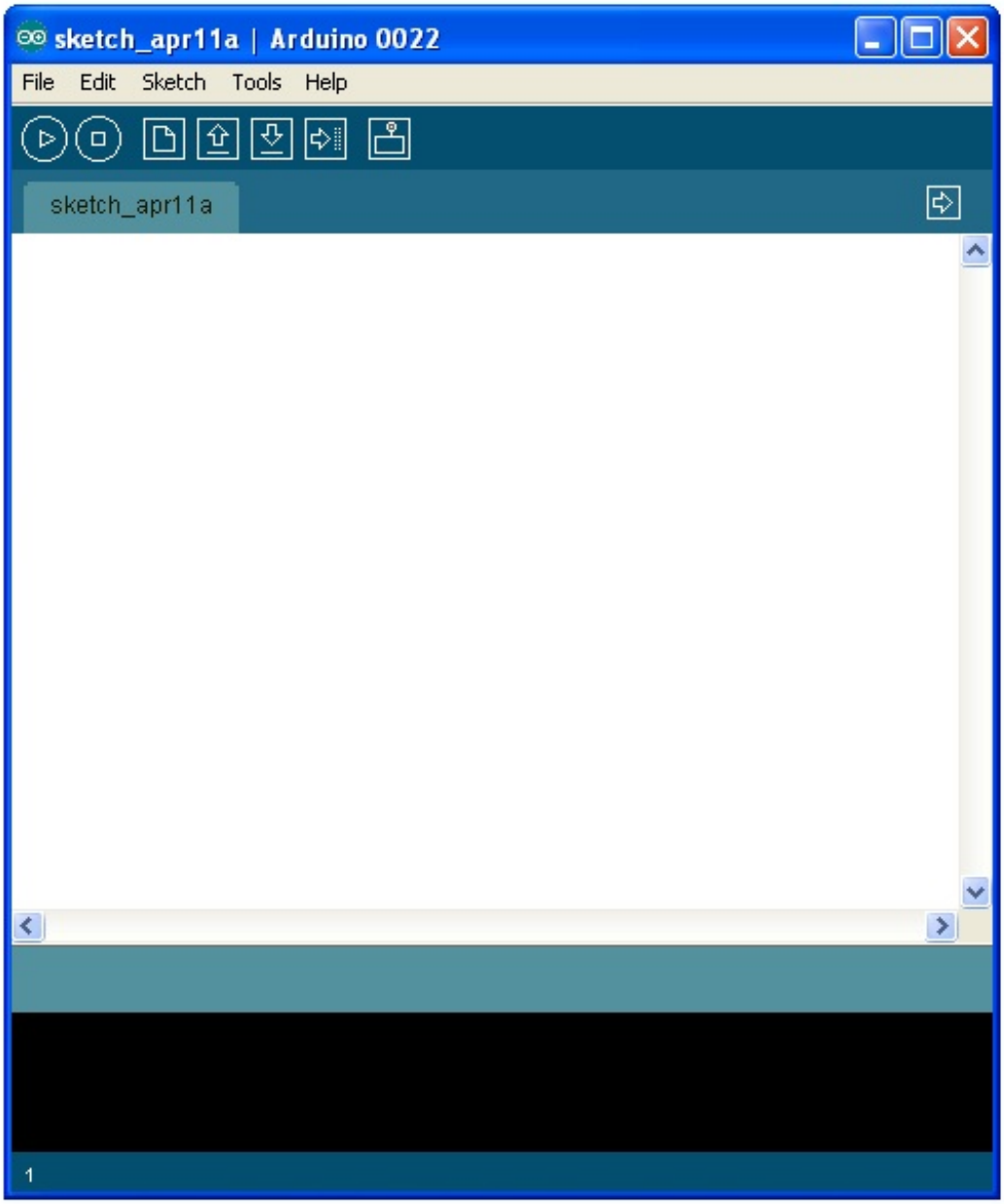



*Figure 6 : fenêtre du logiciel Arduino*

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.

## Présentation du logiciel

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

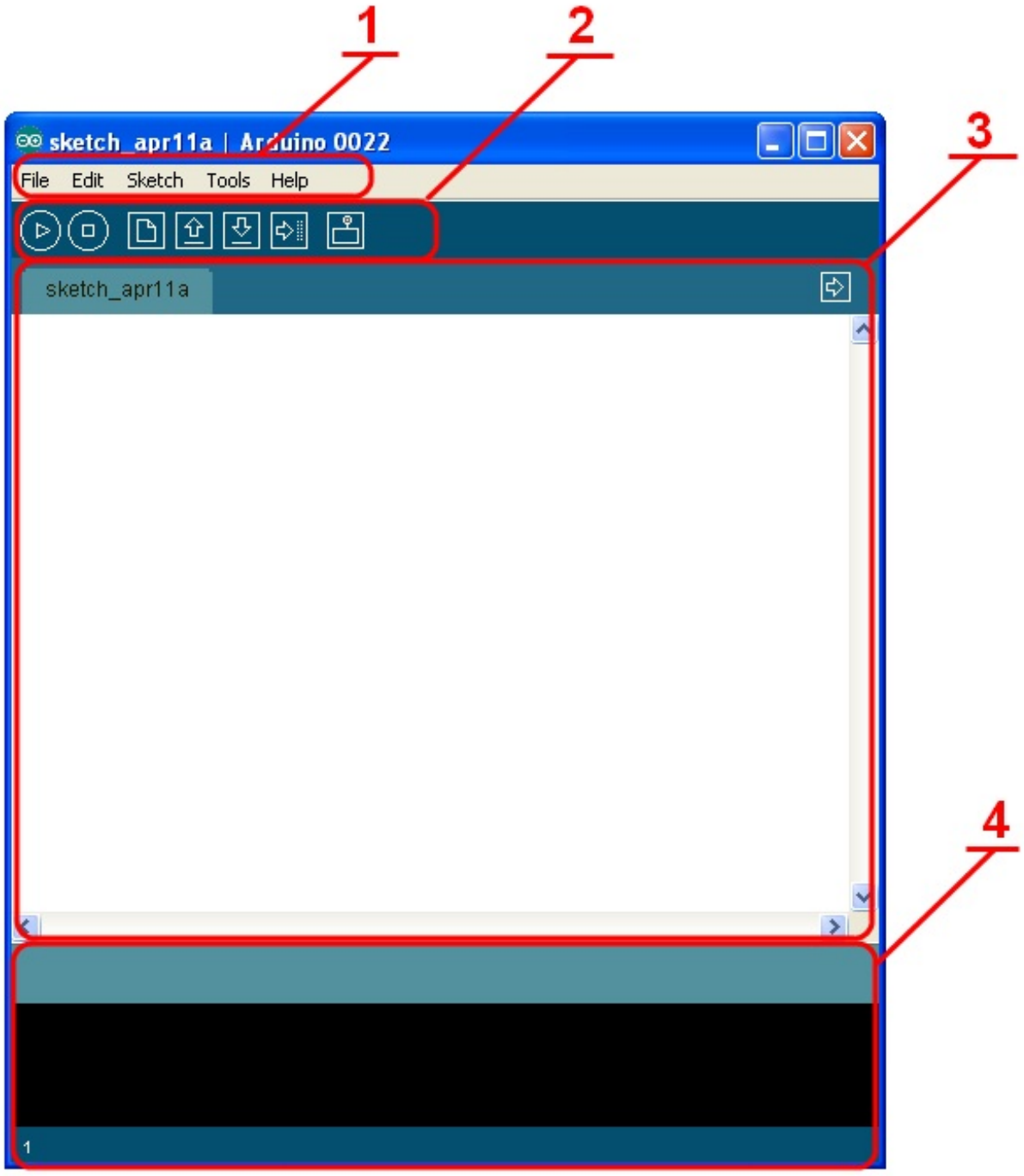



*Figure 7 : Présentation des parties principales du logiciel*

### *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**.

## 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.

## 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 :

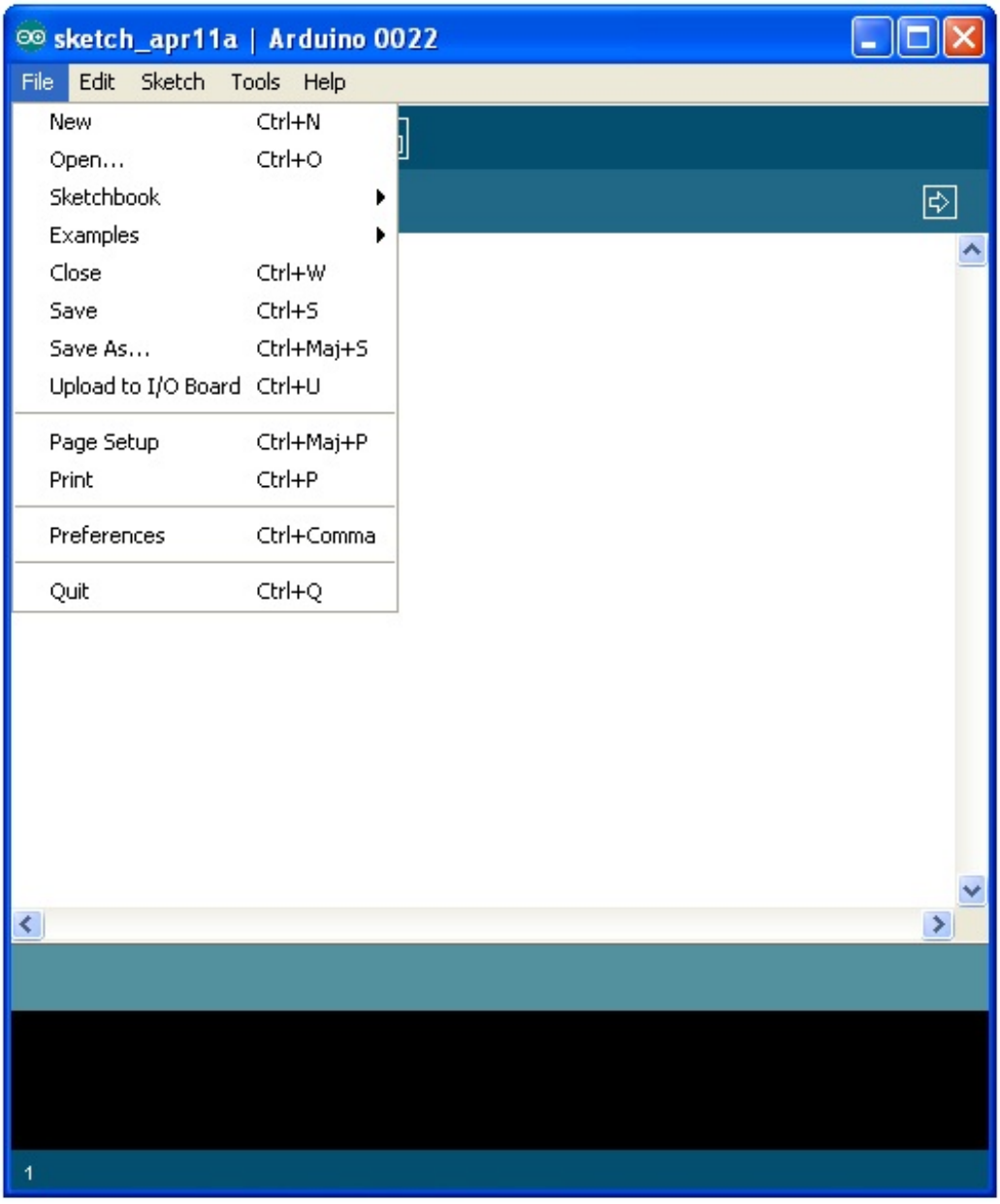



*Figure 8 : contenu du menu "File"*

- *New* (nouveau) : va permettre de créer un nouveau programme. Quand on appuie sur ce bouton, une nouvelle fenêtre, identique à celle-ci, s'affiche à l'écran
- *Open...* (ouvrir) : avec cette commande, nous allons pouvoir ouvrir un programme existant
- *Save* / *Save as...* (enregistrer / enregistrer sous...) : enregistre le document en cours / demande où enregistrer le document en cours
- *Examples* (exemples) : ceci est important, toute une liste se déroule pour afficher les noms d'exemples de programmes existants ; avec çà, vous pourrez vous aider pour créer vos propres programmes

Le reste des menus n'est pas intéressant pour l'instant, on y reviendra plus tard, avant de commencer à programmer.

## Les boutons

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

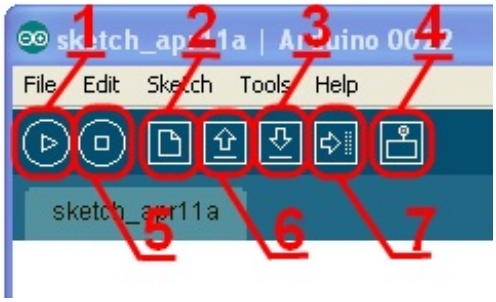



*Figure 9 : Présentation des boutons*

- Bouton 1 : Ce bouton permet de vérifier le programme, il actionne un module qui cherche les erreurs dans votre programme
- Bouton 2 : Créer un nouveau fichier
- Bouton 3 : Sauvegarder le programme en cours
- Bouton 4 : On n'y touche pas pour l'instant 
- Bouton 5 : Stoppe la vérification
- Bouton 6 : Charger un programme existant
- Bouton 7 : Compiler et envoyer le programme vers la carte

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

# Le matériel

J'espère que vous disposez à présent du matériel requis pour continuer le cours car dans ce chapitre, je vais vous montrer comment se présente votre carte, puis comment la tester pour vérifier son bon fonctionnement.

## 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.

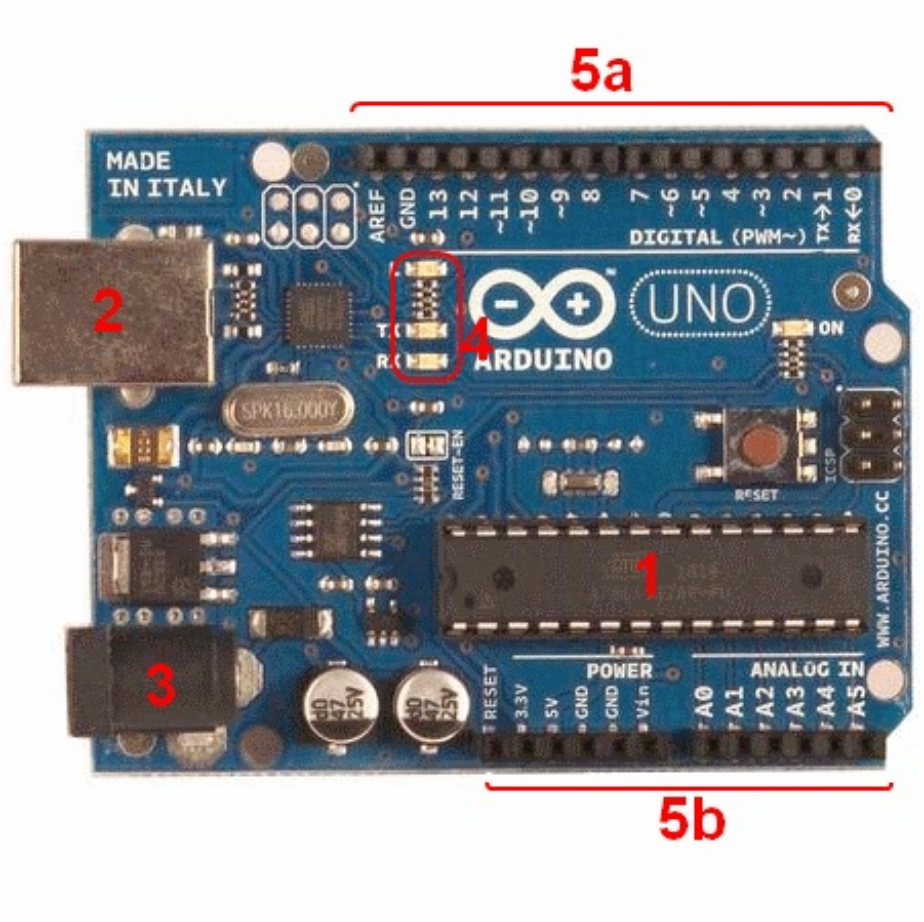



*Figure 1 : Présentation de la carte Arduino*

## Constitution de la carte

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

### *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 micro-contrôleur se faisant par cette voie, on peut les voir clignoter lors du chargement.

### *La connectique*

La carte Arduino ne possédant pas de composants qui peuvent être utilisés pour un programme, mis a par 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 le la connecter, avec une résistance en série, à la carte, sur les fiches de connections 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.

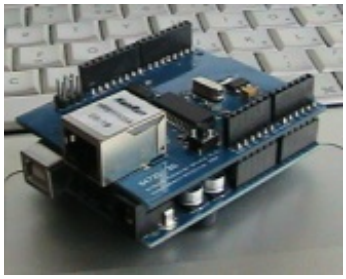


*Figure 2 : Une carte Arduino étendue avec un Ethernet Shield*

## Installation

Afin d'utiliser la carte, il faut l'installer. Normalement, les drivers 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.

## 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 Win 7 comme moi, il se peut que ca ne marche pas du premier coup. Dans ce cas, laisser la carte branchée puis ensuite allez dans le panneau de configuration. Une fois là, cliquez 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-jusqu'au-dossier]\arduino-0022\arduino-0022\drivers*

Il semblerait qu'il y est des problèmes en utilisant la version francaise 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 :

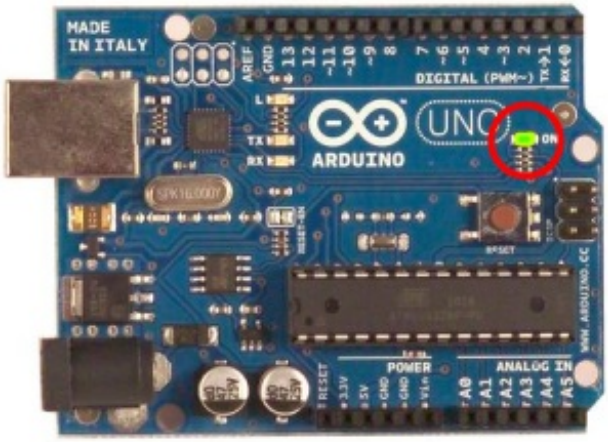



*Figure 3 : carte connectée et alimentée*

## Tester son matériel

Avant de commencer à programmer la tête baissée, il faut, avant toutes choses, tester le bon fonctionnement de la carte. Car 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.

Mais, on n'en a pas encore fait de programmes ?

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.

### *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* :

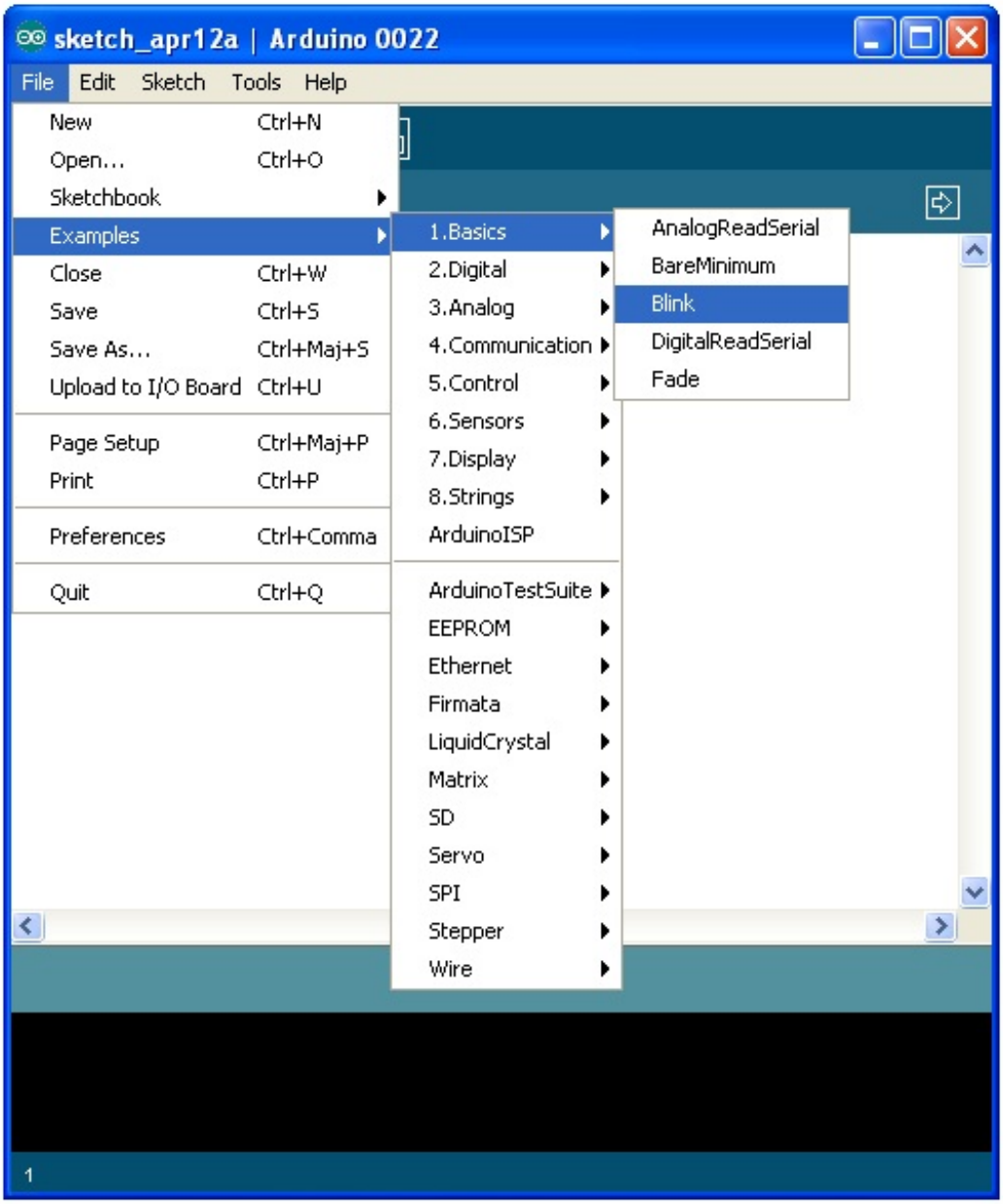



*Figure 4 : Ouvrir le programme Blink*

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.

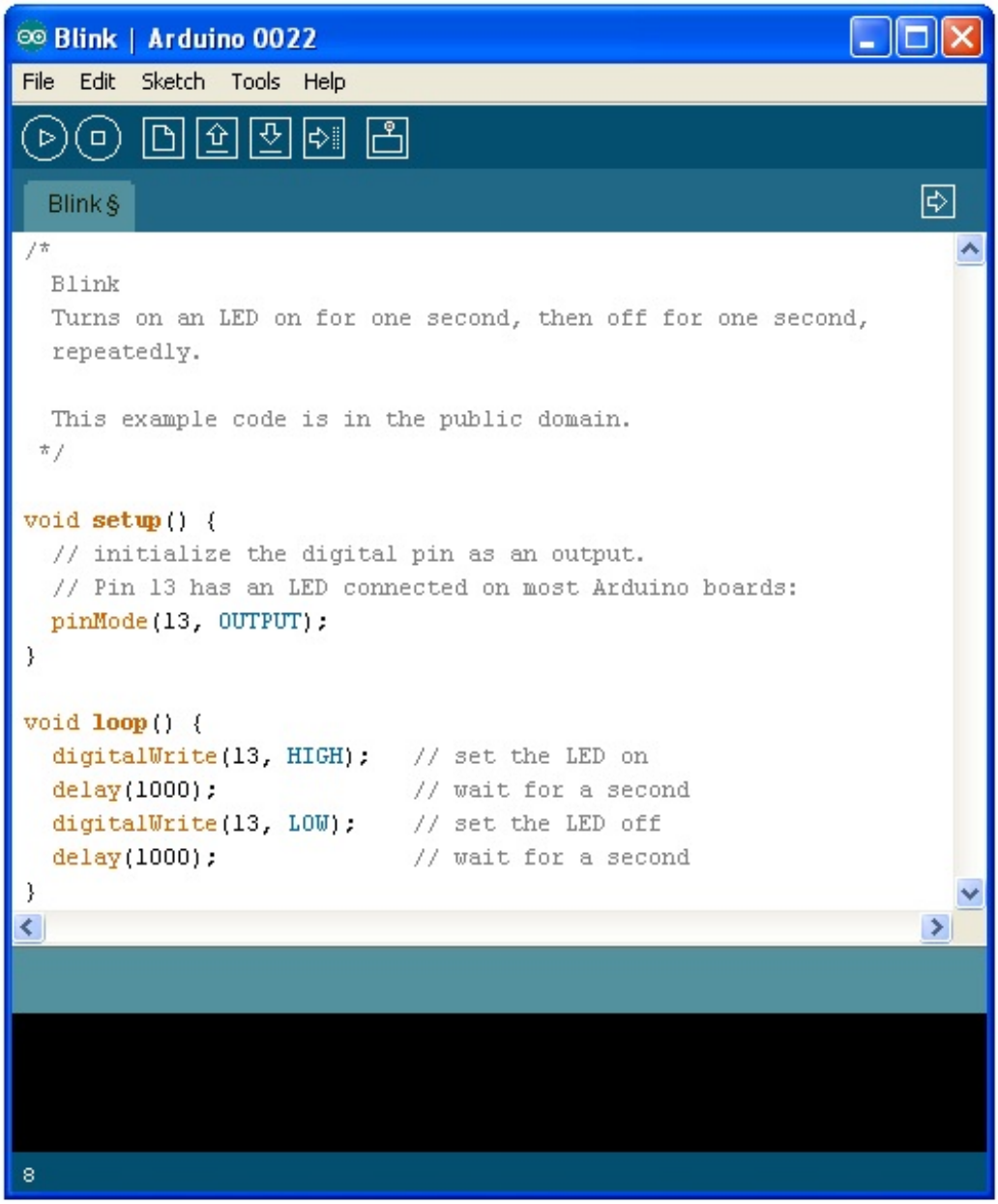



*Figure 5 : Contenu du programme Blink*

### 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.

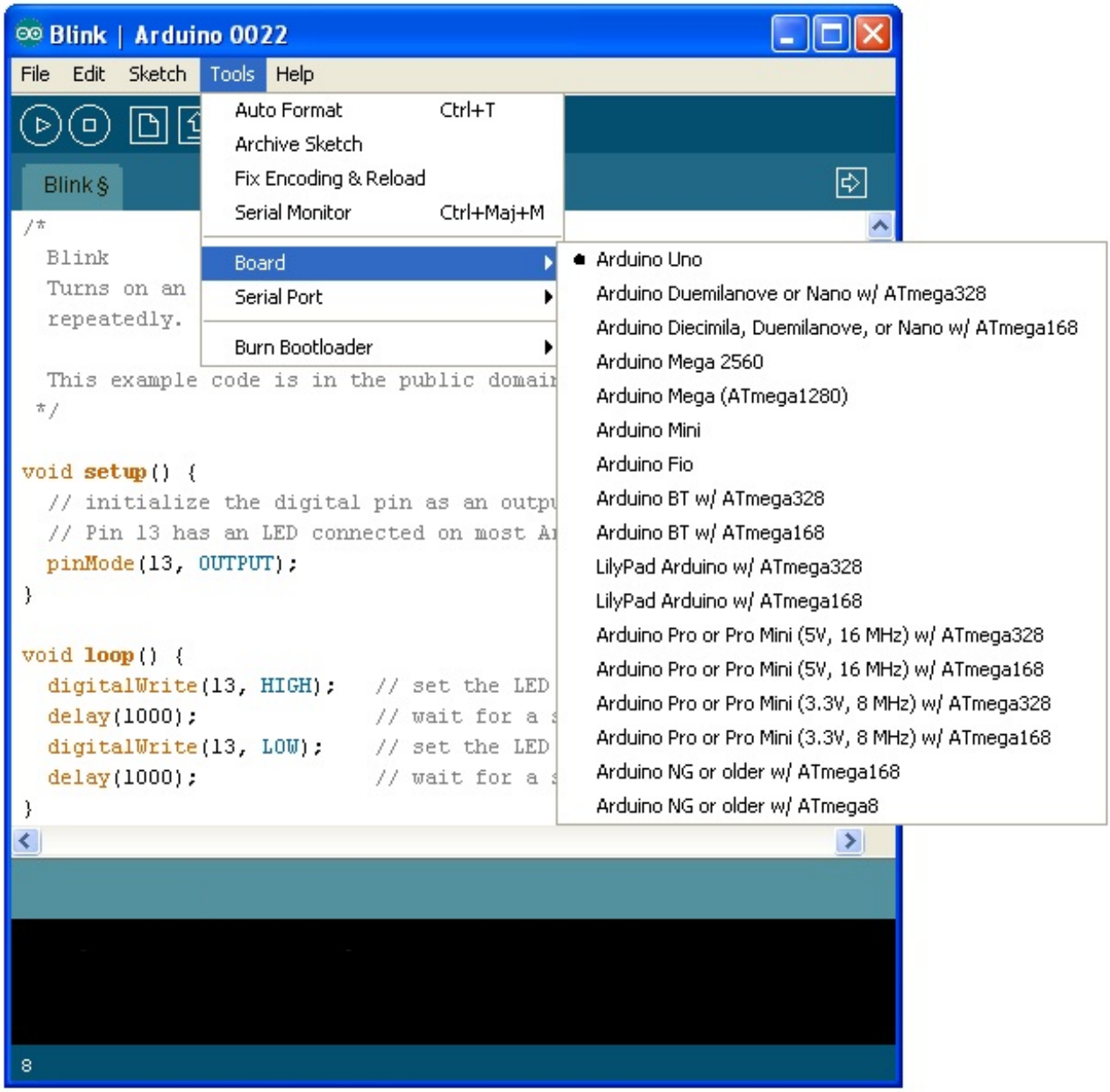



*Figure 6 : Choix de la carte Arduino*

***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 :

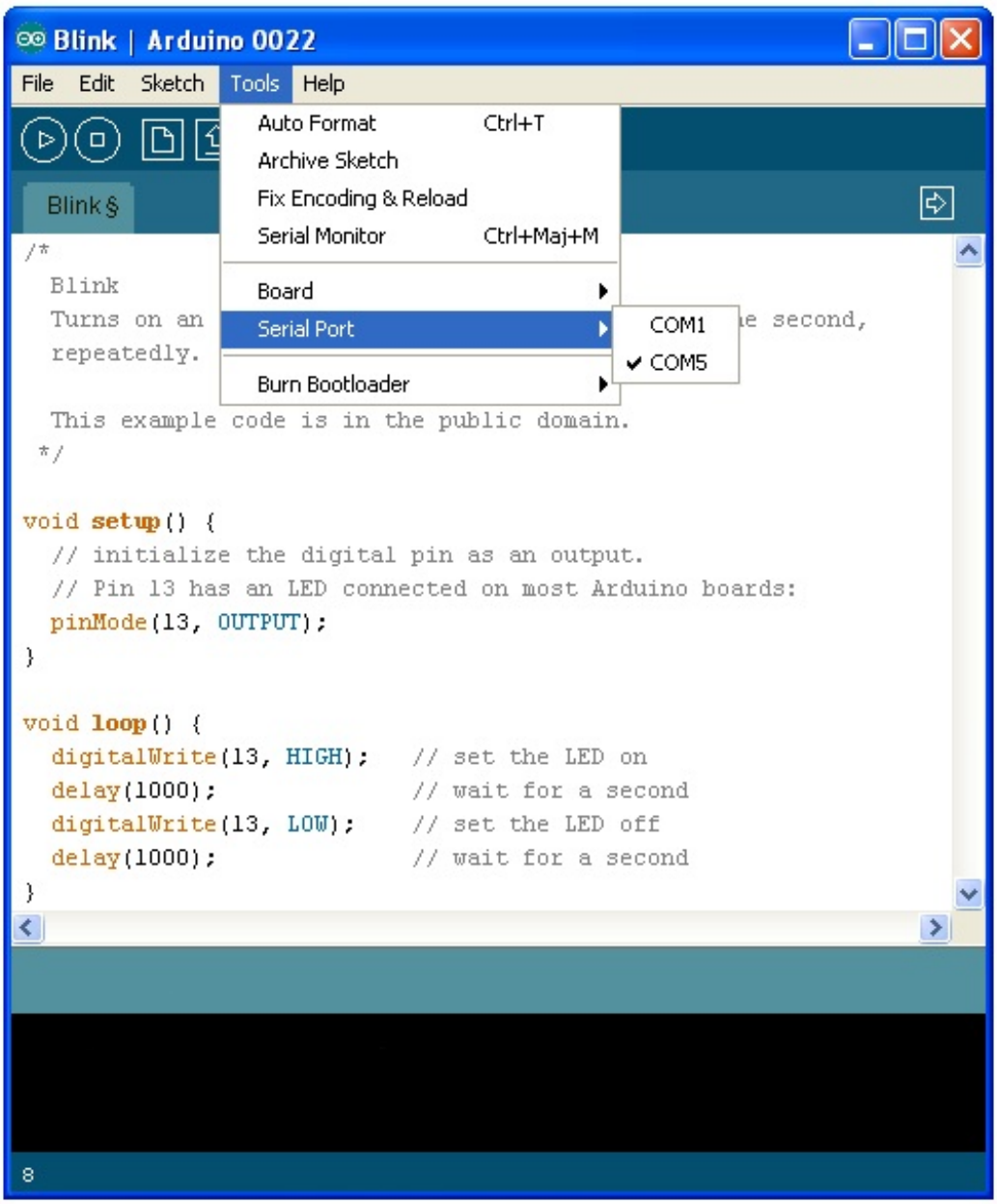



*Figure 7 : Choix du port de connexion de la carte*

Pour trouver le port de connexion de la carte, vous pouvez aller dans le ***gestionnaire de périphérique*** 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 :

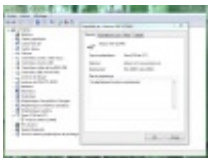


*Figure 8 : Recherche du port de communication de la carte* (Merci à sye pour cette image)

### *Dernière étape*

Très bien. Maintenant, il va falloir envoyer le programme dans la carte. Pour ce faire, il suffit de cliquer sur le bouton *Upload* (ou "Télécharger" en Français), en jaune-orangé sur la photo :

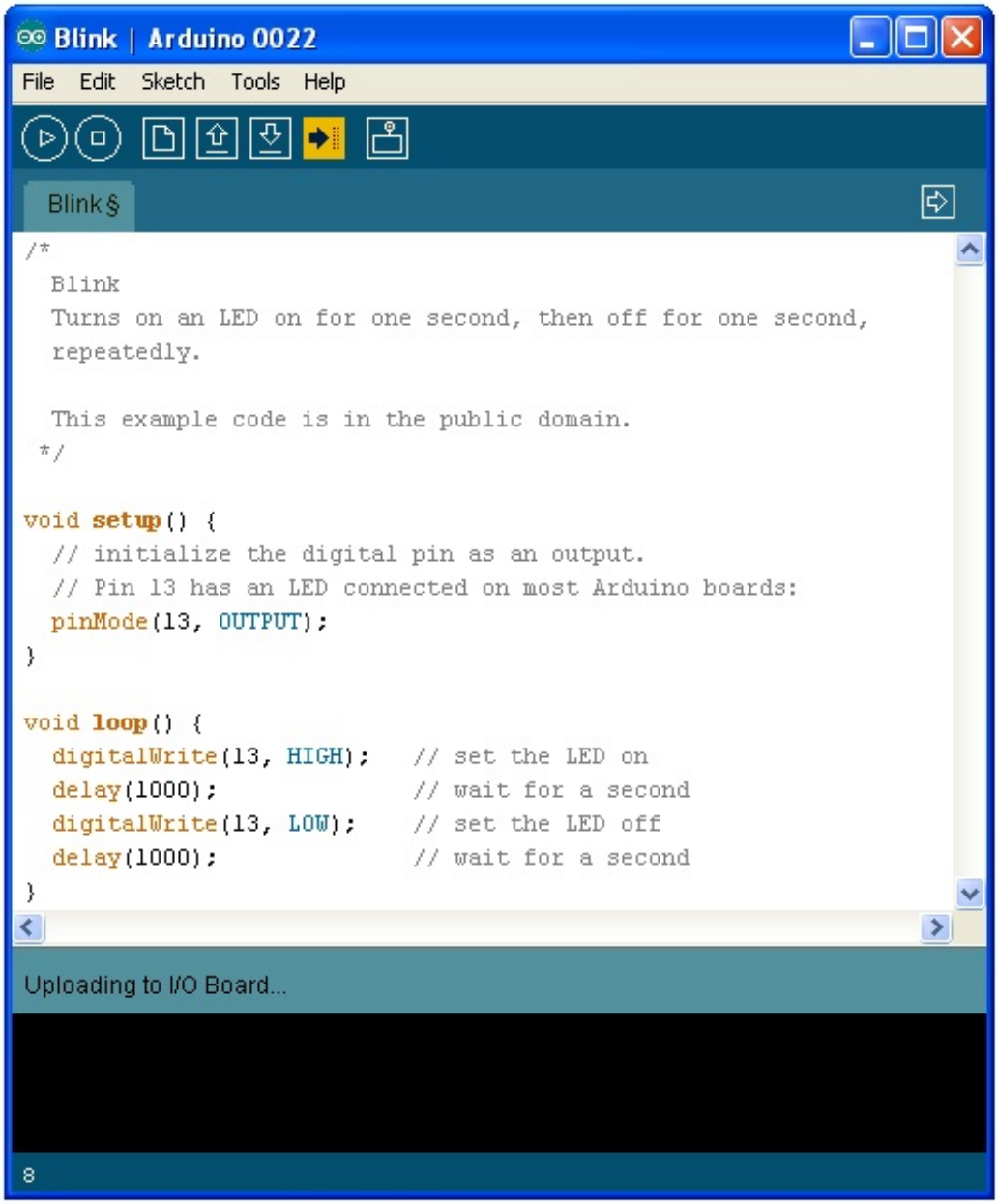



*Figure 9 : Envoi du programme Blink*

En bas dans l'image, vous voyez le texte : "*Uploading to I/O Board...*", cela signifie que le logiciel est en train d'envoyer le programme dans la carte. Une fois qu'il a fini, il affiche un autre message :

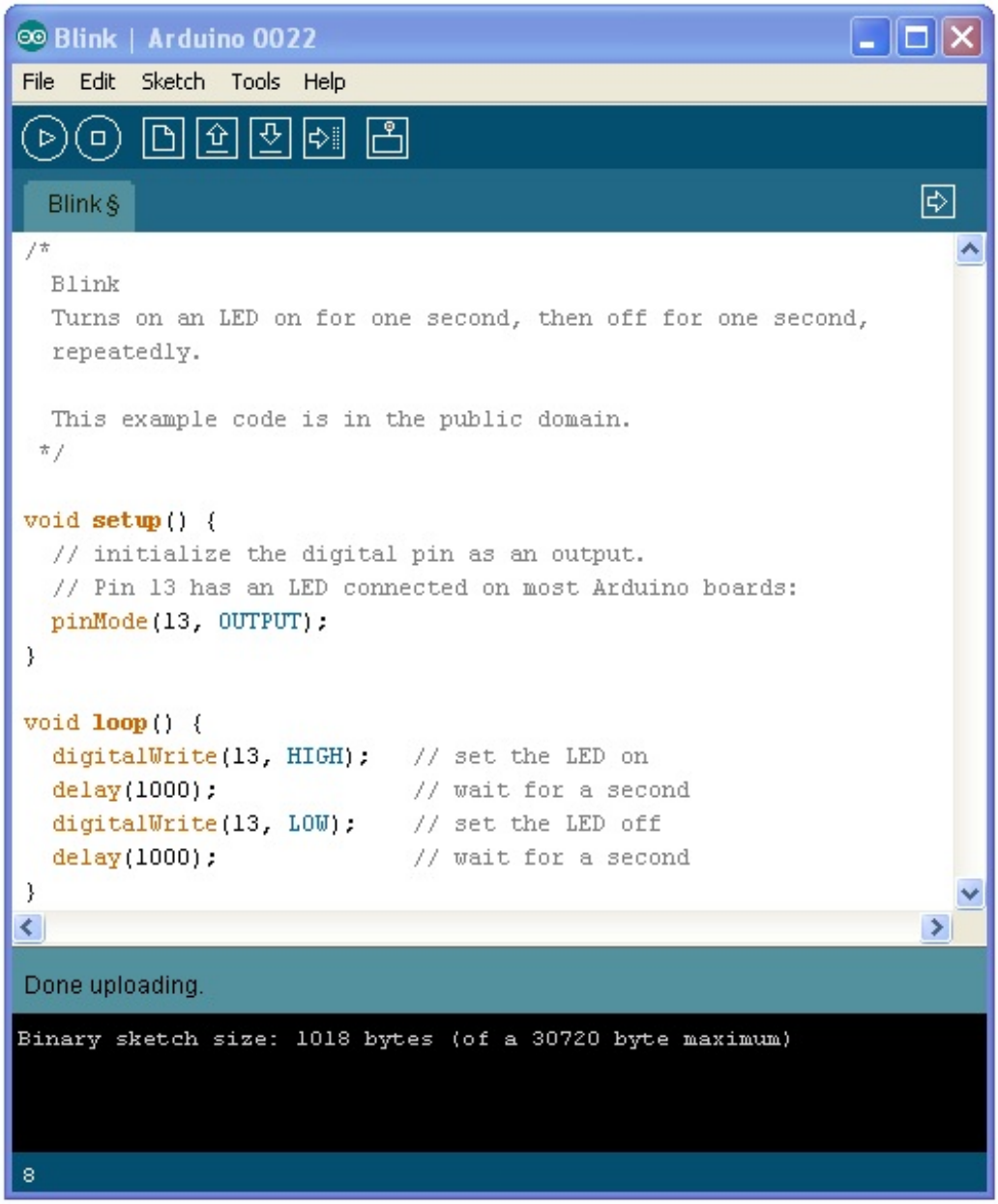



*Figure 10 : fin de l'upload*

Le message afficher : "*Done uploading*" signale que le programme à bien été chargé dans la carte. Si votre matériel fonctionne, vous devriez avoir une LED sur la carte qui clignote :

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, vérifier la présence d'erreur
- - La voie série est peut-être mal choisi, vérifier les branchements et le choix de la voie série
- - l'IDE est codé en JAVA, il peut-être capricieux et bugger de temps en temps (surtout avec la voie série...) : réessayez l'envoi!

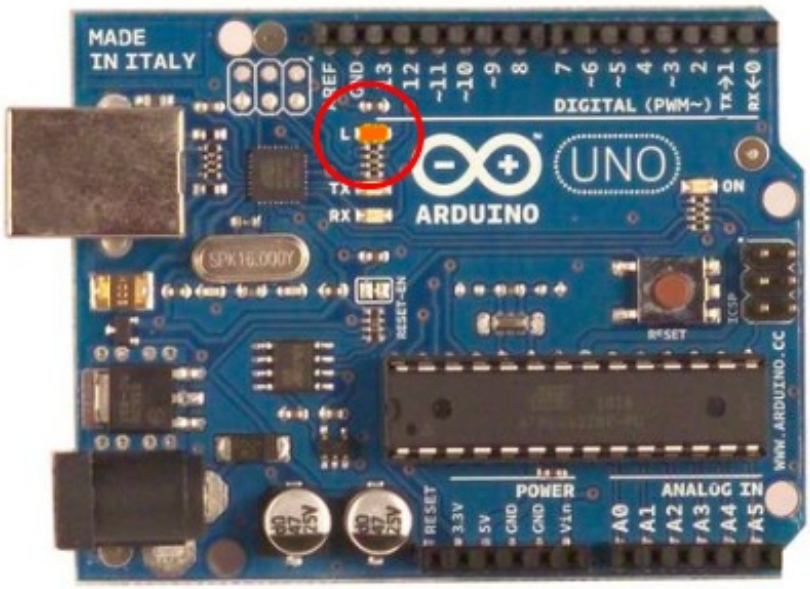



*Figure 11 : LED sur la carte qui clignote*

Toutes ces étapes, vous devrez les faire avant d'utiliser la carte pour vérifier son bon fonctionnement. C'est très important !

# Le langage Arduino (1/2)

Pour pouvoir programmer notre carte, il nous faut trois choses :

- Un ordinateur
- Une carte Arduino
- Et connaitre 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.

Je précise un petit aléa : le langage Arduino n'ayant pas la coloration de sa syntaxe dans le zCode, je le mettrai en tant que code C car leur syntaxe est très proche :

**Code : C**

```
//voici du code Arduino coloré grâce à la balise "code : C"
du zCode

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

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 chapitre sur le langage Arduino. Bien qu'il y ait des points quelques peu important.

## La syntaxe du langage

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

## 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.

**Code : C**

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

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

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 ?

### La fonction

Dans ce code se trouvent deux fonctions. Les fonctions sont en fait des portions de code.

**Code : C**

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

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 :

*- 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.

**Code : C**

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

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**.

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 !

### Les instructions

Dans ces fonctions, on écrit quoi ?

C'est justement l'objet de ce paragraphe.

Dans votre liste pour le diner 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.

### 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.

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.

### *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.

### *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.

Mais alors c'est inutile ? 

Non car cela va nous permettre à nous et aux programmeurs qui lirons 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.

Ligne unique de commentaire :

**Code : C**

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

Ligne ou paragraphe sur plusieurs lignes :

**Code : C**

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

### *Les accents*

Il est formellement interdit de mettre des accents en programmation. Sauf dans les commentaires.

## 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 un problème. 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) ?

### *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.

Chaussures rangées dans des cases numérotées

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 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 |

Une variable est un nombre, c'est tout ?

Ce nombre a la particularité de changer de valeur. Etrange 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..."

### *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)

### *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 :

**Code : C**

```
x = 4;
```

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 | $-3.4 \times 10^{38}$ **à** $+3.4 \times 10^{38}$ | 32 bits | 4 octets |
| **double** | décimale | $-3.4 \times 10^{38}$ **à** $+3.4 \times 10^{38}$ | 32 bits | 4 octets |

Par exemple, si notre variable "x" ne prend que des valeurs décimales, 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**.

**Code : C**

```
char x = 0;
```

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**.

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 il y en a plus, 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 d'octets |
|---|---|---|---|---|
| **unsigned** char | entier non négatif | 0 à 255 | 8 bits | 1 octets |
| **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 |

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 octets |
| **word** | entier non négatif | 0 à 65535 | 16 bits | 2 octets |
| **boolean** | entier non négatif | 0 à 1 | 1 bits | 1 octets |

Pour votre information, vous pouvez retrouver ces tableaux sur cette page.

### *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 :

**Code : C**

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

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 :

**Code : C**

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

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

**Code : C**

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

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

## 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 ! " 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 !

### *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 :

**Code : C**

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

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

Faisons maintenant une addition de variables :

**Code : C**

```
int x = 38;   //définition de la variable x et assignation à la
valeur 38
int y = 10;
int z = 0;
//faisons une addition avec un nombre choisi au hasard

z = x + y;    // on a donc z = 38 + 10 = 48
```

### *La soustraction*

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

**Code : C**

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

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

Soustraction de variables :

**Code : C**

```
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
```

### *La multiplication*

**Code : C**

```
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 toute autre opération) un nombre et
```

```
une variable :

z = z * ( 1 / 10 )    //soit z = 360 * 0.1 = 36
```

### La division

**Code : C**

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

x = 12 / 3;    // x vaut maintenant 12 / 3 = 4

z = x / y;     // on a donc z = 4 / 10 = 0.4
```

### 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.

**Code : C**

```
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
```

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

Autre exemple :

**Code : C**

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

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

## Quelques opérations bien pratiques

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

### L'incrémentation

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

**Code : C**

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

"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.

### *La décrémentation*

C'est l'inverse de l'incrémentation. Autrement dit, on enlève le chiffre 1 à la valeur de var.

**Code : C**

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

### *Les opérations composées*

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

**Code : C**

```
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;
```

Avec un exemple, cela donnerait :

**Code : C**

```
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
```

## 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 :

**Code : C**

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

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

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 :

- 1er temps : x = 1 - x soit x = 1 - 0 donc x = 1
- 2e temps : x = 1 - x or x vaut maintenant 1 donc x = 1 - 1 soit x = 0
- 3e 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) :

**Code : Console**

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

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 :

**Code : C**

```
x = !x;
```

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.

## Les conditions

### *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€.*

### *Quelques symboles*

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

| Symbole | A quoi il sert | Signification |
|---|---|---|
| == | Ce symbole, composé de deux égales, permet de tester l'égalité entre deux variables | ... est égale à ... |
| < | Celui-ci teste l'infériorité d'une variable par rapport à une autre | ...est inférieur à... |
| > | Là c'est la supériorité d'une variable par rapport à une autre | ...est supérieur à... |
| <= | teste l'infériorité ou l'égalité d'une variable par rapport à une autre | ...est inférieur ou égale à... |
| >= | teste la supériorité ou l'égalité d'une variable par rapport à une autre | ...est supérieur ou égal à... |
| != | teste la différence entre deux variables | ...est différent de... |

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

Comment décortiquer cette phrase ? Mmm... Ha ! Je sais !

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.

## If...else

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

### *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€.

**Code : C**

```
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€.

**Code : C**

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

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.

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 :

**Code : C**

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

### *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 :

**Code : C**

```
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
}
```

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

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 :

**Code : C**

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

### *else if*

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 ?

Bien sur 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* :

**Code : C**

```
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
}
```

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* :

**Code : C**

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

Le "else if" ne peut pas être utilisée toute seule, il faut obligatoirement qu'il y ait un "if" avant !

## 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 ? 

 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 |

### *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* :

**Code : C**

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

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

### *OU*

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

 Et oui, les programmeurs sont des flemmards ! 

Rappelons quelles sont ces conditions :

**Code : C**

```
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 :

**Code : C**

```
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
}
```

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".

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.

### *NON*

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 :

**Code : C**

```
int prix_voiture = 5500;

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

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.

## Switch

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

Voilà un exemple :

**Code : C**

```
int options_voiture = 0;

if(options_voiture == 0)
{
   //il n'y a pas d'options dans la voiture
}
if(options_voiture == 1)
{
   //la voiture a l'option GPS
}
if(options_voiture == 2)
{
   //la voiture a l'option climatisation
}
if(options_voiture == 3)
{
   //la voiture a l'option vitre automatique
}
if(options_voiture == 4)
{
   //la voiture a l'option barres de toit
}
if(options_voiture == 5)
{
   //la voiture a l'option décrottage de nez
}
else
{
   //retente ta chance ;-)
}
```

Ce code est indigérable ! C'est infâme ! Grotesque ! 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 :

**Code : C**

```
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 décrottage de nez
          break;
        default:
          //retente ta chance ;-)
          break;
    }
```

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 :

**Code : Console**

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

Si option_voiture vaut maintenant 5 :

**Code : Console**

```
la voiture a l'option décrottage de nez
```

L'instruction **break** est hyper importante, 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.

## 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 :

**Code : C**

```
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 :

**Code : C**

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

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

Cette ligne :

**Code : C**

```
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"

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.

# Le langage Arduino (2/2)

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 ?

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 débugger. Enfin, nous apprendrons à utiliser les tableaux qui nous seront très utiles.

Voilà le programme qui vous attend !

## Les boucles

### *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.

## 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.

**Code : C**

```
/* ICI, un bout de programme permet de faire les choses suivantes :
_ un capteur détecte la tombée de la nuit et la levée du jour
o Si c'est la nuit, alors on doit fermer le volet
o Sinon, si c'est le jour, on doit ouvrir le volet

_ le programme lit l'état du capteur qui indique si le volet est
ouvert ou fermé

_ enregistrement de cet état dans la variable de type String :
position_volet
o Si le volet est ouvert, alors : position_volet = "ouvert";
o Sinon, si le volet est fermé : position_volet = "ferme";
*/

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

### *Comment lire ce code ?*

En anglais, le mot *while* signifie "tant que". Donc si on lit la ligne :

**Code : C**

```
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).

### *Construction d'une boucle while*

Voilà donc la syntaxe de cette boucle qu'il faut retenir :

**Code : C**

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

### *Un exemple*

Prenons un exemple simple, réalisons un compteur !

**Code : C**

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

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

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

**Code : Console**

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

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 continu le programme. Dans notre cas, le programme se termine.

## 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 :

**Code : C**

```
do
{
   //les instructions entre ces accolades sont répétées tant que la condition est vrai

}while(/* condition à tester */);
```

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

Qu'est-ce que ça change ?

Et bien, dans une *while*, si la condition est vraie 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 :

**Code : C**

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

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 fausse. Donc **la boucle est au moins exécutée une fois** ! Et ce quelle que soit la véracité de la condition. En test cela donne :

**Code : Console**

```
compteur = 6
```

### *Concaténation*

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

**Code : C**

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

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

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 !

## 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.

**Code : C**

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

### *Fonctionnement*

**Code : C**

```
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

Le langage Arduino n'accepte pas l'absence de la ligne suivante :

**Code : C**

```
int compteur
```

On est obligé de déclarer la variable que l'on va utiliser (avec son type) **dans la boucle *for*** !

Donc, si on li cette ligne : "POUR compteur = 0 et compteur inférieur à 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.

### *A retenir*

La structure de la boucle :

**Code : C**

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

## 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.

**Code : C**

```
while(1)
{
    //instructions à répéter jusqu'à l'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" :

**Code : C**

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

Cela ne fonctionnera pas avec la valeur *0*. En effet, 0 signifie "condition fausse" donc la boucle s'arrêtera aussitôt...

La fonction loop() se comporte comme une boucle infinie, puisqu'elle se répète après avoir fini d'exécuter ses tâches.

## 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.

### *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 :

**Code : C**

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

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, ...). C'est

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.

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.

J'ai pas trop compris à quoi ça sert ?

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 à 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.

## 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 ?

### *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'appelez *tmp* (pour température ).

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 !

### *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**.

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.

### *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). 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 digitale" 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()` qui prendra en paramètre une variable de type *char* ou *int*, etc. (cela dépendra de la taille de la variable). Cette variable sera donc le paramètre de notre 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.

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.

## Les fonctions vides

On vient de voir qu'une fonction pouvait accepter des paramètres. Mais ce n'est pas obligatoire. Une fonction qui n'accepte pas de paramètres est une fonction vide.

La syntaxe utilisée pour définir une fonction vide est la suivante :

**Code : C**

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

On utilise donc le type `void` pour dire que la fonction n'aura pas de paramètres.

Une fonction de type void ne peut pas retourner de valeur. Par exemple :

**Code : C**

```
void setup()
{

}

void loop()
{
  fonction();
}

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

Ce code ne fonctionnera pas, parce que la fonction `fonction()` est de type void. Or elle doit renvoyer une variable qui est de type `int`. Ce qui est impossible !

Il n'y en a pas plus à savoir. 

## 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, ...)

### *Comment créer une fonction typée ?*

En soit, cela n'a rien de compliqué, il faut simplement remplacer `void` par le type choisi (`int`, `long`, ...)

Voilà un exemple :

**Code : C**

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

*Notez que je n'ai pas mis les deux fonctions principales, à savoir `setup()` et `loop()`, mais elles sont obligatoires !*

Lorsqu'elle sera appelée, la fonction `maFonction()` va tout simplement retourner la variable `resultat`. Voyez cet exemple :

**Code : C**

```
int calcul = 0;

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

int maFonction()
```

```
{
   int resultat = 44;  //déclaration de ma variable résultat
   return resultat;
}
```

Dans la fonction `loop()`, on fait un calcul avec la valeur que nous retourne la fonction `maFonction()`. Autrement dis, le calcul est : `calcul = 10 * 44;` Ce qui nous donne : calcul = 440.

Bon ce n'est qu'un exemple très simple pour vous montrer un peu comment cela fonctionne. Plus tard, lorsque vous serez au point, vous utiliserez certainement cette combinaison de façon plus complexe.

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.

## 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 :

**Code : C**

```
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;
}
```

Que se passe-t-il ?

J'ai défini trois variables : `somme`, `x` et `y`. La fonction `maFonction()` est "typée" et accepte des **paramètres**.

Lisons le code du début :

- On déclare nos variables
- La fonction `loop()` appelle 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 la signature de la fonction.

La signature... de quoi tu parles ?

La signature c'est le "titre complet" de la fonction. Grâce à elle on connait le **nom** de la fonction, le **type** de la valeur retourné, et le type des différents **paramètres**.

**Code : C**

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

La fonction récupère dans des variables les paramètres que l'on lui a envoyés. Autrement dit, dans la variable `param1`, on retrouve la variable `x`. Dans la variable `param2`, on retrouve la variable `y`.

Soit : `param1 = x = 64` *et* `param2 = y = 192`.

Pour finir, on utilise ces deux variables créées "à la volée" dans la signature de la fonction pour réaliser le calcul souhaité (une somme dans notre cas).

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 :

**Code : C**

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

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

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

```
        return resultat;
    }
```

Donc si la variable `operation` vaut 0, on addition les variables `x` et `y`, sinon si `operation` vaut 1, on soustrait `y` à `x`. Simple à comprendre, n'est-ce pas ?

## Les tableaux

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

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 !

### *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 |

### *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 `eleve1`, une autre `eleve2` et ainsi de suite pour chaque élève, on inscrit les notes des élèves dans un tableau.

Mais, concretement 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 précédente |

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. Y'a encore du boulot !

## 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 :

**Code : C**

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

On peut également créer un tableau vide, la syntaxe est légèrement différente :

**Code : C**

```
float notes[] = {};
```

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:

**Code : C**

```
char absenteisme[100];
```

## 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):

**Code : C**

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

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

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 '=' :

**Code : C**

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

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.

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

Vous avez sûrement rencontré des crashs 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 essai de manipuler des trucs qui ne lui appartiennent pas.

## Initialiser un tableau

Au départ, notre tableau était vide :

**Code : C**

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

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 valeur 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. Pour cela, on parcourt toutes les cases en leur mettant la valeur 0 :

**Code : C**

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

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

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

**Code : C**

```
float note[] = {0,0,0,0 /*, etc.*/ };
```

Ou bien même, comme cela :

**Code : C**

```
float note[] = {};

void setup()
{
   note[0] = 0;
   note[1] = 0;
   note[2] = 0;
   note[3] = 0;
   //...
}
```

## Exemple de traitement

Bon c'est bien beau tout ça, on a des notes coincées dans un tableau, on en fait quoi ? 

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  ).

### *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.

**Code : C**

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

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

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

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

**Code : C**

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

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

### *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 :

**Code : C**

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

**Une solution :**

**Secret** (cliquez pour afficher)

**Code : C**

```
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;
}
```

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...

Vous voilà fin prêt pour commencer à utiliser votre carte ! Alors rendez-vous à la prochaine partie du cours.

# Partie 2 : [Pratique] 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. On va commencer simplement, donc vous étonnez pas si vous allez vite dans la lecture des chapitres.

Ne négligez pas les bases, sans quoi vous risquez de ne pouvoir suivre les chapitres plus complexes ! Un conseil aussi, essayez de bien comprendre avant de passer au chapitre suivant, on ne fait pas la course, chacun fait à son rythme.

---> **Matériel nécessaire :** dans la balise secret pour la partie 2.

## 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.

### La diode électroluminescente

#### DEL / LED ?

La question n'est pas de savoir quelle abréviation choisir mais plutôt de savoir qu'est ce que c'est.

Une **DEL** / **LED** : **D**iode **E**lectro-**L**uminescente, ou bien "**L**ight **E**mitting **D**iode" en anglais. C'est un composant électronique qui crée de la lumière quand il est parcouru par un courant électrique. Je vous en ai fait acheter de différentes couleurs. Vous pouvez, pour ce chapitre, utiliser celle que vous voudrez, cela m'est égal. Vous voyez, 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.

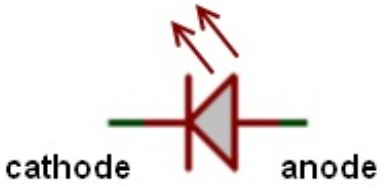
C'est ce composant que nous allons essayer d'allumer avec notre carte Arduino. Mais avant, voyons un peu comment il fonctionne.

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.

##### *Symbole*

Sur un schéma électronique, chaque composant est repéré par un symbole qui lui est propre. Celui de la diode est celui-ci :

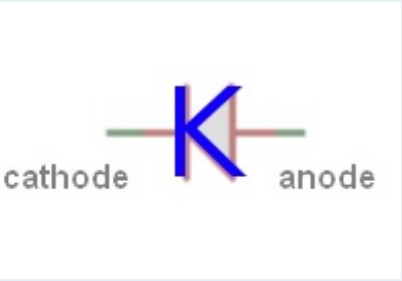
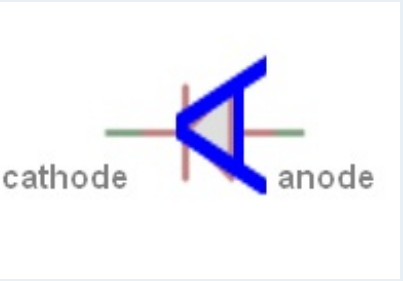

Celui de la LED est :




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.

### *Astuce mnémotechnique*

Pour ce souvenir de quel côté est l'anode ou la cathode, voici une toute simple et en image ...

|  |  |
|---|---|
| K comme K-thode | A comme A-node |

## Fonctionnement

### *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 doit la parcourir de l'anode vers la cathode. Autrement dit, la tension doit être plus élevée à l'anode qu'à la cathode.

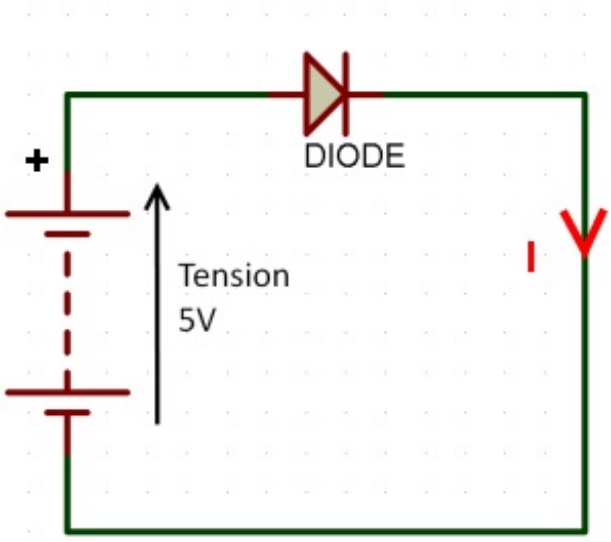



*Figure 1 : diode polarisée directement*

### *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.

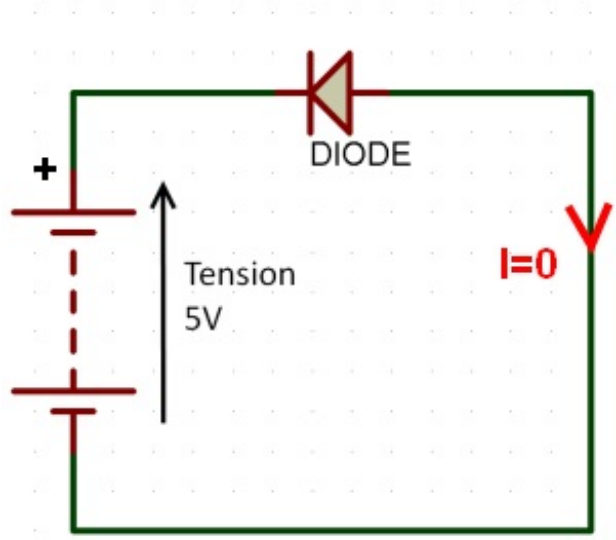



*Figure 2 : diode polarisée en inverse*

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.

### *Utilisation*

Si vous ne voulez pas faire partir votre première diode en fumée, je vous conseille de lire les prochaines lignes attentivement

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 à lui aussi son importance.

### *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.

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.

### *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 peu causer des dommages importants dans nos

appareils électroniques ! Quoi qu'il en soit, on ne manipulera jamais du 75V !

### *Le courant de passage*

Pour une LED, le courant qui la traverse à 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, savamment calculée, lui permettra d'assurer un fonctionnement optimal.

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 :

$$Ur = 5 - 1,2 = 3,8V$$

Enfin, on peut calculer la valeur de la résistance à utiliser :

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

$$R = \frac{3,8}{0,02}$$

$$R = 190\Omega$$

Et voila, vous connaissez la valeur de la résistance à utiliser pour être sur de ne pas griller des LED à tour de bras.

A votre avis, vaut-il mieux utiliser une résistance de plus forte valeur ou de plus faible valeur ?

**Secret** (cliquez pour afficher)

**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 admettrait un courant plus intense qui circulerait en elle et cela pourrait la détruire.

## Par quoi on commence ?

### *Le but*

Le but de ce premier programme est... de vous faire programmer ! 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 !

### *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 !

### *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.

## 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 placant 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) :

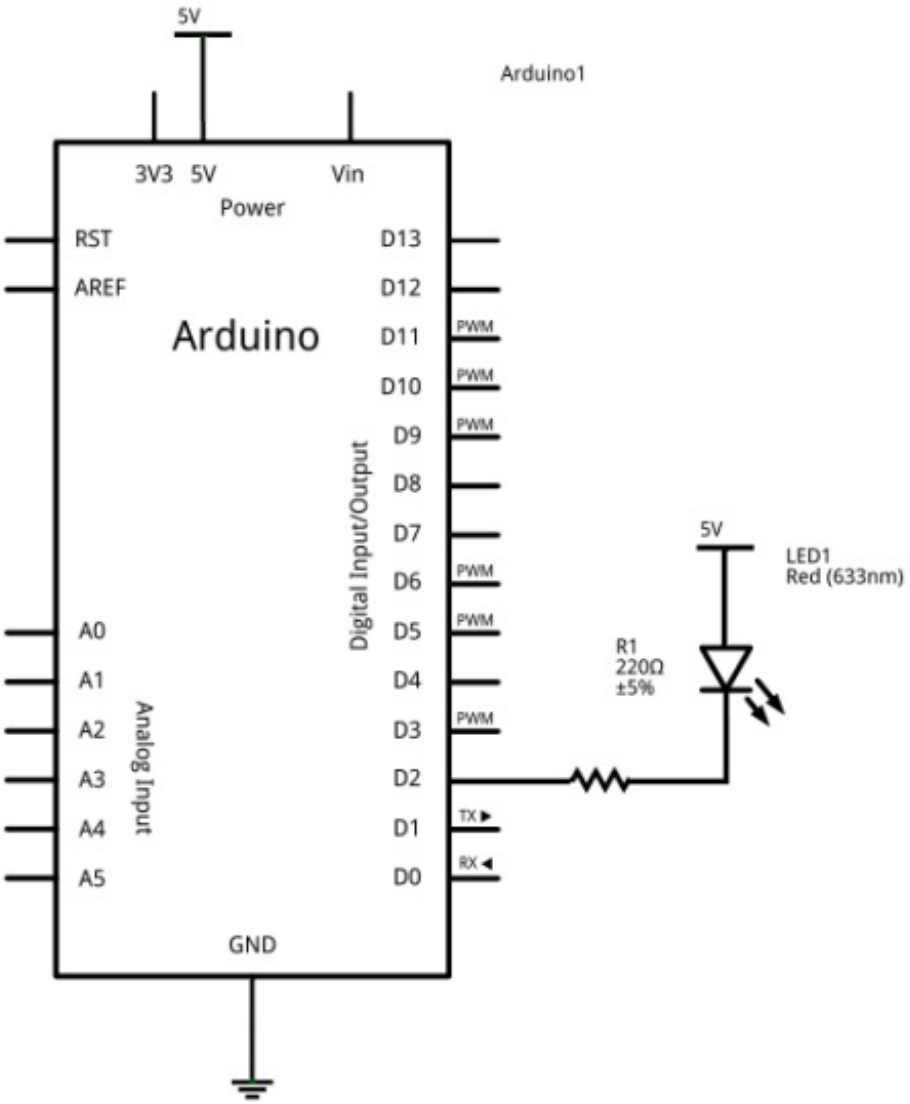
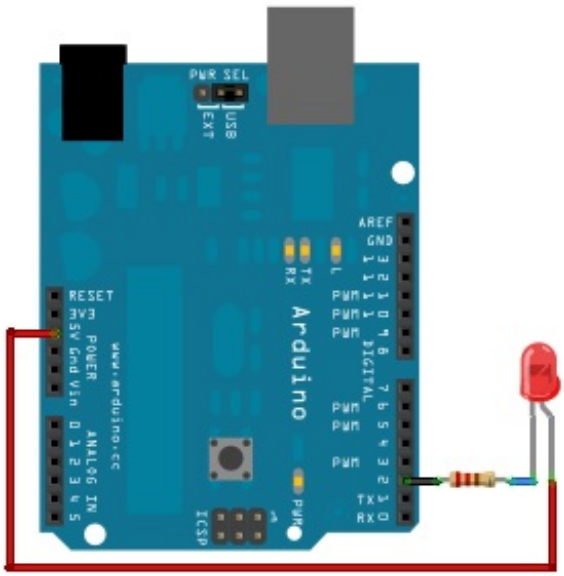
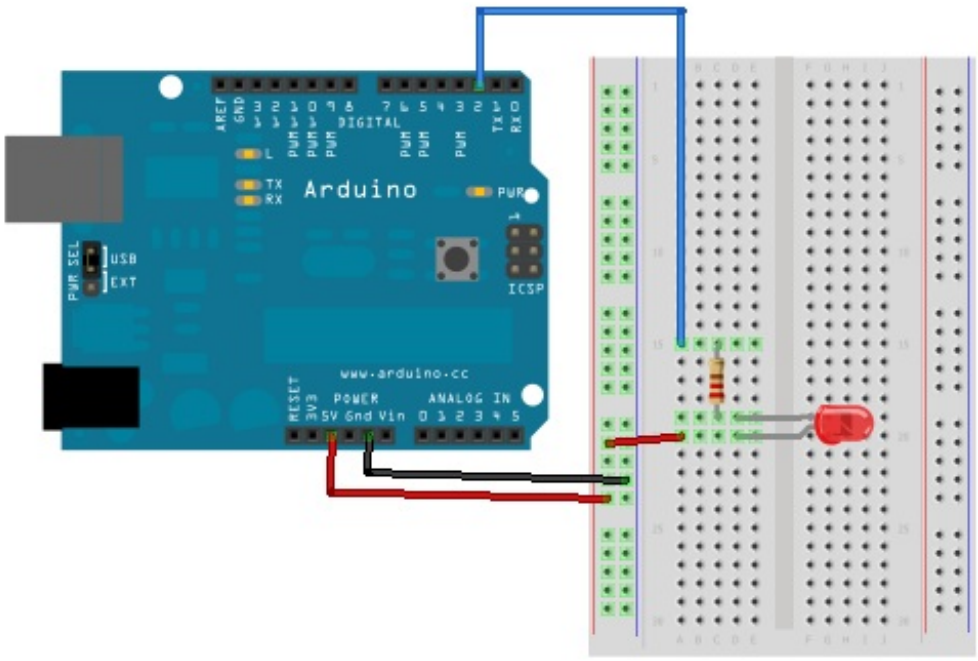


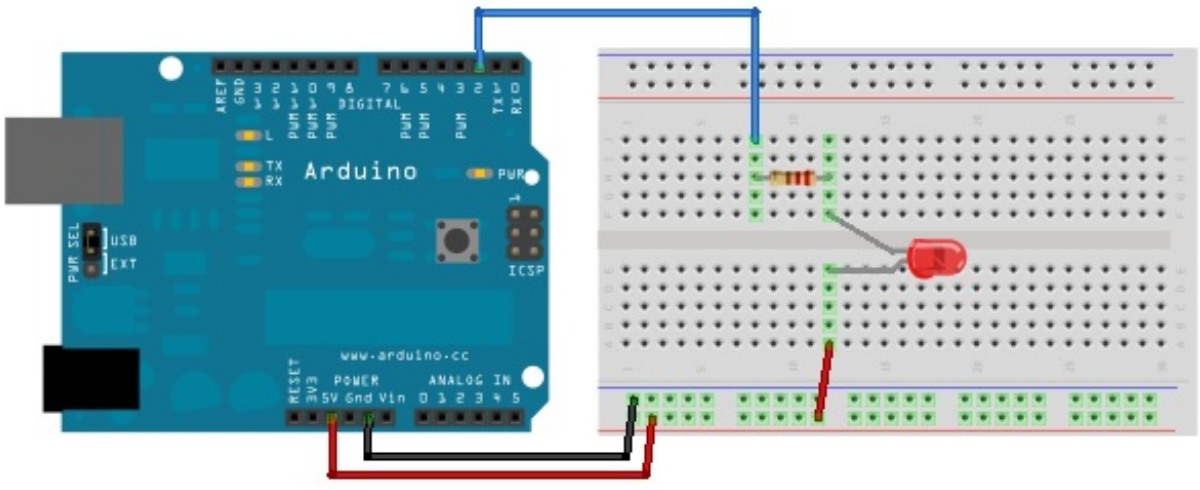

*Figure 3 : réalisation montage, schéma de la carte*




Montage avec une LED et sans breadboard




Montage une LED sur breadboard




Montage une LED sur breadboard

## Créer un nouveau projet

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

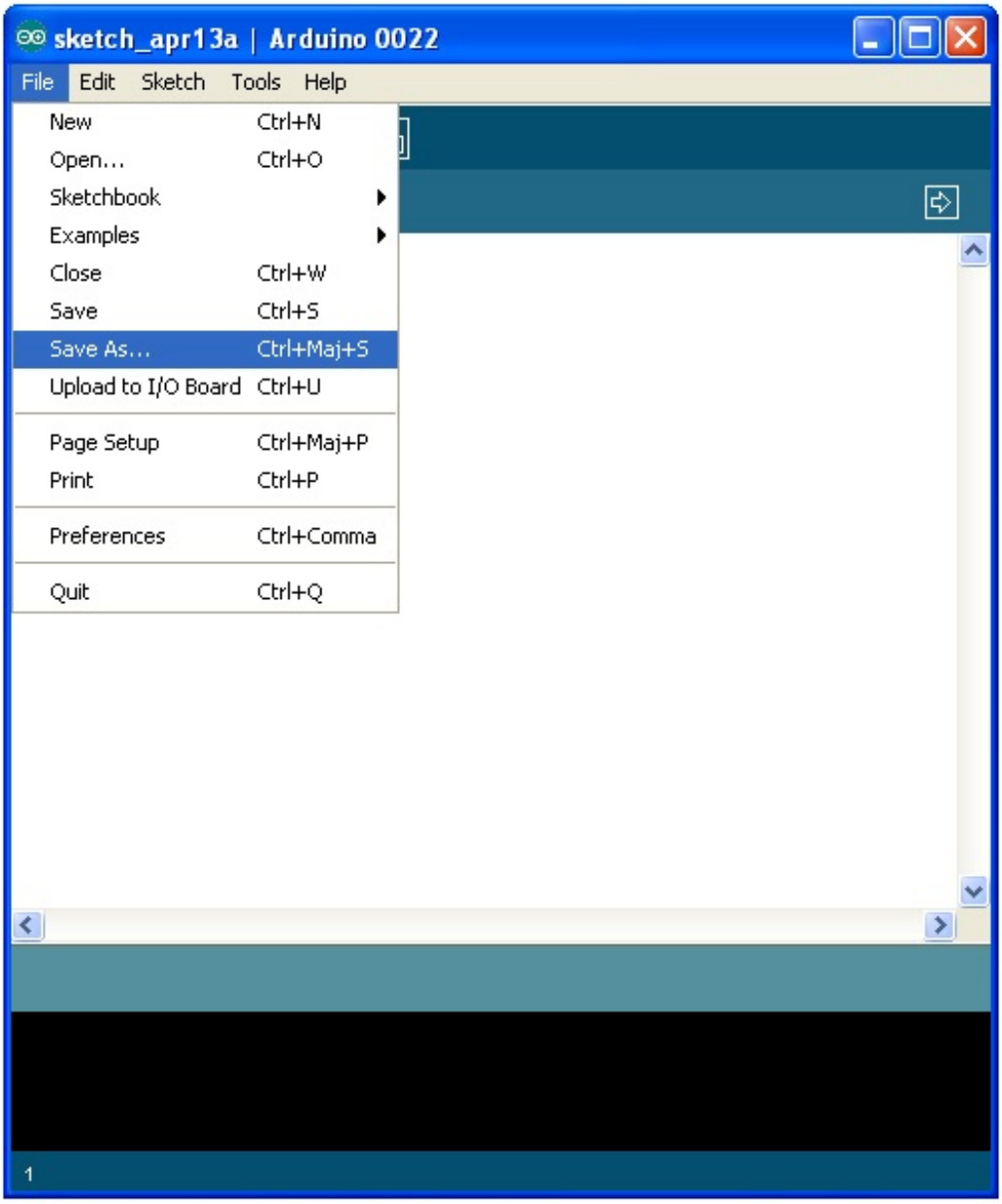



*Figure 4 : Enregistrer sous...*

Vous arrivez dans cette nouvelle fenêtre :

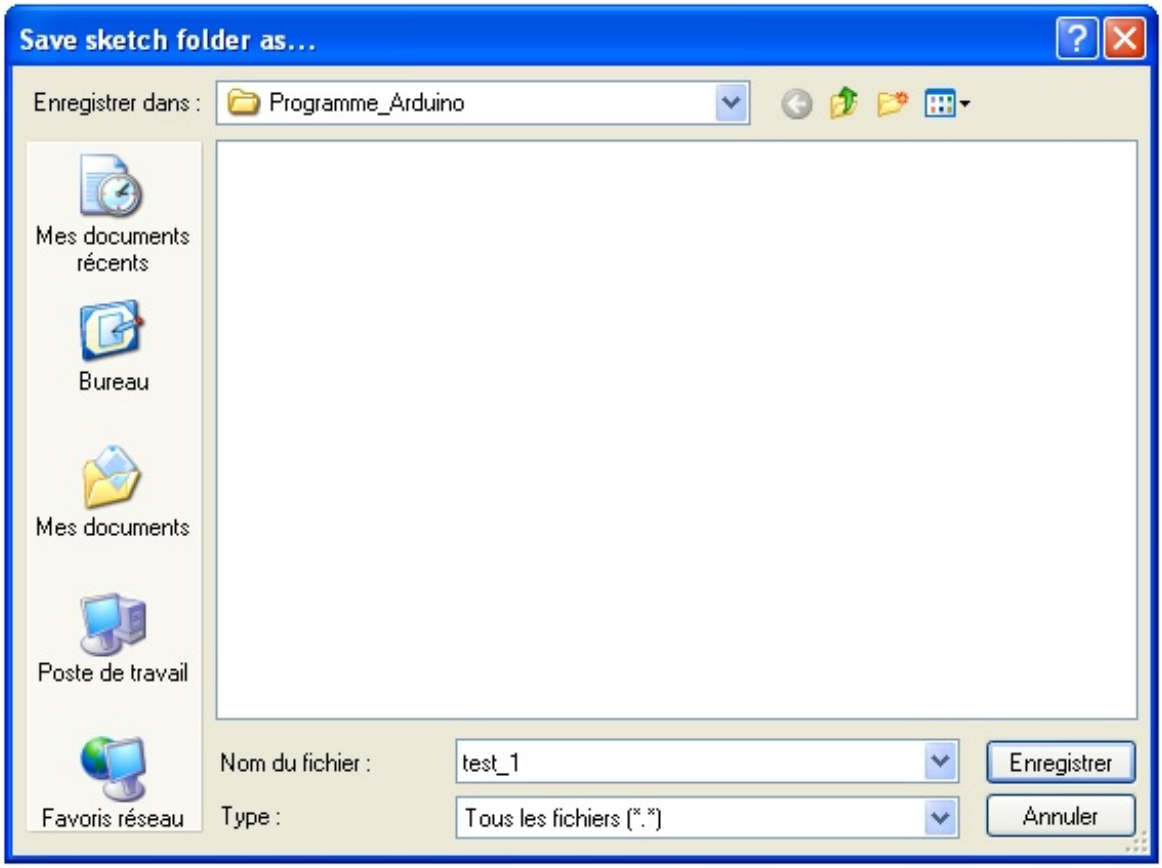



*Figure 5 : Enregistrer*

Tapez le nom du programme, dans mon cas, je l'ai appelé *test_1* . Enregistrez. vous arriver 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 :

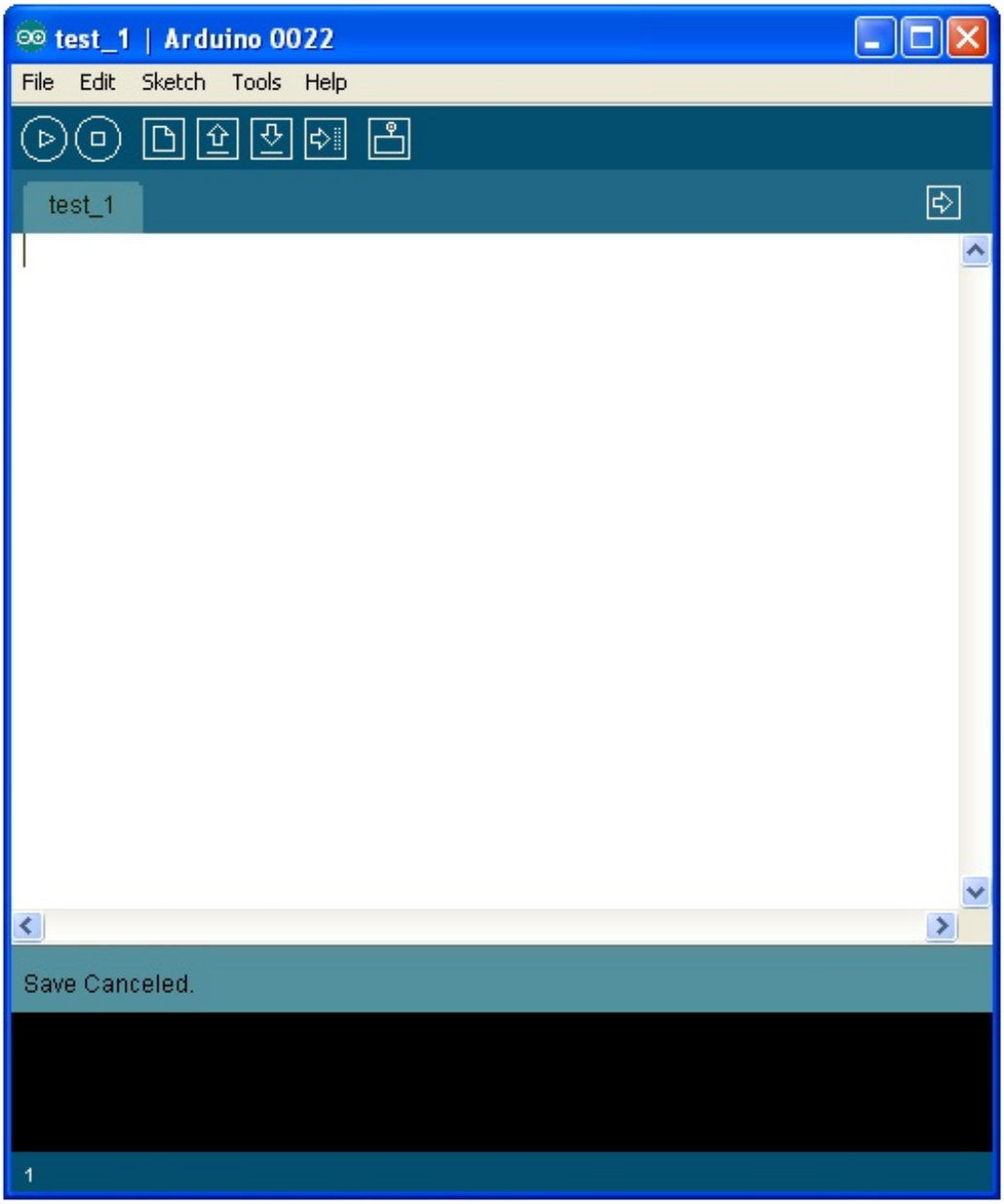



*Figure 6 : Votre nouveau programme !*

### *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 :

**Code : C**

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

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

## Créer le programme : les bons outils !

# La référence Arduino

### *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 ? Et 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.

### *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)

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...

# Allumer notre LED

### *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 micro-contrôleur :

**Code : C**

```
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.
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.

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 donnée) ou en sortie (envoi de donnée).

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)

**Code : C**

```
void setup()
{
   pinMode(led_rouge, OUTPUT); //initialisation de la broche 2
comme étant une 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 :

**Code : C**

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

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

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

### *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 ! Il faut retourner chercher dans la référence Arduino ce dont on a besoin.

Oui, mais là, on ne sait pas ce que l'on veut ?

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 dis 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).

Quel 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 : *Ecriture 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.

Ce 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 :

**Code : C**

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

Si on teste le code entier :

**Code : C**

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

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

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

On voit s'éclairer la LED !!! C'est fantastique !

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

# 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 ! ). 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. Et bien devinez quoi ? Il existe une fonction toute prête là encore ! Je ne vous en dis pas plus, passons à la pratique !

## Comment faire ?

### *Trouver la commande...*

Je vous laisse cherche un peu par vous même, cela vous entrainera !

...

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.

### *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 1 seconde, il va falloir donner à la fonction ce même temps, écrit en millisecondes, soit 1000ms.

Le code est simple à utiliser, il est le suivant :

**Code : C**

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

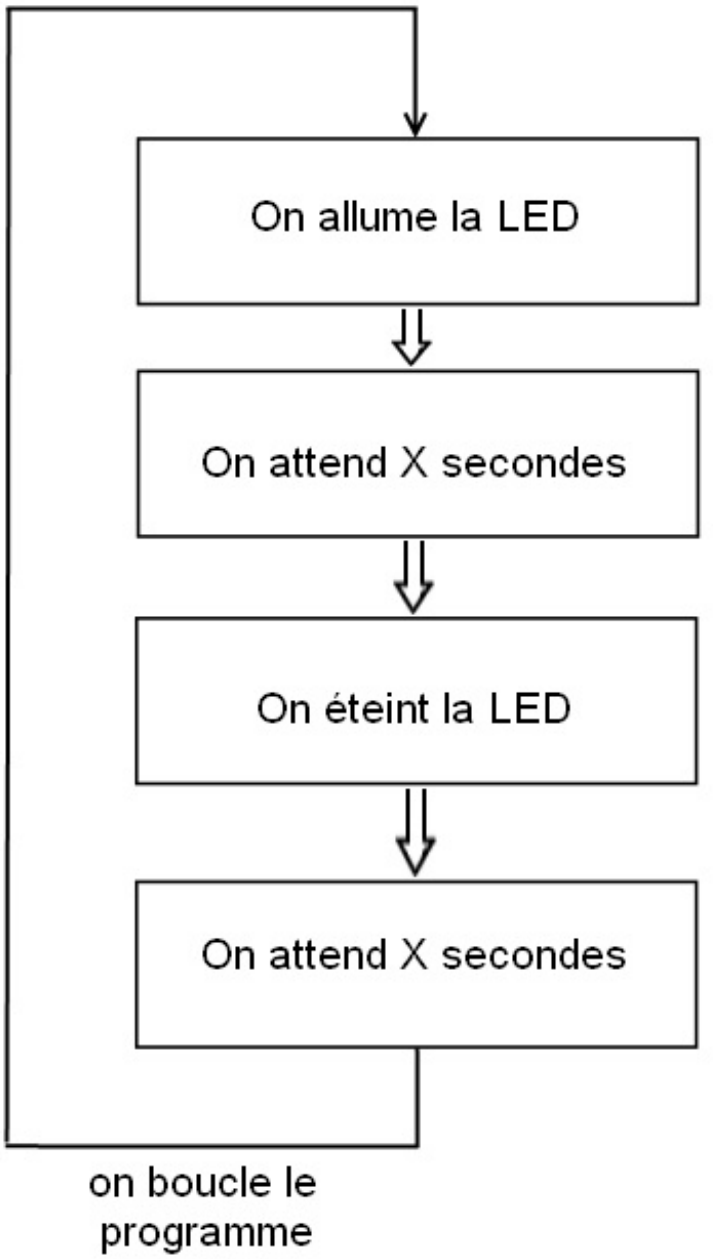
Rien de plus simple donc. Pour 20 secondes de pause, il aurait fallu écrire :

**Code : C**

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

### *Mettre en pratique : faire clignoter une LED*

Du coup, si on veut faire clignoter notre LED, il va falloir 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 somme de commande, qui forme le processus qui fait clignoter la LED.

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 traduis par l'instruction que l'on a vue dans le chapitre précédent :

**Code : C**

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

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

**Code : C**

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

Puis, on traduit la ligne suivante :

**Code : C**

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

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

**Code : C**

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

On se retrouve avec le code suivant :

**Code : C**

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

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

**Code : C**

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

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 !

**Code : C**

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

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

```
   pinMode(led_rouge, OUTPUT); //initialisation de la broche 2
comme étant une sortie
}

void loop()                    //fonction principale, elle se
répète (s'exécute) à l'infini
{
   digitalWrite(led_rouge, LOW);    // allume la LED
   delay(1000);                      // fait une pause de 1 seconde
   digitalWrite(led_rouge, HIGH);     // éteint la LED
   delay(1000);                      // fait une pause de 1 seconde
}
```

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 :

**Code : C**

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

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

void loop()                    //fonction principale, elle se
répète (s'exécute) à l'infini
{
   digitalWrite(led_rouge, LOW);    // allume la LED
   delay(600);                       // fait une pause de 600 milli-
seconde
   digitalWrite(led_rouge, HIGH);     // éteint la LED
   delay(40);                       // fait une pause de 40 milli-
seconde
}
```

Et Hop, une petite vidéo d'illustration !

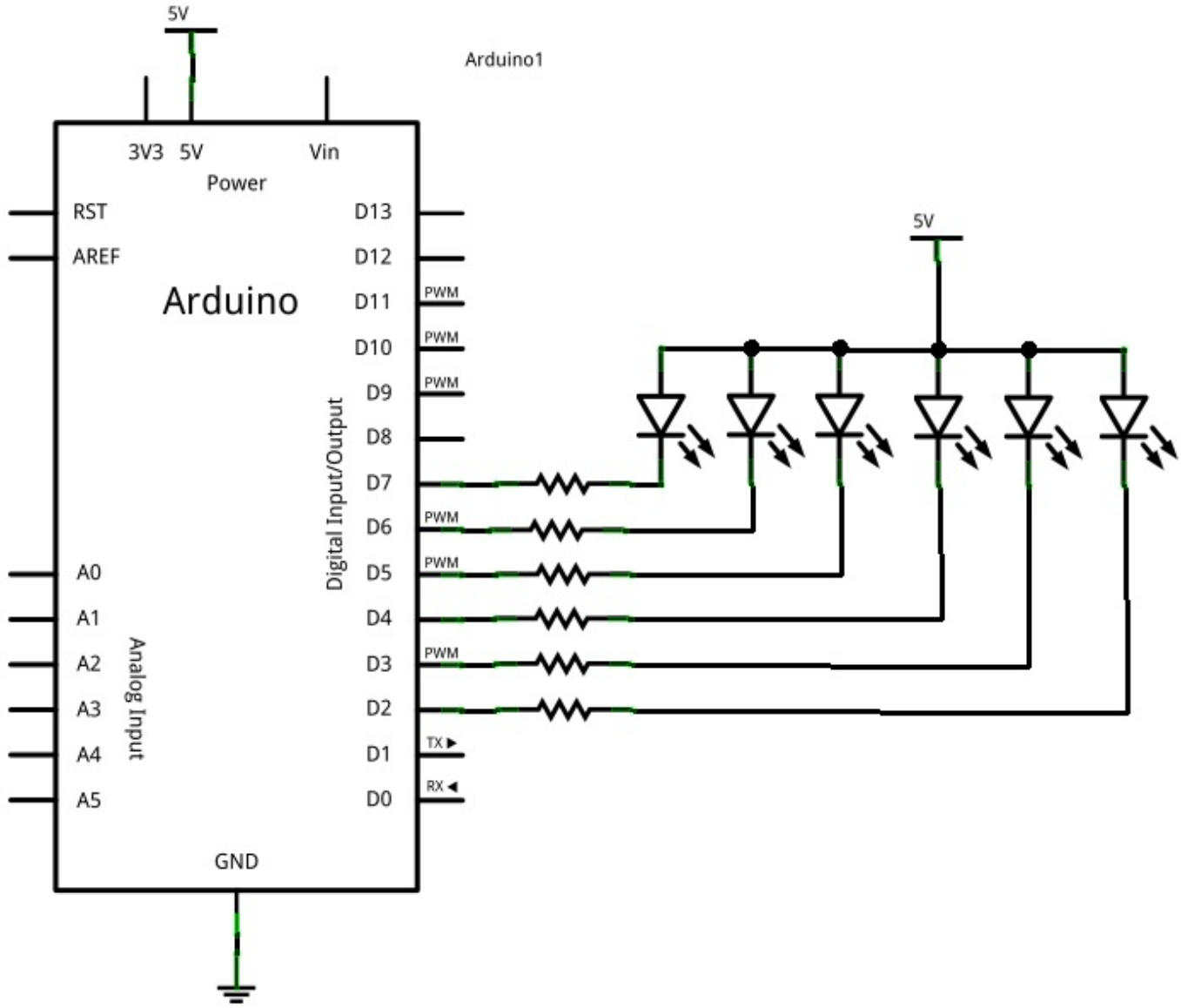
## Faire clignoter un groupe de LED

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.

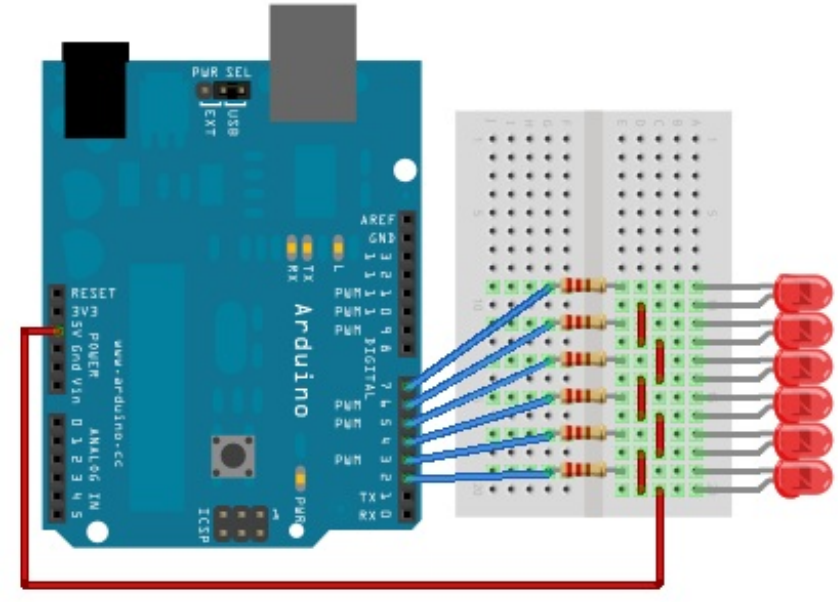
### *Le matériel et les schémas*

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 semblable de résistances.

Le schéma de la réalisation :

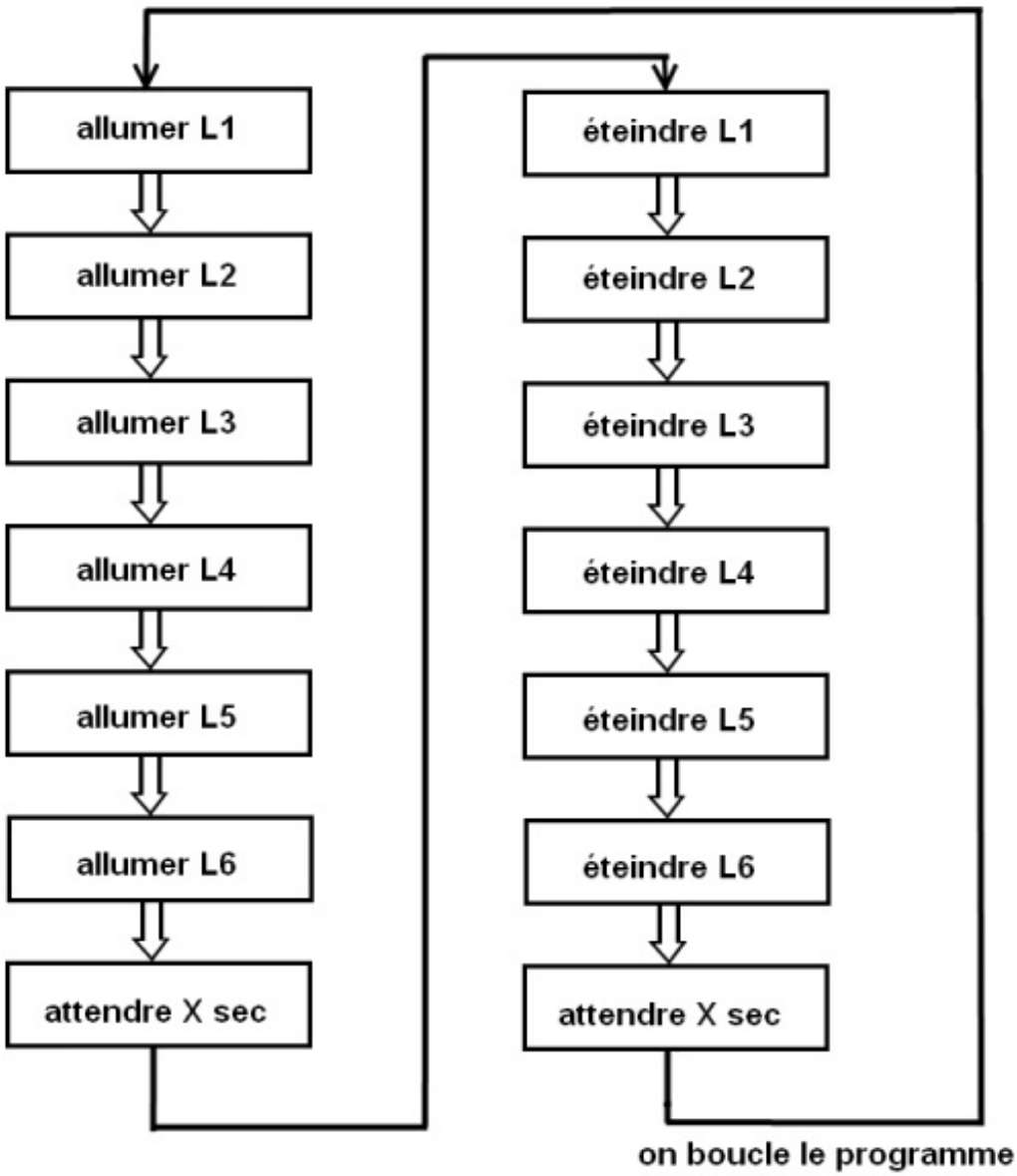



La photo de la réalisation :



### *Le programme*

Le programme est un peu plus long que le précédent, car il ne s'agit plus d'allumer 1 seule LED, mais 6 ! 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 :

**Code : C**

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

Ensuite, on attend 1,5 seconde :

**Code : C**

```
delay(1500);
```

Puis on traduis les six autres instructions :

**Code : C**

```
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 :

**Code : C**

```
delay(4320);
```

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

**Code : C**

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

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

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 :

**Code : C**

```
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;
```

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

**Code : C**

```
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);
```

### *Le programme final*

Il n'est certes pas très beau, mais il fonctionne :

**Code : C**

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

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

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

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

Tu soulèves un point intéressant. Si je comprends bien, tu te demandes 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 !

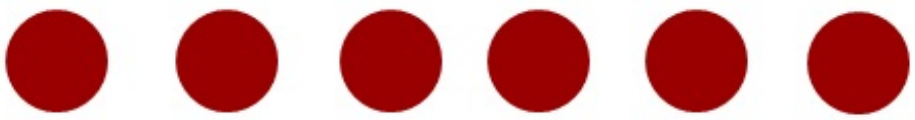
Il y a une solution pour faire ce que tu dis. Nous allons la voir dans quelques chapitres, ne sois pas impatient !

En attendant, voici une vidéo d'illustration du clignotement :

## Réaliser un chenillard

### *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 !

Voilà donc ce qu'est un chenillard. Chaque LED s'allume alternativement et dans l'ordre chronologique. De la gauche vers la droite ou l'inverse, c'est au choix.

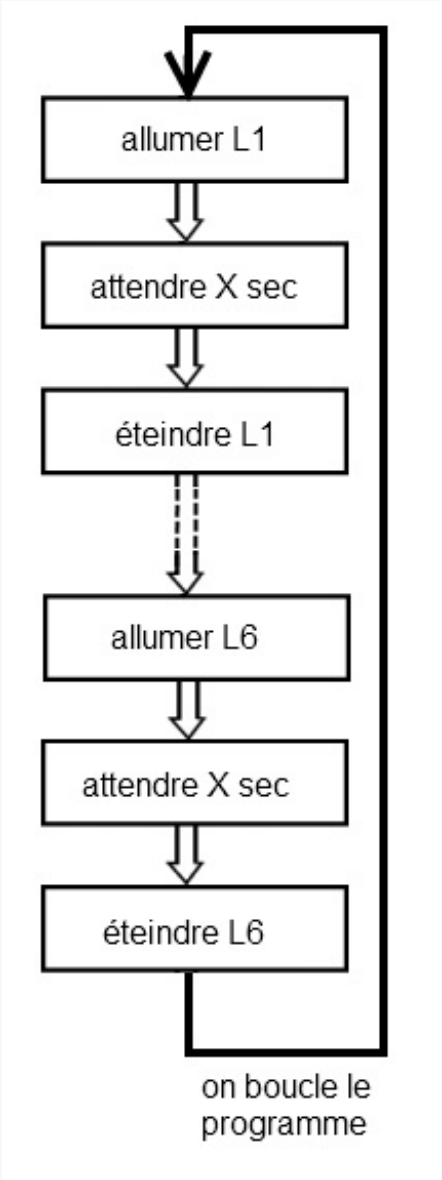
### *Organigramme*

Comme j'en ai marre de faire des dessins avec paint.net, je vous laisse réfléchir tout seul 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** (cliquez pour afficher)




A vous de jouer !

### *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 :

**Code : C**

```
// on garde le même début que le programme précédent

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);
}
```

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 !

## Fonction millis()

Nous allons terminer ce chapitre par un point qui peutêtre utile, notamment dans certaines situations où l'on veut ne 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 `delay()` qui met en pause le programme durant le temps défini.

## 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

**Code : C**

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

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

```
  }
 }
```

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.

## Découvrons et utilisons millis()

Tout d'abord, quelques précisions à son sujet, avant d'aller s'en servir. A 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".

C'est bien gentil mais concrètement on l'utilise comment ?

Et 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 :

**Code : C**

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

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

void loop()
{
 if(digitalRead(bouton)==HIGH) //si le bouton est cliqué (on rentre
dans un mur)
 {
  digitalWrite(moteur, LOW); //on arrête le moteur
 }
 else //sinon on clignote
 {
  //on compare l'ancienne valeur du temps et la valeur sauvée
  //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
  }
```

```
    }
  }
```

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. A la place, on avance dans le programme et test 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.

Attention, au "if" de la ligne 25 ne faites surtout pas "millis() - temp == 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 !

# [TP] Feux de signalisation routière

Vous voilà arrivé pour votre premier TP, que vous ferez seul ! 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.

## Préparation

Ce dont nous avons besoin pour réaliser ces feux.

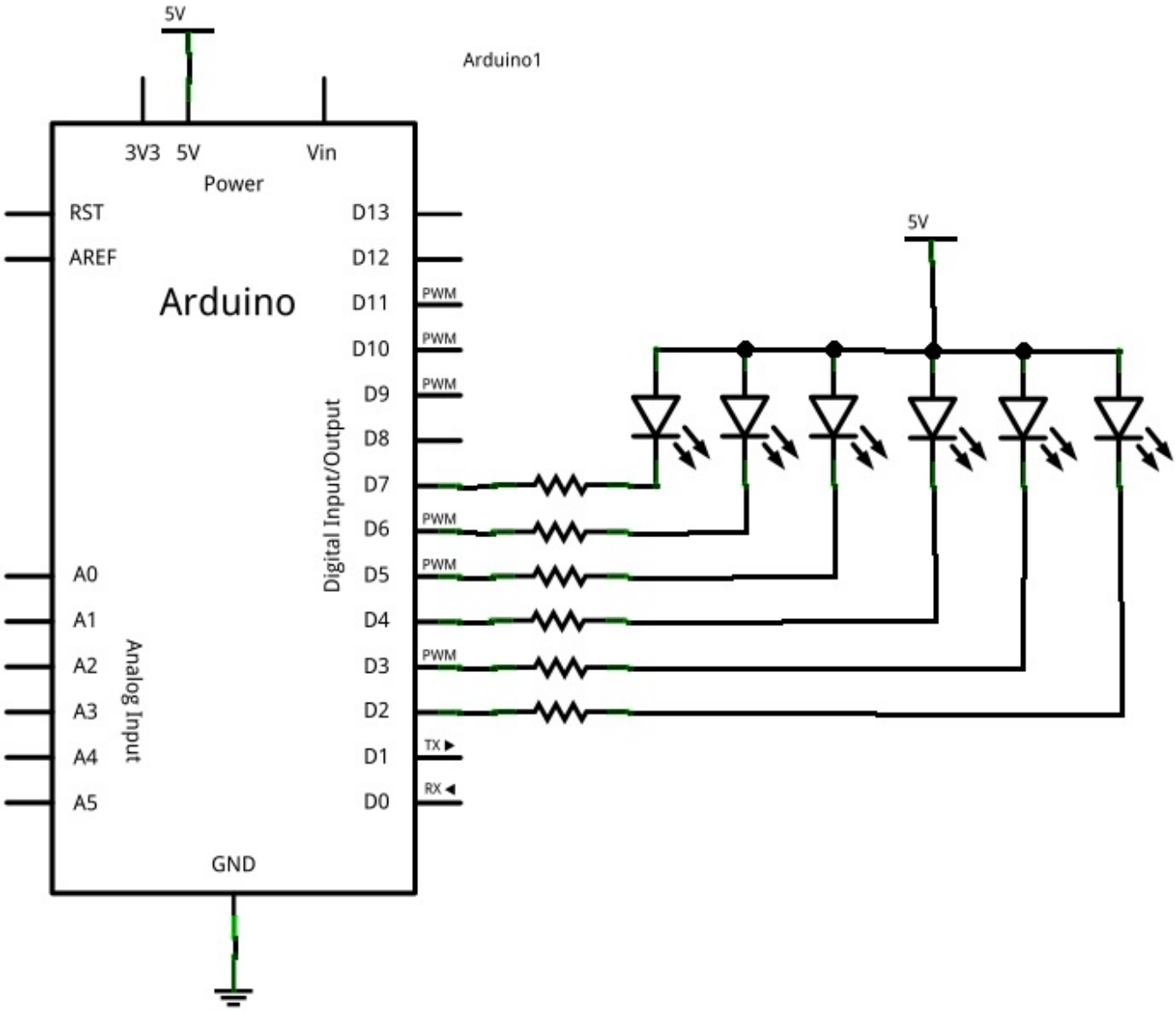
### *Le matériel*

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.

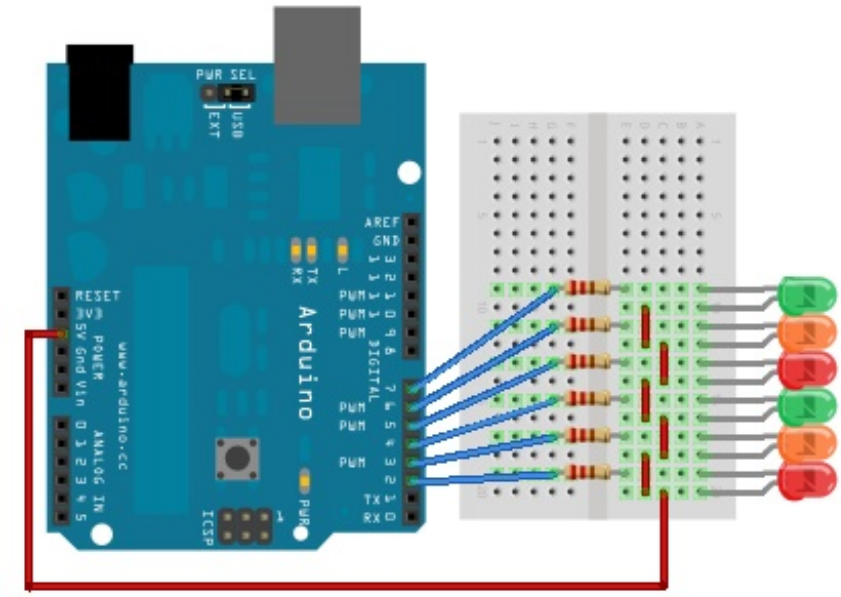
- LED : un nombre de 6, dont 2 **rouges**, 2 **jaune**/**orange** et 2 **vertes**
- Résistors : 6 également, de la même valeur que ceux que vous avez utilisés.
- Arduino : une carte Arduino évidemment !

### *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 :

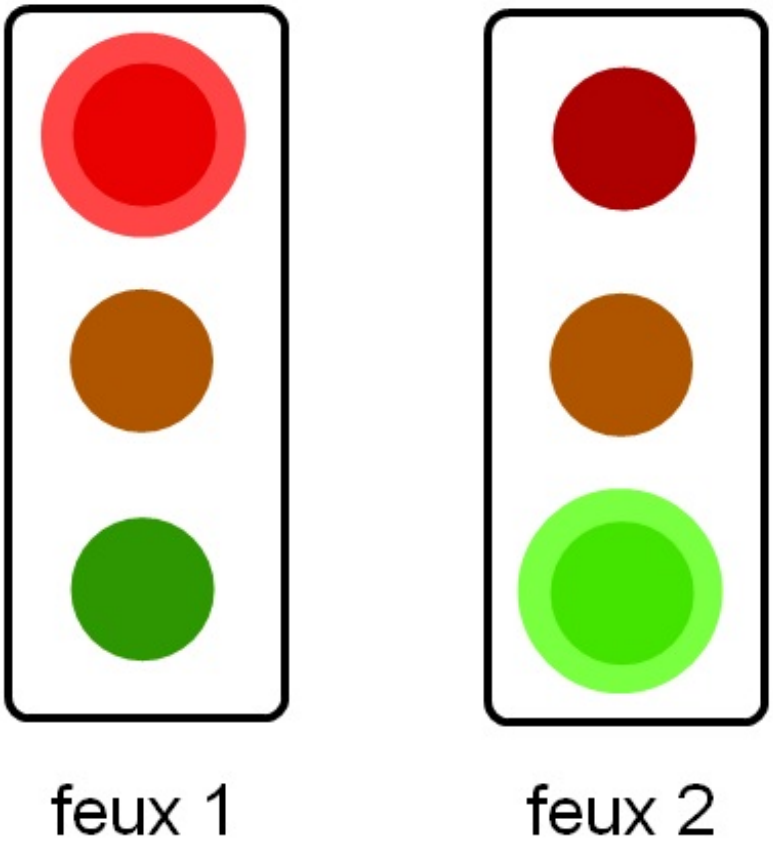



N'oubliez pas de tester votre matériel en chargeant un programme qui fonctionne ! Cela évite de s'acharner à faire un nouveau programme qui ne fonctionne pas à cause d'un matériel défectueux. On est jamais sur de rien, croyez-moi !

## Énoncé de l'exercice

### *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 :



### *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.

### *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.

### *C'est parti !*

Allez, c'est parti ! A vous de m'épater. 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 :

## Correction !

### *Fini !*

Vous avez fini ? Votre code ne fonctionne pas, mais vous avez eu beau cherché 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 !

### *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 !

## La correction, enfin !

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

### *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** (cliquez pour afficher)

**Code : C**

```
//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);
}
```

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

### *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** (cliquez pour afficher)

**Code : C**

```
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
----------------- */
}
```

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.

# Un simple bouton

Dans cette partie, vous allez pouvoir interagir de manière simple avec votre carte. A la fin de ce chapitre, vous serez capable d'utiliser des boutons ou des interrupteurs pour interagir avec votre programme.

## 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.

## 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.

### *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**.

**Retenez bien ces mots de vocabulaire !**

Habituellement le bouton poussoir a deux broches, mais en général ils en ont 4 reliées deux à deux.

### *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**.

### *Les interrupteurs*

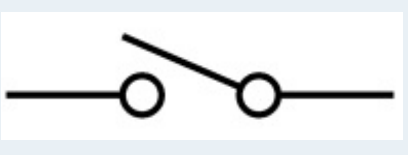
A la différence d'un bouton poussoir, l'interrupteur agit comme une bascule. Un appui ferme le circuit et il faut un second appui pour l'ouvrir de nouveau. Il possède donc des états stables (ouvert ou fermé). On dit qu'un interrupteur est **bistable**. Vous en rencontrez tous les jours lorsque vous allumez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

## L'électronique du bouton

### *Symbole*

Le BP et l'interrupteur ne possèdent pas le même symbole pour les schémas électroniques. Pour le premier, il 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 :

|  |  |  |
|---|---|---|
| Bouton Poussoir NO | Bouton Poussoir NF | Interrupteur |

### *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 )

Il est très fréquent de trouver des boutons dans les starters kit. Souvent ils ont 4 pattes (comme sur l'image ci-dessous). Si c'est le cas, sachez que les broches sont reliées deux à deux. Cela signifie quelles fonctionnent par paire. Il faut donc se méfier lorsque vous le brancher sinon vous obtiendrez le même comportement qu'un fil (si vous connectez deux broches reliés). Utilisez un multimètre pour déterminer quels broches sont distinctes.

**Pour ne pas se tromper, on utilise en général deux broches qui sont opposées sur la diagonale du bouton.**

## Contrainte pour les montages

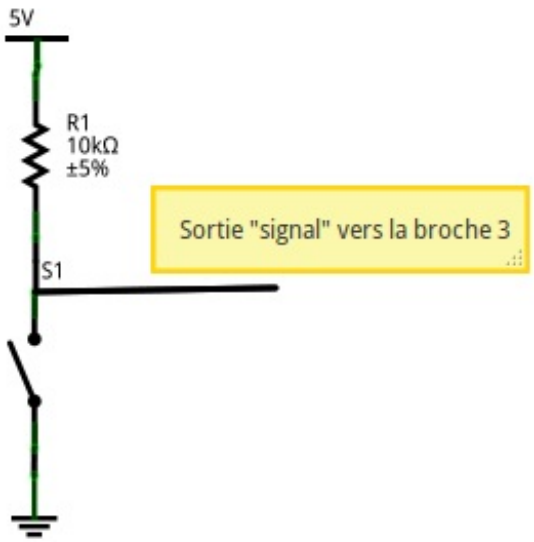
Voici maintenant un point très important, soyez donc attentif car je vais vous expliquer le rôle d'une résistance de pull-up !

C'est quoi st'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 d'en prendre 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, ont 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.

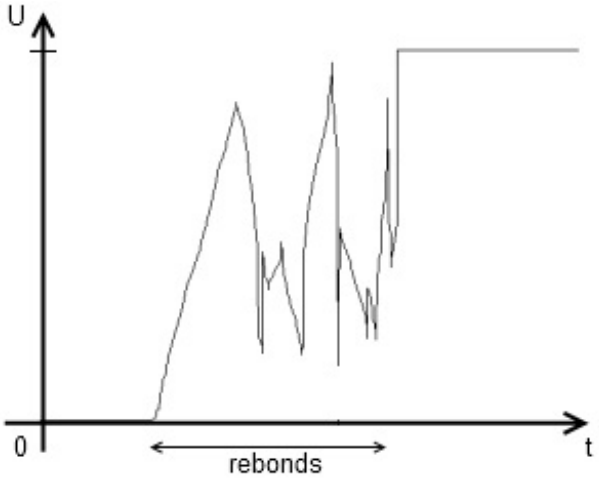
Sur le schéma suivant, on voit ainsi qu'en temps normal le "signal" à un potentiel de 5V. Ensuite, lorsque l'utilisateur appuiera sur le bouton une connexion sera faite avec la masse. On lira alors une valeur de 0V pour le signal. Voici donc un deuxième intérêt de la résistance de pull-up, éviter le court-circuit qui serait généré à l'appui !



### *Filtrer les rebonds*

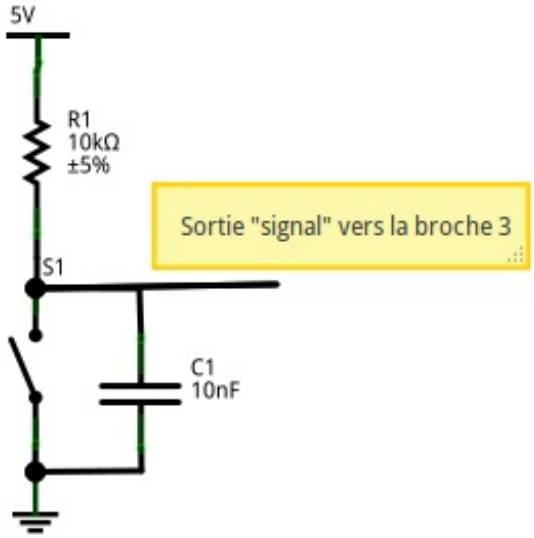
Les boutons ne sont pas des systèmes mécaniques parfaits. Du coup, lorsqu'un appui est fait dessus, le signal ne passe pas immédiatement et proprement de 5V à 0V. En l'espace de quelques millisecondes, le signal va "sauter" entre 5V et 0V plusieurs fois avant de se stabiliser. Il se passe le même phénomène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



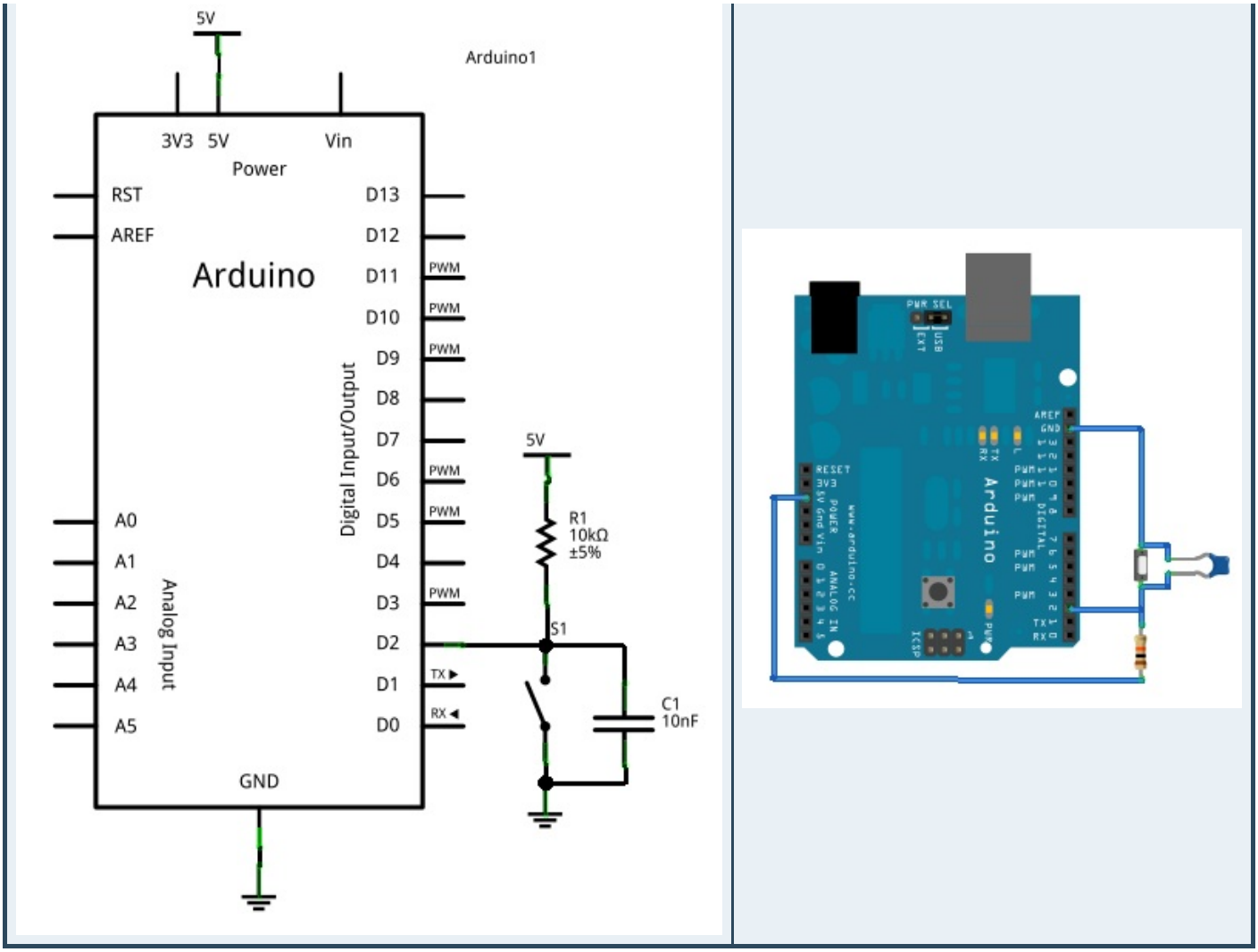

Pour atténuer ce phénomène, nous allons utiliser un condensateur en parallèle avec le bouton. Ce composant servira ici "d'amortisseur" qui absorbera les rebonds (comme sur une voiture avec les cahots de la route). Le condensateur, initialement chargé, va se décharger lors de l'appui sur le bouton. S'il 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 :




### *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 :



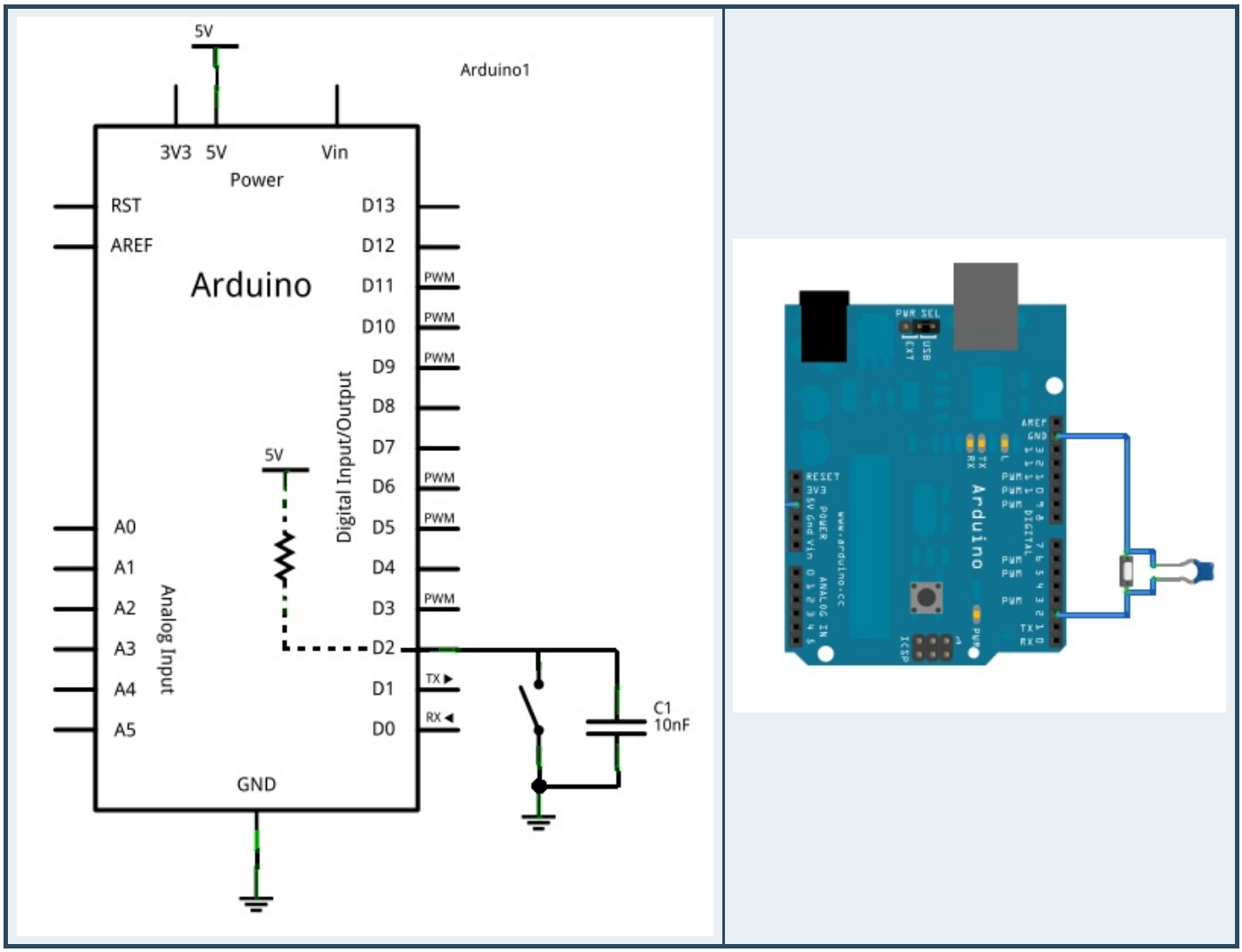

## Les pull-ups internes

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 microcontrolleur 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 utilisés sans aucune contraintes techniques. 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 vient de dire permet de mettre en place ces dernières dans le logiciel :

**Code : C**

```
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
}
```

### *Schéma résumé*




## Récupérer l'appui du bouton

## 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 :

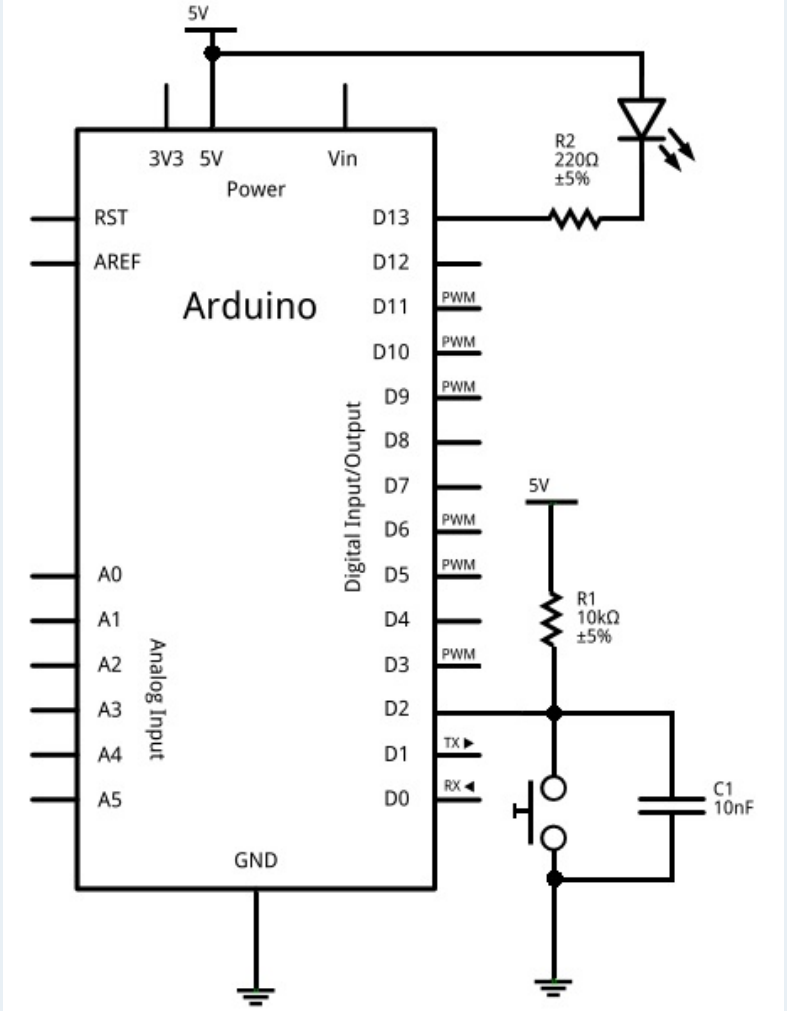
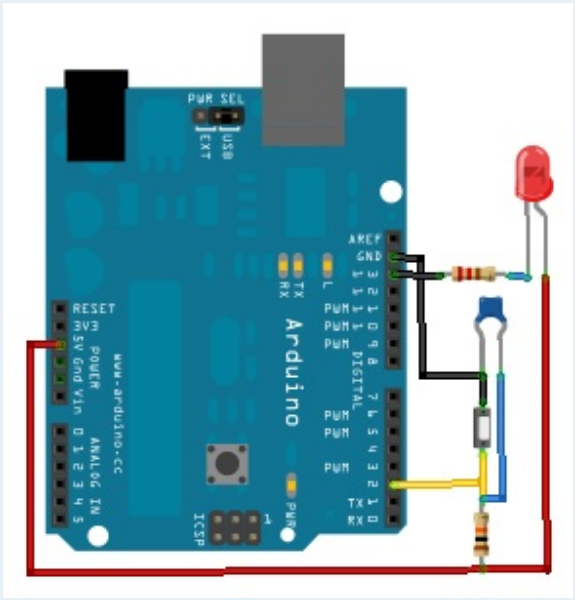



Schéma avec 1 bouton et 1 LED



Montage avec 1 bouton et 1 led

Montage simple avec un bouton et une LED

## 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 :

**Code : C**

```
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 :

**Code : C**

```
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.

## Récupérer l'état du bouton

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

- 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 :

**Code : C**

```
int etat;

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

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

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 `actionRelache()` et `actionAppui()`.

## Test simple

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

### *But*

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

### *Correction*

Allez, c'est vraiment pas dur, en plus je vous donnais le montage dans la première partie...

Voici la correction :

- - Les variables globales

**Code : C**

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

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

- - La fonction setup()

**Code : C**

```
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
"relaché"
}
```

- - La fonction loop()

**Code : C**

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

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

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

## Interagir avec les LEDs

Nous allons maintenant faire un exemple d'application ensemble.

## Montage à faire

Pour cet exercice, nous allons utiliser deux boutons et quatre LEDs 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 4 LEDs seront connectées sur les broches 10 à 13 de l'Arduino.

Voilà donc le montage à effectuer :

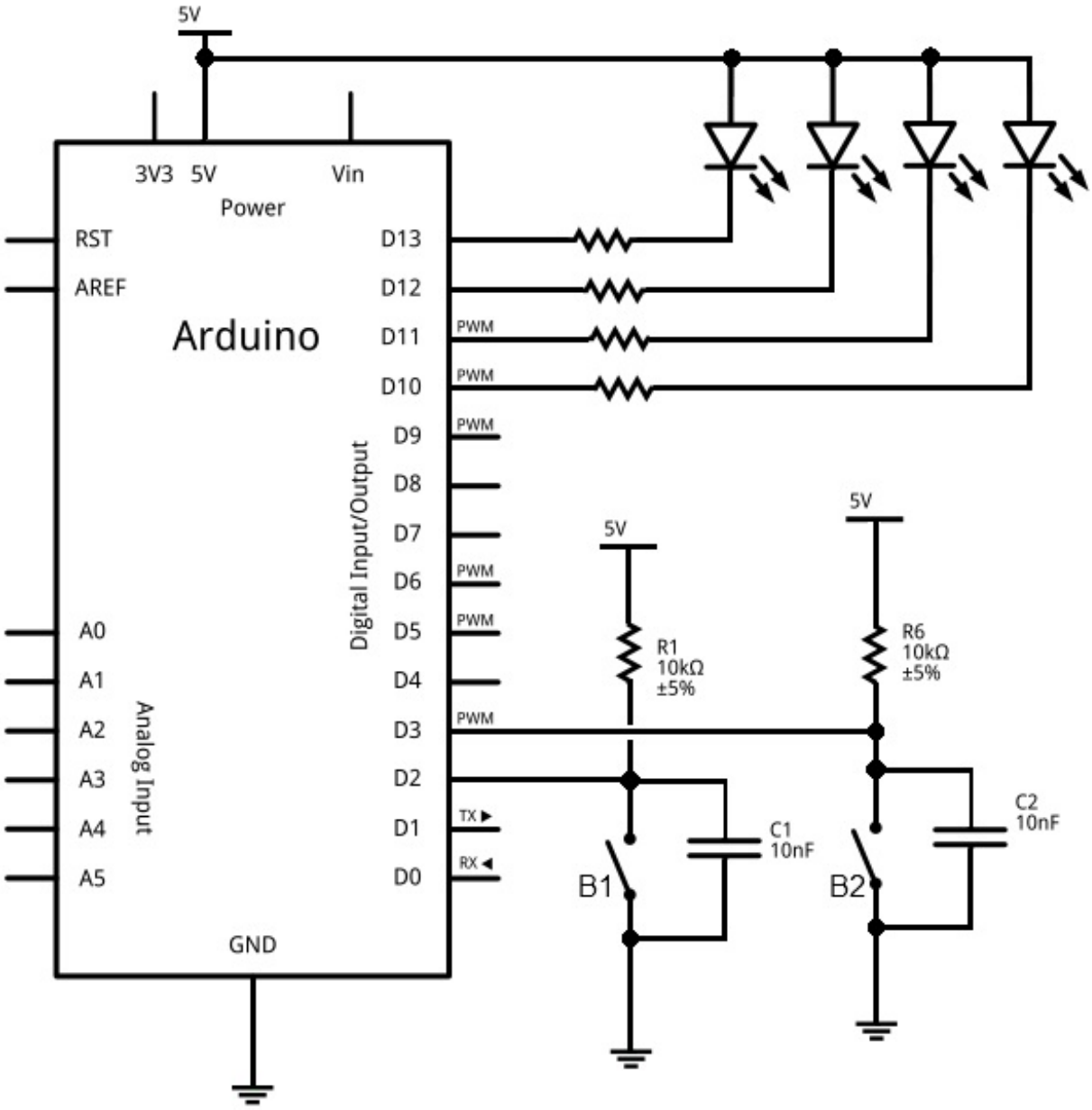
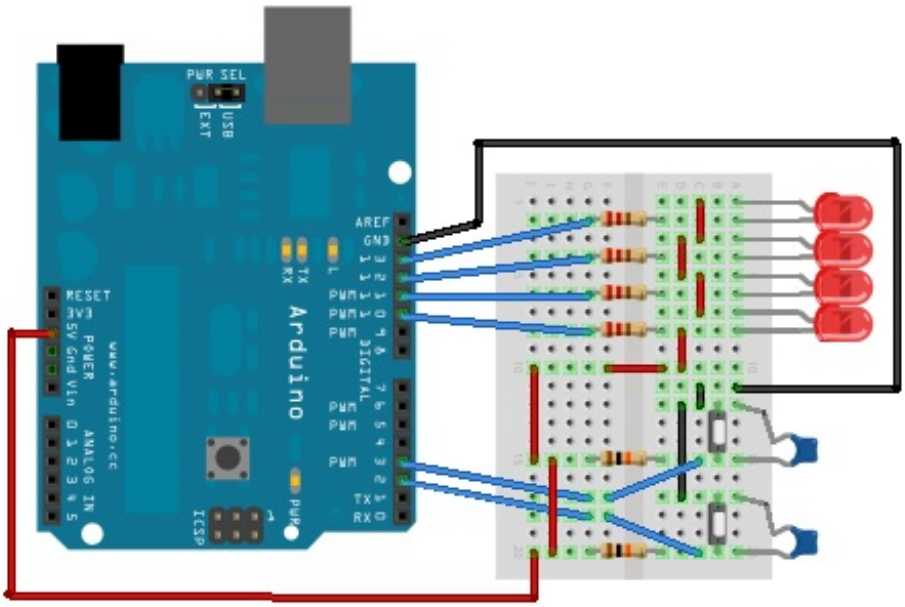



Schéma avec 2 boutons et 4 LEDs



Montage avec 2 boutons et 4 leds

Montage de l'exercice, avec deux boutons et quatre LEDs

# 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 LEDs 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 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.

## *Cahier des charges*

La réalisation prévue devra :

- - posséder 4 LED (ou plus pour les plus téméraires)
- - posséder 2 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 1 et 4 selon l'appui du bouton d'incrémentation ou de décrémentation.

Vous pouvez maintenant vous lancer dans l'aventure. Pour ceux qui se sentiraient encore un peu mal à l'aise avec la programmation, je vous autorise à poursuivre la lecture qui vous expliquera pas à pas comment procéder pour arriver au résultat final. 

# Correction

## *Initialisation*

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

**Code : C**

```
/* 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 */

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

/* 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()
{
    //les instructions de votre programme
}
```

### *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 à 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 :

**Code : C**

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

### *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 :

**Code : C**

```
void loop()
{
    //lecture de l'état du bouton d'incrémentation
    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))
    {
        nombre_led++; //on incrémente la variable qui indique combien
de LED devrons s'allumer
    }

    memoire_plus = etat_bouton; //on enregistre l'état du bouton
pour le tour suivant


    //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
    }
    memoire_minus = etat_bouton; //on enregistre l'état du bouton
```

```
    pour le tour suivant

       //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);
    }
```

Nous avons terminé de créer le squelette du programme et la détection d'évènement, il ne reste plus qu'à afficher le résultat du nombre !

### *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.

A présent, nous devons allumer les LEDs selon la valeur reçue. On sait que l'on doit afficher une LED lorsque l'on reçoit le nombre 1, 2 LEDs lorsqu'on reçoit le nombre 2, ...

**Code : C**

```
    void affiche(int valeur_recue)
    {
        //on éteint toutes les LEDs
        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);
        }
    }
```

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 LEDs.

Le code au grand complet :

**Secret** (cliquez pour afficher)

**Code : C**

```
/* déclaration des constantes pour les nom 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 */

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

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'incrémentation
   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))
   {
      nombre_led++; //on incrémente la variable qui indique
combien de LED devrons s'allumer
   }

   memoire_plus = etat_bouton; //on enregistre l'état du bouton
pour le tour suivant


   //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
   }
   memoire_minus = etat_bouton; //on enregistre l'état du bouton
pour le tour suivant

   //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 leds
         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);
         }
    }
```

Une petite vidéo du résultat que vous devriez obtenir, même si votre code est différent du mien :

## Les interruptions matérielles

Voici maintenant un sujet plus délicat (mais pas tant que ça !  ) 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**.

## 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 évènements 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).

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'infinie. 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'infinie (autrement dit jamais, en théorie). Nous allons donc nous tourner vers les interruptions qui, dès que le bouton sera appuyé, interromprons 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.

## 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.

### *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, function, 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)
- - function : 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 :

**Code : C**

```
attachInterrupt(0, Reagir, FALLING);
```

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.

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).

## 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 !

# 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 lettre de l'alphabet.

### *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.

## Première approche : côté électronique
## Un peu (beaucoup) d'électronique

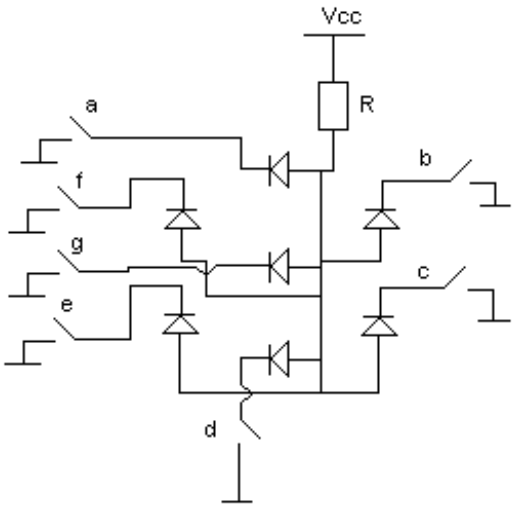
Comme son nom l'indique, l'afficheur 7 segments possède... 7 segments. Mais un segment c'est quoi au juste ? Et bien c'est une portion de l'afficheur, qui est allumée ou éteinte pour réaliser l'affichage. Cette portion n'est en fait rien d'autre qu'une LED qui au lieu d'être ronde comme d'habitude est plate et encastré dans un boiter. 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 :

*Afficheur 7 segments*

### *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).

### *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 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 330 ohms 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.

### *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. 

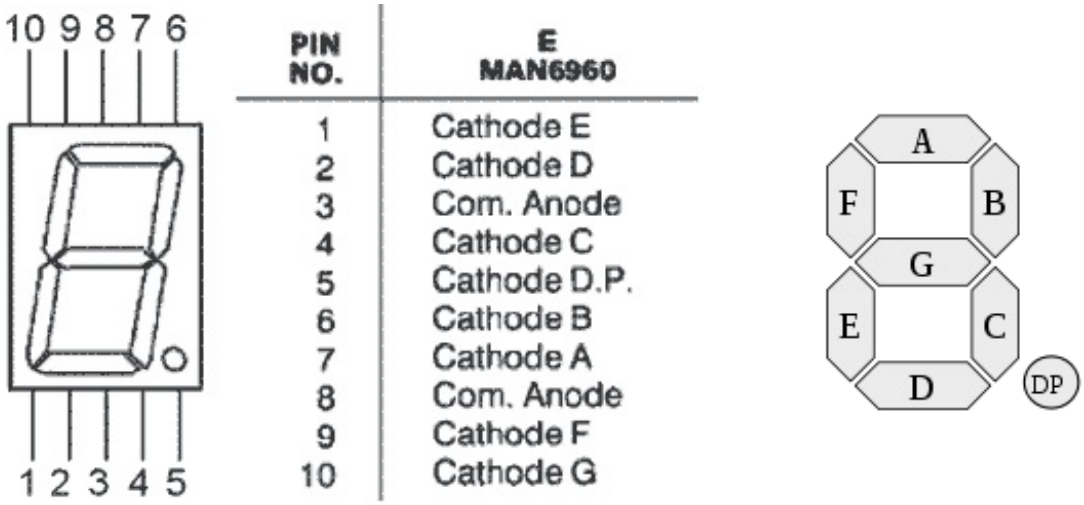
## Branchement "complet" de l'afficheur

Nous allons maintenant voir comment brancher l'afficheur à anode commune.

### *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 boitier).

Voici une représentation de ce dernier (à gauche) :



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

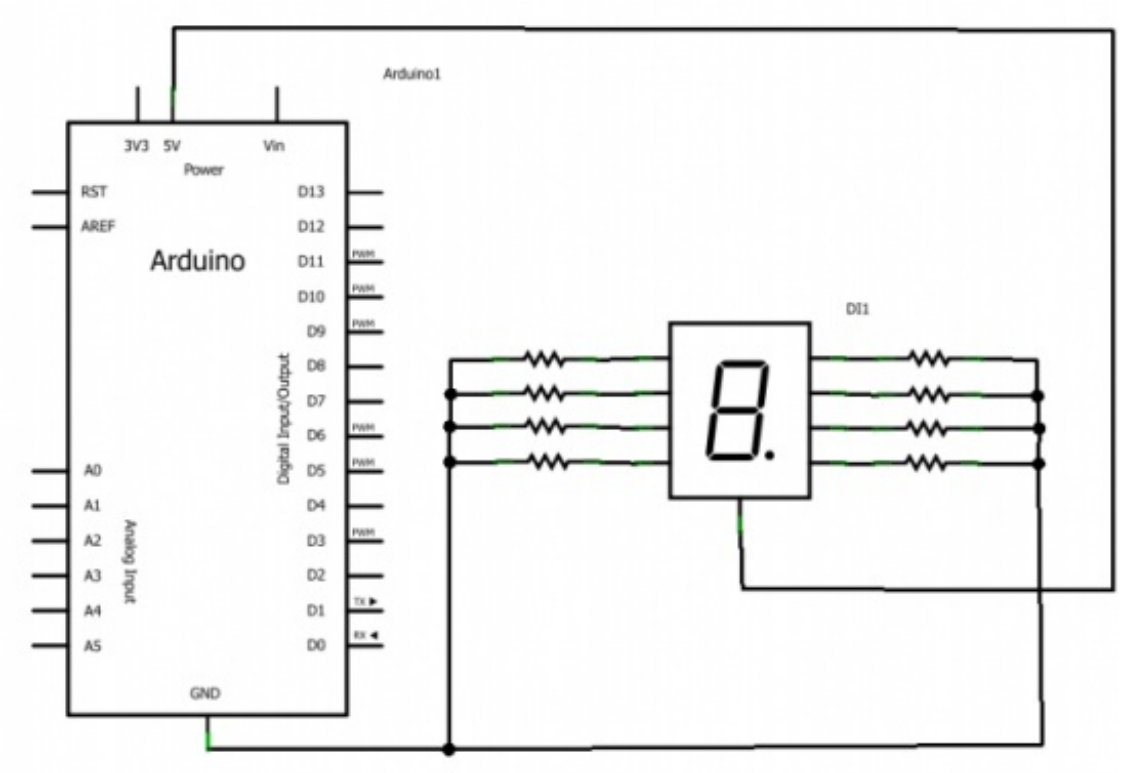
Pour allumer un segment c'est très simple, il suffit de le relier à la masse !

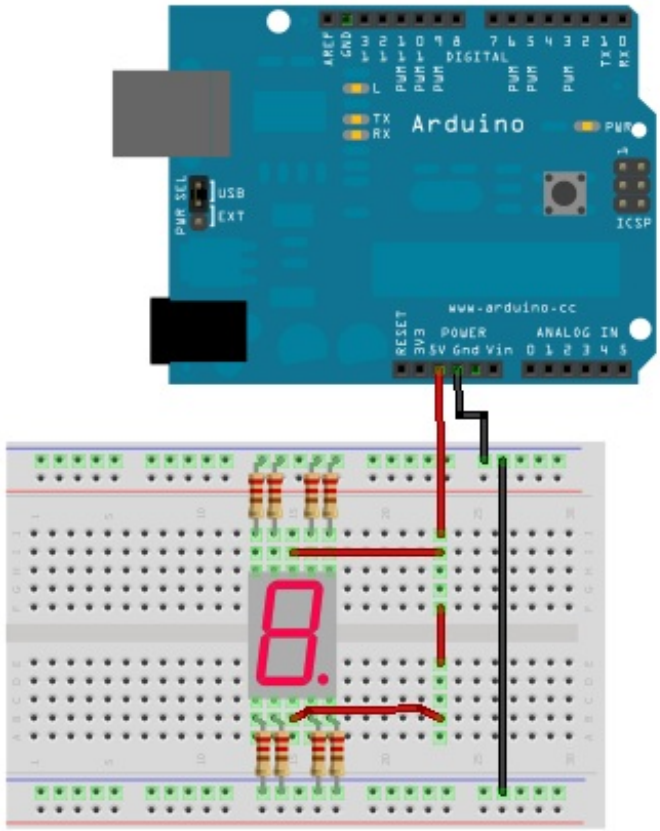
Nous cherchons à allumer les LED de l'afficheur, il est donc impératif de ne pas oubliez les résistances de limitations de courant !

### *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 $330\Omega$ 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 :






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.

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.

### *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 !

| 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 | |
| y | | x | x | x | | x | x |
| ° | x | x | | | | x | x |

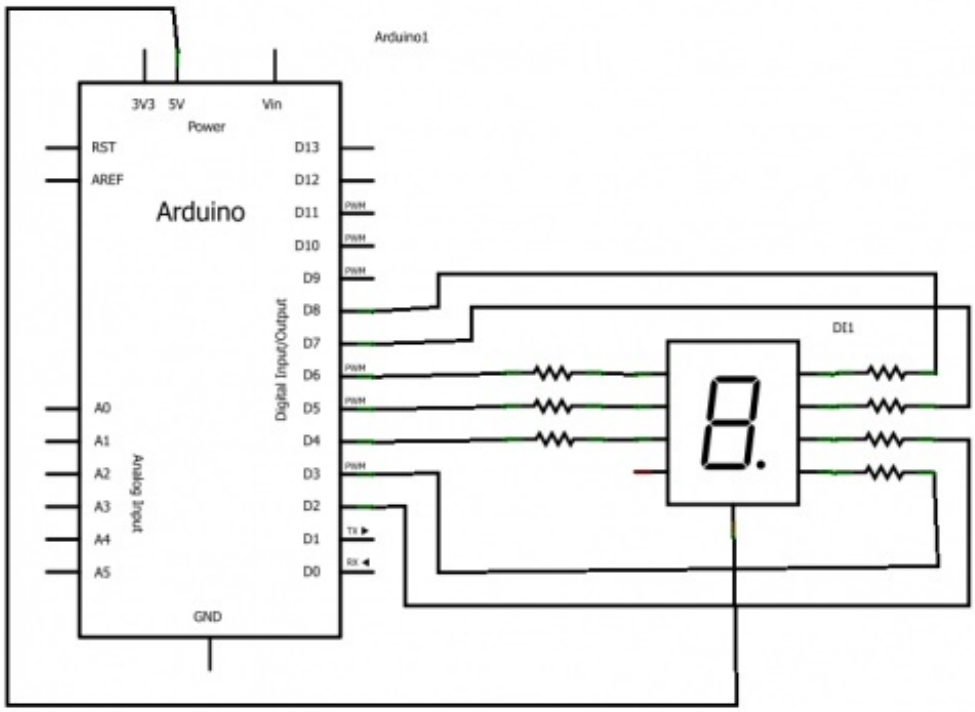
Aidez vous de ce tableau lorsque vous aurez à coder l'affichage de caractères !

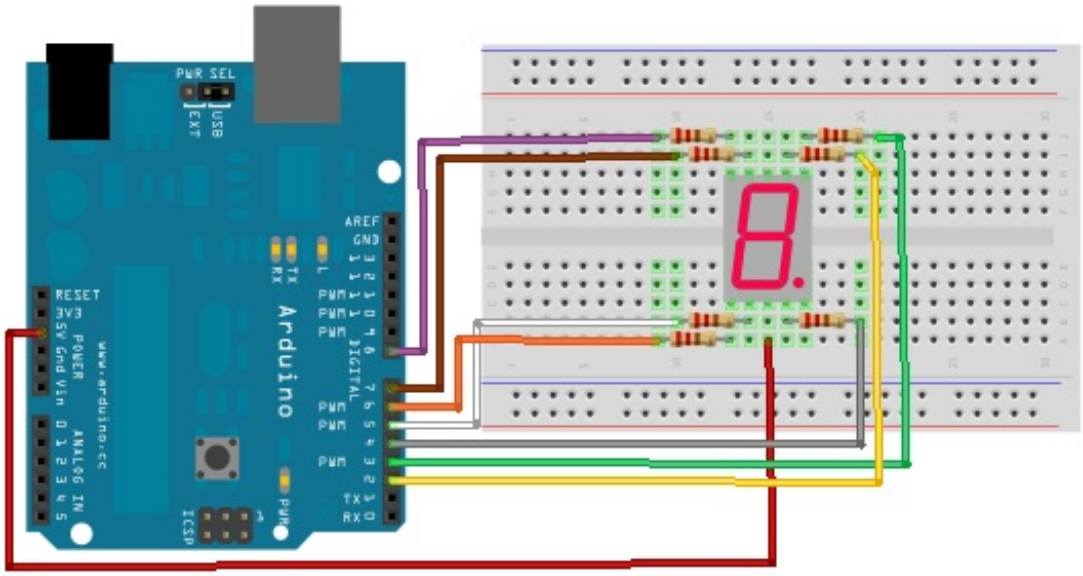
## Afficher son premier chiffre !

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.

## 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 :




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...

## Le programme

L'objectif du programme va être d'afficher un chiffre. Eh bien... c'est partit !

Quoi ?! Vous voulez de l'aide ? 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** (cliquez pour afficher)

Solution :

**Code : C**

```
/* 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 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);
}
```

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.).

## 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 fais rien d'autre que d'utiliser les afficheurs, cela ne nous gène pas, cependant, il est fort probable que vous serez amener 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 !

## 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.

### *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.

### *Principe du décodeur*

Sur un afficheur 7 segments, on peut représenter aisément les chiffres de 0 à 9 (et en insistant un peu les lettres de A à F). 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.

### *Choix du décodeur*

Nous allons utiliser le composant nommé MC14543B comme exemple. Tout d'abord, ouvrez ce lien dans un nouvel onglet, il vous menera 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 :

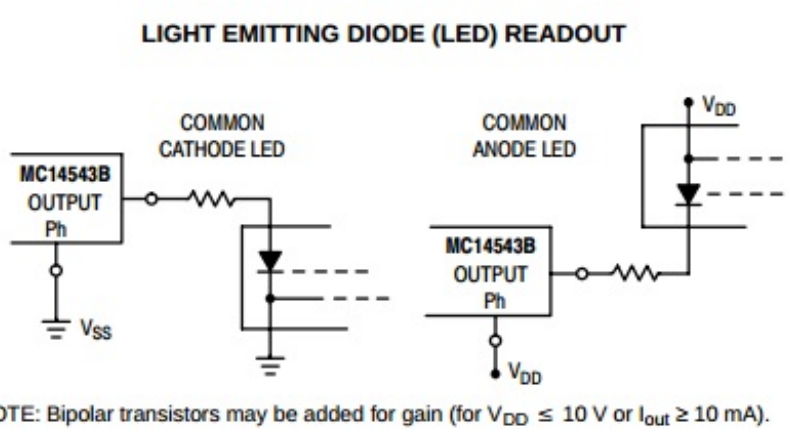
- Si l'on met la broche Bl (Blank, n°7) à un, toutes les sorties passent à zéro. En effet, comme son nom l'indique cette broche sert à effacer l'état de l'afficheur. Si vous ne voulez pas l'utiliser il faut donc la connecter à la masse pour la désactiver.
- Les entrées A, B, C et D (broches 5,3,2 et 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/RBO (n°4) sers 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
- Enfin, les deux broches d'alimentation sont la 8 (GND/VSS, masse) et la 16 (VCC, +5V)

N'oubliez pas de mettre des résistances de limitations de courant entre chaque segment et la broche de signal du circuit!

### *Fonctionnement*

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) défini 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 :



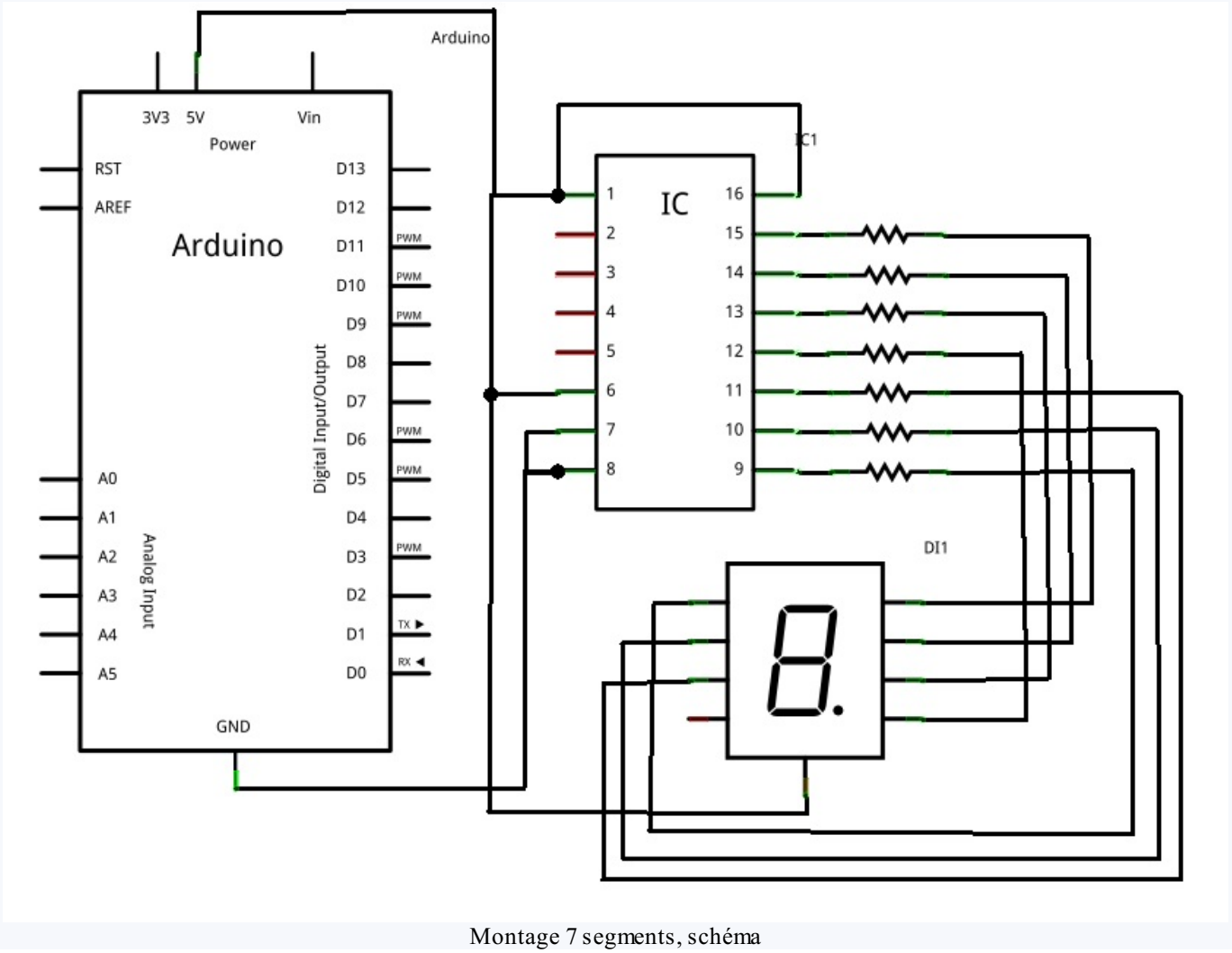

Branchement du MC14543B

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 !

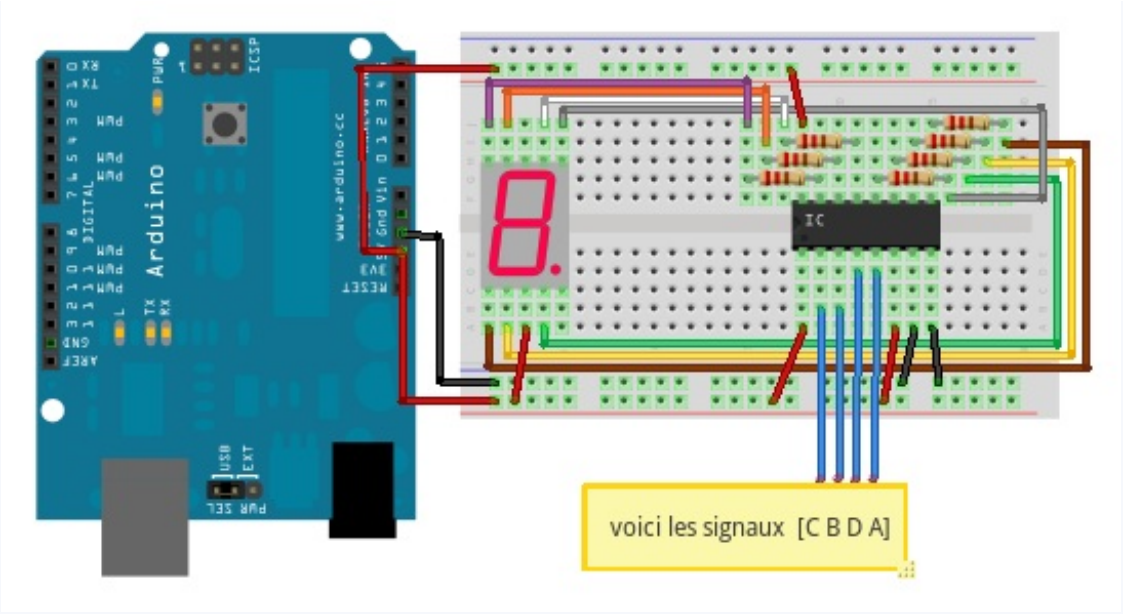
En titre d'exercice afin de vous permettre de mieux comprendre, je vous propose de changer les états des entrées A, B, C et D du décodeur pour observer ce qu'il affiche.

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** (cliquez pour afficher)




Montage 7 segments, schéma




Montage 7

segments, breadboard

## 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 parait fluide. On change une image toutes les 40 ms et votre œil n'a pas le temps de le voir, donc les images semble s'enchainer sans transition. Bref...

Ici, la même stratégie 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 géré un afficheur seul.

## 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.

### *Initialisation*

Vous avez l'habitude maintenant, nous allons commencer par définir les différentes broches d'entrées/sorties. Pour débuter (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 :

**Code : C**

```
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 sorites à
```

```
    l'état bas
       digitalWrite(bit_A, LOW);
       digitalWrite(bit_B, LOW);
       digitalWrite(bit_C, LOW);
       digitalWrite(bit_D, LOW);
    }
```

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 !

### *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 `affichage()` qui s'occupera donc de l'affichage (belle démonstration de ce qui est une évidence ).

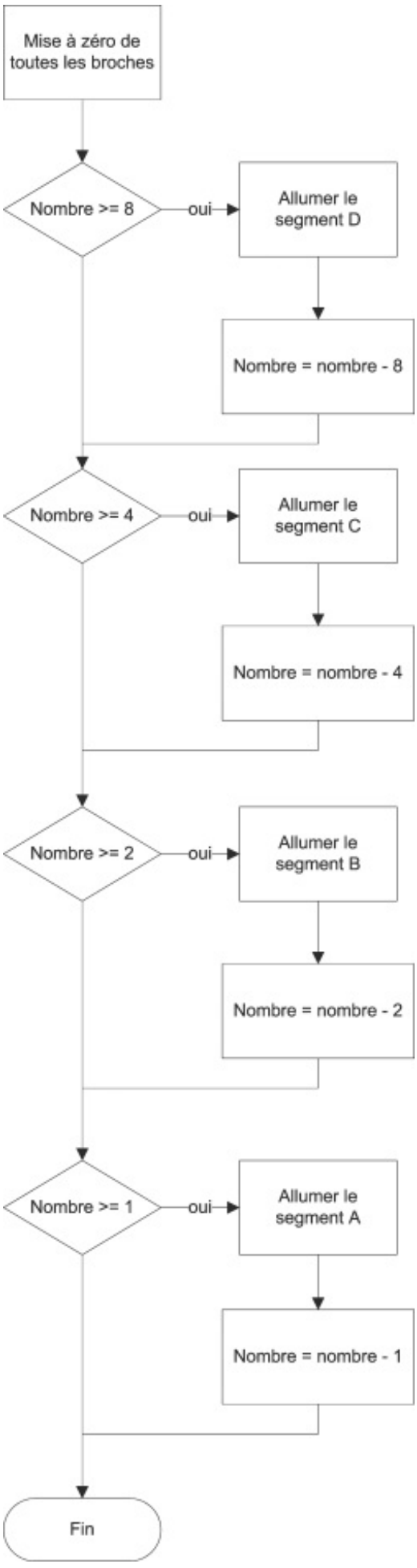
**Code : C**

```
void loop()
{
    char i=0; //variable "compteur"
    for(i=0; i<10; i++)
    {
        affichage(i); //on appel la fonction d'affichage
        delay(1000); //on attend 1 seconde
    }
}
```

### *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 ) 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 ont peu obtenir n'importe quel nombre de 0 à 15. Voici une démarche mathématique envisageable :




Organigramme décodeur 7 segments

On peut coder cette méthode de manière assez simple et direct, en suivant cet organigramme :

**Code : C**

```
//fonction écrivant sur un seul afficheur
void afficher(char chiffre)
{
   //on met à zéro tout les segments
   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;
   }
}
```

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 | $(0101)_2$ |
| 6 | $(0110)_2$ |
| 7 | $(0111)_2$ |
| 8 | $(1000)_2$ |
| 9 | $(1001)_2$ |

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 :

**Code : C**

```
//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;
            }
        }
```

Mais, c'est bien trop lourd à écrire. Enfin c'est vous qui voyez.

## 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*.

### *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 utilisé 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.

## Un peu d'électronique...

Pour faire commuter nos deux afficheurs, vous allez avoir besoin d'un nouveau composant, j'ai nommé : le **transistor** !

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*).

### *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.

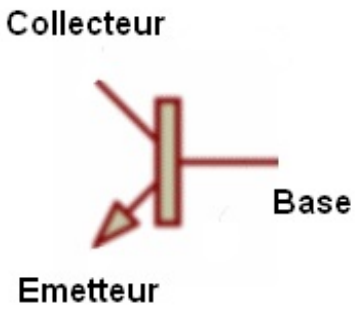
Un transistor, cela ressemble à ça :

*Photo d'un transistor*

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 à une broche d'entrée (**collecteur**), une broche de sortie (**émetteur**) et une broche de commande (**base**).

Son symbole est le suivant :




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. Quoi qu'il en soit, nous n'utiliserons que des transistors NPN dans ce chapitre.

### *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**.

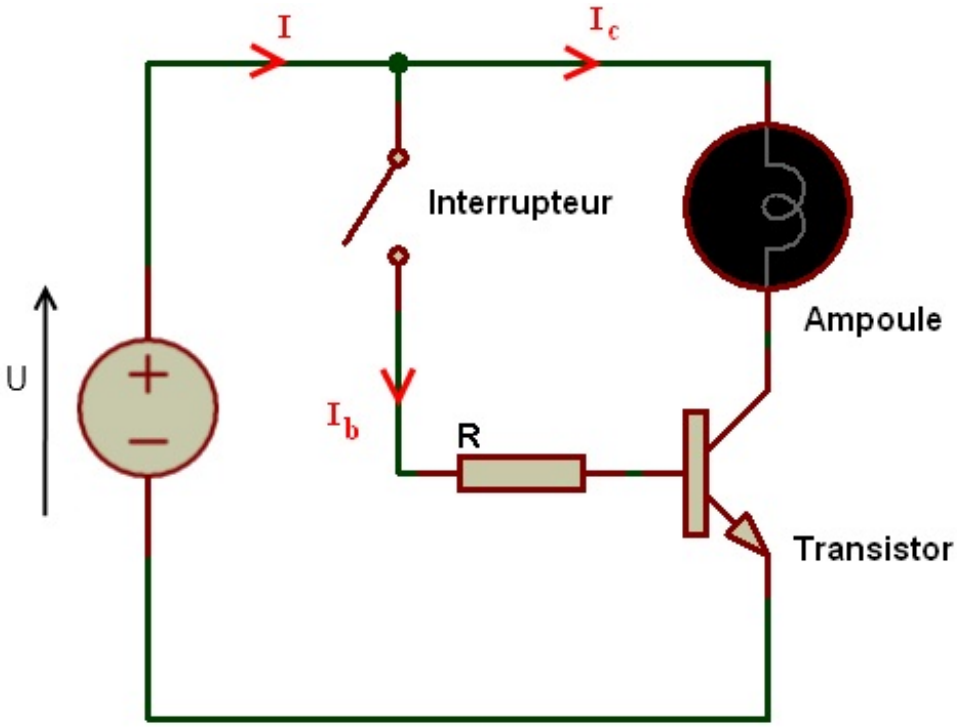
Ceci est une présentation très vulgarisée et simplifiée sur le transistor pour l'utilisation que nous en ferons ici. Les usages et possibilités des transistors sont très nombreux et ils mériteraient un big-tuto à eux seuls ! Si vous voulez plus d'informations, rendez-vous sur le cours sur l'électronique 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 ! 

### *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 :

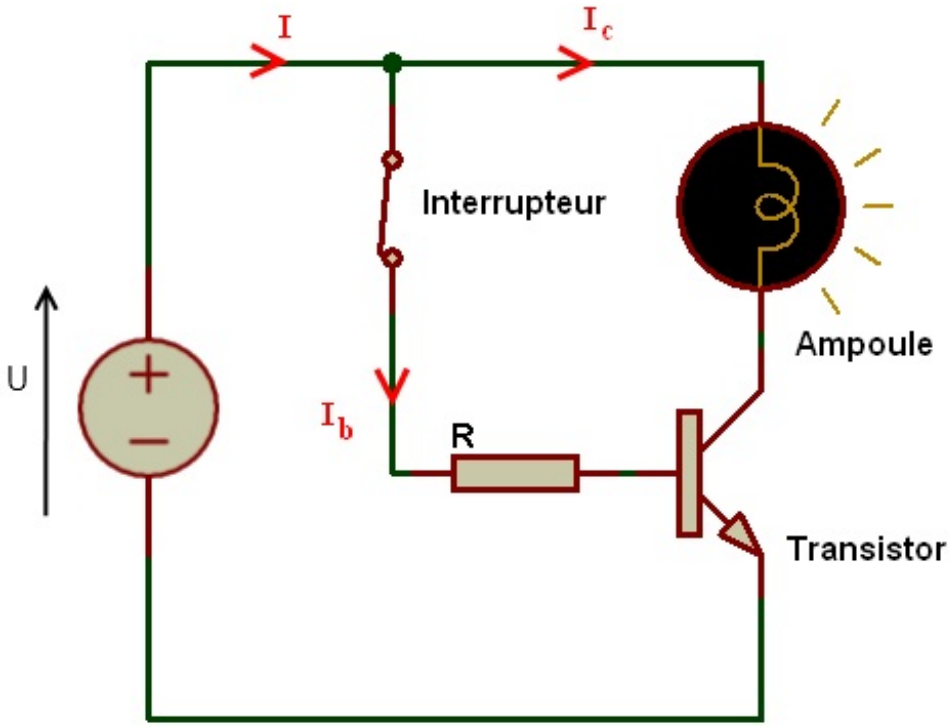



*Câblage du transistor en commutation*

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 :




*L'ampoule est allumée*

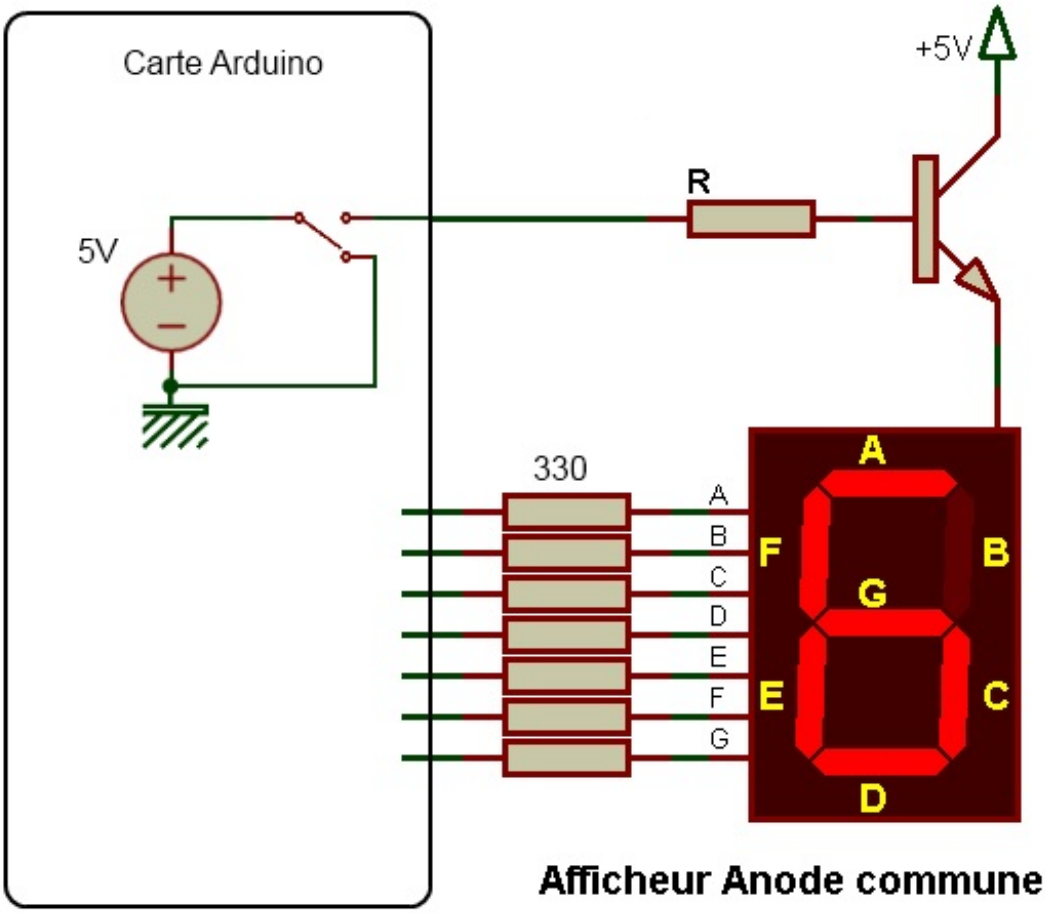
Que se passe-t-il ? Eh bien la base du transistor, qui était jusqu'à présent "en l'air", est parcourue par un courant électrique. Cette cause à pour conséquence de rendre le transistor **passant** ou **saturé** et permet au courant de s'établir à travers l'ampoule. Soit $I_C \neq 0$ car $I_B \neq 0$.

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.

### *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 :

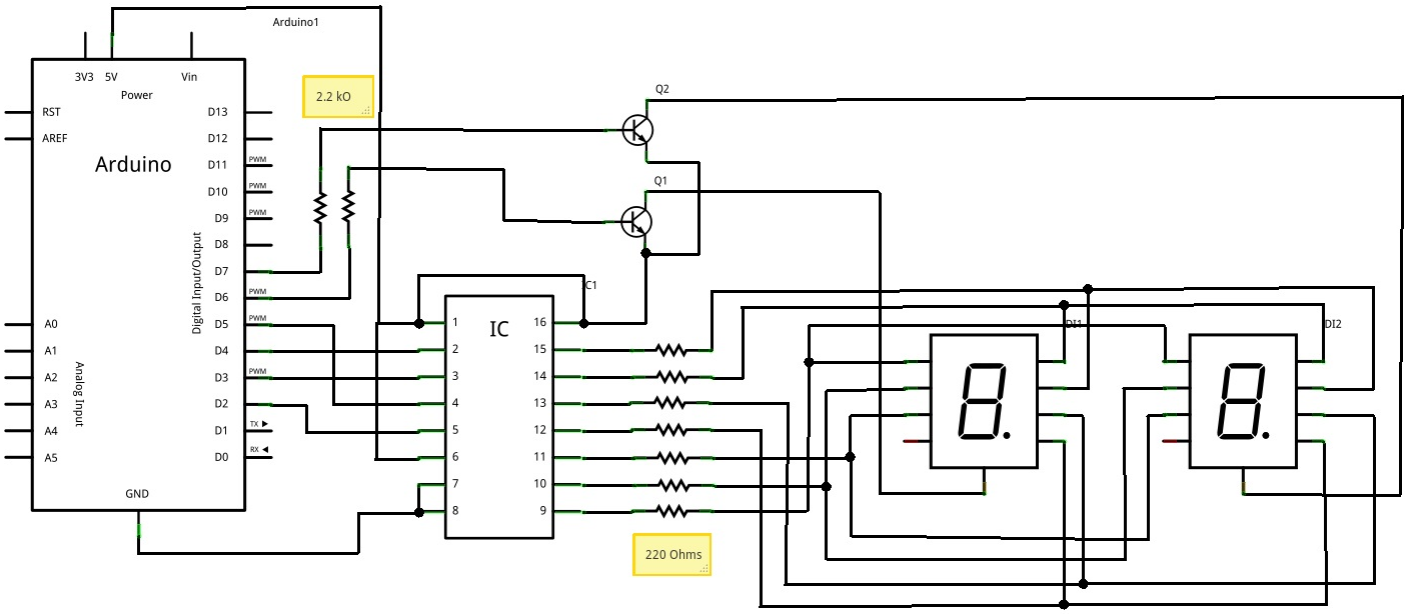



LA carte Arduino va soit mettre à la masse la base du transistor, soit la mettre à +5V. Dans le premier cas, il sera bloqué et l'afficheur sera éteint, dans le second il sera saturé et l'afficheur allumé.

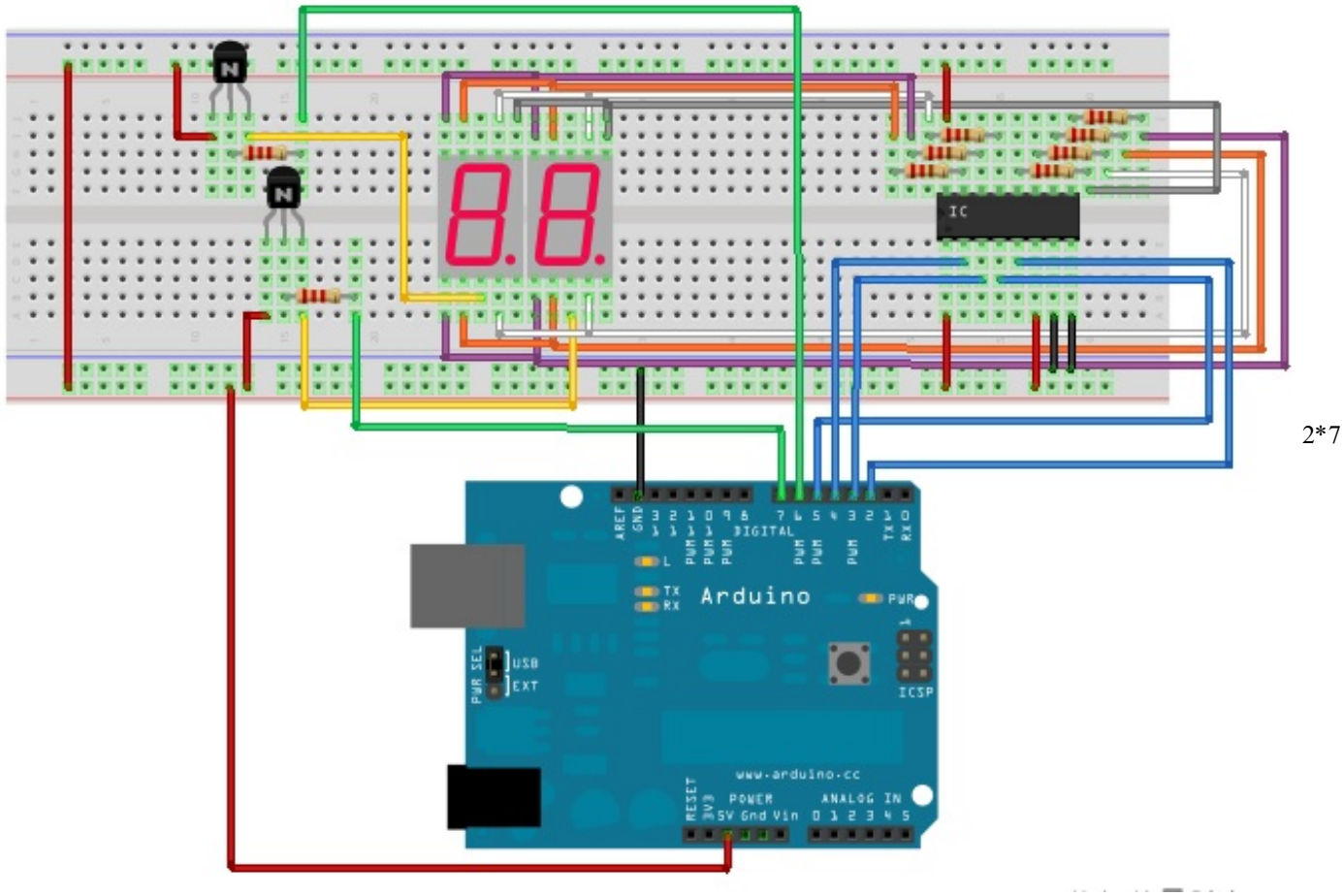
Il en est de même pour chaque broche de l'afficheur. Elles seront au +5V ou à la masse selon la configuration que l'on aura définie dans le programme.

### *Schéma final*

Et comme vous l'attendez surement depuis tout à l'heure, voici le schéma tant attendu (nous verrons juste après comment programmer ce nouveau montage) !




2*7 segments schéma




2*7 segments breadboard

### *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 !

## ...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.

**Code : C**

```
//définition des broches du décodeur 7 segments (vous pouvez changer
les numéros si bon vous semble)
const int bit_A = 2;
const int bit_B = 3;
const int bit_C = 4;
const int bit_D = 5;

//définitions des broches des transistors pour chaque afficheur
(dizaines et unités)
const int alim_dizaine = 6;
const int alim_unite = 7;

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
{
 for(char i = 0; i<100; i++) //boucle qui permet de compter de 0 à
99 (= 100 valeurs)
 {
  afficher_nombre(i); //appel de la fonction affichage avec envoi
du nombre à afficher
 }
}

//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
```

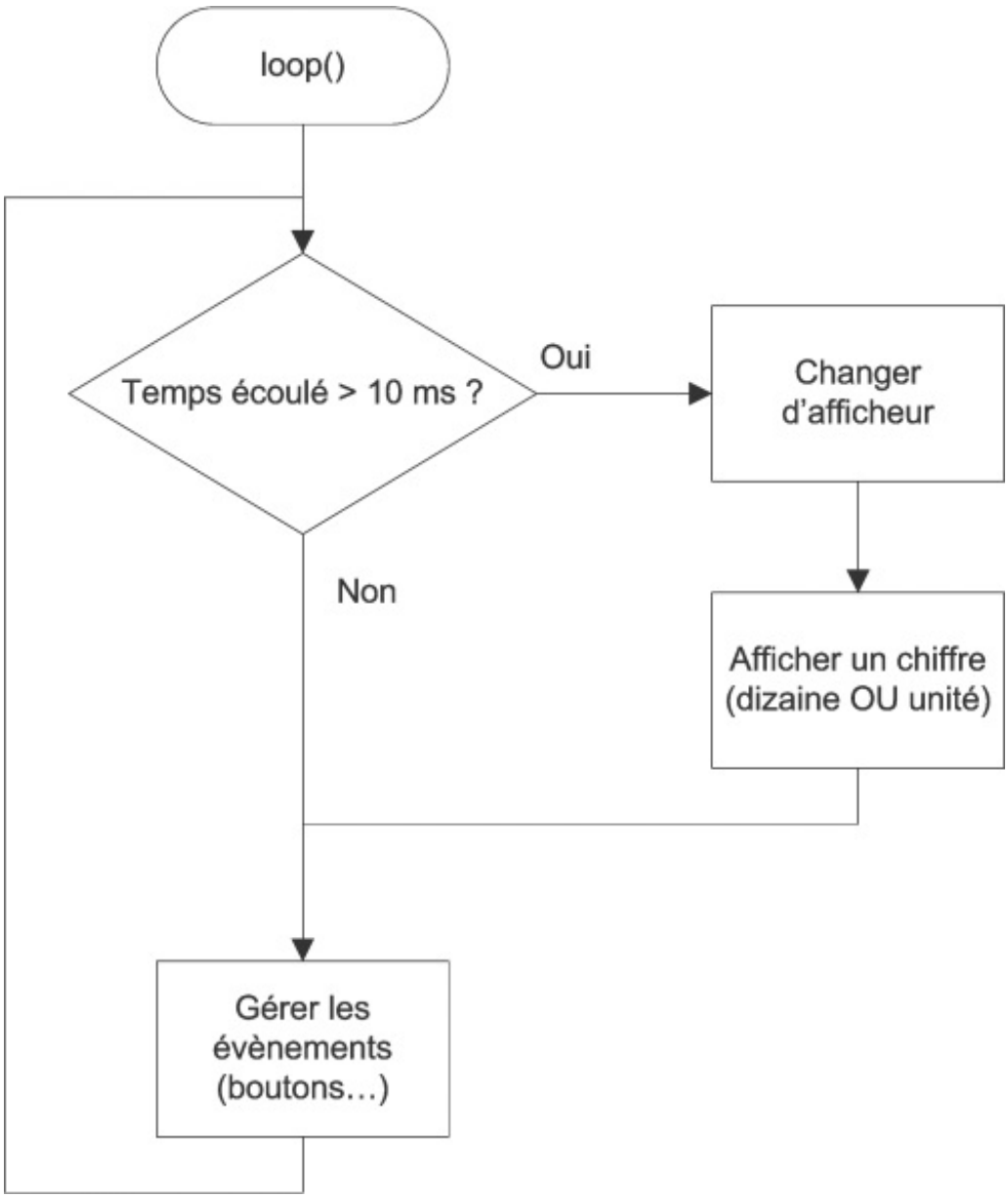
```
        }

 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
  digitalWrite(alim_dizaine, HIGH); /* le transistor de l'afficheur
des dizaines est saturé,
donc l'afficheur est allumé */
  afficher(dizaine); //on appel la fonction qui permet d'afficher
le chiffre dizaine
                digitalWrite(alim_unite, LOW); // l'autre transistor
est bloqué et l'afficheur éteint
                delay(10);

  //puis les unités pendant 10 ms
                digitalWrite(alim_dizaine, LOW); //on éteint le
transistor allumé
  afficher(unite); //on appel la fonction qui permet d'afficher le
chiffre unité
  digitalWrite(alim_unite, HIGH); //et on allume l'autre
                delay(10);
 }
}

//fonction écrivant sur un seul afficheur
//on utilise le même principe que vu plus haut
void afficher(char chiffre)
{
 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;
 }
}

//le code est terminé
```

Voilà donc la vidéo présentant le résultat final :

## Contraintes des évènements

Comme vous l'avez vu juste avant, afficher de manière alternative n'est pas trop difficile. Cependant, vous avez surement 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 trèèèèèès rapide pourra toujours passer inaperçu).

Voici un exemple de programmation de la boucle principal (suivi de ses fonctions annexes) :

**Code : C**

```
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)
  {
   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;
   }
  }
```

Si vous voulez tester le phénomène de persistance rétinienne, vous pouvez changer le temps de la boucle de rafraichissement (ligne 9). Si vous l'augmenter, vous commencerez à vois 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...

# [TP] zParking

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 ) 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.

## 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** !

### *Histoire*

Le maire de zCity à 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 !

### *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 :

# Bon courage !

## 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 électronique). Nous commencerons par voir le schéma électronique, puis ensuite nous rentrerons dans le code.

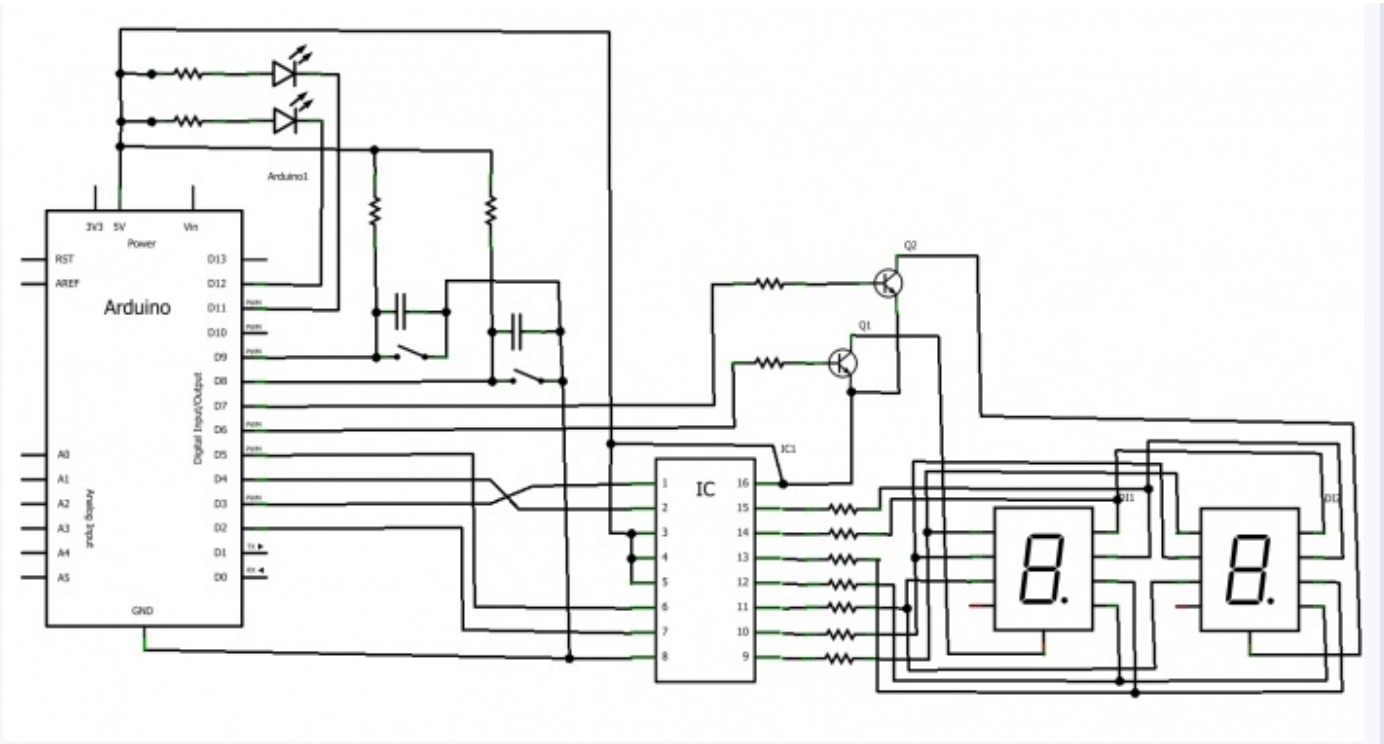
## Montage

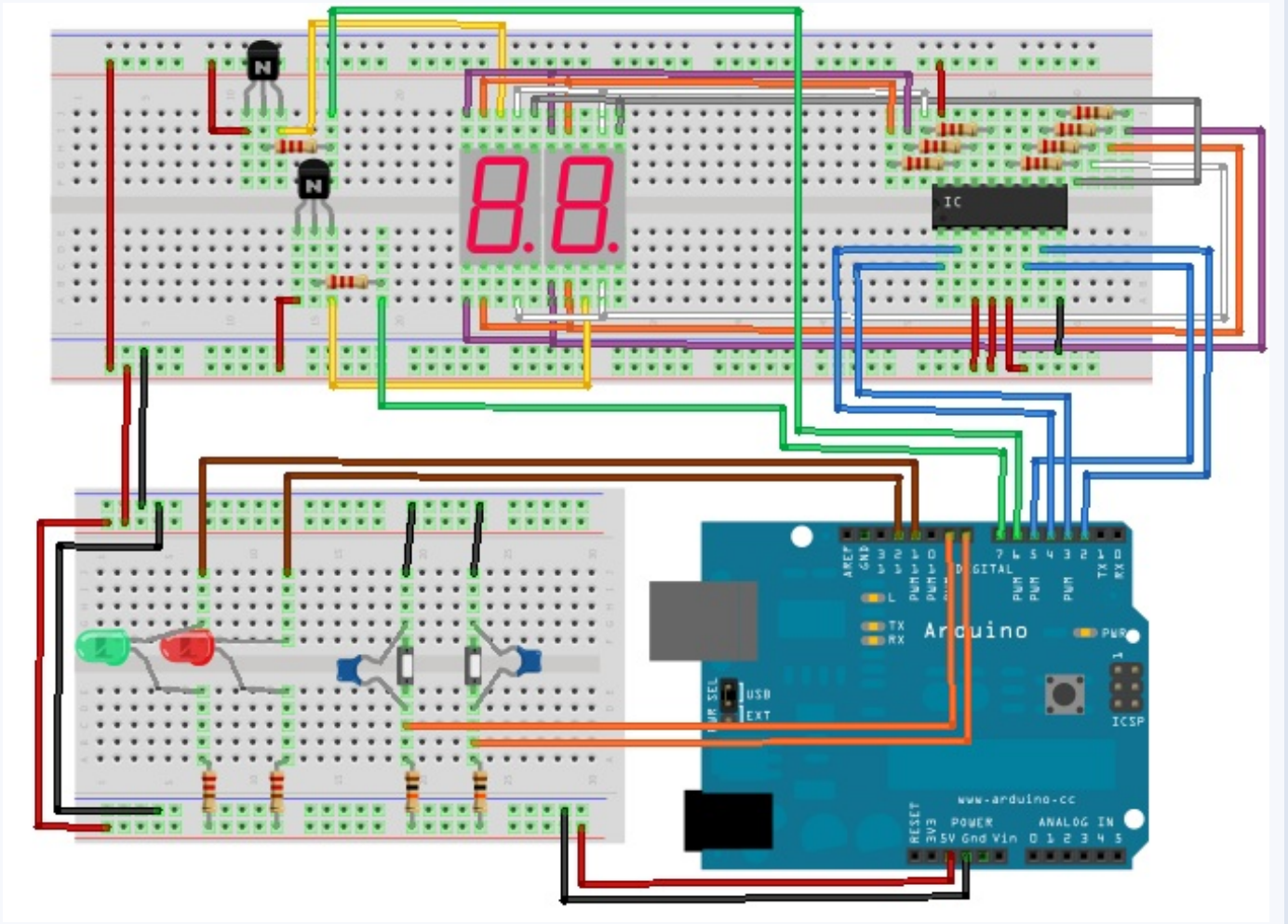
Le montage électronique est la base de ce qui va nous servir pour réaliser le système. Une fois qu'il est terminé on pourra l'utiliser grâce aux entrées/sorties de la carte Arduino et lui faire faire pleins de choses. Mais ça, vous le savez déjà. Alors ici pas de grand discours, il "suffit" de reprendre les différents blocs vus un par un dans les chapitres précédents et de faire le montage de façon simple.

### *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** (cliquez pour afficher)




### *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 !

## Programme

Nous allons maintenant voir une solution de programme pour le problème de départ. La vôtre sera peut-être (voire surement) 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  )

### *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.

- Deux 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** (cliquez pour afficher)

**Code : C**

```
//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 = 99; //contenu des places dispos
bool afficheur = false;
long temps;
```

### *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).

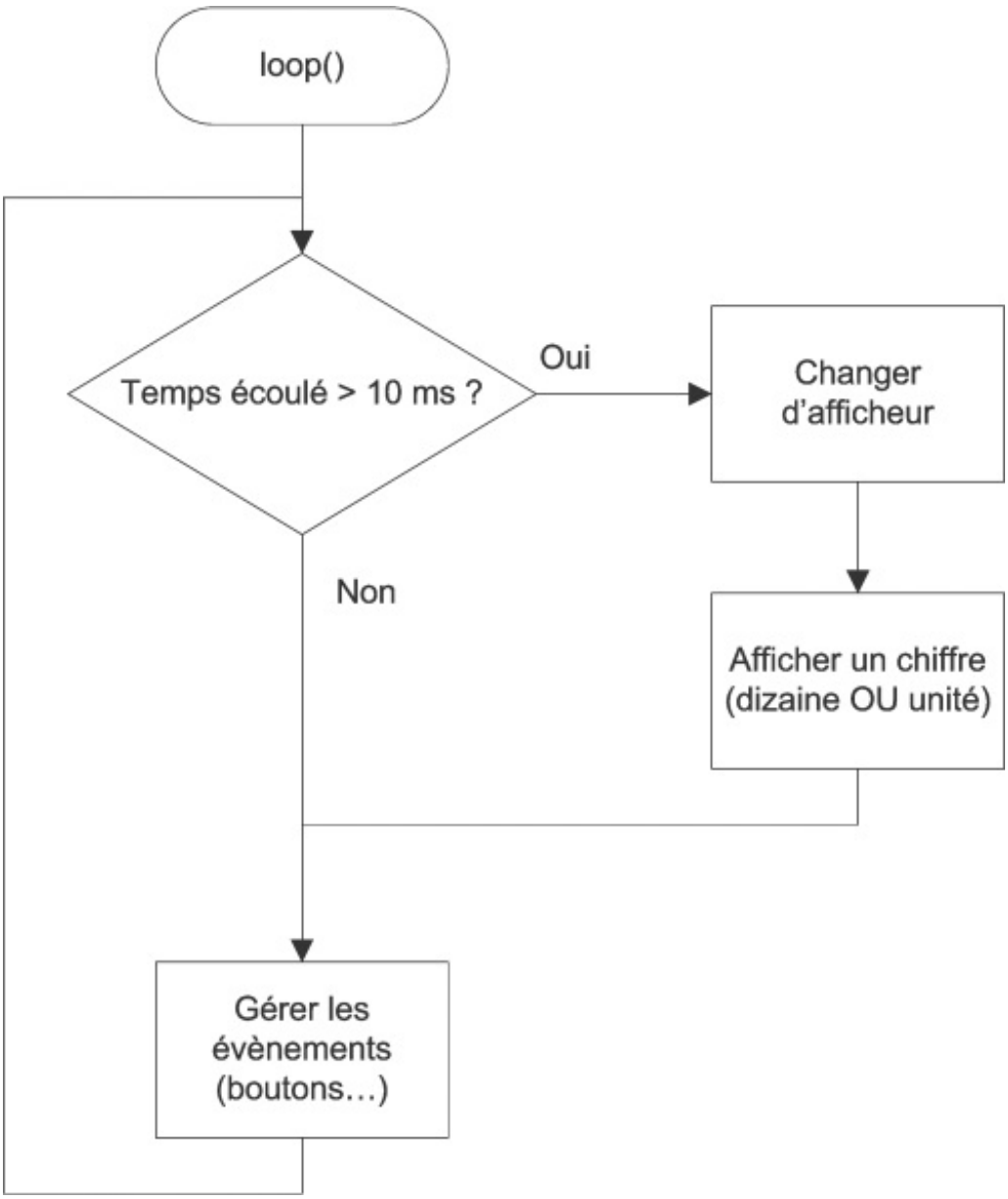
**Secret** (cliquez pour afficher)

**Code : C**

```
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); //rappelons que dans cette
configuration, la LED est éteinte à l'état HIGH
 digitalWrite(led_verte, LOW); //vert par défaut

 temps = millis(); //enregistre "l'heure"
}
```

### *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** (cliquez pour afficher)

**Code : C**

```
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 éteindres
 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.

### *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 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** (cliquez pour afficher)

**Code : C**

```
//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 écriveant 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 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 !)

**Code : C**

```
//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
```

## 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 !

# Ajouter des sorties (numériques) à l'Arduino

Dans ce chapitre "bonus", nous allons vous faire découvrir comment ajouter des sorties numériques à votre carte Arduino. Car en effet, pour vos projets les plus fous, vous serez certainement amené à avoir besoin d'un grand nombre de sorties. Là il y a deux choix : le premier serait d'opter pour une carte Arduino qui dispose de plus de sorties, telle que la Arduino Mega ; mais dans le cas où vous aurez besoin d'un *giga super ultra* grand nombre de sorties, même la Mega ne suffira pas. Le deuxième choix c'est donc... de lire ce chapitre.

Ce que vous allez découvrir se révélera fort utile, soyez-en certains. Prenons l'exemple suivant : dans le cas où vous devrez gérer un grand nombre de LED pour réaliser un afficheur comme l'on en trouve parfois dans les vitrines de magasins, vous serez très vite limité par le nombre de sorties de votre Arduino. Surtout si votre afficheur contient plus de 1000 LED ! Ce chapitre va alors vous aider dans de pareils cas, car nous allons vous présenter un composant spécialisé dans ce domaine : le **74HC595**.

## Présentation du 74HC595

### Principe

Comme je viens de l'énoncer, il peut arriver qu'il vous faille utiliser plus de broches qu'il n'en existe sur un micro-contrôleur, votre carte Arduino en l'occurrence (ou plutôt, l'ATMEGA328 présent sur votre carte Arduino). Dans cette idée, des ingénieurs ont développé un composant que l'on pourrait qualifier de "décodeur série -> parallèle". D'une manière assez simple, cela consiste à envoyer un octet de données (8 bits) à ce composant qui va alors décoder l'information reçue et changer l'état de chacune de ses sorties en conséquence. Le composant que nous avons choisi de vous faire utiliser dispose de huit sorties de données pour une seule entrée de données.

Concrètement, cela signifie que lorsque l'on enverra l'octet suivant : 00011000 au décodeur 74HC595, il va changer l'état (HAUT ou BAS) de ses sorties. On verra alors, en supposant qu'il y a une LED de connectée sur chacune de ses sorties, les 2 LED du "milieu" (géographiquement parlant) qui seront dans un état opposé de leurs congénères. Ainsi, en utilisant seulement deux sorties de votre carte Arduino, on peut *virtuellement* en utiliser 8 (voir beaucoup plus mais nous verrons cela plus tard).

### Le composant

Rentrons maintenant dans les entrailles de ce fameux 595. Pour cela nous utiliserons cette datasheet tout au long du tuto.

#### *Brochage*

Lisons ensemble quelques pages.
La première nous donne, de par le titre, la fonctionnalité du composant. Elle est importante car l'on sait à ce moment à quel composant nous allons avoir affaire.
La seconde apporte déjà quelques informations utiles outre la fonctionnalité. Au-delà du résumé qu'il est toujours bon de lire, les caractéristiques du composant sont détaillées. On apprend également que ce composant peut fonctionner jusqu'à une fréquence de 170MHz. C'est très très rapide par rapport à notre carte Arduino qui tourne à 16MHz, nous sommes tranquilles de ce côté-là.
Continuons...
C'est la page 4 qui nous intéresse vraiment ici. On y retrouve le tableau et la figure suivante :

**PINNING**

| PIN | SYMBOL | DESCRIPTION |
|---|---|---|
| 1, 2, 3, 4, 5, 6, 7 and 15 | $Q_1$, $Q_2$, $Q_3$, $Q_4$, $Q_5$, $Q_6$, $Q_7$ and $Q_0$ | parallel data output |
| 8 | GND | ground (0 V) |
| 9 | $Q_7'$ | serial data output |
| 10 | $\overline{MR}$ | master reset (active LOW) |
| 11 | $SH_{CP}$ | shift register clock input |
| 12 | $ST_{CP}$ | storage register clock input |
| 13 | $\overline{OE}$ | output enable input (active LOW) |
| 14 | $D_S$ | serial data input |
| 16 | $V_{CC}$ | DC supply voltage |

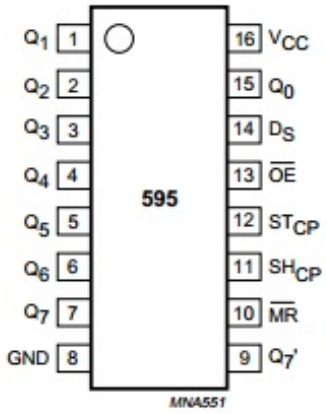



Fig.1 Pin configuration.

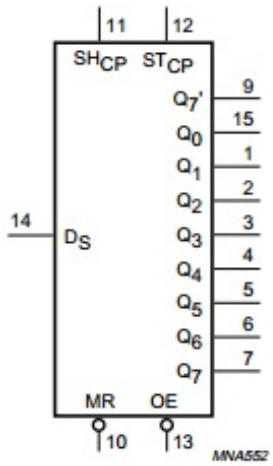



Fig.2 Logic symbol.

Brochage du 595

Avec ce dernier, on va pouvoir faire le lien entre le nom de chaque broche et leur rôle. De plus, nous savons où elles sont placées sur le composant. Nous avons donc les sorties et la masse à gauche et les broches de commande à droite (plus la sortie Q0) et l'alimentation.
Voyons maintenant comment faire fonctionner tout cela.

### *Fonctionnement*

Comme tout composant électronique, il faut commencer par l'alimenter pour le faire fonctionner. Le tableau que nous avons vu juste au-dessus nous indique que les broches d'alimentation sont la broche 16 (VCC) et la broche 8 (masse). Quelques pages plus loin dans la datasheet, page 7 précisément, nous voyons la tension à appliquer pour l'alimenter : entre 2V et 5.5V (et idéalement 5.0V). Une fois que ce dernier est alimenté, il faut se renseigner sur le rôle des broches pour savoir comment l'utiliser correctement. Pour cela il faut revenir sur le tableau précédent et la table de vérité qui le suit.

On découvre donc que les sorties sont les broches de 1 à 7 et la broche 15 (*Qn*) ; l'entrée des données série, qui va commander les sorties du composant, se trouve sur la broche 14 (*serial data input*) ; une sortie particulière est disponible sur la broche 9 (*serial data output*, nous y reviendrons à la fin de ce chapitre).

Sur la broche 10 on trouve le *Master Reset*, pour mettre à zéro toutes les sorties. Elle est active à l'état BAS. Vous ferez alors attention, dans le cas où vous utiliseriez cette sortie, de la forcer à un état logique HAUT, en la reliant par exemple au +5V ou bien à une broche de l'Arduino que vous ne mettrez à l'état BAS que lorsque vous voudrez mettre toutes les sorties du 74HC595 à l'état bas. Nous, nous mettrons cette sortie sur le +5V.

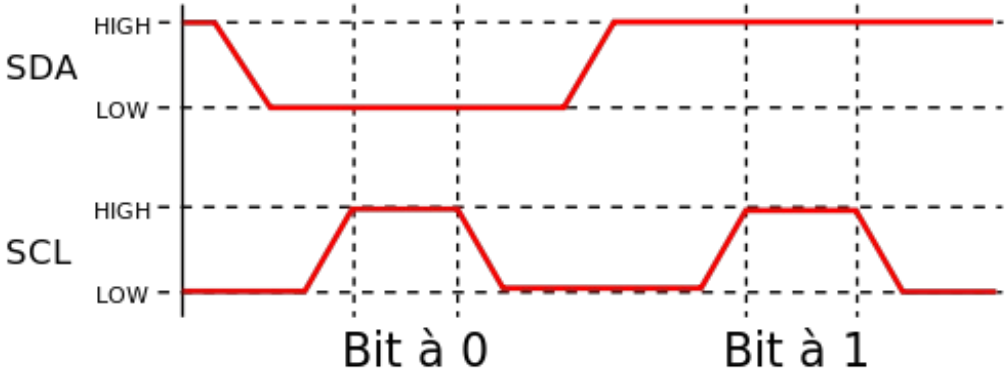
La broche 13, *output enable input*, est un broche de sélection qui permet d'inhiber les sorties. En clair, cela signifie que lorsque cette broche n'a pas l'état logique requis, les sorties du 74HC595 ne seront pas utilisables. Soit vous choisissez de l'utiliser en la connectant à une sortie de l'Arduino, soit on la force à l'état logique BAS pour utiliser pleinement chaque sortie. Nous, nous la relierons à la masse.

Deux dernières broches sont importantes. La n°11 et la n°12. Ce sont des "horloges". Nous allons expliquer quelle fonction elles remplissent.

Lorsque nous envoyons un ordre au 74HC595, nous envoyons cet ordre sous forme d'états logiques qui se suivent. Par exemple l'ordre 01100011. Cet ordre est composé de 8 états logiques, ou bits, et forme un octet. Cet ordre va précisément définir l'état de sortie de chacune des sorties du 74HC595. Le problème c'est que ce composant ne peut pas dissocier chaque bit qui arrive.

Prenons le cas des trois zéros qui se suivent dans l'octet que nous envoyons. On envoie le premier 0, la tension sur la ligne est alors de 0V. Le second 0 est envoyé, la tension est toujours de 0V. Enfin le dernier zéro est envoyé, avec la même tension de 0V puis vient un changement de tension à 5V avec l'envoi du 1 qui suit les trois 0. Au final, le composant n'aura vu en entrée qu'un seul 0 puisqu'il n'y a eu aucun changement d'état. De plus, il ne peut pas savoir quelle est la durée des états logiques qu'on lui envoie. S'il le connaissait, ce temps de "vie" des états logiques qu'on lui envoie, il pourrait aisément décoder l'ordre transmis. En effet, il pourrait se dire: "tiens ce bit (état logique) dépasse 10ms, donc un deuxième bit l'accompagne et est aussi au niveau logique 0". Encore 10ms d'écoulée et toujours pas de changement, eh bien c'est un troisième bit au niveau 0 qui vient d'arriver. C'est dans ce cas de figure que l'ordre reçu sera compris dans sa totalité par le composant.

Bon, eh bien c'est là qu'intervient le **signal d'horloge**. Ce signal est en fait là dans l'unique but de dire si c'est un nouveau bit qui arrive, puisque le 74HC595 n'est pas capable de le voir tout seul. En fait, c'est très simple, l'horloge est un signal carré fixé à une certaine fréquence. À chaque front montant (quand le signal d'horloge passe du niveau 0 au niveau 1), le 74HC595 saura que sur son entrée, c'est un nouveau bit qui arrive. Il pourra alors facilement voir s'il y a trois 0 qui se suivent. Ce chronogramme vous aidera à mettre du concret dans vos idées :



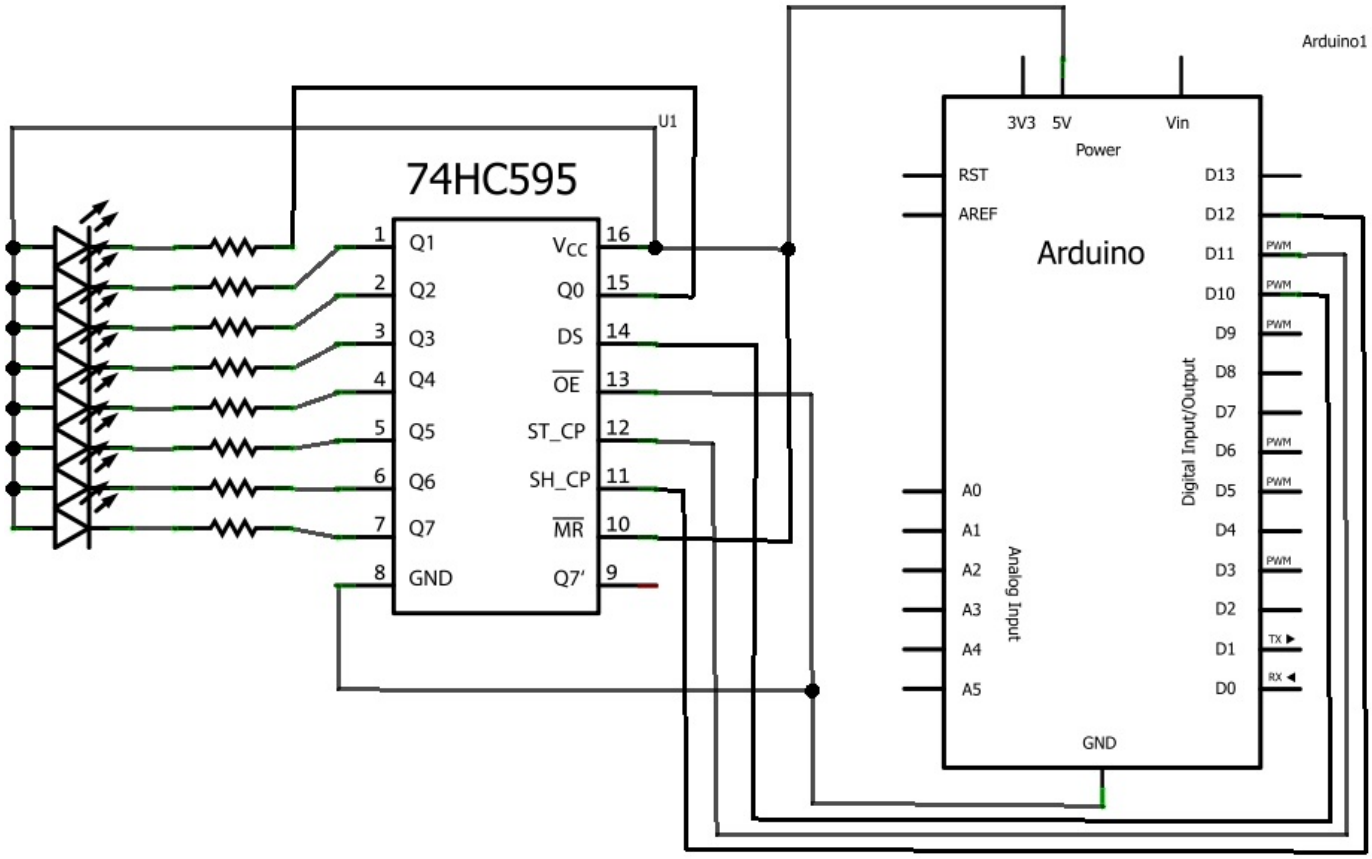

Source : Wikipédia -
SDA est le signal de données, l'ordre que l'on envoie ; SCL est le signal d'horloge

Pour câbler cette horloge, il faudra connecter une broche de l'Arduino à la broche numéro 11 du 74HC595. Ce signal travaillera donc en corrélation avec le signal de données relié sur la broche 14 du composant.

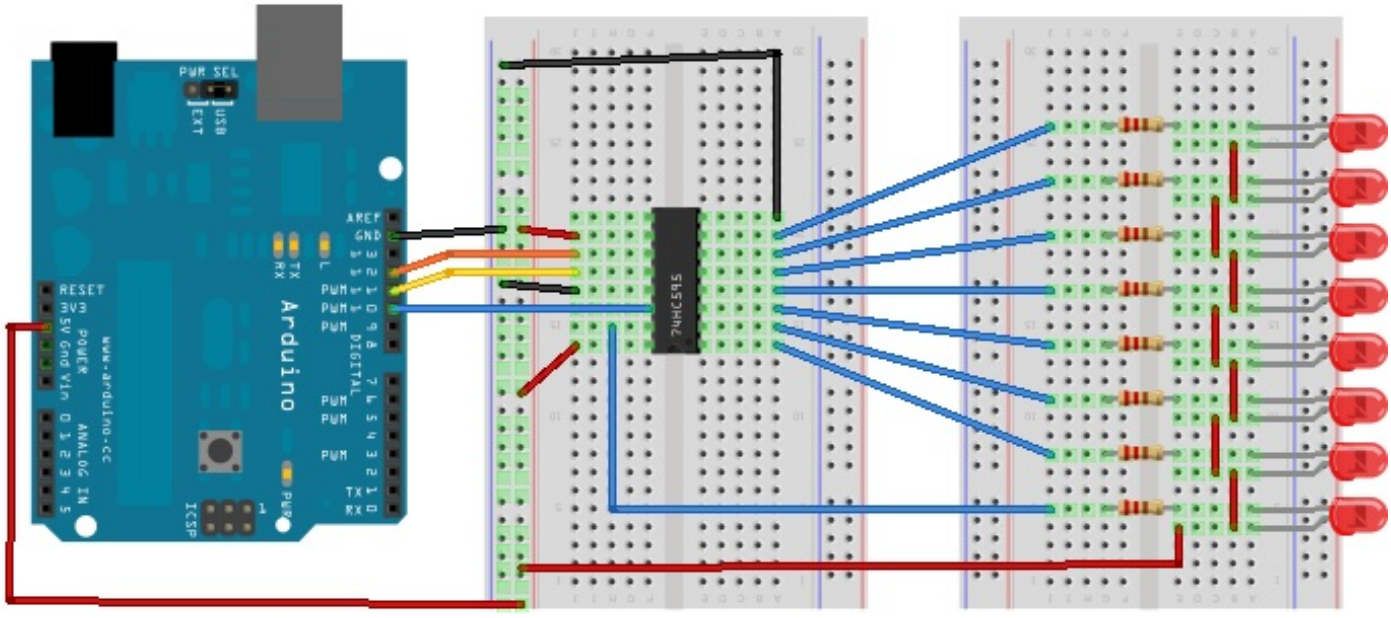
La seconde horloge pourrait aussi s'appeler "verrou". Elle sert à déterminer si le composant doit mettre à jour les états de ses sorties ou non, en fonction de l'ordre qui est transmis. Lorsque ce signal passe de l'état BAS à l'état HAUT, le composant change les niveaux logiques de ses sorties en fonction des bits de données reçues. En clair, il copie les huit derniers bits transmis sur ses sorties. Ce verrou se présente sur la broche 12.

### *Montage*

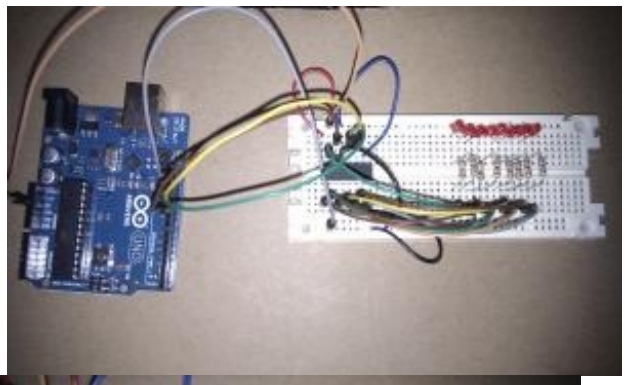
Voici un petit montage à titre d'illustration que nous utiliserons par la suite. Je vous laisse faire le câblage sur votre breadboard comme bon vous semble, pendant ce temps je vais aller me siroter un bon petit café.



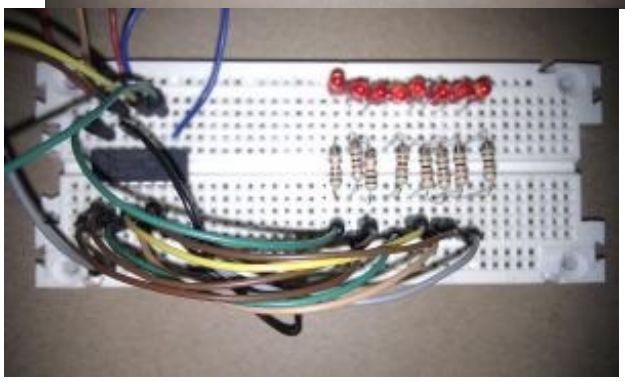

Montage du 595 schéma




Montage du 595 breadboard



Montage du HC595 et 8 LEDs



Montage du HC595 et 8 LEDs (zoom)

## Programmons pour utiliser ce composant
## Envoyer un ordre au 74HC595

Nous allons maintenant voir comment utiliser le composant de manière logicielle, avec Arduino. Pour cela, je vais vous expliquer la façon de faire pour lui envoyer un ordre. Puis, nous créerons nous-mêmes la fonction qui va commander le 74HC595.

### *Le protocole*

Nous le verrons dans le chapitre sur la liaison série plus en détail, le protocole est en fait un moyen qui permet de faire communiquer deux dispositifs. C'est une sorte de convention qui établit des règles de langage. Par exemple, si deux personnes parlent deux langues différentes, elles vont avoir un mal fou à se comprendre l'une de l'autre. Eh bien le protocole sert à imposer un langage qui leur permettra de se comprendre. En l'occurrence, il va s'agir de l'anglais.

Bon, cet exemple n'est pas parfait et a ses limites, c'est avant tout pour vous donner une vague idée de ce qu'est un protocole. Comme je vous l'ai dit, on en reparlera dans la partie suivante.

Nous l'avons vu tout à l'heure, pour envoyer un ordre au composant, il faut lui transmettre une série de bits. Autrement dit, il faut envoyer des bits les uns après les autres sur la même broche d'entrée. Cette broche sera nommée "data".

Ensuite, rappelez-vous, le composant a besoin de savoir quand lire la donnée, quand est-ce qu'un nouveau bit est arrivé ? C'est donc le rôle de l'horloge, ce que je vous expliquais plus haut. On pourrait s'imaginer qu'elle dit au composant : " Top ! tu peux lire la valeur car c'est un autre bit qui arrive sur ton entrée ! ".

Enfin, une troisième broche où l'on va amener l'horloge de verrou sert à dire au composant : " Nous sommes en train de mettre à jour la valeur de tes sorties, alors le temps de la mise à jour, garde chaque sortie à son état actuel ". Quand elle changera d'état, en passant du niveau BAS au niveau HAUT (front montant), cela donnera le "top" au composant pour qu'il puisse mettre à jour ses sorties avec les nouvelles valeurs.

Si jamais vous voulez économiser une broche sur votre Arduino, l'horloge de verrou peut être reliée avec l'horloge de données. Dans ce cas l'affichage va "scintiller" lors de la mise à jour car les sorties seront rafraîchies en même temps que la donnée arrive. Ce n'est pas gênant pour faire de l'affichage sur des LEDs mais ça peut l'être beaucoup plus si on a un composant qui réagit en fonction du 595.

### *Création de la fonction d'envoi*

Passons à la création de la fonction d'envoi des données. C'est avec cette fonction que nous enverrons les ordres au 74HC595, pour lui dire par exemple d'allumer une LED sur sa sortie 4. On va donc faire un peu de programmation, aller zou !

Commençons par nommer judicieusement cette fonction : `envoi_ordre()`.

Cette fonction va prendre quatre paramètres. Le premier sera le numéro de la broche de données. Nous l'appellerons "dataPin". Le second sera similaire puisque ce sera le numéro de la broche d'horloge. Nous l'appellerons "clockPin". Le troisième sera le "sens" d'envoi des données, je reviendrai là-dessus ensuite. Enfin le dernier paramètre sera la donnée elle-même, donc un char (sur 8 bits, exactement comme l'ordre qui est à envoyer), que nous appellerons "donnee". Le prototype de la fonction sera alors le suivant :

**Code : C**

```
void envoi_ordre(int dataPin, int clockPin, boolean sens, char
donnee)
```

Le code de la fonction ne sera pas très compliqué. Comme expliqué plus tôt, il suffit de générer une horloge et d'envoyer la bonne donnée pour que tout se passe bien.
Le 74HC595 copie le bit envoyé dans sa mémoire lorsque le signal d'horloge passe de 0 à 1. Pour cela, il faut donc débuter le cycle par une horloge à 0. Ensuite, nous allons placer la donnée sur la broche de donnée. Enfin, nous ferons basculer la broche d'horloge à l'état haut pour terminer le cycle. Nous ferons ça huit fois pour pouvoir envoyer les huit bits de l'octet concerné (l'octet d'ordre). Schématiquement le code serait donc le suivant :

**Code : C**

```
for(int i=0; i<8; i++) //on va parcourir chaque bit de l'octet
  {
     //départ du cycle, on met l'horloge à l'état bas
     digitalWrite(clockPin, LOW);
     //on met le bit de donnée courant en place
     digitalWrite(dataPin, le_bit_a_envoyer);
     //enfin on remet l'horloge à l'état haut pour faire prendre en
compte ce dernier et finir le cycle
     digitalWrite(clockPin, HIGH);
  }  //et on boucle 8 fois pour faire de même sur chaque bit de
l'octet d'ordre
```

### *Envoyer un `char` en tant que donnée binaire*

Maintenant que l'on a défini une partie de la fonction `envoi_ordre()`, il va nous rester un léger problème à régler : envoyer une donnée de type `char` en tant que suite de bit (ou donnée binaire).

Prenons un exemple : le nombre 231 s'écrit aussi sous la forme 11100111 en base 2 (et oui, c'est le moment de se rappeler ce que l'on a vu ici ). Seulement, en voulant envoyer ce nombre sur la broche de donnée pour commander le 74HC595, cela ne marchera pas d'écrire :

**Code : C**

```
digitalWrite(dataPin, 231);
```

En faisant de cette façon, la carte Arduino va simplement comprendre qu'il faut mettre un état HAUT (car 231 est différent de 0) sur sa broche de sortie que l'on a nommée `dataPin`. Pour pouvoir donc envoyer ce nombre sous forme binaire, il va falloir ajouter à la fonction que l'on a créé un morceau de code supplémentaire.

Ce que nous allons va faire va être une vraie boucherie : on va découper ce nombre en huit tranches et envoyer chaque morceau un par un sur la sortie `dataPin`.

Pour découper ce nombre, ça va pas être de la tarte... euh... je m'égare. On va utiliser une technique qui se nomme, tenez-vous bien, le **masquage**. On va en fait utiliser un masque qui va cacher la véritable valeur du nombre 231. Bon bon, je vous explique.

Tout d'abord, on va considérer que le nombre 231 est vu sous sa forme binaire, qui je le rappel est 11100111, par votre carte Arduino. Donc, lorsque l'on va passer en paramètre `donnee` le nombre 231, le programme verra la suite de 1 et de 0 : 11100111. Jusque-là, rien de bien sorcier.

Voilà donc notre suite de 1 et de 0 que l'on va devoir découper. Alors, il n'existe pas de fonction toute prête spécialement conçue pour découper un nombre binaire. Non, ça va être à nous de faire cela. Et c'est pourquoi je vous parlais du masquage. Cette technique ne porte pas son nom par hasard, en effet, nous allons réellement utiliser un **masque**. Quelques précisions s'imposent, je le sens bien.

Reprenons notre suite binaire :


Notre objectif étant d'envoyer chaque bit un par un, on va faire croire à l'Arduino que cette suite n'est composée que d'un seul bit. En clair, on va cacher les 7 autres bits en utilisant un masque :

Ce qui, au final, donnera :


L'Arduino ne verra donc qu'un seul bit.

Et les autres, il les voit pas, comment on peut envoyer les 8 bits alors ?

Bien sûr, les autres, l'Arduino ne les voit pas. C'est pourquoi l'on va faire évoluer le masque et révéler chaque bit un par un. En faisant cela huit fois, on aura envoyé les 8 bits à la suite :


On peut aussi faire évoluer le masque dans le sens opposé :

L'étape qui suit est donc d'identifier le bit à envoyer en premier. C'est là que rentre en jeu le paramètre sens. On a le choix d'envoyer soit le bit de poids fort (on l'appelle **MSB**, Most Significant Bit) en premier et finir par le bit de poids faible (Less Significant Bit, **LSB**) ; soit dans le sens opposé, du LSB vers le MSB. On parle alors d'envoi MSB First (pour "bit de poids fort en premier") ou LSB First.

À présent, voyons comment appliquer la technique de masquage que je viens de vous présenter

### *Les masques en programmation*

Maintenant que vous connaissez cela, nous allons pouvoir voir comment isoler chacun des bits pour les envoyer un par un.

En programmation, il est évident que l'on ne peut pas mettre un masque papier sur les bits pour les cacher. Il existe donc un moyen de les cacher. Cela va faire appel à la **logique binaire**. Nous n'entrerons pas dans le détail, mais sachez que nous allons employer des **opérateurs logiques**. Il en existe plusieurs, dont deux très utilisés, même dans la vie courante, l'opérateur **ET** et **OU**.

Commençons par l'opérateur logique ET (je vous laisse regarder le OU tout seul, nous n'en aurons pas besoin ici). Il s'utilise avec le symbole **&** que vous trouverez sous la touche 1 au-dessus de la lettre "a" sur un clavier azerty.

Pour envoyer le premier bit de notre donnée, nous allons effectuer le masquage avec cet opérateur logique dont la table de vérité se trouve être la suivante :

| Table de vérité du ET | | |
|---|---|---|
| **Bit 1** | **Bit 2** | **Résultat** |
| **0** | **0** | **0** |
| **0** | **1** | **0** |
| **1** | **0** | **0** |
| **1** | **1** | **1** |

Je ne comprends pas trop où tu veux en venir ?

Je vais vous expliquer.

Pour faire le masquage, on va faire une opération avec ce fameux ET logique. Il s'agit de la même chose que si l'on additionnait deux nombres ensemble, ou si on les multipliait. Dans notre cas l'opération est "un peu bizarre". Disons que c'est une opération évoluée.

Cette opération va utiliser deux nombres : le premier on le connaît bien, il s'agit de la suite logique 11100111, quant au second, il s'agira du masque. Pour l'instant, vous ne connaissez pas la valeur du masque, qui sera lui aussi sous forme binaire. Pour définir cette valeur, on va utiliser la table de vérité précédente.

Afin que vous ne vous perdiez pas dans mes explications, on va prendre pour objectif d'envoyer le bit de poids faible de notre nombre 11100111 (celui tout à droite).

Le code qui suit est un pseudo-code, mis sous forme d'une opération mathématique telle que l'on en ferait à l'école :

**Code : C**

```
  . 11100111 (donnée à transmettre)
  & 00000001 (on veut envoyer uniquement le bit de poids faible)
    --------
    00000001 (donnée à transmettre au final) -> soit 1
```

Pour comprendre ce qui vient de se passer, il faut se référer à la table de vérité de l'opérateur ET : on sait que lorsque l'on fait 1 et 0 le résultat est 0. Donc, pour cacher tous les bits du nombre à masquer, il n'y a qu'à mettre que des 0 dans le masque. Là,

l'Arduino ne verra que le bit 0 puisque le masque aura caché au complet le nombre du départ.
On sait aussi que 1 ET 1 donne 1. Donc, lorsque l'on voudra montrer un bit à l'Arduino, on va mettre un 1 dans le masque, à l'emplacement du bit qui doit être montré.

Pour monter ensuite le bit supérieur au bit de poids faible, on procède de la même manière :

**Code : C**

```
. 11100111 (donnée à transmettre)
& 00000010 (on veut envoyer uniquement le deuxième bit)
  --------
  00000010 (donnée à transmettre au final) -> soit 1
```

Pour le quatrième bit en partant de la droite :

**Code : C**

```
. 11100111 (donnée à transmettre)
& 00001000 (on veut envoyer uniquement le quatrième bit)
  --------
  00000000 (donnée à transmettre au final) -> soit 0
```

Dans le cas où vous voudriez montrer deux bits à l'Arduino (ce qui n'a aucun intérêt dans notre cas, je fais ça juste pour vous montrer) :

**Code : C**

```
. 11100111 (donnée à transmettre)
& 01000100 (on veut envoyer uniquement le quatrième bit)
  --------
  01000100 (donnée à transmettre au final) -> soit 68 en base
décimale
```

### *L'évolution du masque*

Ce titre pourrait être apparenté à celui d'un film d'horreur, mais n'indique finalement que nous allons faire évoluer le masque automatiquement à chaque fois que l'on aura envoyé un bit.

Cette fois, cela va être un peu plus simple car nous n'avons qu'à rajouter un opérateur spécialisé dans le décalage. Si l'on veut déplacer le 1 du masque (qui permet de montrer un bit à l'Arduino) de la droite vers la gauche (pour le LSBFirst) ou dans l'autre sens (pour le MSBFirst), nous avons la possibilité d'utiliser l'opérateur << pour décaler vers la gauche ou >> pour décaler vers la droite. Par exemple :

**Code : C**

```
.  00000001 (masque initial)
<<        3 (on décale de trois bits)
   --------
   00001000 (masque final, décalé)
```

Et dans le sens opposé :

**Code : C**

```
.  10000000 (masque initial)
>>        3 (on décale de trois bits)
   --------
   00010000 (masque final, décalé)
```

Avouez que ce n'est pas très compliqué maintenant que vous maîtrisez un peu les masques.

On va donc pouvoir isoler un par un chacun des bits pour les envoyer au 74HC595. Comme le sens dépend d'un paramètre de la fonction, nous rajoutons un test pour décaler soit vers la droite, soit vers la gauche.
Voici la fonction que nous obtenons à la fin :

**Code : C**

```
void envoi_ordre(int dataPin, int clockPin, boolean sens, char
donnee)
{
  for(int i=0; i<8; i++) //on va parcourir chaque bit de l'octet
  {
     //on met l'horloge à l'état bas
     digitalWrite(clockPin, LOW);
     //on met le bit de donnée courante en place
     if(sens)
        //envoie la donnée en allant de droite à gauche, en partant
d'un masque de type "00000001"
        digitalWrite(dataPin, donnee & 0x01<<i);
     else
        //envoie la donnée en allant de gauche à droite, en partant
d'un masque de type "10000000"
        digitalWrite(dataPin, donnee & 0x80>>i);
     //enfin on remet l'horloge à l'état haut pour faire prendre en
compte cette dernière
     digitalWrite(clockPin, HIGH);
  }
}
```

Oula ! Hé ! Stop ! C'est quoi ce 0x01 et ce 0x80 ? Qu'est-ce que ça vient faire là, c'est pas censé être le masque que l'on doit voir ?

Si, c'est bien cela. Il s'agit du masque... écrit sous sa forme hexadécimale. Il aurait été bien entendu possible d'écrire : `0b00000001` à la place de `0x01`, ou `0b10000000` à la place de `0x80`. On a simplement opté pour la base hexadécimale qui est plus facile à manipuler.

Cette technique de masquage peut sembler difficile au premier abord mais elle ne l'est pas réellement une fois que l'on a compris le principe. Il est essentiel de comprendre comment elle fonctionne pour aller loin dans la programmation de micro-contrôleur (pour paramétrer les registres par exemple), et vous en aurez besoin pour les exercices du chapitre suivant. Pour plus d'informations un bon tuto plus complet mais rapide à lire est rédigé ici... en PHP, mais c'est pareil.

### *Un petit programme d'essai*

Je vous propose maintenant d'essayer notre belle fonction. Pour cela, quelques détails sont à préciser/rajouter.

Pour commencer, il nous faut déclarer les broches utilisées. Il y en a trois : verrou, horloge et data. Pour ma part elles sont branchées respectivement sur les broches 11, 12 et 10. Il faudra donc aussi les déclarer en sortie dans le setup(). Si vous faites de même vous devriez obtenir le code suivant :

**Code : C**

```
//Broche connectée au ST_CP du 74HC595
const int verrou = 11;
//Broche connectée au SH_CP du 74HC595
const int horloge = 12;
//Broche connectée au DS du 74HC595
const int data = 10;

void setup() {
  //On met les broches en sortie
  pinMode(verrou, OUTPUT);
  pinMode(horloge, OUTPUT);
  pinMode(data, OUTPUT);
}
```

Ensuite, nous allons nous amuser à afficher un nombre allant de 0 à 255 en binaire. Ce nombre peut tenir sur un octet, ça tombe bien car nous allons justement transmettre un octet ! Pour cela, nous allons utiliser une boucle for() allant de 0 à 255 et qui appellera notre fonction.
Avant cela, je tiens à rappeler qu'il faut aussi mettre en place le verrou en encadrant l'appel de notre fonction. Rappelez-vous, si nous ne le faisons pas, l'affichage risque de scintiller.

**Code : C**

```
//On active le verrou le temps de transférer les données
digitalWrite(verrou, LOW);
//on envoi toutes les données grâce à notre belle fonction (octet
inversée avec '~' pour piloter les LED à l'état bas)
envoi_ordre(data, horloge, 1, ~j);
//et enfin on relâche le verrou
digitalWrite(verrou, HIGH);
```

Et voici le code complet que vous aurez surement deviné :

**Code : C**

```
//Broche connectée au ST_CP du 74HC595
const int verrou = 11;
//Broche connectée au SH_CP du 74HC595
const int horloge = 12;
//Broche connectée au DS du 74HC595
const int data = 10;

void setup() {
  //On met les broches en sortie
  pinMode(verrou, OUTPUT);
  pinMode(horloge, OUTPUT);
  pinMode(data, OUTPUT);
}

void loop() {
  //on affiche les nombres de 0 à 255 en binaire
  for (char i = 0; i < 256; i++) {
    //On active le verrou le temps de transférer les données
    digitalWrite(verrou, LOW);
    //on envoi toutes les données grâce à notre belle fonction
    envoi_ordre(data, horloge, 1, ~i);
    //et enfin on relâche le verrou
    digitalWrite(verrou, HIGH);
    //une petite pause pour constater l'affichage
    delay(1000);
  }
}
```

```
void envoi_ordre(int dataPin, int clockPin, boolean sens, char
donnee)
{
  for(int i=0; i<8; i++) //on va parcourir chaque bit de l'octet
  {
     //on met l'horloge à l'état bas
     digitalWrite(clockPin, LOW);
     //on met le bit de donnée courante en place
     if(sens)
        digitalWrite(dataPin, donnee & 0x01<<i);
     else
        digitalWrite(dataPin, donnee & 0x80>>i);
     //enfin on remet l'horloge à l'état haut pour faire prendre en
compte cette dernière
     digitalWrite(clockPin, HIGH);
  }
}
```

Et voila le travail ! :

## La fonction magique, ShiftOut

Vous êtes content ? vous avez une belle fonction qui marche bien et fait le boulot proprement ? Alors laissez-moi vous présenter une nouvelle fonction qui s'appelle `shiftOut()`. Quel est son rôle ? Faire exactement la même chose que la fonction dont l'on vient juste de finir la création.


*#@"e !!

Alors oui je sais, c'est pas sympa de ma part de vous avoir fait travailler mais admettez que c'était un très bon exercice de développement non ? À présent vous comprenez comment agit cette fonction et vous serez mieux capable de créer votre propre système que si je vous avais donné la fonction au début en disant : "voilà, c'est celle-là, on l'utilise comme ça, ça marche, c'est beau... mais vous avez rien compris".

Comme je vous le disais précédemment, cette fonction sert à faire ce que l'on vient de créer, mais elle est déjà intégrée à l'environnement Arduino (donc a été testée par de nombreux développeurs, ne laissant pas beaucoup de place pour les bugs !).

Cette fonction prend quatre paramètres :

- La broche de donnée
- La broche d'horloge
- Le sens d'envoi des données (utiliser avec deux valeurs symboliques, MSBFIRST ou LSBFIRST)
- L'octet à transmettre

Son utilisation doit maintenant vous paraître assez triviale. Comme nous l'avons vu plutôt, il suffit de bloquer le verrou, envoyer la donnée avec la fonction puis relâcher le verrou pour constater la mise à jour des données.
Voici un exemple de loop avec cette fonction :

**Code : C**

```
void loop()
{
  //on affiche les nombres de 0 à 255 en binaire
  for (int i = 0; i < 256; i++)
  {
    //On active le verrou le temps de transférer les données
    digitalWrite(verrou, LOW);
    //on envoi toutes les données grâce à shiftOut (octet inversée
avec '~' pour piloter les LED à l'état bas)
    shiftOut(data, horloge, LSBFIRST, ~i);
    //et enfin on relache le verrou
    digitalWrite(verrou, HIGH);
    //une petite pause pour constater l'affichage
    delay(1000);
  }
}
```

## Exercices : encore des chenillards !

Je vous propose maintenant trois exercices pour jouer un peu avec ce nouveau composant et tester votre habileté au code. Le but du jeu est d'arriver à reproduire l'effet proposé sur chaque vidéo. Le but second est de le faire intelligemment... Autrement dit, tous les petits malins qui se proposeraient de faire un "tableau de motif" contenant les valeurs "affichages binaires" successives devront faire autrement.

Amusez vous bien !

PS : Les corrections seront juste composées du code de la loop avec des commentaires. Le schéma reste le même ainsi que les noms de broches utilisés précédemment.

PPS : La bande son des vidéos est juste là pour cacher le bruit de la télé... je n'y ai pas pensé quand je faisais les vidéos et Youtube ne permet pas de virer la bande audio...

## "J'avance et repars !"

### *Consigne*

Pour ce premier exercice, histoire de se mettre en jambe, nous allons faire une animation simple. Pour cela, il suffit de faire un chenillard très simple, consistant en une LED qui "avance" du début à la fin de la ligne. Arrivée à la fin elle repart au début. Si ce n'est pas clair, regardez la vidéo ci-dessous ! (Éventuellement vous pouvez ajouter un bouton pour inverser le sens de l'animation).

#### *Correction*

**Secret** (cliquez pour afficher)

**Code : C**

```
void loop() {
  for (int i = 0; i < 8; i++) {
    //On active le verrou le temps de transférer les données
    digitalWrite(verrou, LOW);
    //on envoie la donnée
    //ici, c'est assez simple. On va décaler l'octet 00000001 i
fois puis l'envoyer
    shiftOut(data, horloge, LSBFIRST, ~(0x01 << i));
    //et enfin on relache le verrou
    digitalWrite(verrou, HIGH);
    //une petite pause pour constater l'affichage
    delay(250);
  }
}
```

## "J'avance et reviens !"

#### *Consigne*

Cette seconde animation ne sera pas trop compliquée non plus. La seule différence avec la première est que lorsque la "lumière" atteint la fin de la ligne, elle repart en arrière et ainsi de suite. Là encore si ce n'est pas clair, voici une vidéo :

### *Correction*

**Secret** (cliquez pour afficher)

Dans cet exercice, le secret est d'utiliser de manière intelligente le paramètre LSBFIRST ou MSBFIRST pour pouvoir facilement inverser le sens de l'animation sans écrire deux fois la boucle for.

**Code : C**

```
char sens = MSBFIRST; //on commence à aller de droite vers gauche

void loop() {
  for (int i = 0; i < 7; i++) { //on ne fait la boucle que 7 fois
pour ne pas se répéter au début et à la fin
    //On active le verrou le temps de transférer les données
    digitalWrite(verrou, LOW);
    //on envoie la donnée
    //On va décaler l'octet 00000001 i fois puis l'envoyer
    shiftOut(data, horloge, sens, ~(0x01 << i));
    //et enfin on relache le verrou
    digitalWrite(verrou, HIGH);
    //une petite pause pour constater l'affichage
    delay(250);
  }
  sens = !sens; //on inverse le sens d'affichage pour la
prochaine fois (MSBFIRST <-> LSBFIRST)
}
```

## Un dernier pour la route !

### *Consigne*

Pour cette dernière animation, il vous faudra un peu d'imagination. Imaginez le chenillard numéro 1 allant dans les deux sens en même temps... C'est bon ? si non alors voici la vidéo :

### *Correction*

**Secret** (cliquez pour afficher)

**Code : C**

```
void loop() {
  char donnee = 0;

  for (int i = 0; i < 8; i++) {
    //on saute la boucle si i vaut 4 (pour une histoire de
fluidité de l'animation, tester sans et vous verrez)
    if(i == 4)
       continue;

    //calcule la donnée à envoyer
    donnee = 0;
    donnee = donnee | (0x01 << i); // on calcule l'image du
balayage dans un sens
    donnee = donnee | (0x80 >> i); // et on ajoute aussi l'image
du balayage dans l'autre sens

    //On active le verrou le temps de transférer les données
    digitalWrite(verrou, LOW);
    //on envoie la donnée
    shiftOut(data, horloge, LSBFIRST, ~donnee);
    //et enfin on relache le verrou
    digitalWrite(verrou, HIGH);
    //une petite pause pour constater l'affichage
    delay(250);
  }
}
```

## Exo bonus

### *Consigne*

Ici le but du jeu sera de donner un effet de "chargement / déchargement" en alternance...
Comme d'habitude, voici la vidéo pour mieux comprendre...

### *Correction*

**Secret** (cliquez pour afficher)

Dans cet exercice, tout repose sur l'utilisation du MSBFIRST ou LSBFIRST ainsi que du complément appliqué sur la donnée. Ce dernier permet d'activer ou non les LEDs et le premier atout permet d'inverser l'effet.

**Code : C**

```
char extinction = 0; //on commence à aller de droite vers gauche

void loop() {
  char donnee = extinction; //on démarre à 0 ou 1 selon...
  for (int i = 0; i < 8; i++) {
    //On active le verrou le temps de transférer les données
    digitalWrite(verrou, LOW);
    //si on est en train d'éteindre
    if(extinction)
      shiftOut(data, horloge, MSBFIRST, ~donnee);//on envoie la
donnée inversé
    //sinon
    else
      shiftOut(data, horloge, LSBFIRST,  donnee);//on envoie la
donnée normale
    //et enfin on relache le verrou
    digitalWrite(verrou, HIGH);
    //une petite pause pour constater l'affichage
    delay(250);
    donnee = donnee | (0x01 << i); //et on met à jour la donnée
en cumulant les décalages
  }
  extinction = !extinction; //permet d'inverser "MSBFIRST <->
LSBFIRST" comme dans l'exercice 2
}
```

## Pas assez ? Augmenter encore !

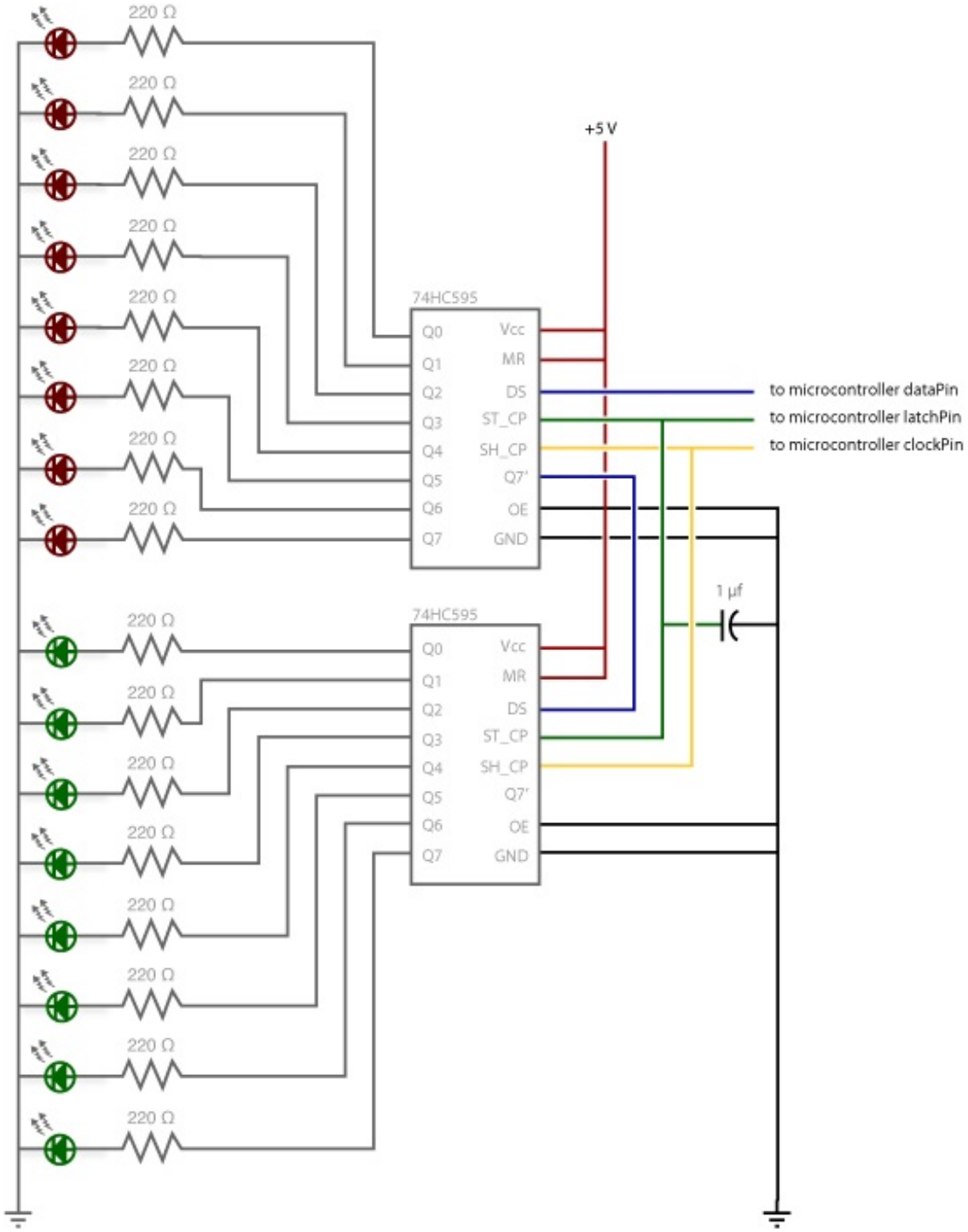
Si jamais 8 nouvelles sorties ne vous suffisent pas (bien que cela n'en face que 5 au total puisque trois sont prises pour communiquer avec le composant), les ingénieurs ont déjà tout prévu. Ainsi il est possible de mettre en cascade plusieurs 74HC595 !

Pour cela, le 595 dispose d'une broche appelée "débordement". Lorsque vous envoyez un seul octet au 74HC595, rien ne se passe sur cette broche. Cependant, si vous envoyez plus d'un octet, les huit derniers bits seront conservés par le composant, tandis que les autres vont être "éjectés" vers cette fameuse sortie de débordement (numéro 9). Le premier bit envoyé ira alors vers le 74HC595 le plus loin dans la chaine. Souvenez-vous, elle s'appelle "serial data output" et j'avais dit qu'on reviendrait dessus. D'une manière très simple, les bits éjectés vont servir aux éventuels 74HC595 qui seront mis en aval de celui-ci.

### *Branchement*

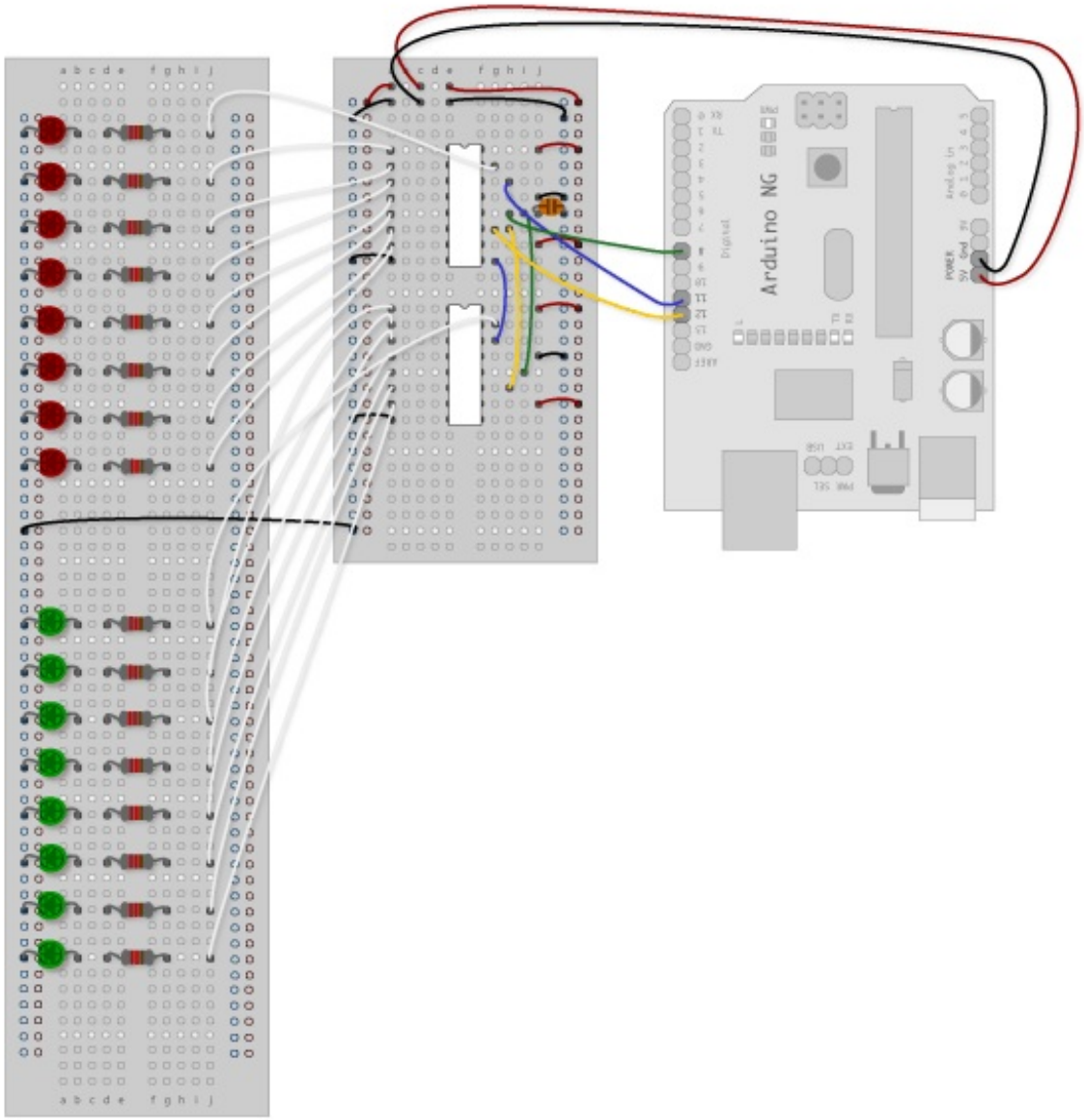
Il suffit dons de mettre deux 595 bout-à-bout en reliant la broche de débordement du premier sur la broche de donnée du second. Ainsi, les bits "en trop" du premier arriveront sur le second. Afin que le second fonctionne, il faut aussi également relier les mêmes broches pour l'horloge et le verrou (reliées en parallèle entre les deux).

Les images proviennent d'une explication du site Arduino. Attention, dans ce schéma les LEDs sont branchées "à l'envers" de ce que nous avons l'habitude de faire.




Deux 595 en cascade,

schéma




Deux 595 en cascade, breadboard

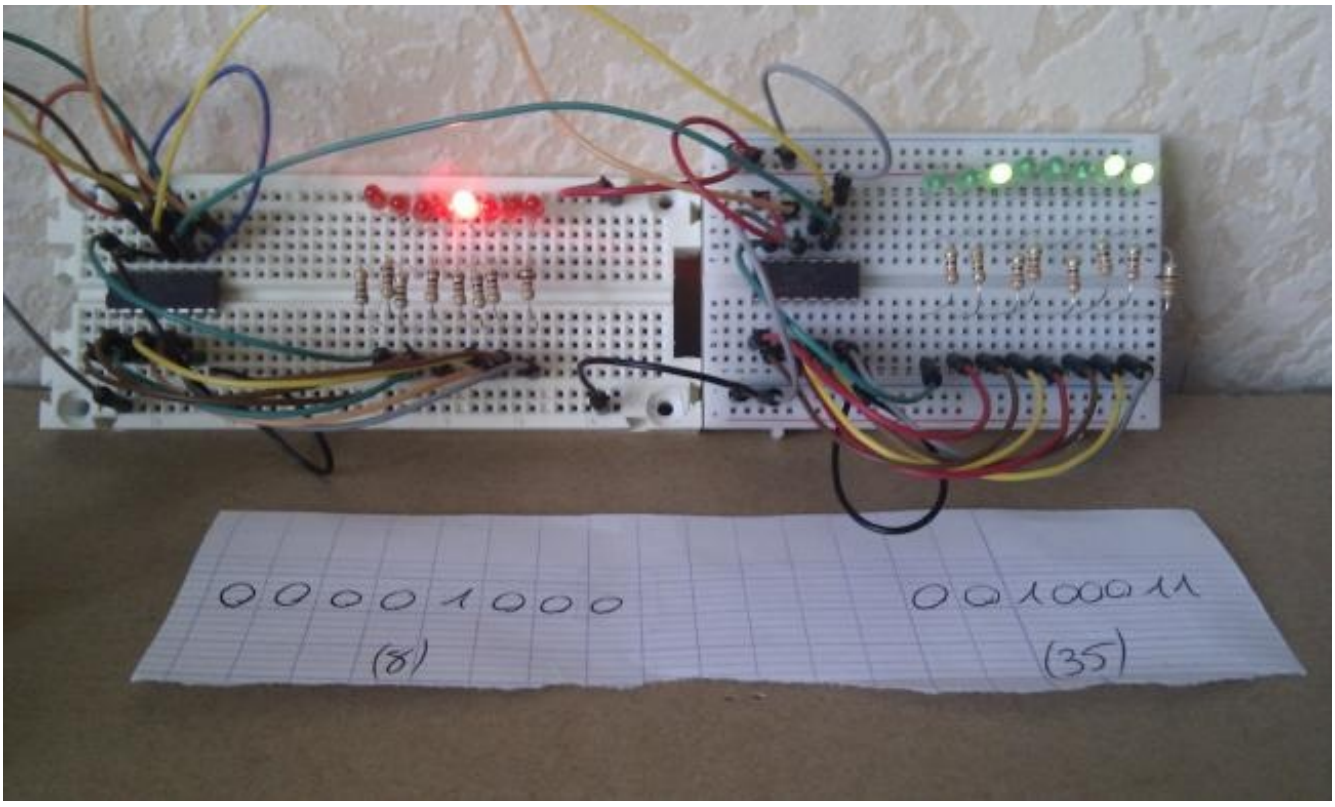
### *Exemple d'un affichage simple*

Au niveau du programme, il suffira de faire appel deux fois de suite à la fonction shiftOut pour tout envoyer (2 fois 8 bits). Ces deux appels seront encadrés par le verrou pour actualiser l'affichage des données. On commence par envoyer la donnée qui doit avancer le plus pour atteindre le second 595, puis ensuite on fait celle qui concerne le premier 595.
Voici un exemple :

**Code : C**

```
const int verrou = 11;
const int donnee = 10;
const int horloge = 12;

char premier = 8; //en binaire : 00001000
char second = 35; //en binaire : 00100011

void setup()
{
   //on déclare les broches en sortie
   pinMode(verrou, OUTPUT);
   pinMode(donnee, OUTPUT);
   pinMode(horloge, OUTPUT);
```

```
    //puis on envoie les données juste une fois

    //on commence par mettre le verrou
    digitalWrite(verrou, LOW);

    //on envoie la seconde donnée d'abord
    shiftOut(donnee, horloge, LSBFIRST, ~second); //les LEDs vertes
du montage
    //on envoie la première donnée
    shiftOut(donnee, horloge, LSBFIRST, ~premier); //Les LEDs rouges
du montage

    //et on relache le verrou pour mettre à jour les données
    digitalWrite(verrou, HIGH);
}

void loop()
{
    //rien à faire
}
```




### *Exemple d'un chenillard*

Voici maintenant un petit exemple pour faire un chenillard sur 16 LEDs. Pour cela, j'utiliserai un int qui sera transformé en char au moment de l'envoi. Il faudra donc le décaler vers la droite de 8 bits pour pouvoir afficher ses 8 bits de poids fort. Voici une loop pour illustrer mes propos (le setup étant toujours le même).

**Code : C**

```
void loop()
{
    int masque = 0;

    for(int i=0; i<16; i++)
    {
      masque = 0x01 << i; //on décale d'un cran le masque
```

```
            //on commence par mettre le verrou
            digitalWrite(verrou, LOW);

            //on envoie la seconde donnée d'abord
            shiftOut(donnee, horloge, LSBFIRST, ~(masque & 0x00FF)); //On
      envoie les 8 premiers bits
            //on envoie la première donnée
            shiftOut(donnee, horloge, LSBFIRST, ~((masque & 0xFF00) >> 8));
      //On envoie les 8 derniers bits

            //et on relache le verrou pour mettre à jour les données
            digitalWrite(verrou, HIGH);
            delay(500);
         }
      }
```

Ce composant peut vous paraître un peu superflu mais il existe en fait de très nombreuses applications avec. Par exemple, si vous voulez réaliser un cube de LED (disons 4x4x4 pour commencer gentiment). Si vous vouliez donner une broche par LED vous seriez bloquer puisque Arduino n'en possède pas autant (il vous en faudrait 32). Ici le composant vous permet donc de gérer plus de sorties que vous ne le pourriez initialement.

On achève enfin cette deuxième partie où vous avez pu acquérir un ensemble de connaissances nécessaires pour poursuivre la lecture de ce tutoriel. La prochaine partie va traiter sur la communication entre une Arduino et un ordinateur ou même entre deux Arduino. Cela risque d'être prometteur ! 

# Partie 3 : [Pratique] Communication par la liaison série

Maintenant que nous avons de bonne bases, nous allons pouvoir passer à quelque chose d'un tout petit peu plus difficile (mais pas de quoi avoir peur pour autant).
Cette partie va vous apprendre à utiliser un moyen de communication, afin de faire "parler" votre carte Arduino avec un autre matériel ou un ordinateur.

---> **Matériel nécessaire :** dans la balise secret pour la partie 3.

## Généralités

Saviez-vous que l'USB ne sert pas qu'à alimenter la carte Arduino ? Dans ce chapitre, nous allons apprendre à utiliser la liaison série, au travers de l'USB. Grâce à elle, vous pourrez faire communiquer entre eux, votre ordinateur et la carte Arduino.

Mais juste avant de commencer à utiliser la liaison série avec Arduino, je vous propose ce petit chapitre sur les généralités de cette liaison. Elles vous seront utiles lorsque vous aurez besoin de faire communiquer des appareils entre eux pour faire des commandes domotiques par exemple, ou bien tester des appareils fonctionnant avec cette liaison, etc.

La lecture de ce chapitre n'est donc pas obligatoire, mais vivement conseillée. Après, vous n'êtes pas obligé de retenir tout ce qui va être dit sur les normes, les tensions, etc. de la liaison série.

Voyons maintenant tout cela !

### Protocole de communication

### Principe de la voie série

Pour faire des communications entre différents supports, il existe différents moyens. Pour n'en citer que quelques-uns, on retrouve les bus CAN, le bus I²C, l'Ethernet, etc. et la liste est longue. Dans notre cas, nous allons étudier la **communication série**, aussi appelée **RS232**, puisqu'elle est intégrée par défaut dans la carte Arduino.

#### *À quoi ça va nous servir ?*

La voie série permet de communiquer de manière directe et unique entre deux supports. Ici, elle se fera entre un ordinateur et la platine Arduino, mais elle pourrait aussi se faire par exemple entre deux cartes Arduino. Dans sa forme la plus simple, elle ne nécessite que 3 fils : 2 pour l'émission/réception et 1 pour la masse afin d'avoir un référentiel électrique commun.

Dans des formes plus évoluées, on retrouve des fils de contrôle de flux. Ces liaisons permettent de s'assurer que la communication se passe correctement en utilisant des systèmes de synchronisation. Mais on ne verra pas ce dernier point car la carte Arduino ne le supporte tout simplement pas. On va uniquement utiliser l'émission/réception de données.

Ainsi, voilà où je voulais en venir, on va faire communiquer notre carte Arduino avec notre ordinateur ! Vous verrez, c'est génial !!  En effet, une fois que vous aurez bien saisi comment fonctionne la liaison série, il vous sera facile de l'utiliser et difficile de vous en passer (idéal pour faire du debug par exemple). Et pour les plus téméraires, vous pourrez créer un logiciel complet qui communique des ordres à votre carte Arduino pour effectuer des actions plus ou moins complexes (par exemple, créer un système de maison intelligente).

### Avant de commencer...

#### *Qu'est-ce qu'un protocole de communication ?*

En informatique, lorsque l'on parle de **protocole de communication**, il s'agit de règles prédéfinies pour un type de communication. Ici ce sera le type liaison série. Pour simplifier, je vous parle en français. Seuls ceux qui comprennent le français pourront lire ce que j'écris. Sauf dans le cas où la personne qui lit ce qui est écrit, connait le français ou dispose d'un traducteur. Eh bien, lorsque la carte Arduino communiquera avec l'ordinateur, il faudra que ces deux dispositifs puissent se comprendre, donc "parler le même langage". C'est notre fameuse liaison série.

#### *Les types de liaison série*

Le premier type est la liaison **simplex**. Il n'y a qu'un émetteur et un seul récepteur. Par exemple, seul l'ordinateur peut envoyer des données à la carte Arduino. Ça nous n'est pas très utile si on veut faire le contraire. On n'utilisera donc pas ce type de liaison.

Le deuxième est la **liaison half-duplex**. En fait, c'est un peu lorsque l'on communique à quelqu'un avec un talkie-walkie. L'un parle pendant que l'autre écoute. Nous n'utiliserons pas ce type de communication.

Le dernier est la liaison **full-duplex**. Là, c'est un peu comme le téléphone, chacun peut parler et écouter en même temps ce que l'autre dit. Avec Arduino, c'est de ce type de communication que nous disposons. Ce qui est bien pratique afin d'éviter d'attendre que l'on ait réceptionné ce que l'ordinateur envoie pour ensuite lui émettre des données.

### *Le support de liaison*

Tout comme votre téléphone ou votre télécommande, pour communiquer, les appareils ont besoin d'un support de transmission. Par exemple, un fil électrique, une liaison infrarouge ou hertzienne. Je ne m'étends pas, ce n'est pas l'objet de ce chapitre. On utilisera, pour cette partie, uniquement la liaison filaire.

On en termine là, vous trouverez d'autres informations plus complètes sur internet, le but étant de vous faire utilise la liaison série. Donc il n'y a pas besoin de grosses connaissances.

## Fonctionnement de la communication série

On va enfin voir comment fonctionne cette liaison et ce qu'elle fait.

### *Les données*

D'abord, on va voir sous quelle forme sont envoyées les données. Oui, car le but de la liaison série est bien de permettre l'échange de données entre deux dispositifs.

Nous allons prendre l'exemple de la lettre 'P' majuscule. Voilà, ce sera la donnée que nous transmettrons. Saviez-vous que chaque lettre du clavier peut se coder avec des chiffres ou des chiffres et des lettres ? Ces codes sont définis selon la table ASCII.

En haut à gauche de la table ASCII, on observe la ligne : "Code en base..." et là vous avez : 10, 8, 16, 2. Respectivement, ce sont les bases **décimale** (10), **octale** (8), **hexadécimale** (16) et **binaire** (2).

Nous, ce qui va nous intéresser, c'est la base **binaire**. Oui car le binaire est une succession de 0 et de 1, qui sont en fait des états logiques, tel que LOW (0) et HIGH (1). En sortie du micro-contrôleur de la carte Arduino, ces états se traduisent par une tension de 0V pour l'état logique LOW et une tension de 5V pour un état logique HIGH. Ces états sont ce qu'on appelle des **bits**. Un bit est donc la traduction d'un état logique (bit à 0 pour un état logique LOW ; bit à 1 pour un état logique HIGH).

Reprenons notre lettre 'P'. Elle se traduit, en binaire, par la succession de 1 et 0, comme ceci : 01010000. Il y a donc 8 bits accolés les uns aux autres. On appelle cela un **octet**. En informatique, un octet, c'est comme un **mot** pour nous. D'ailleurs, quand on parle de mots transmis sur une liaison, on parle d'octets.

Pour votre culture, sachez que la table ASCII est à l'origine codée sur 7 bits. Pour plus d'information sur le binaire, consultez cette page.

### *Le protocole*

Bon, après cette brève introduction, on va pouvoir regarder comment est transmise la lettre 'P', qui sera notre mot, ou plutôt notre octet.

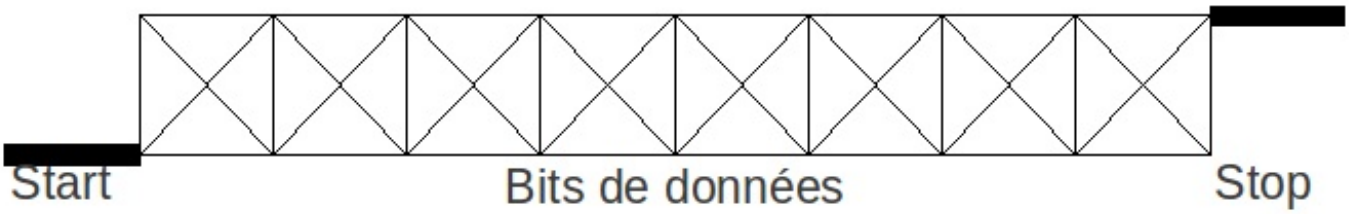
On va prendre un exemple assez simple :

- Lorsque vous passez un coup de fil, vous commencez souvent par dire "Bonjour" ou "Allo". Ce message s'appellera, dans notre cas, le **bit de départ** ou **bit de start**. Il possède un niveau logique 0 (NL0).
- Ensuite, vous allez dire des mots, donc l'information que vous avez à transmettre.
- Enfin, à la fin de la communication vous dites "Au revoir" ou "Salut !" "A plus !" etc. Cette information sera le **bit de fin** ou **bit de stop**, et aura un niveau logique 1 (NL1).

C'est sous cette "norme" que la communication série fonctionne comme ça. D'ailleurs, savez-vous pourquoi la liaison série s'appelle ainsi ?

Parce que l'ordinateur est branché en série ?

Non, ce n'est pas pour ça. 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. Voilà un petit schéma pour résumer ce que l'on vient d'affirmer :




Enchaînement des données

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 : 001010001.

Eh bien... c'est presque ça. Sauf que les petits malins 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 :


On ne le voit pas sur ce chronogramme, mais l'échelle des abscisses est en unité de temps (ici ce sont des bits, la durée d'émission d'un bit dépendant de la vitesse de transmission) et l'échelle des ordonnées est en Volt (enfin, ici, on représente l'état des bits : 1 ou 0)

Sur une liaison série, les données sont toujours envoyées sous forme d'octet. Mais on peut très bien envoyer seulement 7 bits. Par exemple, pour envoyer le caractère '?', on enverra : 00111111 en octet, ou bien sur 7 bits : 0111111. Avec Arduino ce paramètres est réglé à 8 bits de données (un octet). Donc le jour ou vous écrirez une application de réception des données ou utiliserez un logiciel de voie série, vérifiez qu'il est bien à 8 (toujours par défaut cependant).

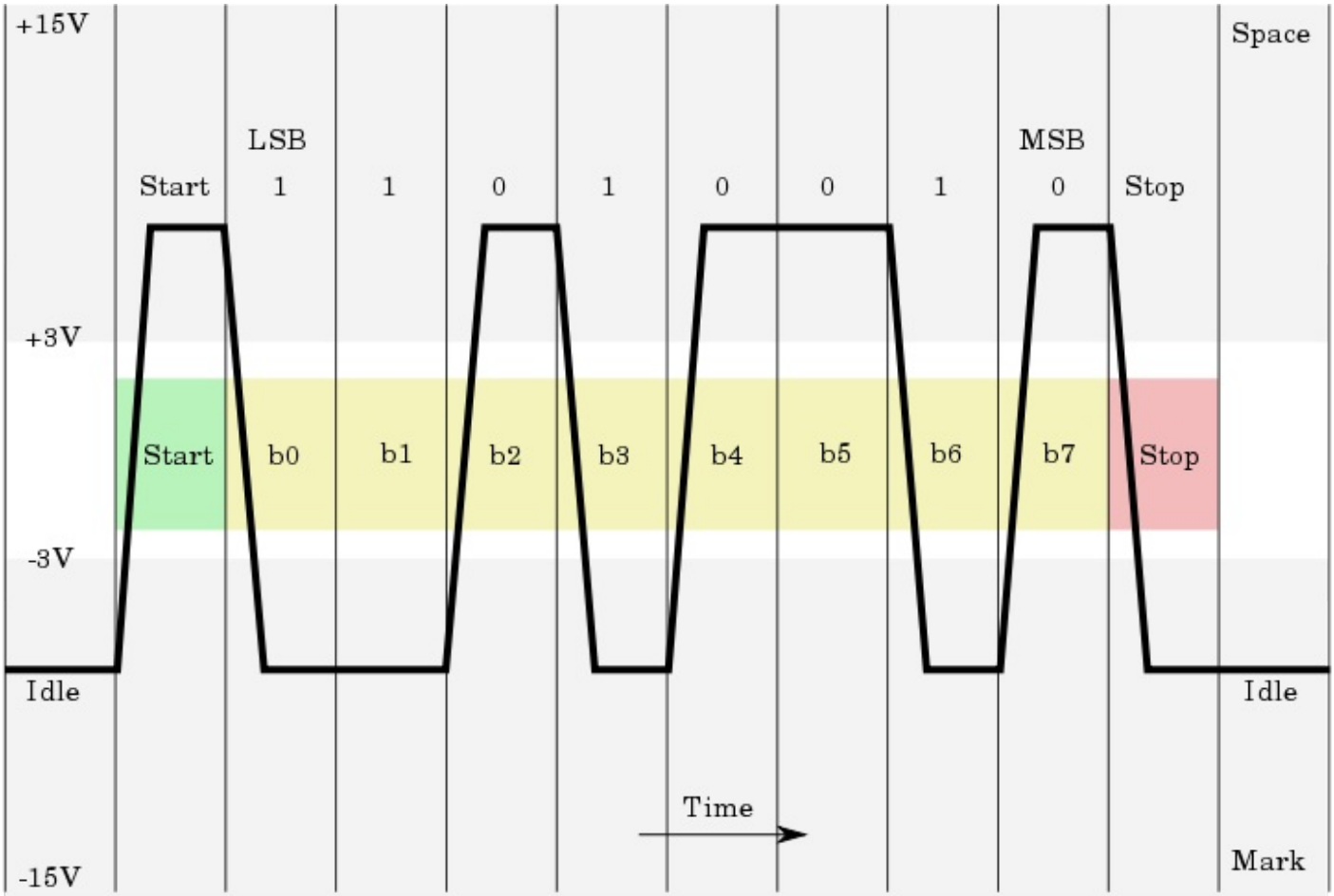
### *La norme RS232*

Qu'est-ce que c'est que cette bête-là ? A priori, il s'agit d'une norme. Bon, soit. Que fait-elle ? Cette norme définit les niveaux de tension qui doivent être utilisés pour l'échange de données. Je le disais tout à l'heure, le micro-contrôleur sur la carte Arduino n'utilise que des tensions de 0 et 5V (sauf pour ses entrées analogiques). Or, la norme RS232 nous impose ceci :

- Le NL1 doit être une tension comprise entre -3V et -25V
- Le NL0 doit être une tension comprise entre +3V et +25V

Encore un raisonnement logique de la part des concepteurs de cette liaison...

Bon, ben c'est à peu près tout ce qu'il y a à savoir là dessus. Je vais résumer tout ce que l'on vient de dire avec cette image, extraite de la page Wikipédia :




Petite précision, le **MSB** et le **LSB** sont les bits de **poids fort** (Most Significant Bit) et de **poids faible** (Less Significant Bit). En fait, lorsqu'on lit 0001010 (donc 'P'), le bit LSB est celui qui est tout à droite, tandis que le MSB est celui tout à gauche.

### *La vitesse de communication*

Quand on va utiliser la voie série, on va définir la vitesse à laquelle sont transférées les données. En effet, comme les bits sont transmis un par un, la liaison série envois les données en un temps prédéfini. Par exemple, on pourra envoyer une totalité de 9600 bits par secondes (9600 bps). Avec cette liaison, on peut envoyer entre 75 et 115200 bits par secondes ! Ce sera à nous de définir cette vitesse.

Il faut faire attention de ne pas confondre les bps et les bauds. Vous trouverez de plus amples informations à ce sujet sur cette page.

## Fonctionnement de la liaison série

Maintenant que l'on sait comment fonctionne le protocole de communication de la liaison série, je vais vous en dire un peu plus sur cette mystérieuse liaison, qui, depuis tout à l'heure n'a toujours pas révélé où elle se cachait.

### 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 connaitttrrreuhhh... Ah ben là, chui

pas d'accord !

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**.

### *Qu'est-ce que c'est ?*

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. A 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.

*Photos extraites du site Wikipédia - Connecteur DB9*
*Mâle à gauche ; Femelle à droite*

### *A quoi ça sert ?*

Si je vous parle de ça dans le chapitre sur la liaison série, c'est qu'il doit y avoir un lien, non ? Juste, car la liaison série (je parle là de la transmission des données) est véhiculée par ce connecteur. Donc, notre ordinateur dispose d'un connecteur DB9, qui permet de relier, via un câble adapté, sa connexion série à un autre matériel.

Mais alors, pourquoi tant de broches puisque tu nous as dit que la liaison série n'utilisait que 3 fils ?

Eh bien, toutes ces broches ont une fonction. 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.

Dans une communication, il arrive quelques fois qu'il y ait des erreurs de transmission (par exemple, dans une conversation téléphonique, il n'est pas anodin de mal avoir compris le nom de la personne, on lui redemande alors de l'énoncer). Sur la liaison série il peu se passer la même chose. Cependant, si on utilise la liaison telle que l'on l'a vu, on ne pourra pas vérifier la présence

d'erreurs. C'est là qu'interviennent les moyens mis en place pour la **gestion des erreurs**.

## La gestion des erreurs

### *Bit de parité*

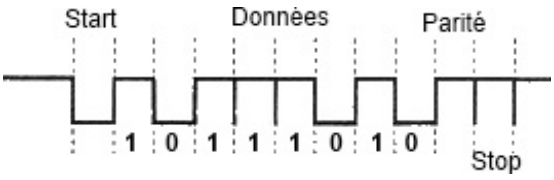
Le premier moyen, et le plus simple à mettre en œuvre pour diminuer le risque de réceptionner un signal sans erreur de transmission est d'utiliser un **bit de parité**. Ici, plus question de parler d'électronique, mais plutôt de logique et d'algorithme.

Comme vue précédemment, une transmission est faite d'un enchainement de plusieurs bits : *bit de start*, *bits de données* puis *bit de stop*. Afin de vérifier s'ils ont tous été bien transmis correctement, on va ajouter un bit de parité juste avant le bit de stop.

Ça a un rapport avec le fait que ce soit pair ou impair ? Mais alors, si oui, c'est quoi qui est pair et impair ? 

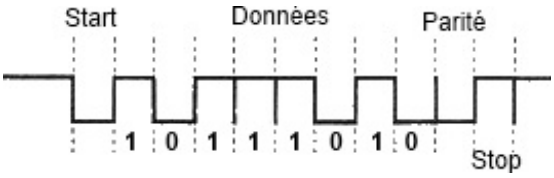
Tout à fait, il s'agit bien de cela. Regardons ensemble plus en détail ce que cela signifie.

Le bit de parité va en fait servir pour indiquer que le nombre de bit au niveau logique 1 soit bon. Plus exactement, si je choisis un bit de parité paire pour ma transmission série, alors ce bit aura un niveau logique (0 ou 1) qui dépend du nombre de bits transmis qui sont à l'état haut, pour donner au final un nombre pair de bits à 1 y compris avec le bit de parité. Voilà une petite image pour résumer ça :




On voit que le bit de parité est à 1, sachant qu'on l'a choisi pour qu'il soit pair et si on compte le nombre de 1, on a bien un nombre pair.

Il en est de même pour le bit de parité impaire, celui-ci est à 0 (pour les mêmes données), ce qui indique bien qu'on a un nombre impair de 1 :




Ceci est donc le premier moyen mis en œuvre pour éviter certaines erreurs de transmission. Après, c'est le programme qui va voir si le bit de parité est bon ; s'il est mauvais alors on demande à ce que les données soient renvoyées. Il se peut également que se soit le bit de parité qui soit mauvais (erreur de transmission).

## Désolé, je suis occupé...

Dans certains cas, et il n'est pas rare, les dispositifs communicant entre eux par l'intermédiaire de la liaison série ne traitent pas les données à la même vitesse. Tout comme lorsque l'on dicte quelque chose à quelqu'un et qu'il en prend note, celui qui dicte sera plus rapide que celui qui écrit. Celui qui dicte dictera alors moins vite pour attendre que celui qui écrit puisse intercepter toutes les informations dictées. Pour la liaison série, il existe quelque chose de semblable qui s'appelle le **contrôle de flux**.

### *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 logiciellement.

Le fonctionnement très simple de ce contrôle de flux utilise des caractères de la table ASCII, le caractère 17 et 19, respectivement

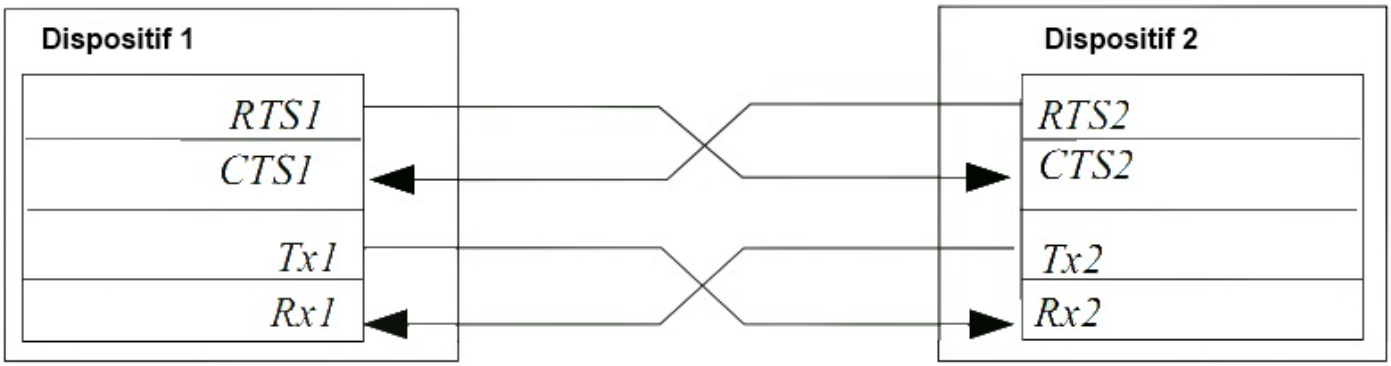
nommés **XON** et **XOFF**.

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 stockent 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 à a lui dire.

### *Contrôle de flux matériel*

On n'utilisera pas le contrôle de flux matériel avec Arduino, 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**.




Voilà le branchement adéquat pour utilise 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 très compliqué et est utilisé lorsque le contrôle de flux logiciel ne l'est pas.

## Mode de fonctionnement

Pour terminer, parlons du mode fonctionnement. Ce sera très rapide. 

### *Mode asynchrone*

Le mode asynchrone est en fait l'utilisation de la liaison série comme je viens de l'expliquer dans ce chapitre. Les données sont envoyées sur un fil et lues "à la volée". L'émetteur peut donc envoyer des informations plus rapidement que le récepteur ne les traite, sans contrôle de flux.

### *Mode synchrone*

Le mode synchrone utilise un signal d'horloge pour synchroniser l'émetteur et le récepteur lors d'une transmission. Ainsi, les deux dispositifs (ou plus) connaissent exactement la durée d'un bit et sont ainsi capable de dissocier les parasitent des bits de données. Cependant cette solution a ses limites lorsque l'on veut utiliser la liaison série sur de longues distances. D'autres

moyens sont envisageables, en utilisant seulement trois fils et en envoyant le signal d'horloge sur le fil de transmission des données.

Je ne vous en dirait pas plus, n'étant pas au point sur ce sujet et puis cela ne relève que de la culture électronique, on utilisera jamais, nous, cette méthode de transmission.

## Arduino et la communication

### *Les différentes cartes Arduino*

Selon les cartes Arduino que vous utilisez, vous pourrez utiliser une seule ou plusieurs liaisons séries. Par exemple, la carte Arduino Mega propose 4 voies séries différentes. La carte Arduino ADK (interfacer avec Android) propose elle aussi 4 voies séries. Lorsque vous utilisez les voies séries, vous faites appel à un objet Serial (nous verrons ça plus loin dans le cours). Ainsi, lorsqu'il n'y a qu'une seule voie série, l'objet utilisé est "Serial". Ensuite, s'il y a d'autres voies séries on aura les objets "Serial1", "Serial2" puis "Serial3".

### *Les autres moyens de communication*

Comme énoncé brièvement plus tôt, la voie série n'est pas le seul moyen de communication existant sur Arduino. En effet, il existe une multitude de types de connexion, natives ou non et plus ou moins difficiles à mettre en place. On citera par exemple l'I²C, qui est une communication de type "Maître/Esclave" et est intégré nativement à Arduino grâce à la librairie "Wire".

De manière native, il y a aussi la librairie "SPI" qui permet d'utiliser la communication du même nom.

Enfin, le Shield Ethernet vous permet de raccorder une liaison de type Ethernet à votre carte Arduino.

## Utiliser la liaison série avec Arduino

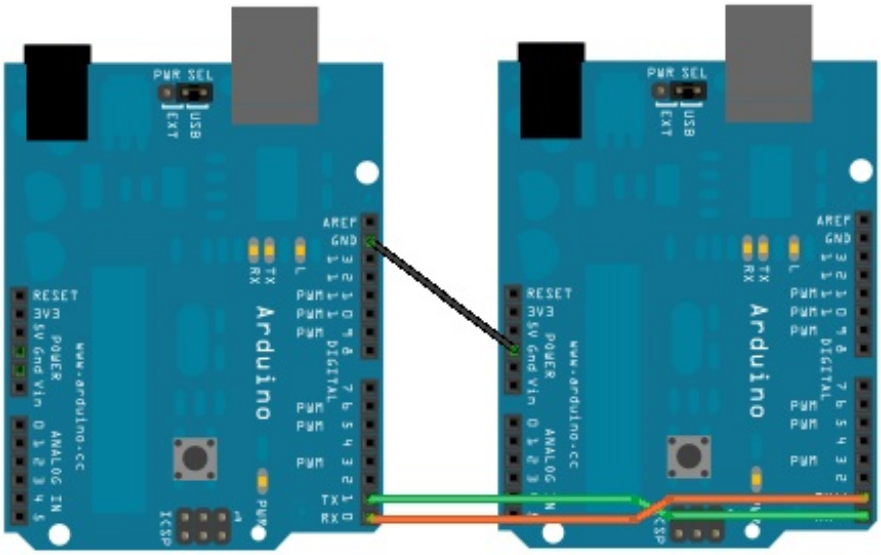
### *Entre l'ordinateur et la carte Arduino*

La liaison série entre la carte Arduino et l'ordinateur est établie à travers le port USB. En fait, ce port USB n'est pas utilisé avec le protocole USB, mais avec celui de la liaison série !

Ceci est donc géré par la carte Arduino et il n'y a rien à paramétrer.

### *Entre deux cartes Arduino*

Pour relier deux cartes Arduino en liaison série, rien de plus simple ! En effet, il suffit de connecter les broches Tx et Rx ensemble, de cette manière :



*Sur la première carte : Tx en vert ; Rx en orange*
*Sur la deuxième, c'est inversé !*

### *Entre une carte Arduino et un autre micro contrôleur*

Là, c'est la même chose que pour connecter deux Arduino ensemble. Il faut relier le Tx et le Rx de la carte Arduino au Rx et au Tx du micro-contrôleur.

## Différence entre Ordinateur et Arduino

## Les niveaux électriques

La transmission par voie série se fait, bien entendu, grâce à l'électricité. Cependant, les niveaux électriques (les tensions) ne sont pas les mêmes du côté de l'ordinateur ou du côté de Arduino. En effet, l'ordinateur utilise des tensions entre -12V et +12V (moyenne) alors que Arduino utilise pour sa part des tensions de 0 ou +5V.

Mais alors comment font-ils pour se comprendre ?

Bonne question, à laquelle nous allons répondre maintenant.

### *L'ordinateur*

Comme dit ci-dessus, l'ordinateur utilise des niveaux de -12V à +12V (de manière habituelle, mais ils sont en réalité entre -3/-24V et +3/+24V). Et dans ce petit monde, tout est à l'envers. Les niveaux "positifs" représentent un état bas (un '0' logique), alors qu'un niveau haut (le '1' logique) est représenté par les tensions négatives.

### *Arduino*

En électronique, et donc dans le cas de l'Arduino, on n'aime pas trop les tensions élevées et/ou négatives. En revanche, on apprécie énormément les tensions de 0V ou 5V (que l'on appelle niveau "TTL").

Pour que les deux composants puissent communiquer, on effectue une "adaptation de niveau", que l'on va étudier (rapidement) maintenant.

### *Adaptation de niveaux*

Afin de faire cette conversion, un composant est placé entre les deux supports. Le but de ce composant sera de faire l'adaptation afin que tout le monde se comprenne. Dans le cas de l'Arduino, c'est un cas un peu particulier puisque ce même composant sert

aussi à *émuler* une voie série. Ainsi, lorsque vous branchez la carte sur votre USB d'ordinateur, ce dernier détecte automatiquement un nouvel appareil avec lequel il est possible de communiquer par voie série.

## Cas d'utilisation

### *Avec un ordinateur*

Pour faire une communication avec un ordinateur, rien de plus simple... ou pas ! Depuis le début je vous parle d'un port série puis de prise type DB9. De nos jours, elles sont en voie d'extinction ! Mais les développeurs ont pensé à cet évènement. La carte Arduino, plutôt que d'être branché sur un port série classique sera donc branché sur l'USB. Les niveaux seront donc toujours du 5V maximum. Ensuite, un composant intégré à Arduino se chargera de simuler une voie série et tout devient transparent pour votre ordinateur. Il vous suffit donc juste d'utiliser le câble USB et de le relier.

### *Avec un autre système électronique*

Pour communiquer avec un autre appareil électronique en voie série (une autre carte Arduino par exemple), il faut juste suivre quelques étapes :

1. Coupez l'alimentation de chacune des cartes
2. Branchez le Tx de l'un sur le Rx de l'autre et vice-versa
3. Relié un fil de masse entre les deux cartes si l'alimentation est différente entre les deux (cela permet d'avoir une référence électrique entre les deux systèmes, une sorte de 'zéro commun')

### *Mise en garde*

Lorsque vous faites des montages "Voie Série <-> Ordinateur", ne branchez **JAMAIS** de fils sur les broches 0 et 1 de votre carte (si c'est une UNO, sinon se référer aux broches Tx/Rx de votre carte). Cela perturberait votre communication voir endommager la carte.

Vous savez maintenant quasiment tout du principe de communication de la liaison série.

Nous allons maintenant pouvoir passer à la pratique et commencer à utiliser cette liaison avec Arduino et envoyer et recevoir nos premières données.

# Envoyer/Recevoir des données

Dans ce chapitre, nous allons apprendre à utiliser la liaison 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 liaison série avec Arduino est quasiment un jeu d'enfant, puisque tout est opaque aux yeux de l'utilisateur...

## Préparer la liaison série

Petite introduction sur la liaison série : la liaison série est un moyen de communication utilisé pour faire communiquer entre eux plusieurs dispositifs. On retrouve cette liaison sur les ordinateurs, par exemple, ou sur des appareils électroniques (onduleurs, ...). Cette liaison est aussi utilisée dans le milieu industriel.

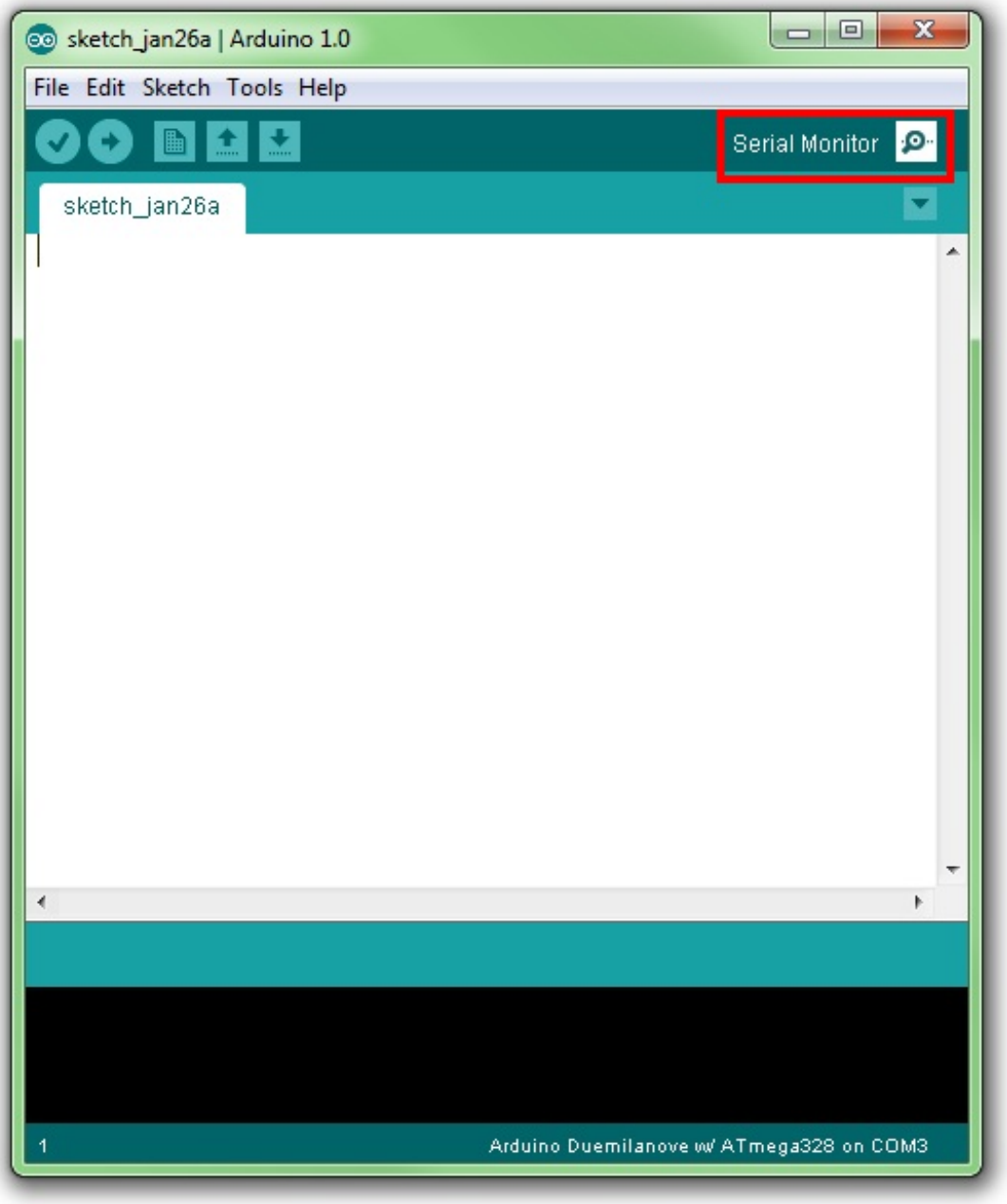
L'avantage de la liaison série, c'est de pouvoir émettre des informations d'un dispositif à un autre pour, par exemple, créer un système de domotique, afficher la température extérieure sur l'écran de son ordinateur, etc. On trouve une infinité de possibilités d'utilisation.

J'ai choisi d'introduire la liaison série avant les grandeurs analogiques car nous allons l'utiliser pour communiquer la tension présente sur une broche analogique de l'Arduino vers l'ordinateur.

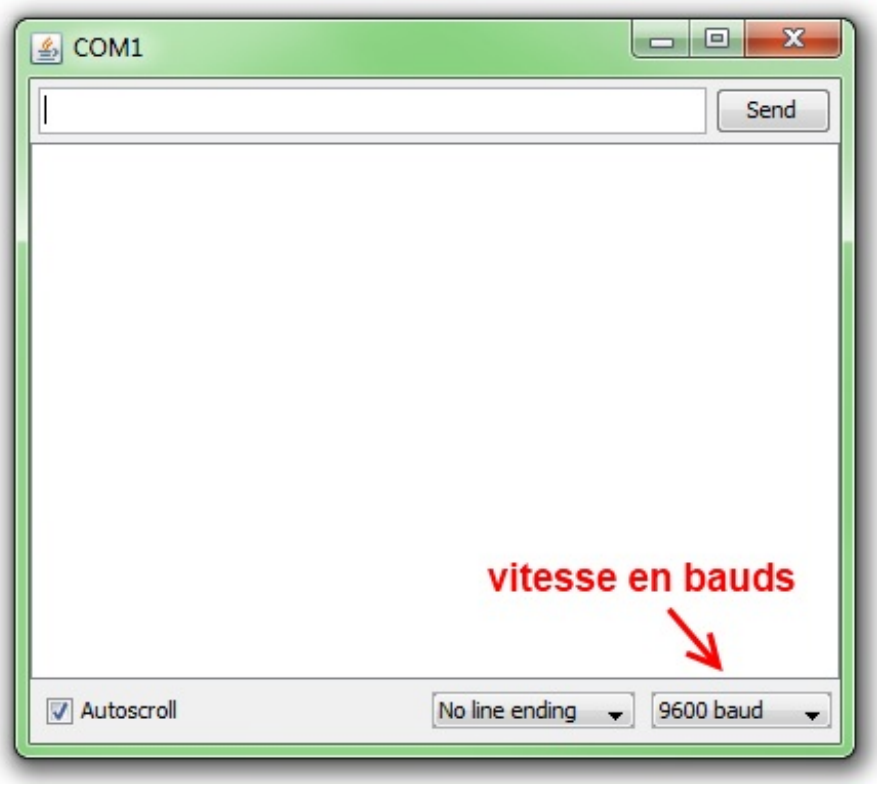
Notre objectif, pour le moment, est de communiquer des informations de la carte Arduino vers l'ordinateur et inversement. Pour ce faire, on va d'abord devoir préparer le terrain.

## 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 liaison série de votre ordinateur (qui est émulée 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.

## Du côté du programme

### *L'objet Serial*

Pour utiliser la liaison 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**.

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 tous 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 !

### *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 liaison série.

Le logiciel Arduino à prévu, dans sa *bibliothèque Serial*, tout un tas de fonctions qui vont nous êtres très utiles, voir même indispensables afin de bien utiliser la liaison 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.

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.

**Code : C**

```
void setup()
{
   Serial.begin(9600); //on démarre la liaison en la réglant à une vitesse de 9600 bits par seconde.
}
```

À présent, votre carte Arduino a ouvert une nouvelle communication vers l'ordinateur. Ils vont pouvoir communiquer ensemble.

## 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 envoyé des données à partir de l'ordinateur vers la carte Arduino il suffit d'ouvrir le terminal série et de taper le texte dedans ! Donc, on va bien programmer et voir comment faire pour que votre carte Arduino envoie des données à l'ordinateur.

Et ces données, elles proviennent d'où ?

Eh bien de la carte Arduino... En fait, lorsque l'on utilise la liaison 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.

## 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 !).

### *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 **chaine 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 Zozor !"

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" !

### *`print()` et `println()`*

La fonction que l'on va utiliser pour débuter, s'agit de `print()` et de son acolyte `println()`. Ces deux fonctions sont quasiment identiques, mais à quoi servent-elles ?

- `print()` : cette fonction permet d'envoyer des données sur la liaison série. On peut par exemple envoyer un caractère,

une chaine 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 :

**Code : C**

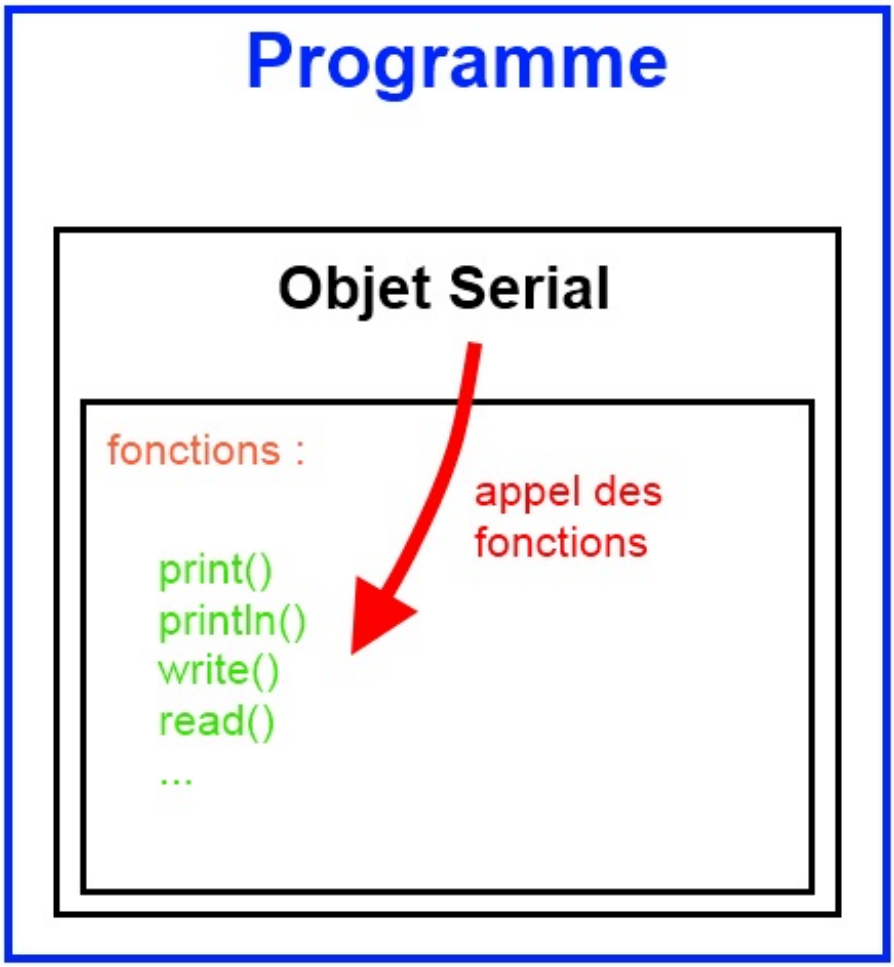
```
Serial.print("Salut les zéros !");
```

Bien sûr, au préalable, vous devrez avoir "déclaré/créé" votre objet Serial et définis une valeur de vitesse de communication :

**Code : C**

```
void setup()
{
   Serial.begin(9600); //création de l'objet Serial (=établissement d'une nouvelle communication série)

   Serial.print("Salut les zéros !"); //envoie de la chaine "Salut les zéros !" sur 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 propre chef :



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 : `Serial.print("Salut les zéros !");` eh bien je demande à mon objet Serial d'exécuter la fonction `print()` en lui passant pour paramètre la chaine de caractère : "Salut les zéros !".

On peut compléter le code précédent comme ceci :

**Code : C**

```
void setup()
{
   Serial.begin(9600);

   Serial.print("Salut les zéros ! "); //l'objet exécute une
première fonction
   Serial.println("Vive Zozor !");    //puis une deuxième fonction,
différente cette fois-ci
   Serial.println("Cette phrase passe en dessous des deux
précédentes"); //et exécute à nouveau la même
}
```

Sur le terminal série, on verra ceci :

**Code : Console**

```
Salut les zéros ! Vive Zozor !
Cette phrase passe en dessous des deux précédentes
```

## 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()`.

Petite précision, je vais utiliser de préférence `println()` pour sauter des lignes, mais je rappel que cette fonction fait la même chose que `print()`.

Résumons un peu ce que nous venons d'apprendre : on sait maintenant envoyer des caractères sur la liaison 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.

### *Envoyer des nombres*

Avec la fonction `print()`, il est aussi possible d'envoyer des chiffres ou des nombres car ce sont des caractères :

**Code : C**

```
void setup()
{
   Serial.begin(9600);

   Serial.println(9);            //chiffre
   Serial.println(42);           //nombre
   Serial.println(32768);        //nombre
   Serial.print(3.1415926535);   //nombre à virgule
}
```

**Code : Console**

```
9
42
32768
3.14
```

Tiens, le nombre pi n'est pas affiché correctement ! C'est quoi le bug ? 

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 :

**Code : C**

```
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);
}
```

**Code : Console**

```
3
3.14
3.1415
3.1415926535
```

### *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 :

**Code : C**

```
int variable = 512;
char lettre = 'a';

void setup()
{
   Serial.begin(9600);

   Serial.println(variable);
   Serial.print(lettre);
}
```

**Code : Console**

```
512
```

```
a
```

Trop facile n'est-ce pas ?

### *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 liaison série.

Prenons l'exemple d'un nombre choisi judicieusement : 65.

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.

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 "**A**merican **S**tandard **C**ode for **I**nformation **I**nterchange", soit en bon français : "Code américain normalisé pour l'échange d'information" est, selon Wikipédia :

**Citation : Wikipédia**

> "la norme de codage de caractères en informatique la plus connue, la plus ancienne et la plus largement compatible"

En somme, c'est un tableau de valeurs codées sur 8bits qui à chaque valeur associent 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, donc je vous donne une table ASCII "allégée".

**Secret** (cliquez pour afficher)

> 

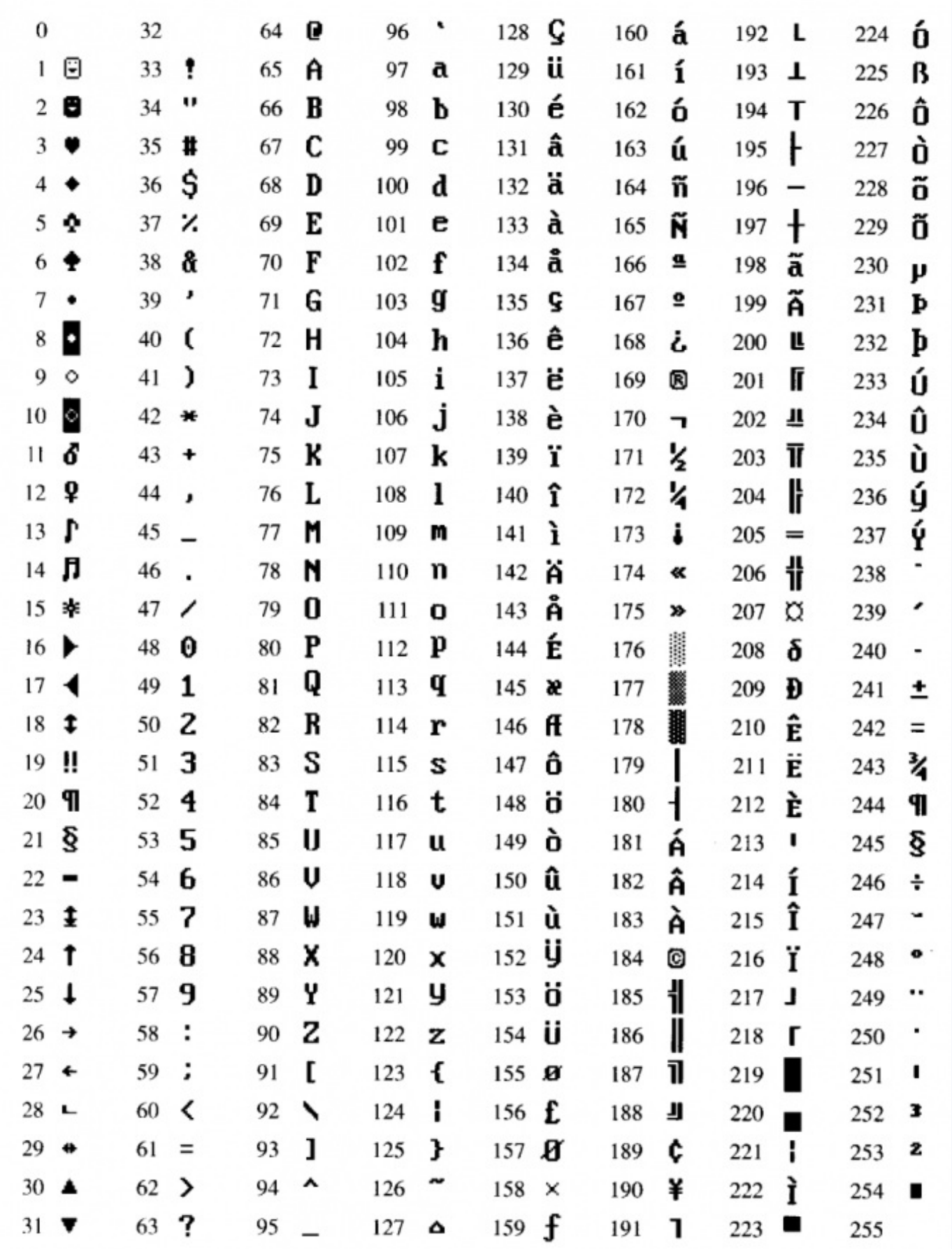
| Dec | Hex | Name | Char | Ctrl-char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | Null | NUL | CTRL-@ | 32 | 20 | Space | 64 | 40 | @ | 96 | 60 | ` |
| 1 | 1 | Start of heading | SOH | CTRL-A | 33 | 21 | ! | 65 | 41 | A | 97 | 61 | a |
| 2 | 2 | Start of text | STX | CTRL-B | 34 | 22 | " | 66 | 42 | B | 98 | 62 | b |
| 3 | 3 | End of text | ETX | CTRL-C | 35 | 23 | # | 67 | 43 | C | 99 | 63 | c |
| 4 | 4 | End of xmit | EOT | CTRL-D | 36 | 24 | $ | 68 | 44 | D | 100 | 64 | d |
| 5 | 5 | Enquiry | ENQ | CTRL-E | 37 | 25 | % | 69 | 45 | E | 101 | 65 | e |
| 6 | 6 | Acknowledge | ACK | CTRL-F | 38 | 26 | & | 70 | 46 | F | 102 | 66 | f |
| 7 | 7 | Bell | BEL | CTRL-G | 39 | 27 | ' | 71 | 47 | G | 103 | 67 | g |
| 8 | 8 | Backspace | BS | CTRL-H | 40 | 28 | ( | 72 | 48 | H | 104 | 68 | h |
| 9 | 9 | Horizontal tab | HT | CTRL-I | 41 | 29 | ) | 73 | 49 | I | 105 | 69 | i |
| 10 | 0A | Line feed | LF | CTRL-J | 42 | 2A | * | 74 | 4A | J | 106 | 6A | j |
| 11 | 0B | Vertical tab | VT | CTRL-K | 43 | 2B | + | 75 | 4B | K | 107 | 6B | k |
| 12 | 0C | Form feed | FF | CTRL-L | 44 | 2C | , | 76 | 4C | L | 108 | 6C | l |
| 13 | 0D | Carriage feed | CR | CTRL-M | 45 | 2D | - | 77 | 4D | M | 109 | 6D | m |
| 14 | 0E | Shift out | SO | CTRL-N | 46 | 2E | . | 78 | 4E | N | 110 | 6E | n |
| 15 | 0F | Shift in | SI | CTRL-O | 47 | 2F | / | 79 | 4F | O | 111 | 6F | o |
| 16 | 10 | Data line escape | DLE | CTRL-P | 48 | 30 | 0 | 80 | 50 | P | 112 | 70 | p |
| 17 | 11 | Device control 1 | DC1 | CTRL-Q | 49 | 31 | 1 | 81 | 51 | Q | 113 | 71 | q |
| 18 | 12 | Device control 2 | DC2 | CTRL-R | 50 | 32 | 2 | 82 | 52 | R | 114 | 72 | r |
| 19 | 13 | Device control 3 | DC3 | CTRL-S | 51 | 33 | 3 | 83 | 53 | S | 115 | 73 | s |
| 20 | 14 | Device control 4 | DC4 | CTRL-T | 52 | 34 | 4 | 84 | 54 | T | 116 | 74 | t |
| 21 | 15 | Neg acknowledge | NAK | CTRL-U | 53 | 35 | 5 | 85 | 55 | U | 117 | 75 | u |
| 22 | 16 | Synchronous idle | SYN | CTRL-V | 54 | 36 | 6 | 86 | 56 | V | 118 | 76 | v |
| 23 | 17 | End of xmit block | ETB | CTRL-W | 55 | 37 | 7 | 87 | 57 | W | 119 | 77 | w |
| 24 | 18 | Cancel | CAN | CTRL-X | 56 | 38 | 8 | 88 | 58 | X | 120 | 78 | x |
| 25 | 19 | End of medium | EM | CTRL-Y | 57 | 39 | 9 | 89 | 59 | Y | 121 | 79 | y |
| 26 | 1A | Substitute | SUB | CTRL-Z | 58 | 3A | : | 90 | 5A | Z | 122 | 7A | z |
| 27 | 1B | Escape | ESC | CTRL-[ | 59 | 3B | ; | 91 | 5B | [ | 123 | 7B | { |
| 28 | 1C | File separator | FS | CTRL-\ | 60 | 3C | < | 92 | 5C | \ | 124 | 7C | \| |
| 29 | 1D | Group separator | GS | CTRL-] | 61 | 3D | = | 93 | 5D | ] | 125 | 7D | } |
| 30 | 1E | Record separator | RS | CTRL-^ | 62 | 3E | > | 94 | 5E | ^ | 126 | 7E | ~ |
| 31 | 1F | Unit separator | US | CTRL-_ | 63 | 3F | ? | 95 | 5F | _ | 127 | 7F | DEL |

| Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char | Dec | Hex | Char |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 128 | 80 | Ç | 160 | A0 | á | 192 | C0 | └ | 224 | E0 | α |
| 129 | 81 | ü | 161 | A1 | í | 193 | C1 | ┴ | 225 | E1 | ß |
| 130 | 82 | é | 162 | A2 | ó | 194 | C2 | ┬ | 226 | E2 | Γ |
| 131 | 83 | â | 163 | A3 | ú | 195 | C3 | ├ | 227 | E3 | π |
| 132 | 84 | ä | 164 | A4 | ñ | 196 | C4 | ─ | 228 | E4 | Σ |
| 133 | 85 | à | 165 | A5 | Ñ | 197 | C5 | ┼ | 229 | E5 | σ |
| 134 | 86 | å | 166 | A6 | ª | 198 | C6 | ╞ | 230 | E6 | µ |
| 135 | 87 | ç | 167 | A7 | º | 199 | C7 | ╟ | 231 | E7 | τ |
| 136 | 88 | ê | 168 | A8 | ¿ | 200 | C8 | ╚ | 232 | E8 | Φ |
| 137 | 89 | ë | 169 | A9 | ⌐ | 201 | C9 | ╔ | 233 | E9 | Θ |
| 138 | 8A | è | 170 | AA | ¬ | 202 | CA | ╩ | 234 | EA | Ω |
| 139 | 8B | ï | 171 | AB | ½ | 203 | CB | ╦ | 235 | EB | δ |
| 140 | 8C | î | 172 | AC | ¼ | 204 | CC | ╠ | 236 | EC | ∞ |
| 141 | 8D | ì | 173 | AD | ¡ | 205 | CD | ═ | 237 | ED | φ |
| 142 | 8E | Ä | 174 | AE | « | 206 | CE | ╬ | 238 | EE | ε |
| 143 | 8F | Å | 175 | AF | » | 207 | CF | ╧ | 239 | EF | ∩ |
| 144 | 90 | É | 176 | B0 | ░ | 208 | D0 | ╨ | 240 | F0 | ≡ |
| 145 | 91 | æ | 177 | B1 | ▒ | 209 | D1 | ╤ | 241 | F1 | ± |
| 146 | 92 | Æ | 178 | B2 | ▓ | 210 | D2 | ╥ | 242 | F2 | ≥ |
| 147 | 93 | ô | 179 | B3 | │ | 211 | D3 | ╙ | 243 | F3 | ≤ |
| 148 | 94 | ö | 180 | B4 | ┤ | 212 | D4 | Ö | 244 | F4 | ⌠ |
| 149 | 95 | ò | 181 | B5 | ╡ | 213 | D5 | ╒ | 245 | F5 | ⌡ |
| 150 | 96 | û | 182 | B6 | ╢ | 214 | D6 | ╓ | 246 | F6 | ÷ |
| 151 | 97 | ù | 183 | B7 | ╖ | 215 | D7 | ╫ | 247 | F7 | ≈ |
| 152 | 98 | ÿ | 184 | B8 | ╕ | 216 | D8 | ╪ | 248 | F8 | ° |
| 153 | 99 | Ö | 185 | B9 | ╣ | 217 | D9 | ┘ | 249 | F9 | ∙ |
| 154 | 9A | Ü | 186 | BA | ║ | 218 | DA | ┌ | 250 | FA | · |
| 155 | 9B | ¢ | 187 | BB | ╗ | 219 | DB | █ | 251 | FB | √ |
| 156 | 9C | £ | 188 | BC | ╝ | 220 | DC | ▄ | 252 | FC | ⁿ |
| 157 | 9D | ¥ | 189 | BD | ╜ | 221 | DD | ▌ | 253 | FD | ² |
| 158 | 9E | Pts | 190 | BE | ╛ | 222 | DE | ▐ | 254 | FE | ■ |
| 159 | 9F | ƒ | 191 | BF | ┐ | 223 | DF | ▀ | 255 | FF | |

Source de la table : http://www.commfront.com/ascii-chart-table.htm

Voici une deuxième table avec les caractères et symboles affichés :

**Secret** (cliquez pour afficher)



| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | | 32 | | 64 | @ | 96 | ` | 128 | Ç | 160 | á | 192 | └ | 224 | Ó |
| 1 | ☺ | 33 | ! | 65 | A | 97 | a | 129 | ü | 161 | í | 193 | ┴ | 225 | ß |
| 2 | ☻ | 34 | " | 66 | B | 98 | b | 130 | é | 162 | ó | 194 | ┬ | 226 | Ô |
| 3 | ♥ | 35 | # | 67 | C | 99 | c | 131 | â | 163 | ú | 195 | ├ | 227 | Ò |
| 4 | ♦ | 36 | $ | 68 | D | 100 | d | 132 | ä | 164 | ñ | 196 | ─ | 228 | õ |
| 5 | ♣ | 37 | % | 69 | E | 101 | e | 133 | à | 165 | Ñ | 197 | ┼ | 229 | Õ |
| 6 | ♠ | 38 | & | 70 | F | 102 | f | 134 | å | 166 | ª | 198 | ã | 230 | µ |
| 7 | • | 39 | ' | 71 | G | 103 | g | 135 | ç | 167 | º | 199 | Ã | 231 | Þ |
| 8 | ◘ | 40 | ( | 72 | H | 104 | h | 136 | ê | 168 | ¿ | 200 | ╚ | 232 | þ |
| 9 | ○ | 41 | ) | 73 | I | 105 | i | 137 | ë | 169 | ® | 201 | ╔ | 233 | Ú |
| 10 | ◙ | 42 | * | 74 | J | 106 | j | 138 | è | 170 | ¬ | 202 | ╩ | 234 | Û |
| 11 | ♂ | 43 | + | 75 | K | 107 | k | 139 | ï | 171 | ½ | 203 | ╦ | 235 | Ù |
| 12 | ♀ | 44 | , | 76 | L | 108 | l | 140 | î | 172 | ¼ | 204 | ╠ | 236 | ý |
| 13 | ♪ | 45 | - | 77 | M | 109 | m | 141 | ì | 173 | ¡ | 205 | ═ | 237 | Ý |
| 14 | ♫ | 46 | . | 78 | N | 110 | n | 142 | Ä | 174 | « | 206 | ╬ | 238 | ¯ |
| 15 | ☼ | 47 | / | 79 | O | 111 | o | 143 | Å | 175 | » | 207 | ¤ | 239 | ´ |
| 16 | ► | 48 | 0 | 80 | P | 112 | p | 144 | É | 176 | ░ | 208 | ð | 240 |  |
| 17 | ◄ | 49 | 1 | 81 | Q | 113 | q | 145 | æ | 177 | ▒ | 209 | Ð | 241 | ± |
| 18 | ↕ | 50 | 2 | 82 | R | 114 | r | 146 | Æ | 178 | ▓ | 210 | Ê | 242 | ‗ |
| 19 | ‼ | 51 | 3 | 83 | S | 115 | s | 147 | ô | 179 | │ | 211 | Ë | 243 | ¾ |
| 20 | ¶ | 52 | 4 | 84 | T | 116 | t | 148 | ö | 180 | ┤ | 212 | È | 244 | ¶ |
| 21 | § | 53 | 5 | 85 | U | 117 | u | 149 | ò | 181 | Á | 213 | ı | 245 | § |
| 22 | ▬ | 54 | 6 | 86 | V | 118 | v | 150 | û | 182 | Â | 214 | Í | 246 | ÷ |
| 23 | ↨ | 55 | 7 | 87 | W | 119 | w | 151 | ù | 183 | À | 215 | Î | 247 | ¸ |
| 24 | ↑ | 56 | 8 | 88 | X | 120 | x | 152 | ÿ | 184 | © | 216 | Ï | 248 | ° |
| 25 | ↓ | 57 | 9 | 89 | Y | 121 | y | 153 | Ö | 185 | ╣ | 217 | ┘ | 249 | ¨ |
| 26 | → | 58 | : | 90 | Z | 122 | z | 154 | Ü | 186 | ║ | 218 | ┌ | 250 | · |
| 27 | ← | 59 | ; | 91 | [ | 123 | { | 155 | ø | 187 | ╗ | 219 | █ | 251 | ¹ |
| 28 | ∟ | 60 | < | 92 | \ | 124 | ¦ | 156 | £ | 188 | ╝ | 220 | ▄ | 252 | ³ |
| 29 | ↔ | 61 | = | 93 | ] | 125 | } | 157 | Ø | 189 | ¢ | 221 | ¦ | 253 | ² |
| 30 | ▲ | 62 | > | 94 | ^ | 126 | ~ | 158 | × | 190 | ¥ | 222 | Ì | 254 | ■ |
| 31 | ▼ | 63 | ? | 95 | _ | 127 | ⌂ | 159 | ƒ | 191 | ┐ | 223 | ▀ | 255 | |

Source de cette table : http://www.lyceedupaysdesoule.fr/infor [...] ble_ascii.htm

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 :

**Code : C**

```
maVariable = 'A'; //l'ordinateur stocke la valeur 65 dans sa
mémoire (cf. table ASCII)
```

Si vous faites ensuite :

**Code : C**

```
maVariable = maVariable + 1; //la valeur stockée passe à 66 (= 65 +
1)

//à l'écran, on verra s'afficher la lettre "B"
```

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).

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.

**Code : C**

```
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
}
```

Vous pouvez donc manipuler les données que vous envoyez à travers la liaison série ! C'est là qu'est l'avantage de cette fonction.

## Exercice : Envoyer l'alphabet

### *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.

**Bon courage !**

### *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** (cliquez pour afficher)

**Code : C**

```
void loop()
{
 char i = 0;
 char lettre = 'a'; // ou 'A' pour envoyer en majuscule

 Serial.println("------ L'alphabet des Zéros ------"); //petit
message d'accueil

 //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
}
```

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 ! 

## 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 conversion, l'objet Serial dispose d'une fonction pour "écouter" la liaison série afin de savoir si oui ou non il y a une communication de données.

## Réception de données

### *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 liaison série.

Voici un exemple de traitement :

**Code : C**

```
void loop()
{
    int donneesALire = Serial.available(); //lecture du nombre de
caractères disponibles dans le buffer
    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
}
```

Cette fonction de l'objet Serial, available(), renvoie la valeur -1 quand il n'y a rien à lire sur le buffer de réception.

### *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 les urgences traitent la première personne arrivée dans la salle d'attente avant de passer au suivant). On accède donc caractère par caractère aux données reçues. Si jamais rien n'est à lire (personne dans la file d'attente), je le disais, la fonction renverra -1 pour le signaler.

**Code : C**

```
void loop()
{
   char choseLue = Serial.read(); //on lit le premier caractère non
traité du buffer

   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
   }
}
```

Ce code est une façon simple de se passer de la fonction available().

### *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 a 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 :

**Code : C**

```
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 tite 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
{
    while(Serial.read() != -1); //lit toutes les données (vide le
buffer de réception)
    digitalWrite(maLed, LOW); //on allume la LED
}
```

## 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).

**Code : C**

```
void setup()
{
 Serial.begin(9600);
}

void loop()
{
 char carlu = 0; //variable contenant le caractère à lire
 int cardispo = 0; //variable contenant le nombre de caractère
disponibles dans le buffer

 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
}
```

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. 

## [Exercice] Attention à la casse !

### 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 :

## Correction

Je suppose que grâce au superbe tutoriel qui précède vous avez déjà fini sans problème, n'est-ce pas ?

### *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 liaison série à 9600bauds :

**Secret** (cliquez pour afficher)

**Code : C**

```
int carlu; //stock le caractère lu sur la voie série

void setup()
{
    Serial.begin(9600);
}
```

### *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** (cliquez pour afficher)

**Code : C**

```
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

      if(carlu >= 'a' && carlu <= 'z') //Est-ce que c'est un
caractère minuscule ?
      {
         carlu = carlu - 'a'; //on garde juste le "numéro de
lettre"
         carlu = carlu + 'A'; //on passe en majuscule
      }
      else if(carlu >= 'A' && carlu <= 'Z') //Est-ce que c'est un
caractère MAJUSCULE ?
      {
         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);
   }
}
```

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 de carlu |
|---|---|---|---|
| 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.

A la ligne 19, j'utilise la fonction `write()` 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.

# [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, liaison 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 !

## Sujet du TP

## 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 ! "Requiiiinn" 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.

## 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 liaison série) aux sauveteurs sur la plage pour qu'ils changent les couleurs en temps réel. Voici les mots-clés et leurs impacts :

- meduse, tempete, requin : Des animaux dangereux ou la météo rendent la zPlage dangereuse. Baignade interdite
- vague : La natation est réservée aux bons nageurs
- surveillant, calme : Tout baigne, les zSauveteurs sont là et la mer est cool

## Conseil

Voici quelques conseils pour mener à bien votre objectif.

### *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 `clignoter()` ou `changerDeCouleur()` peuvent-être les bienvenues. 

### *Précision sur les chaines 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 :

**Code : C**

```
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 :

| | |
|---|---|
| mot[0] | 'c' |
| mot[1] | 'o' |
| mot[2] | 'u' |
| mot[3] | 'c' |
| mot[4] | 'o' |
| mot[5] | 'u' |
| mot[6] | '\0' |

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 :

**Code : C**

```
int resultat = strcmp(motRecu, "requin"); //utilisation de la
fonction strcmp(chaine1, chaine2) pour comparer des mots

if(resultat == 0)
   Serial.print("Les chaines sont identiques");

else
   Serial.print("Les chaines sont différentes");
```

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 chaine tapée sur la liaison série, il faut lui rajouter ce fameux '\0'.

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 !

## 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.

Voici une photo d'illustration du montage ainsi qu'une vidéo du montage en action.

# Bon Courage !

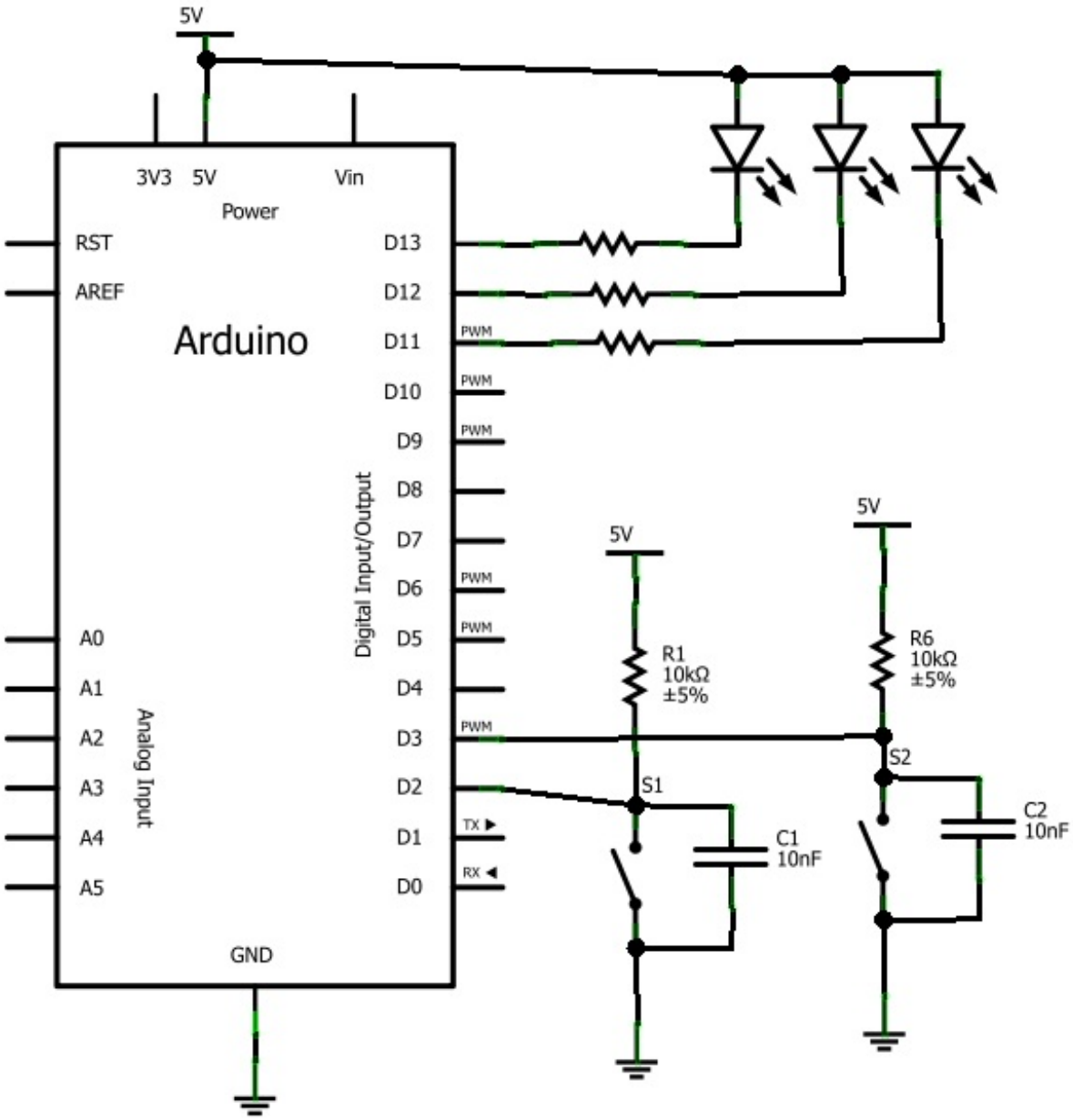
## Correction !

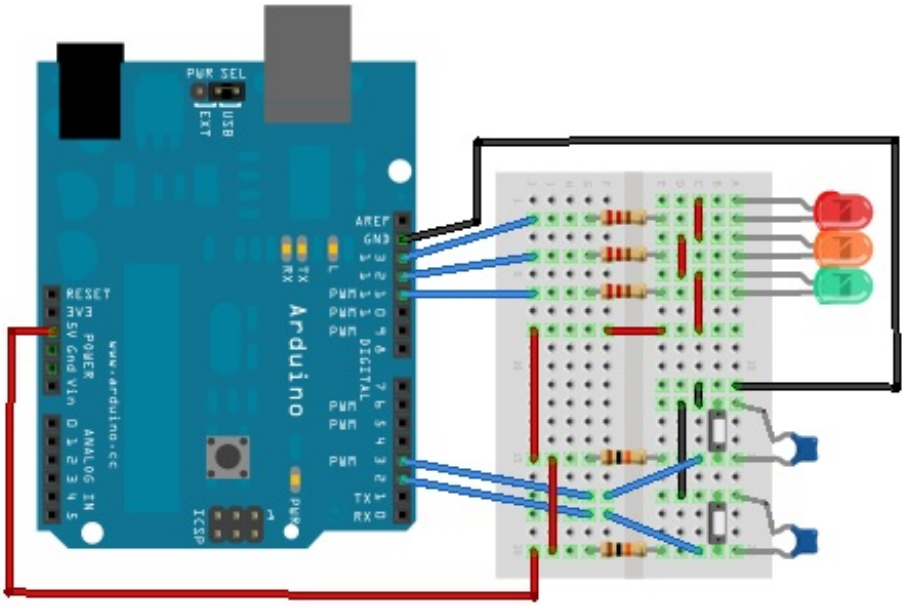
On corrige ?

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. 

## 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 :


5V
3V3
5V
Vin
Power
RST
AREF
Arduino
D13
D12
D11
D10
D9
D8
D7
D6
D5
D4
D3
D2
D1
D0
PWM
Digital Input/Output
A0
A1
A2
A3
A4
A5
Analog Input
GND
R1
10kΩ
±5%
R6
10kΩ
±5%
S1
S2
C1
10nF
C2
10nF
TX
RX



Quelles raisons nous ont poussés à faire ces branchements ? Eh bien :

- On utilise la liaison 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)

## 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().

### *Les variables globales*

**Code : C**

```
#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 liaison 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 des LED
```

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.

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;`

### *La fonction `setup()`*

Rien de particulier dans la fonction `setup()` par rapport à ce que vous avez vu précédemment, on initialise les variables

**Code : C**

```
void setup()
{
 Serial.begin(9600); //On démarre la voie série avec une vitesse de 9600 bits/seconde

 //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);
           }
}
```

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 :

**Code : C**

```
//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);
```

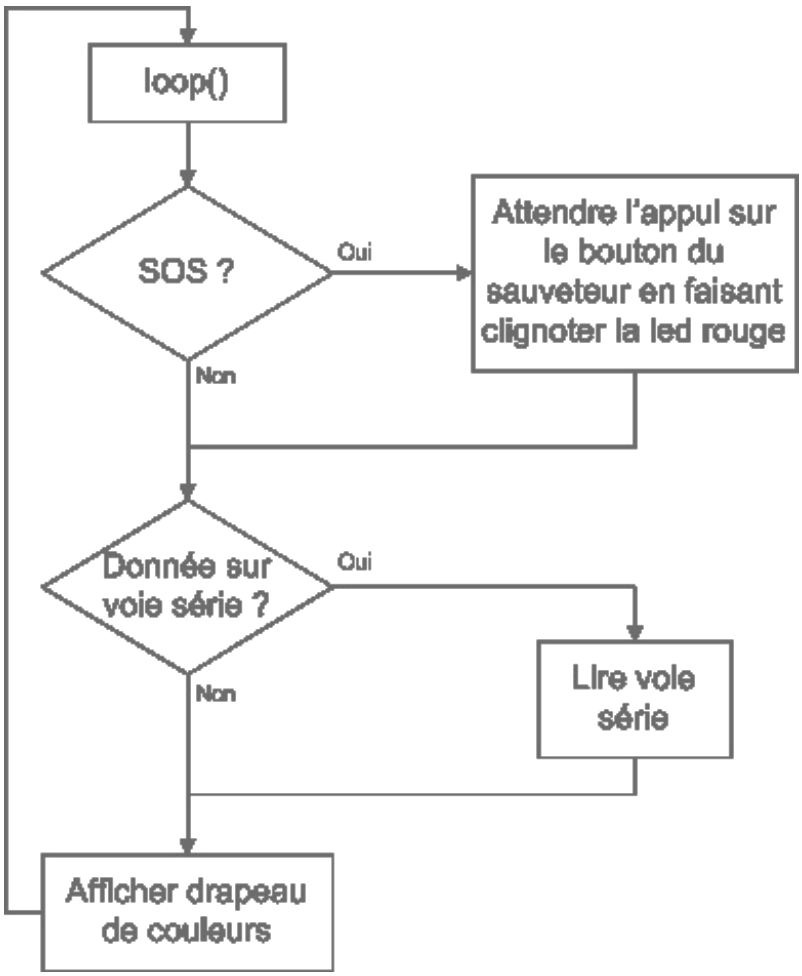
Si vous n'utilisez pas cette astuce dans notre cas, ce n'est pas dramatique. En fait, cela est utilisé lorsque vous avez 20 ou même 100 LED et broches à initialiser ! C'est moins fatigant comme ça... Qui a dit programmeur ? 

## La fonction principale et les autres

### *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.

### *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 quelles aient toutes un nom explicite pour que le programme soit facilement compréhensible sans même connaître le code qu'elles contiennent.

**Code : C**

```
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 continu en vérifiant la présence de caractère sur la
liaison série
 //s'il y a des données disponibles sur la liaison 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);
}
```

### *Lecture des données sur la liaison série*

Afin de garder la fonction loop "légère", nous avons rajouté quelques fonctions annexes. La première sera celle de lecture de la liaison 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 :

**Code : C**

```
//lit un mot sur la liaison 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
 }
        mot[i] = '\0'; //on supprime le caractère '\n' et on le
remplace par celui de fin de chaine '\0'
}
```

### *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.

**Code : C**

```
/*
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
*/

}
```

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 !

### *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.

**Code : C**

```
//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
   clignotant = !clignotant; //si clignotant était FALSE, il devient
TRUE et inversement
   digitalWrite(leds[ROUGE], clignotant); //la LEd est allumée au
gré de la variable clignotant
   temps = millis(); //on se rappel de la date de dernier passage
  }
 }
}
```

### *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 liaison 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.

**Code : C**

```
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 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.

**Code : C**

```
#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 liaison 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 des LED


void setup()
{
 Serial.begin(9600); //On démarre la voie série avec une vitesse de
9600 bits/seconde

 //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 continu en vérifiant la présence de caractère sur la
liaison série
 //s'il y a des données disponibles sur la liaison 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 liaison 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
 }
        mot[i] = '\0'; //on supprime le caractère '\n' et on le
remplace par celui de fin de chaine '\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
     clignotant = !clignotant; //si clignotant était FALSE, il devient
TRUE et inversement
     digitalWrite(leds[ROUGE], clignotant); //la LEd est allumée au
gré de la variable clignotant
     temps = millis(); //on se rappel de la date de dernier passage
    }
   }
}


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;
}
```

Je rappel 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.

## 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 :

### *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 liaison série du micro-contrôleur. Je vous laisse le soin de chercher à comprendre comment elle fonctionne et s'utilise, sur cette page.

### *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 liaison 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 !

Voila 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 !

# [Annexe] Votre ordinateur et sa liaison série dans un autre langage de programmation

Maintenant que vous savez comment utiliser la liaison 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 ).

Cette annexe a donc pour but de vous montrer comment utiliser la liaison 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 :

- - C++ et Qt (librairie QextSerialPort)
- - Java
- - C# (donc .Net plus globalement)

(Je suis désolé je ne connais pas le python pour l'instant)

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.

## En C++ avec Qt

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 le tutoriel C++ !

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 liaison 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.

## 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.

### *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 :

**Code : Console**

```
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 !)

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 !

### *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.

### *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.

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.

### *Installer la librairie : Sous Windows*

Ce point est en cours de rédaction, merci de patienter avant sa mise en ligne. 

### *Infos à rajouter dans le .pro*

Dans votre nouveau projet Qt pour traiter avec la liaison série, vous aller rajouter les lignes suivantes à votre .pro :

**Code : Autre**

```
INCLUDEPATH += QextSerialPort

CONFIG(debug, debug|release):LIBS  += -lqextserialportd
else:LIBS  += -lqextserialport
```

La ligne "INCLUDEPATH" représente le dossier où vous avez mis les fichiers sources de QextSerialPort. Les deux autres lignes font le lien vers les librairies copiées plus tôt (les .so ou les .dll selon votre OS).

## Les trucs utiles

### *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 :


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 |
| QButton | btnconnexion | (Dé)Connecte la voie série (bouton "checkable") |
| QTextEdit | boxEmission | Nous écrirons ici le texte à envoyer |
| QTextEdit | boxReception | Ici apparaitra le texte à recevoir |

### *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 `QextSerialEnumerator`. En parallèle, nous allons utiliser un autre objet pour stocker les informations des ports, il s'appelle `QextPortInfo`. Voici un exemple de code leur permettant de fonctionner ensemble :

**Code : C++**

```
QextSerialEnumerator enumerateur; //L'objet mentionnant les infos
QList<QextPortInfo> ports = enumerateur.getPorts();//on met ces
infos dans une liste
```

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 :

**Code : C++**

```
//on parcourt la liste des ports
    for(int i=0; i<ports.size(); i++)
        ui->ComboPort->addItem(ports.at(i).physName);
```

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".

**Code : C++**

```
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");
```

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.

### *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 fonction instanciera un objet QextSerialPort pour créer la communication, règlera cet objet et enfin ouvrira le canal. Dans le cas où le bouton était déjà coché (puisqu'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 :

**Code : C++**

```
private:
    QextSerialPort * port; //l'objet représentant le port

    BaudRateType getBaudRateFromString(QString baudRate); //une
fonction utile que j'expliquerais après
```

```
    private slots:
        void on_btnconnexion_clicked(); //le slot automatique du bouton
    de connexion
```

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 liaison 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) :

**Code : C++**

```
    //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 règlages 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");
        }
    }
```

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 chaines et le type particulier attendu, on crée et utilise la fonction "getBaudRateFromString". A partir d'un simple **switch**, elle fera la traduction entre `QString` et `BaudRateType`.

**Code : C++**

```
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;
    }
}
```

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 liaison série et par la boite 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 **else** qui permet de faire une déconnexion. Pour cela on utilise la méthode `close()` et ensuite on supprime l'objet QextSerialPort pour ne pas encombrer inutilement la mémoire. Ces deux opérations sont aussi à faire dans le destructeur de la classe `Fenetre` qui affiche l'ensemble (en s'assurant que l'objet port n'est pas `NULL`).

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).

## 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.

### *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 boite avant de l'envoyer sur la liaison série. L'envoi de texte se fait à partir de la fonction `write()` et prend en paramètre un tableau de `char`, ou un `QByteArray`. Bonne nouvelle, les `QString` peuvent générer des QByteArray en utilisant la méthode `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) :

**Code : C++**

```
void Fenetre::sendData() {
    QString caractere = ui->boxEmission->toPlainText().right(1);
//On récupère le dernier caractère tapé
    if(port != NULL) //si le port est instancié (donc ouvert a
priori)
        port->write(caractere.toAscii());
```

```
}
```

### *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 liaison 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 liaison série. C'est la forme qui est utilisée par défaut par l'objet `QextSerialPort`. Lorsqu'une donnée arrive, un signal (`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 `setMode()` de l'objet `QextSerialPort`. Le paramètre à passer sera `QextSerialPort::Polling` ou `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 `read()` ou `readall()` de la classe `QextSerialPort`
- Le copier dans la boite de texte "réception"

**Code : C++**

```
void Fenetre::readData() {
    QByteArray array = port->readAll();
    ui->boxReception->insertPlainText(array);
}
```

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.

Nous vous serons donc reconnaissants de ne pas nous harceler de commentaires relatifs au tuto pour nous dire "bwaaaa c'est mal codéééééé". Merci !

## En C# (.Net)

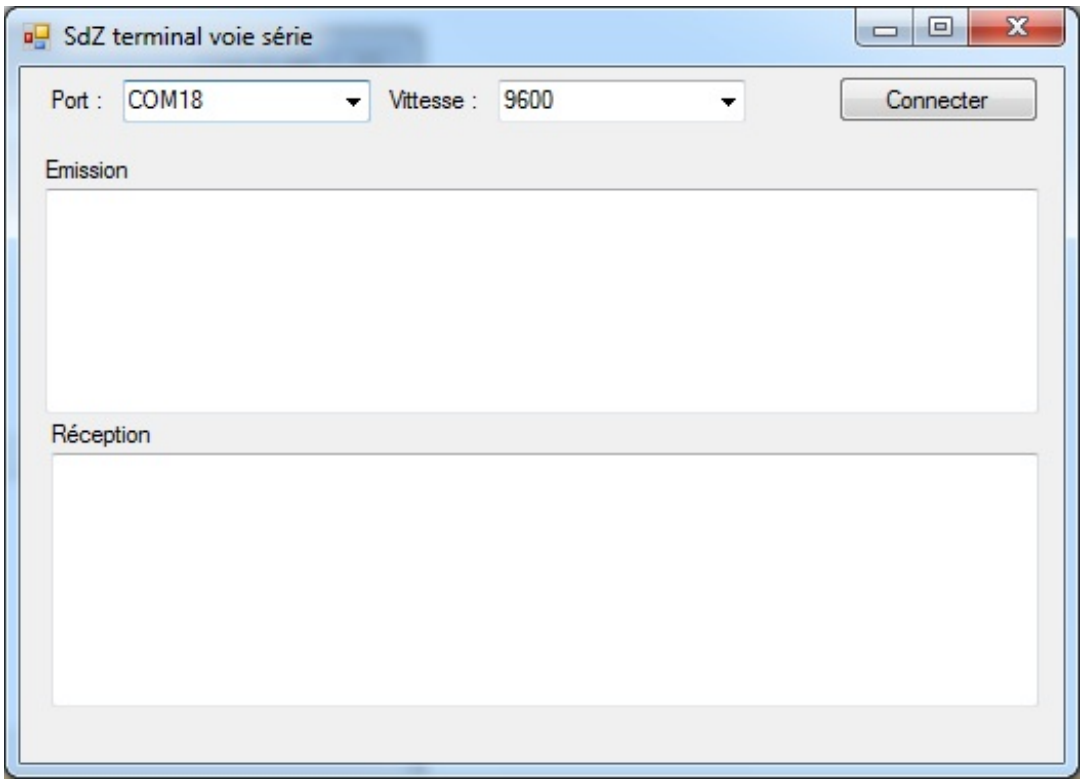
Dans cette partie (comme dans les précédentes) je pars du principe que vous connaissez le langage et avez déjà dessiné des interfaces et créé des actions sur des boutons par exemple. Cette sous-partie n'est pas là pour vous apprendre le C# !

Là encore je vais reprendre la même structure que les précédentes sous-parties.

## Les trucs utiles

### *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).




L'interface en C#

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 liaison série (bouton "checkable") |
| System.Windows.Forms.TextBox | boxEmission | Nous écrirons ici le texte à envoyer |
| System.Windows.Forms.TextBox | boxReception | Ici apparaitra le texte à recevoir |

Avant de commencer les choses marrantes, nous allons d'abord devoir ajouter une librairie : celle des liaisons séries. Elle se nomme simplement `Ports` et vous aurez donc la ligne suivante à rajouter en haut de votre projet : `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 !

**Code : C#**

```
SerialPort port
```

Maintenant que tous les outils sont prêts, nous pouvons commencer !

### *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 `SerialPort`. Cette fonction se nomme `GetPortNames()` et nous renvoie un tableau de `String`. Chaque case du tableau sera une chaîne de caractère comportant le nom d'une liaison 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 `Form_Load` qui se déclenche lorsque la fenêtre s'ouvre. Vous pouviez aussi très bien le mettre dans le constructeur, juste après la méthode `InitializeComponent()` qui charge les composants.

**Code : C#**

```
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");
}
```

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).

### *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 surement, 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 "Deconnecter").

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 `SerialPort` pour notre variable port. Ensuite, nous règlerons cette liaison série avec les différents paramètres (vitesse, parité, nom...) et enfin on pourra ouvrir le port. Chacune de ces étapes est en fait une propriété de notre objet `SerialPort`. Par exemple, pour le nom du port à utiliser, c'est la propriété `PortName` qui est à changer, pour celle des vitesses se sera `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.

**Code : C#**

```
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
            port.BaudRate =
int.Parse(comboVitesse.SelectedItem.ToString()); //parse en int le
combo des vitesses
            port.DataBits = 8;
            port.StopBits = StopBits.One;
            port.Parity = Parity.None;
            port.PortName = comboPort.SelectedItem.ToString();
//récupère le nom sélectionné

            //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 = "Deconnecter";
        }
        else //sinon on déconnecte
        {
            port.Close(); //ferme la voie série
            btnConnexion.Text = "Connecter";
        }
    }
```

Le point qui peut paraître étrange est la ligne 16, avec la propriété `DataReceived`. En effet, elle est un peu particulière puisqu'en fait on lui ajoute une fonction nommée `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.

## 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. 

### *Envoyer des données*

Pour envoyer des données, une fonction toute prête existe pour les objets `SerialPort`. Cette fonction (vous le devinez surement) est : `write()`. En argument il nous faut passer soit une chaine de caractère (`string`) soit un tableau de `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.

**Code : C#**

```
    //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]};
        if(port!=null && port.IsOpen) //on s'assure que le port est
existant et ouvert
            port.Write(car,0,1); //envoie le tableau de caractère,
depuis la position 0, et envoie 1 seul élément
    }
```

### *Recevoir des données*

La dernière étape pour pouvoir gérer de manière complète notre liaison 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 `SerialPort`, nous ajoutons à la propriété `DataReceived` une fonction (en faisant un += ). Faire cela équivaut à dire "*Va à cette fonction lorsque tu reçois des données*". Cette fonction aura ensuite deux choses à faire. Lire le contenu du buffer de réception de la liaison série puis ajouter ces nouvelles données (en théorie un seul caractère) à la boîte de texte `boxReception`.

Dans l'idéal nous aimerions faire de la façon suivante :

**Code : C#**

```
//gestionnaire de la réception de caractère
private void DataReceivedHandler(object sender,
SerialDataReceivedEventArgs e)
{
    String texte = port.ReadExisting();
    boxReception.Text += texte;
}
```

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 :

**Code : C#**

```
//une déléguée pour pouvoir mettre à jour le texte de la boite de
réception de manière "thread-safe"
delegate void SetTextCallback(string text);
```

Le code de la réception devient alors le suivant :

**Code : C#**

```
//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;
        }
```

Je suis désolé si mes informations sont confuses. Je ne suis malheureusement pas un maitre dans l'art des threads UI de C#. Cependant, un tas de documentation mieux expliqué existe sur internet si vous voulez plus de détails.

Une fois tout cela instancié, vous devriez avoir un terminal liaison 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.
Cette annexe vous aura permis de comprendre un peu comment utiliser la liaison 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...

Vous savez tout, ou presque, sur la liaison série. Ce domaine va vous ouvrir des portes vers des possibilités encore plus grande, telle que la création d'interface graphique pour communiquer par l'intermédiaire de votre ordinateur avec Arduino. Vous pourrez également créer un réseau complet pour, par exemple, faire un système domotique ou je ne sais quoi d'autre tout aussi amusant !

N'hésitez pas à faire part de vos projet sur les forums ! 

# Partie 4 : [Pratique] Les grandeurs analogiques

Dans cette partie, je vais introduire la notion de grandeur analogique qui sont opposées au grandeurs logiques. Grâce à ce chapitre, vous serez ensuite capable d'utiliser des capteurs pour interagir avec l'environnement autour de votre carte Arduino (enfin pas tout à fait puisqu'il faudra pour cela lire le chapitre sur les capteurs ).

---> **Matériel nécessaire :** dans la balise secret pour la partie 4.

## 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 !

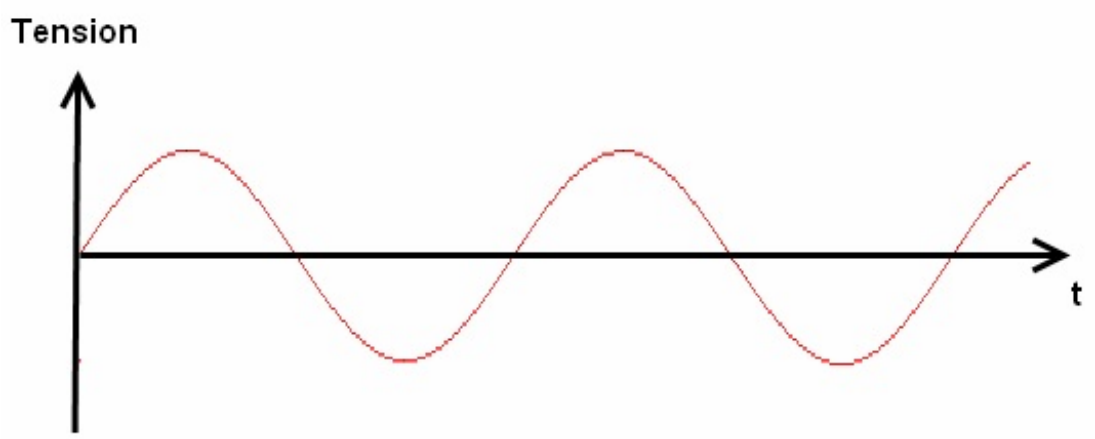
### Un signal analogique : petits rappels

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 :



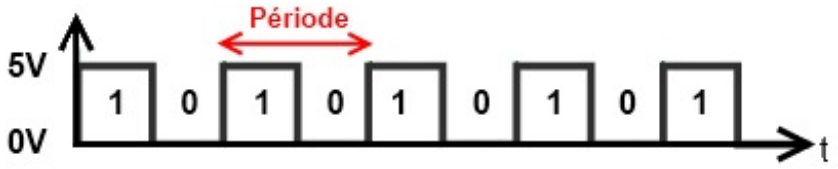

On retiendra que l'on ne s'occupe que de la tension et non du courant, en fonction du temps.

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.

#### *Signal périodique*

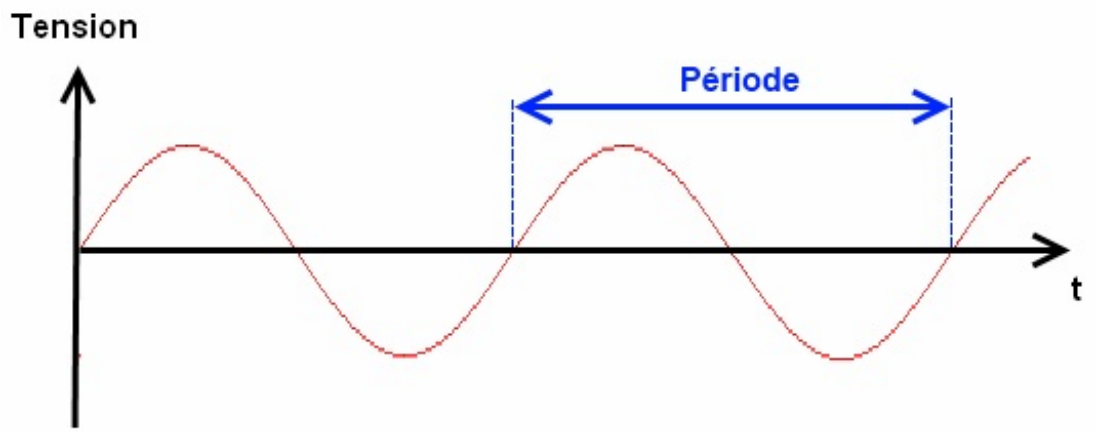
Dans la catégorie des signaux analogiques et même numériques (dans le cas d'horloge de signal pour le cadencement des micro-contrôleurs par exemple) on a les **signaux dits périodiques**.

La période d'un signal est en fait un bout 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, ...




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 1, les possibilités sont infinies !

Pour un signal analogique, il en va de même. Reprenons le signal de tout à l'heure :




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.

### *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...

## Les convertisseurs analogiques -> numérique ou CAN

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.

### *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 cours 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...

## 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).

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 ! 

### *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 connait 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 :

**Code : Console**

```
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.

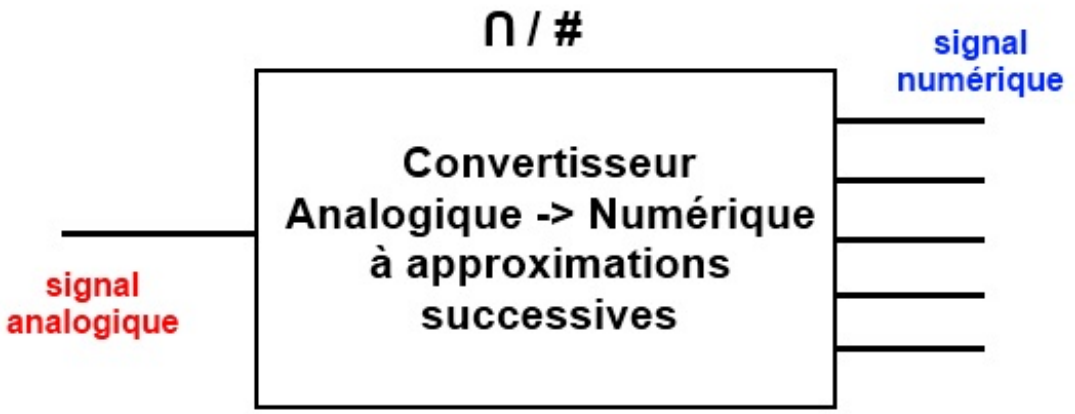
Cet exemple n'est qu'à titre indicatif pour bien comprendre le concept.

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.

## 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'abrégerai par CAN dorénavant).




Notez le symbole du CAN qui se trouve juste au-dessus de l'image. Il s'agit d'un "U" renversé et du caractère #.

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.

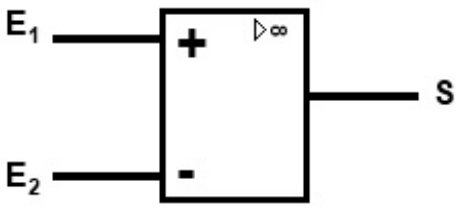
### *Le comparateur*

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.

Comparer, oui, mais quoi ?

Des tensions ! 

Regardez son symbole, je vous explique ensuite...




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 :

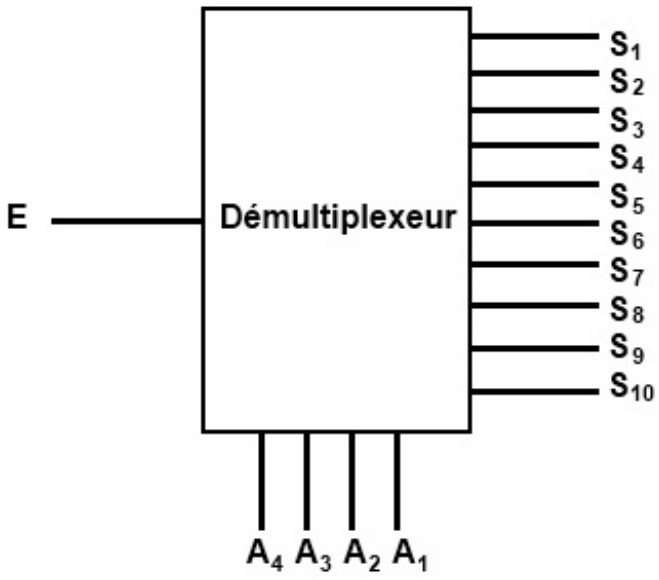
- $E_1 > E_2$ alors $S = 1$
- $E_1 < E_2$ alors $S = 0$
- $E_1 = E_2$, alors $S = indefini$

Simple n'est-ce pas ?

### *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).

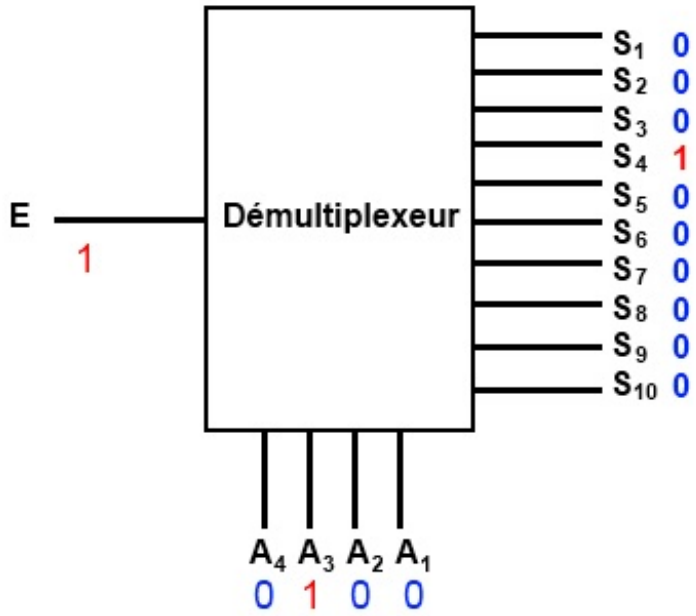
Le principe est là encore très simple. Le démultiplexeur à plusieurs sorties, une entrée et des entrées de sélection :




- $E$ 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 $A_4$




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.

### *La mémoire*

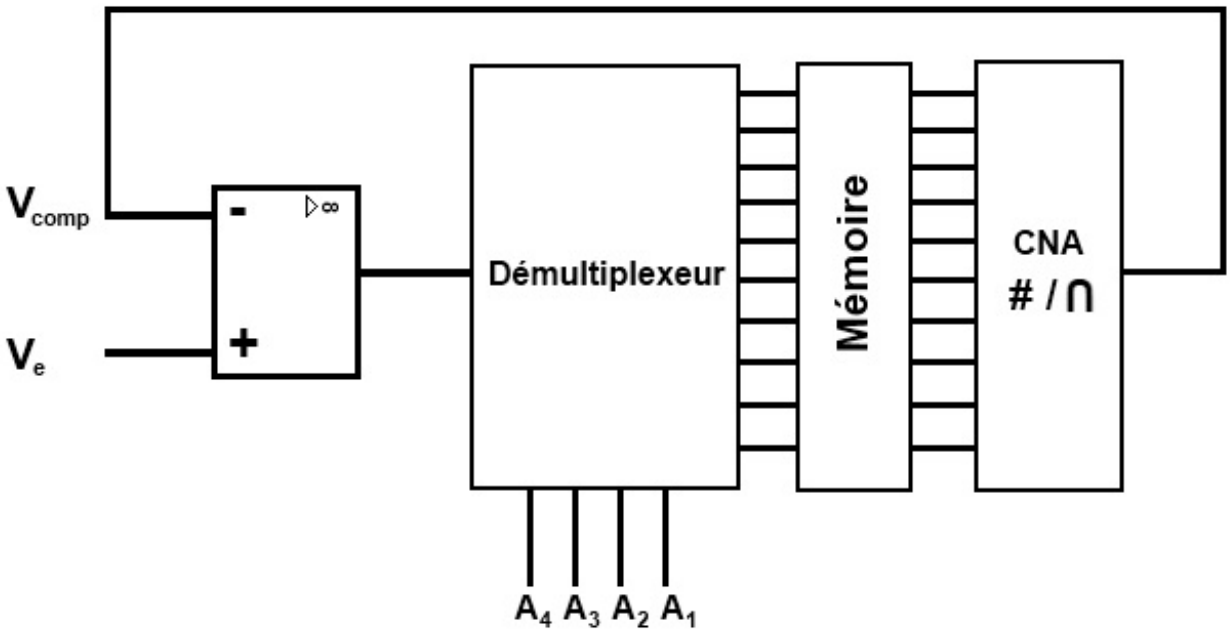
Ce composant électronique sert simplement à stocker des données sous forme binaire.

### *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.

### *Fonctionnement global*

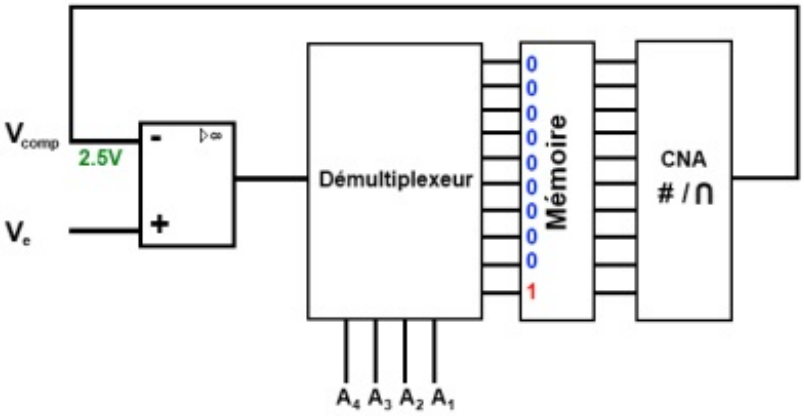
Rentrons dans les explications du fonctionnement d'un CAN à approximations successives. Je vous ai fait un petit schéma rassemblant les éléments précédemment présentés :




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 :

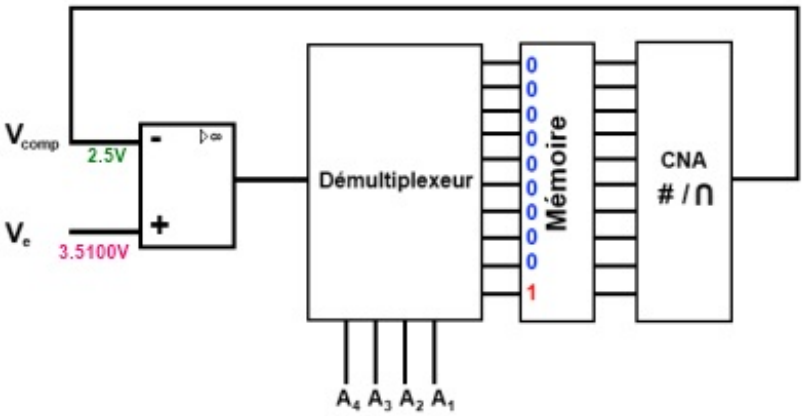
- $V_e$ est la tension analogique d'entrée, celle que l'on veut mesurer en la convertissant en signal numérique.
- **La mémoire** contient pour l'instant que des **0** sauf pour le bit de poids fort ($S_{10}$) qui est à **1**. Ainsi, le convertisseur numérique -> analogique va convertir ce nombre binaire en une tension analogique qui aura pour valeur 2.5V.
- Pour l'instant, le démultiplexeur n'entre pas en jeu.




Suivons le fonctionnement étape par étape :

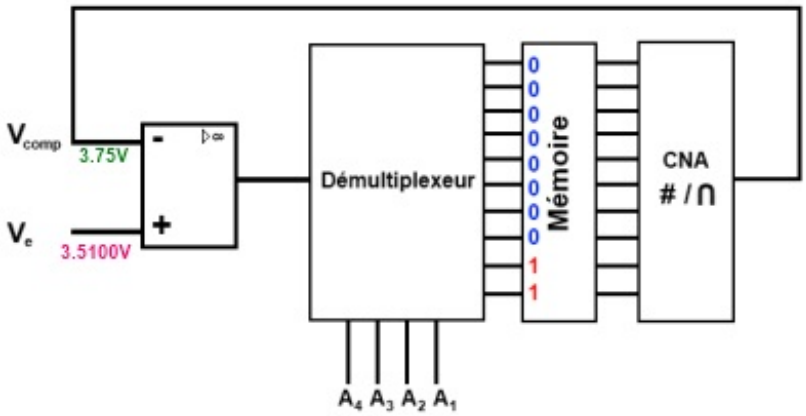
**Étape 1 :**

- 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 jeux. 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**.



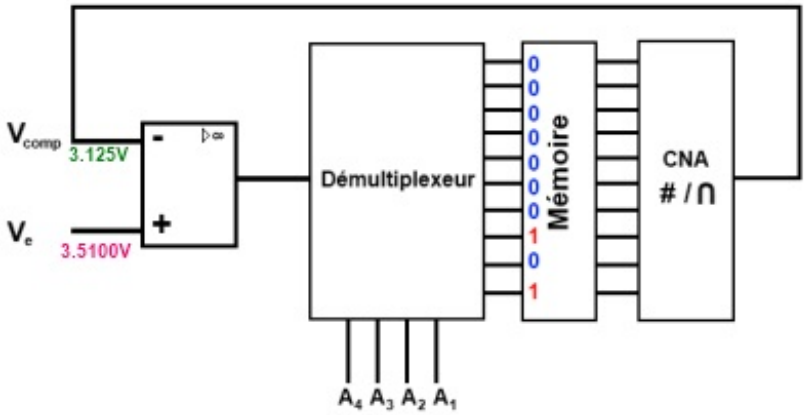
**Étape 2 :**

- 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

- 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 CNA (en V) |
|---|---|---|---|
| 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 |
| 2 | 1 | 1 | 3.505859375 |
| 1 | 0 | 0 | 3.5107421875 |

**Résultat :** Le résultat de la conversion donne :

| Résultat de conversion (binaire) | Résultat de conversion (décimale) | Résultat de conversion (Volts) |
|---|---|---|
| 1011001110 | 718 | 3,505859375 |

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.

### *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.

### *Les inconvénients*

Pour terminer avant de passer à l'utilisation du CNA avec Arduino, je vais vous parler de ses inconvénients. Il en existe deux 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).

## Lecture analogique, on y vient...
## Lire la tension sur une broche analogique

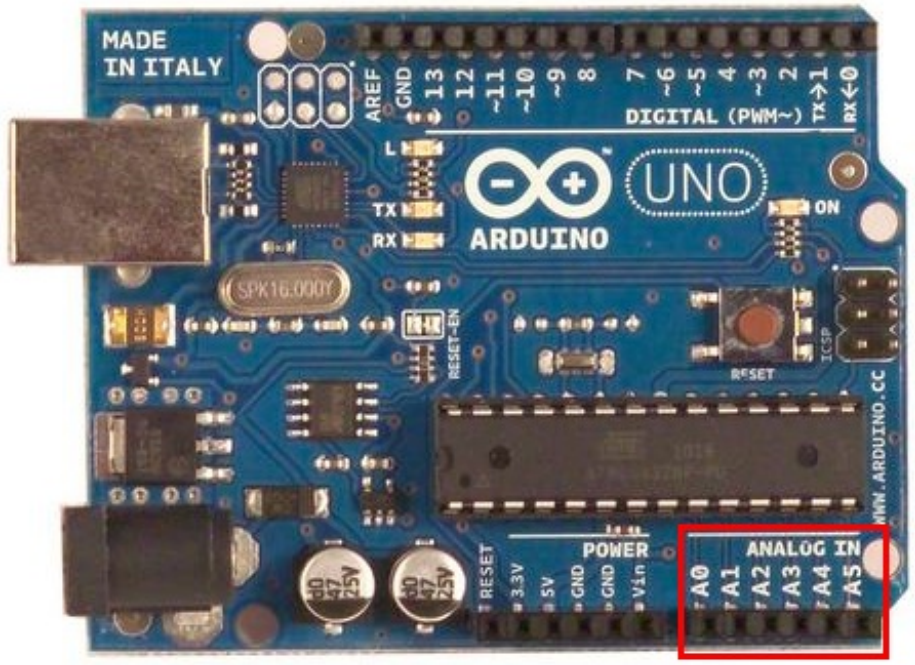
Un truc très sympa avec Arduino, c'est que c'est facile à prendre en main. Et ça se voit une fois de plus avec l'utilisation des convertisseurs numérique -> analogique ! En effet, vous n'avez qu'une seule nouvelle fonction à retenir : `analogRead()` !

### `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);`.

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 :

**Code : C**

```
const int monCapteur = 3; //broche analogique 3 OU broche numérique 3

int valeurLue = 0; //la valeur lue sera comprise entre 0 et 1023

//fonction setup()

void loop()
{
   valeurLue = analogRead(monCapteur); //on mesure la tension du capteur sur la broche analogique 3

   //du code et encore du code
}
```

## 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 ?

### *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 : $\frac{458 \times 5}{1024} = 2.235V$
- En utilisant le pas calculé plus haut on obtient : $458 \times 4.88 = 2.235V$

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 :

**Code : C**

```
int valeurLue; //variable stockant la valeur lue sur le CAN
float tension; //résultat stockant la conversion de valeurLue en
Volts

void loop()
{
   valeurLue = analogRead(uneBrocheAvecUnCapteur);
   tension = valeurLue * 4.88; //produit en croix, ATTENTION, donne
un résultat en mV !
   tension = valeurLue * (5 / 1024); //formule à aspect "physique",
donne un résultat en mV !
}
```

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 :

**Code : C**

```
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
```

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 :

**Code : C**

```
tension = map(valeurLue, 0, 1023, 0, 5000); //conversion de la
valeur lue en tension en mV
```

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 :

**Code : C**

```
tension = map(valeurLue, 0, 1023, 0, 5000); //conversion de la
valeur lue en tension en mV
tension = tension / 1000; //conversion des mV en V
```

Au retour de la liaison série (seulement si on envoie les valeurs par la liaison série) on aurait donc (valeurs à titre d'exemple) :

**Code : Console**

```
valeurLue = 458

tension = 2.290V
```

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".

## Une meilleure précision ?

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.

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.

### Solution 1 : modifier la plage d'entrée du convertisseur

C'est la solution la plus simple ! Voyons deux choses...

### *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 2.5V. Pour améliorer la précision de lecteur, car la tension maximale d'entrée est divisée par deux, on va utiliser la fonction : `analogReference()`.

Pour ce faire, il suffit d'appeler cette fonction comme ceci :

**Code : C**

```
void setup()
{
   analogReference(INTERNAL); //permet de choisir une tension de référence de 2.56V
}
```

La tension de référence interne est de 2.56V lorsque l'on appelle la fonction comme précédemment et de 5V par défaut.

### *Tension de référence externe*

On va utiliser la même fonction, mais comme ceci :

**Code : C**

```
void setup()
{
   analogReference(EXTERNAL); //permet de choisir une tension de référence externe à la carte
}
```

Seulement, il faut mettre la tension de référence sur la broche **AREF** de l'Arduino, toujours comprise entre 0 et 5V !!

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.

Mais, si je veux que ma tension d'entrée varie au-delà de +5V, comment je fais ? Y a-t-il un moyen d'y parvenir ?

Oui, il y en a un, mais il requiert quelques connaissances en électronique. Je ne parlerai donc que de son fonctionnement théorique.

## Solution 2 : présentation théorique d'une solution matérielle (nécessite des composants supplémentaires)

Cette deuxième solution est assez simple à comprendre, mais un peu moins à mettre en œuvre. En tous cas, avec vos connaissances actuelles vous ne pouvez pas utiliser cette solution. À moins, bien sûr, d'avoir quelques connaissances bien

fondées en électronique. C'est pour cela que j'énoncerai seulement le principe, ceux qui voudront utiliser cette solution se débrouilleront avec leurs connaissances.

### *Principe*

Pour arriver à améliorer la précision de conversion du CAN, on va utiliser une "astuce".

Prenons un capteur qui délivre une tension analogique comprise entre 0V et +10V. A cette tension, on va en soustraire une que l'on aura créée, pour "la faire rentrer" dans la plage d'entrée du CAN d'Arduino. Cette tension créée ne l'est pas par hasard et à une valeur déterminée. Comment ? Eh bien, par exemple, je vais soustraire 0.5V à la tension d'entrée du capteur à chaque fois que la tension résultante de cette soustraction est supérieure à 0.5V.

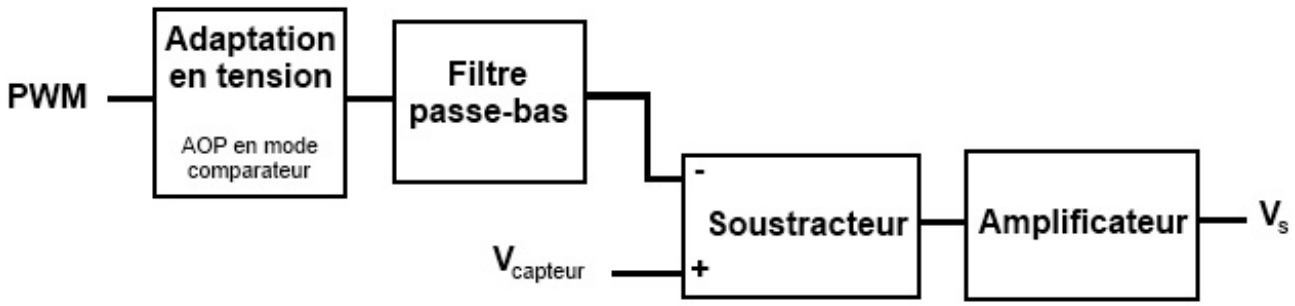
Bon, d'accord, mais ça veut dire que la tension lue sera toujours comprise entre 0V et 0.5V, alors quel est l'intérêt puisque, du coup, on perd énormément en précision ! Et même si on descend la tension de référence du CAN à 0.5V, on aura la même précision qu'au départ ! Je comprends pas !!

C'est là qu'est toute l'astuce, après avoir soustrait la tension, on va l'amplifier ! Et cette amplification sera d'un facteur 10. Comme cela, on retrouve bien nos 5V à l'entrée du CAN de l'Arduino. Et de cette manière, on aura gagné en précision et d'un facteur 10 de surcroit !

Je pense que je vais vous faire un petit schéma avec un bon exemple et quelques calculs théoriques pour que vous puissiez mieux assimiler mes explications.

### *Un schéma, un exemple...*

Pour cette solution je vais aller un peu vite car il s'agit d'une technique avancée qui demande un certain niveau en électronique et que vous n'avez pas en ayant pour seules connaissances en le domaine que la lecture de ce cours. Elle est donc destinée aux plus téméraires d'entre vous.




- Le fonctionnement est très simple, on créer une PWM qui passe dans un filtre passe-bas afin de créer un palier de tension.
- Cette tension alors créée va être soustraite à la tension en sortie du capteur que l'on récupère.
- Enfin, on amplifie la tension résultante.

**Prenons l'exemple suivant :**

- Le capteur fournit une tension de 0.856V, l'amplification du montage est de 10 fois. Chaque palier de tension créé à partir de la PWM correspond à un niveau de tension approximativement égal à 0.5V
- En sortie du soustracteur on a donc $V_{capteur} - V_{palier}$, soit $0.856 - 0.5 = 0.356V$
- Enfin, en sortie de l'amplificateur on a donc une tension de $0.356 \times 10 = 3.56V$

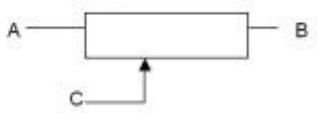
Cette dernière valeur est bien comprise entre 0V et 5V, exactement comme on le souhaitait pour que l'on puisse convertir cette valeur grâce au CAN de l'Arduino. De ce fait, on a augmenté la précision d'un facteur 10, le CAN de l'Arduino sera donc capable de "voir" des tensions 10 fois plus faibles sur un seul bit, soit : $0.000488V = 488\mu V$

## Exemple d'utilisation

## Le potentiomètre

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 :

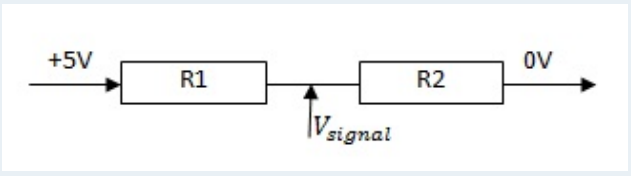
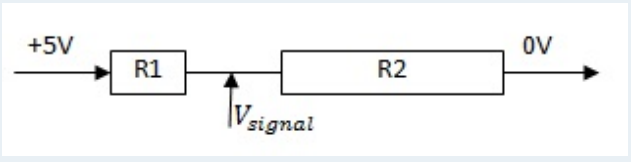
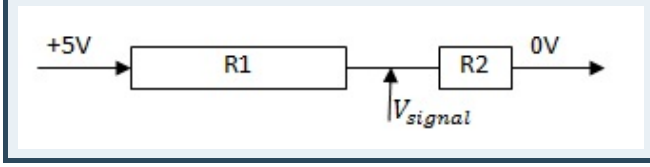



### *Cas n°1 : le pont diviseur de tension*

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 la broche C |
|---|---|---|
|   | Curseur à **la moitié** | $V_{signal} = (1 - \frac{50}{100}) \times 5 = 2.5V$ |
|   | Curseur à **25% du départ** | $V_{signal} = (1 - \frac{25}{100}) \times 5 = 3.75V$ |
|   | Curseur à **75% du départ** | $V_{signal} = (1 - \frac{75}{100}) \times 5 = 1.25V$ |

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.

### *Cas n°2 : la résistance variable*

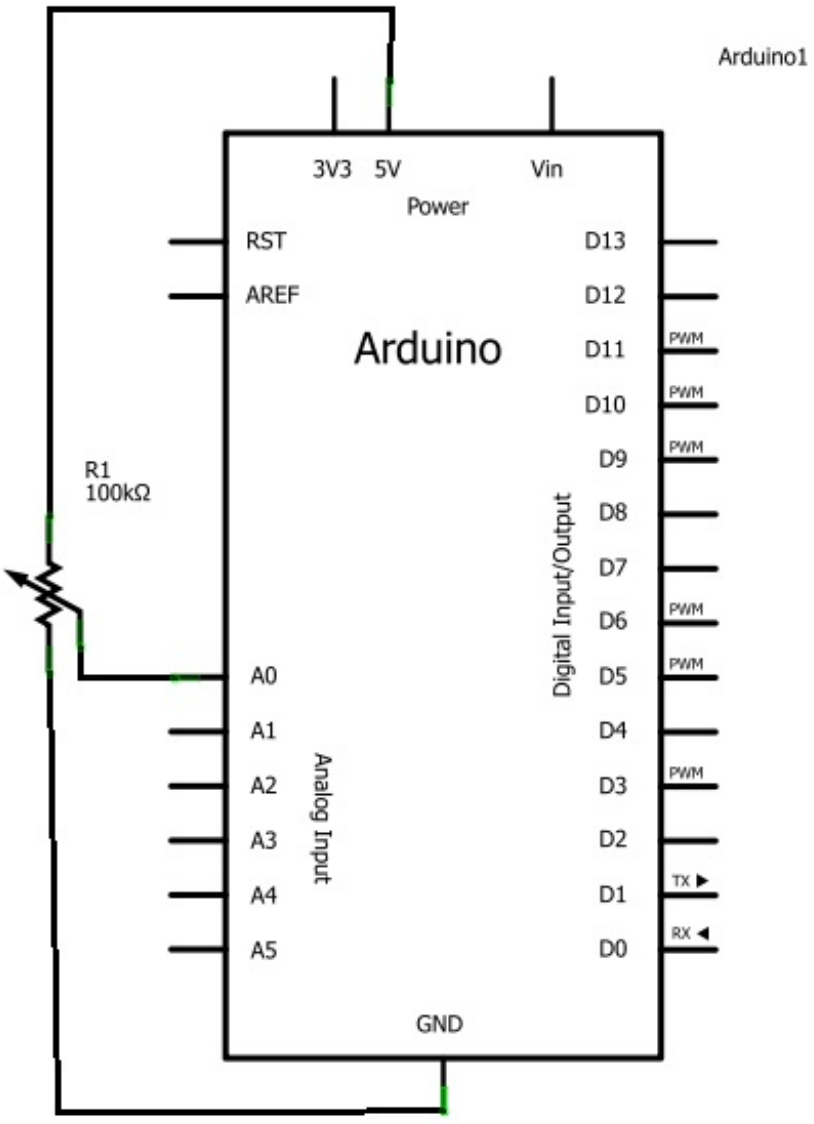
Le deuxième cas d'utilisation du potentiomètre est la **résistance variable**. Cette configuration est très simple, il suffit d'utiliser le potentiomètre comme une simple résistance dont les bornes sont A et C ou B et C. On pourra alors faire varier la valeur ohmique de la résistance grâce à l'axe du potentiomètre.

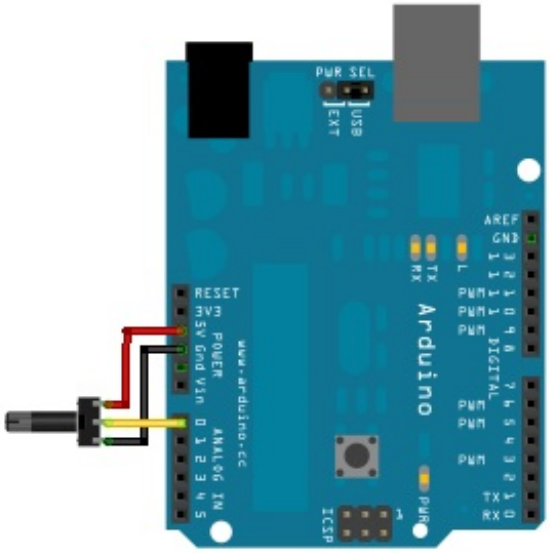
Attention, il existe des potentiomètres **linéaires** (la valeur de la tension évolue de manière proportionnelle au

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.

## 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 !

...

Voici la correction, c'est le programme que j'ai fait, peut-être que le vôtre sera mieux :

**Code : C**

```
const int potar = 0; // le potentiomètre, branché sur la broche
analogique 0
int valeurLue; //variable pour stocker la valeur lue après
conversion
float tension; //on convertit cette valeur en une 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);

 //on traduit la valeur brute en tension (produit en croix)
 tension = valeurLue * 5.0 / 1024;

       //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");

 Serial.println(); //on saute une ligne entre deux affichages
 delay(500); //on attend une demi-seconde pour que l'affichage ne
soit pas trop rapide
}
```

Vous venez de créer votre premier Voltmètre ! 
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.

# [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 !

## Consigne

### *Vu-mètre, ça vous parle ?*

Dans ce TP, nous allons réaliser un **vu-mètre**. Même si le nom ne vous dit rien, je suis sur que vous en avez déjà rencontré. Par exemple, sur une chaîne hi-fi ou sur une table de mixage on voit souvent des loupiotes 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).

### *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 !

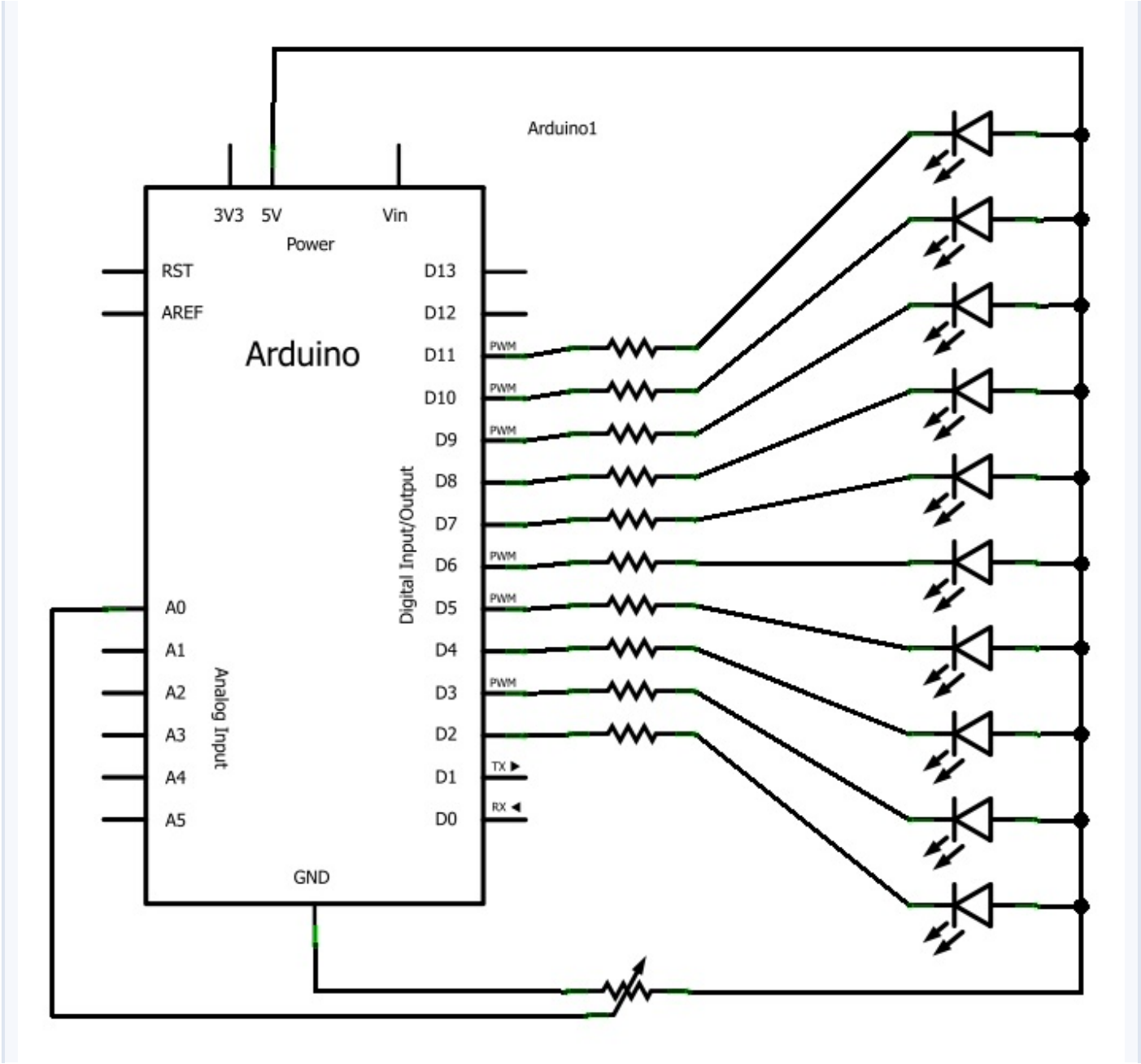
## Correction !

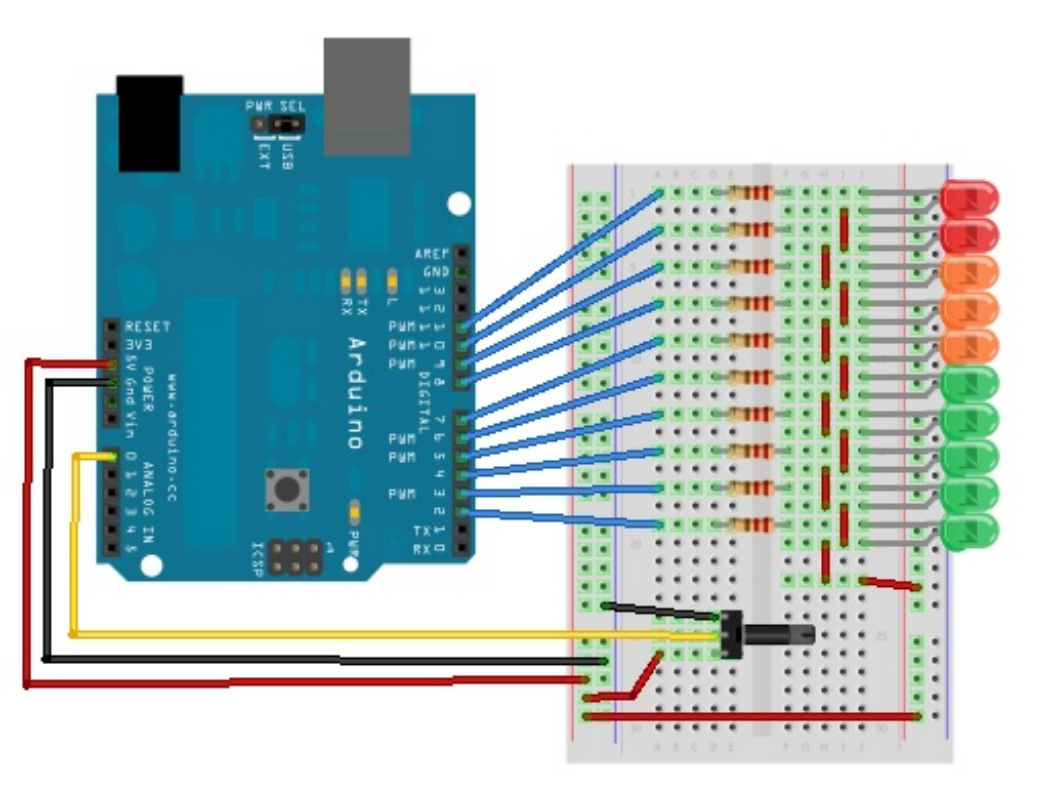
J'espère que tout c'est bien passé pour vous et que l'affichage cartonne ! Voici maintenant venu l'heure de la correction, en espérant que vous n'en aurez pas besoin et que vous la consulterez juste pour votre culture. Comme d'habitude nous allons commencer par voir le schéma puis ensuite nous étudierons le code.

## 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 :

**Secret** (cliquez pour afficher)


Arduino1
3V3
5V
Vin
Power
RST
AREF
Arduino
D13
D12
D11
D10
D9
D8
D7
D6
D5
D4
D3
D2
D1
D0
PWM
TX
RX
Digital Input/Output
A0
A1
A2
A3
A4
A5
Analog Input
GND




## 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.

### *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 !

**Code : C**

```
// Déclaration et remplissage du tableau...
// ...représentant les broches des LEDs
const int leds[10] = {2,3,4,5,6,7,8,9,10,11};
const int potar = 0; //le potentiomètre sera branché sur la broche analogique 0
int tension; //variable stockant la tension mesurée
```

Une fois que l'on à 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).

**Code : C**

```
void setup()
{
 int i = 0;
 for(i=0; i<10; i++)
 {
  pinMode(leds[i], OUTPUT); //déclaration de la broche en sortie
  digitalWrite(leds[i], HIGH); //mise à l'état haut
 }
}
```

### *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 :

**Code : C**

```
void loop()
{
    tension = analogRead(potar); //on récupère la valeur de la
tension du potentiomètre
    afficher(tension); //et on affiche sur les LEDs cette tension
}
```

Encore plus fort, la même écriture mais en une seule ligne !

**Code : C**

```
void loop()
{
    afficher(analogRead(potar)); //la même chose qu'avant même en
une seule ligne !
}
```

### *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 mesuré. Par exemple, si la tension mesuré 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 multiplier par 100 (ce qui représentera nos différents pas). Si le test vaut VRAI, on allume la LED i, sinon on l'éteint.
Démonstration :

**Code : C**

```
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
 }
}
```

## 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.

**Secret** (cliquez pour afficher)

**Code : C**

```
//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;
//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 relation 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 l'affichage avec les vieilles fonctions
 tension = tension*10;

 //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(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_unite, HIGH);
  afficher(unite);
 }
}

//fonction écriveant 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...

# 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...

## 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.

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 entré analogique ).

Tiens, en voilà un projet intéressant ! Je vais me le garder sous la main...

Alors ! alors ! alors !! Comment on fait !?

Serait-ce un léger soupçon de curiosité que je perçois dans vos yeux frétillants ?

Comment fait-on ? Suivez-le guide !

## Convertisseur Numérique->Analogique

La première méthode consiste en l'utilisation d'un convertisseur Numérique->Analogique (que je vais abréger 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 !

## 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->N ? Mais concrètement, c'est quoi ?

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.

### *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).

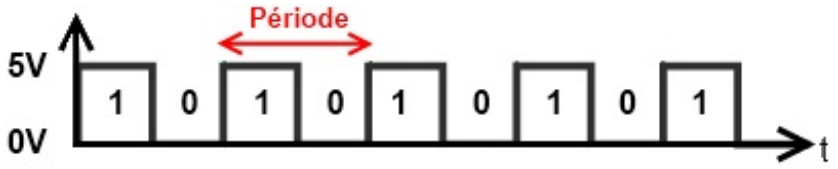
La PWM est en fait un signal numérique qui, à une **fréquence** donnée, a un **rapport cyclique** qui change.

Y'a plein de mots que je comprends pas, c'est normal ?

Oui, car pour l'instant je n'en ai nullement parlé. Voilà donc notre prochain objectif.

### *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 :

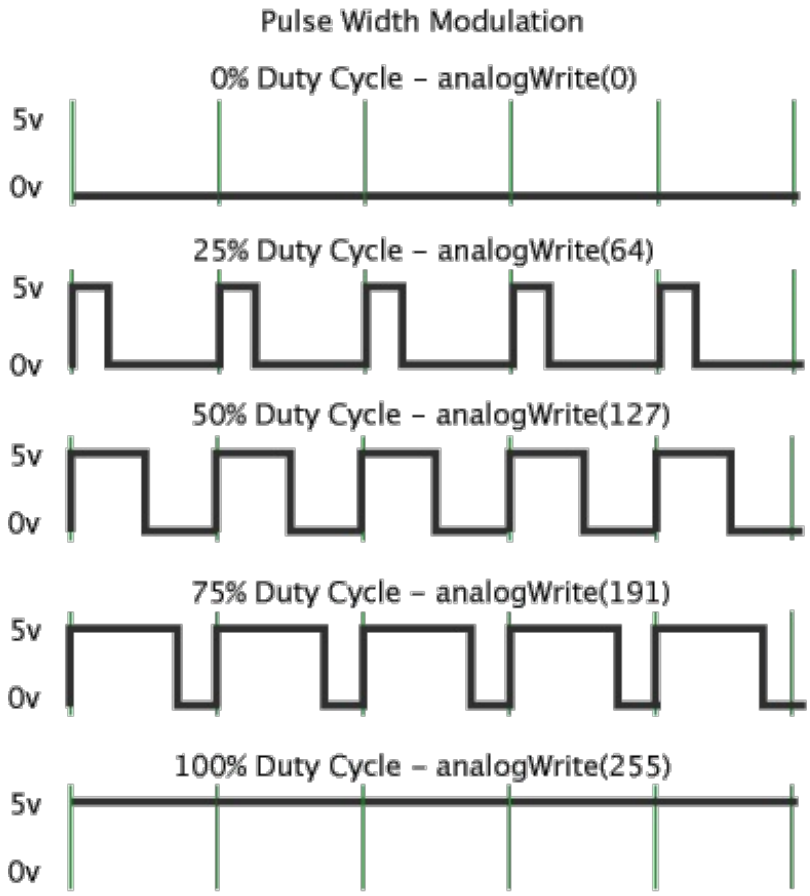
$$F = \frac{1}{T}$$

Avec :

- $F$ : 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 pour cent (%). 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 :




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.

## La PWM de l'Arduino

## Avant de commencer à programmer

### *Les broches de la PWM*

Sur votre carte Arduino, vous devriez disposer de 6 broches qui soient 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.

### *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.

### *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 :

**Citation : Moi**

> la PWM est un signal de fréquence fixe qui a un rapport cyclique qui varie avec le temps suivant "les ordres qu'elle reçoit"

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.

### *Utilisation*

Voilà un petit exemple de code illustrant tout ça :

**Code : C**

```
const int sortieAnalogique = 6; //une sortie analogique sur la broche 6

void setup()
{
    pinMode(sortieAnalogique, OUTPUT);
}

void loop()
{
    analogWrite(sortieAnalogique, 107); //on met un rapport cyclique de 107/255 = 42 %
}
```

## 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.

### *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.

### *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.

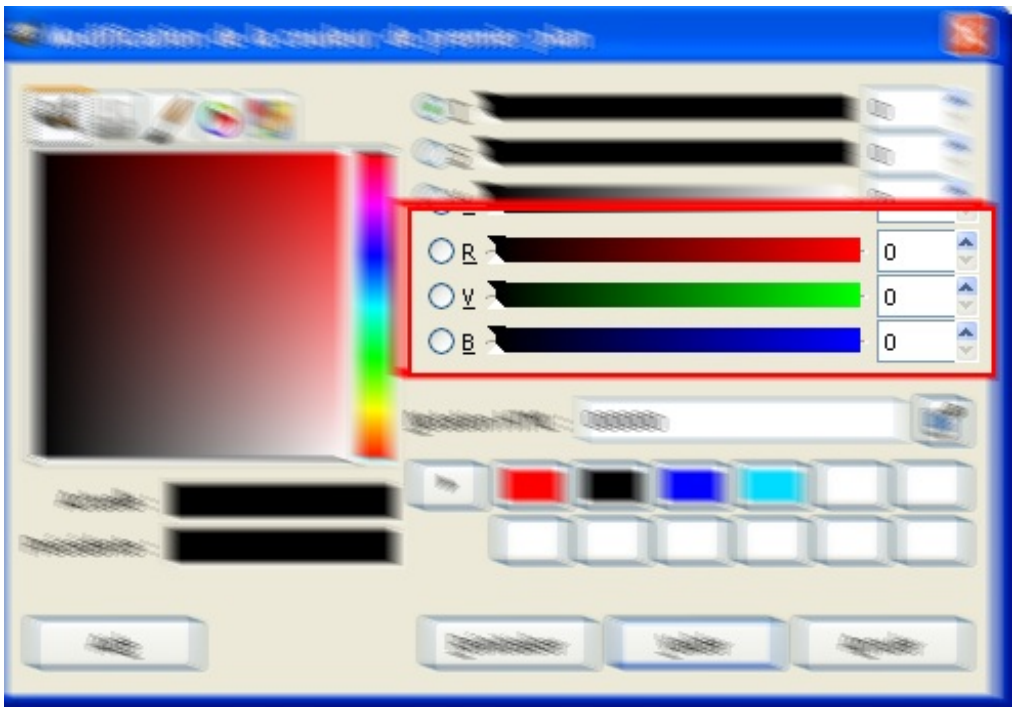
Par exemple, pour faire de l'orange on va mélanger du rouge (2/3 du volume final) et du vert (à 1/3 du volume final).

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 :

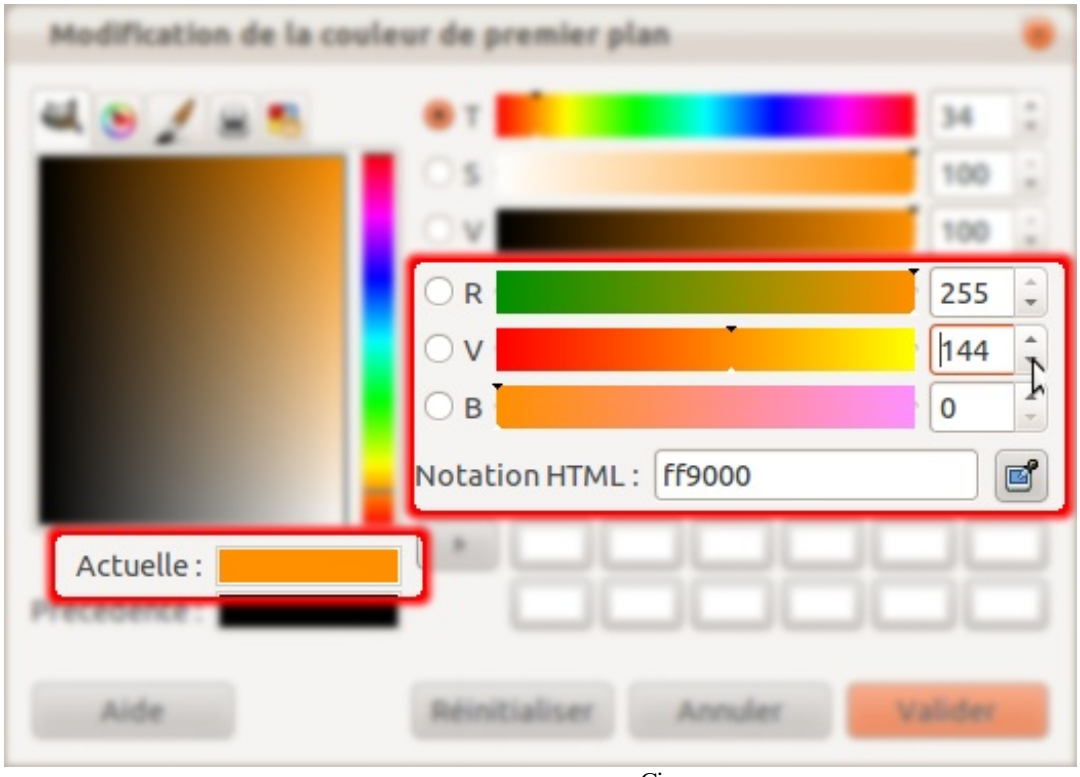
1. Tout d'abord on sélectionne la "boîte à couleurs" dans la boîte à outils
2. Ensuite, en jouant sur les valeurs R, G et B on peut voir la couleur obtenue

gimp toolbox -->




gimp rgb

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 :




La couleur orange avec 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 :

**Code : C**

```
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);
 analogWrite(ledBleue, 0);
}

void loop()
{
//on ne change pas la couleur donc rien à faire dans la boucle
principale
}
```

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 :

**Code : C**

```
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, 255-255);
 analogWrite(ledVerte, 255-144);
 analogWrite(ledBleue, 255-0);
}
```

On en a fini avec les rappels, on va pouvoir commencer un petit exercice.

## À vos claviers, prêt... programmez !

### *L'objectif*

L'objectif est assez simple, vous allez générer trois PWM différentes (une pour chaque LED de couleur) et créer 7 couleurs (le

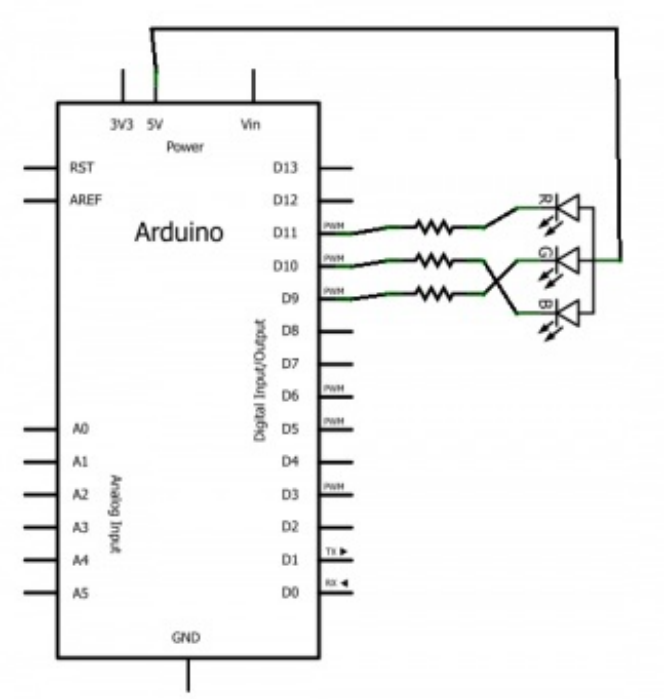
noir ne compte pas ! ) distinctes qui sont les suivantes :

- rouge
- vert
- bleu
- jaune
- bleu ciel
- violet
- blanc

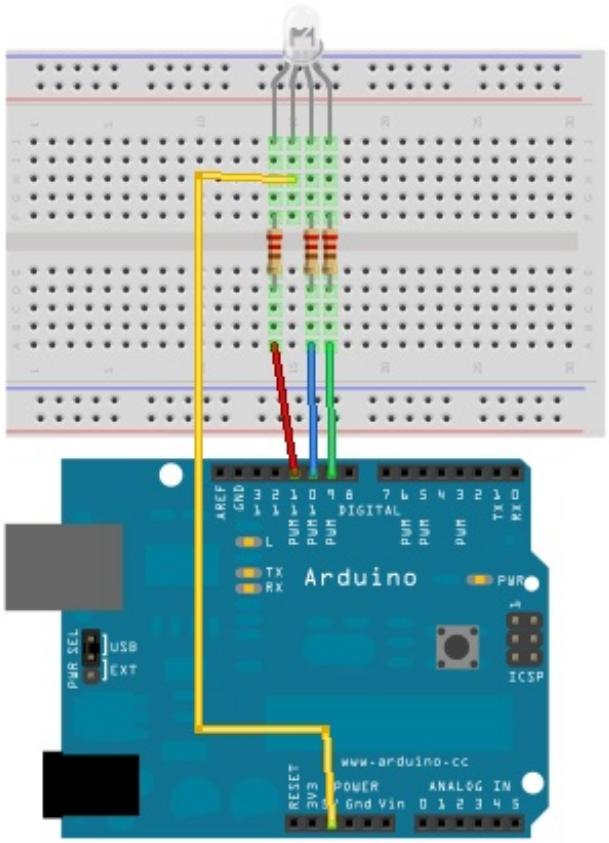
Ces couleurs devront "défiler" une par une (dans l'ordre que vous voudrez) toutes les 500ms.

### *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.




RGB schema




RGB montage

## C'est parti !

### *Correction*

Voilà le petit programme que j'ai fait pour répondre à l'objectif demandé :

**Code : C++**

```
//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()
{
   //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
   if(compteur_defilement > 6) compteur_defilement = 0; //si le
compteur dépasse 6 couleurs
```

```
    delay(500);
}

void couleur(int numeroCouleur)
{
   switch(numeroCouleur)
   {
      case 0 : //rouge
         analogWrite(led_rouge, 0); //rapport cyclique au minimum
pour une meilleure luminosité de la LED
                                    //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;
   }
}
```

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.

## 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.

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.

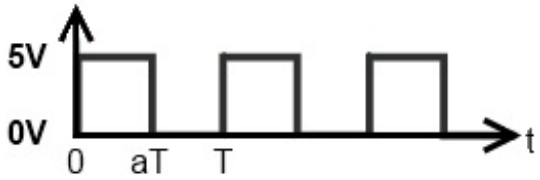
Cependant, avant de continuer, je tiens à vous informer que l'on va aborder des notions plus profondes en électronique et que vous n'êtes pas obligé de lire cette sous-partie si vous ne vous en sentez pas capable. Revenez plus tard si vous le voulez.
Pour ceux qui cela intéresserait vraiment, je ne peux que vous encourager à vous accrocher et éventuellement lire ce chapitre pour mieux comprendre certains points essentiels utilisés dans cette sous-partie.

## 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.

### *Le signal carré*

Reprenons notre signal carré :




J'ai modifié un peu l'image pour vous faire apparaitre 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 :

$$< V_{moyenne} >= \frac{U_1 \times aT + U_2 \times (T - aT)}{T}$$

La valeur moyenne d'un signal se note avec des chevrons <, > autour de la lettre indiquant de quelle grandeur physique il s'agit.

### *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 : $< V_{moyenne} >= a \times U_1 + U_2 - a \times U_2$

Et cela se simplifie encore en :

$$< V_{moyenne} >= a \times (U_1 - U_2) + U_2$$</math>

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 : $< V_{moyenne} >= a \times U_1$, car $U_2 = 0$

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 maitrisée, 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.

## Extraire cette valeur moyenne

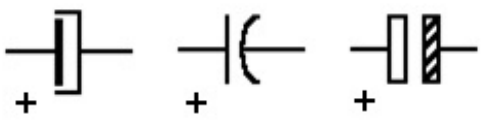
Alors, mais comment faire pour extraire la valeur moyenne du signal de la PWM, me direz-vous. Eh bien on va utiliser les propriétés d'un certain couple de composants très connu : le **couple RC** ou **résistance-condensateur**.

La résistance on connait, mais, le condensateur... tu nous avais pas dit qu'il servait à supprimer les parasites ?

Si, bien sûr, mais il possède plein de caractéristiques intéressantes. C'est pour cela que c'est un des composants les plus utilisé en électronique. Cette fois, je vais vous montrer une de ses caractéristiques qui va nous permettre d'extraire cette fameuse valeur moyenne.

### *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 !

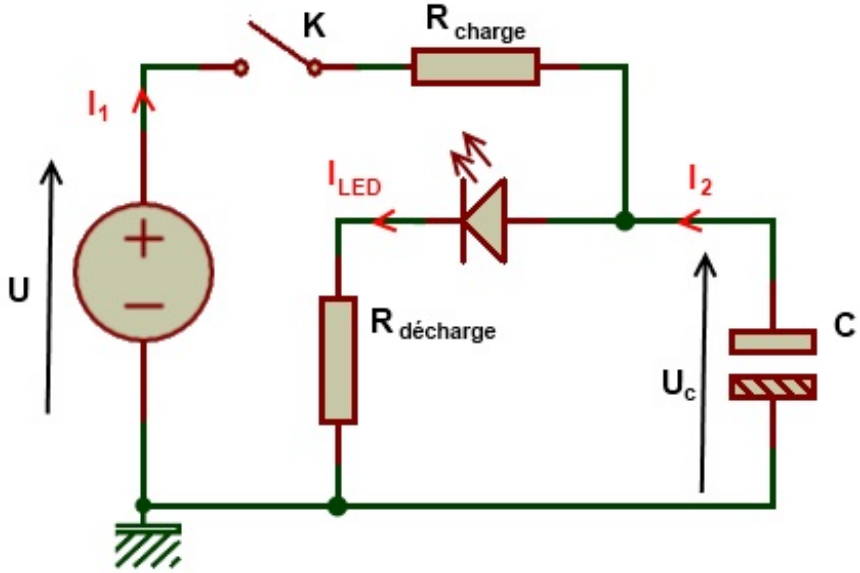
Le condensateur, je rappel ses symboles :  est constitué de deux plaques métalliques, des **armatures**, posées face à face et isolées par... un isolant ! Donc, en somme le condensateur est équivalent à un interrupteur ouvert puisqu'il n'y a pas de courant qui peut passer entre les deux armatures.

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.

### *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...

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 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 ?

Vous pouvez réitérer l'expérience en changeant la valeur des composants, sans jamais descendre en dessous de 220 Ohm pour la résistance de décharge.

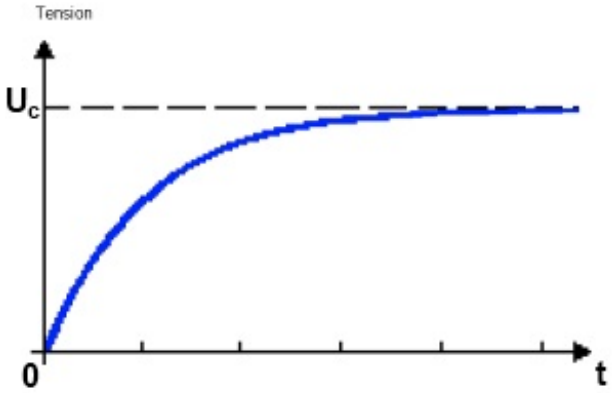
### *Explications*

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.

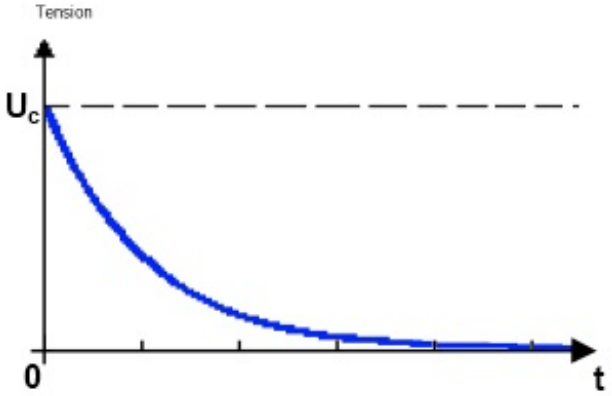
Du coup, lorsqu'on lui applique une tension de 5V sur une des ses armatures et l'autre armature est reliée à la masse, il met du

temps à accepter la tension. Et plus la tension croit, 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 de façon exponentielle 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 :




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 :

- $\tau$ : (prononcez "to") temps de charge/décharge en secondes (s)
- $R$ : valeur de la résistance en Ohm ($\Omega$)
- $C$ : valeur de la capacité du condensateur en Farad (F)

Cette formule donne le temps $\tau$ qui correspond à 63% de la charge à la tension appliquée au condensateur. On considère que le condensateur est complètement chargé à partir de $3\tau$ (soit 95% de la tension de charge) ou $5\tau$ (99% de la tension de charge).

### *Imposons notre PWM !*

Bon, très bien, mais quel est le rapport avec la PWM ?

Ha, haa !

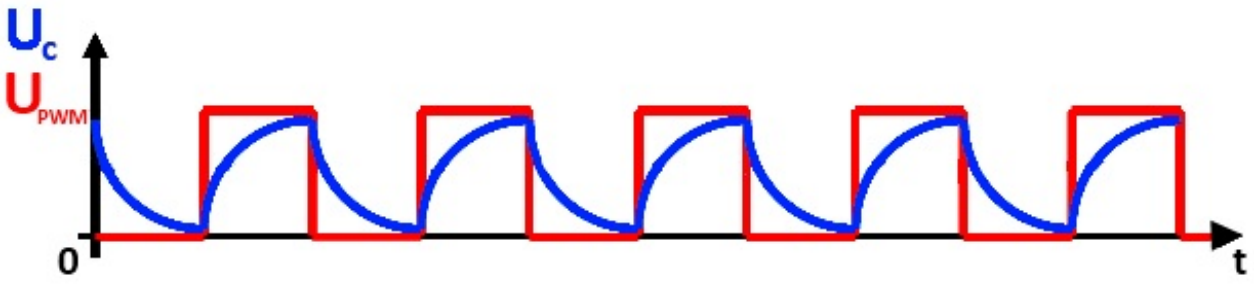
Alors, pour commencer, vous connaissez la réponse.

Depuis quand ?

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, 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 :



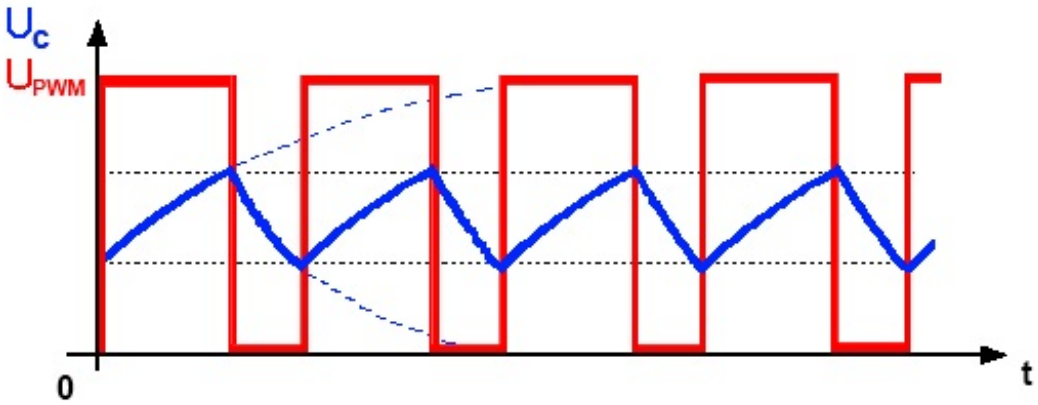

Qu'y a-t-il de nouveau par rapport au signal carré, à part sa forme bizarroïde !?

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 ).

Précisons que dans ce cas, encore une fois, le temps de charge/décharge $3\tau$ 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 ?

### *Constante de temps $\tau$ supérieure à la période*

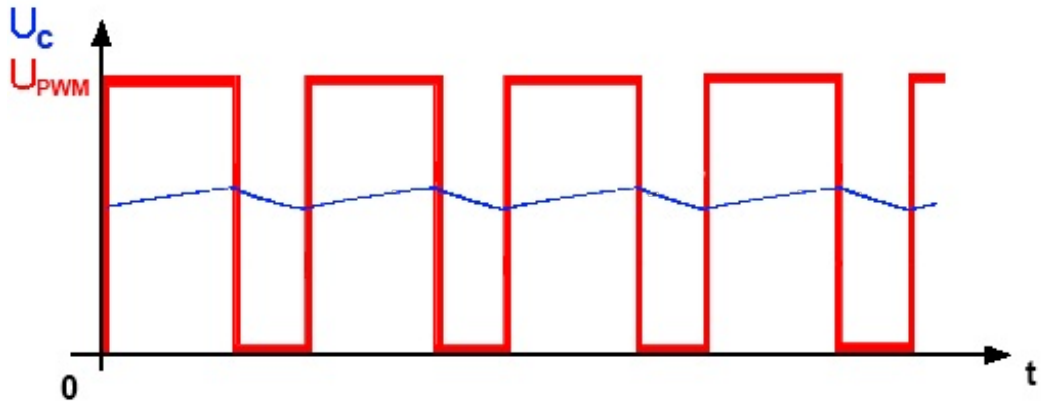
Voilà le chronogramme lorsque la constante de temps de charge/décharge du condensateur est plus grande que la période du signal :




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 :




Et ce signal, Mesdames et Messieurs, c'est la valeur moyenne du signal de la PWM !!

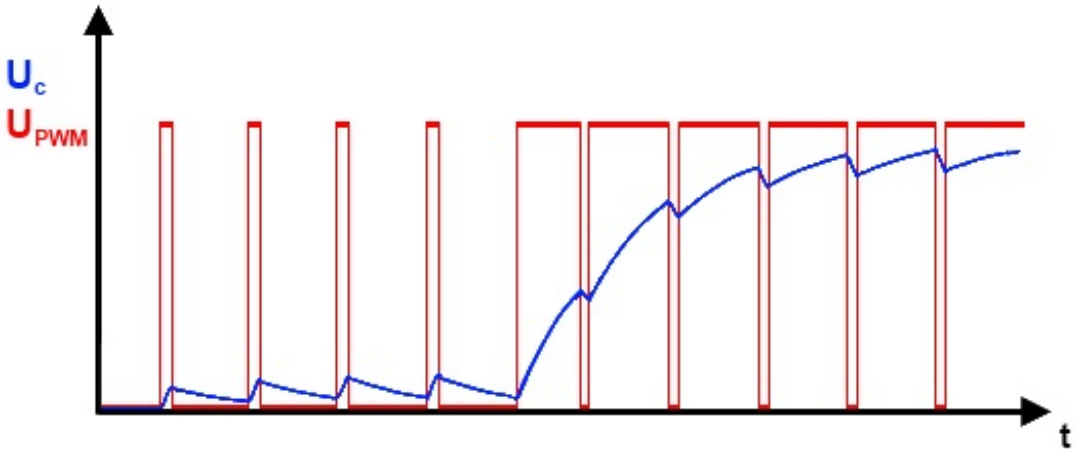
## Calibrer correctement la constante RC

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*".

### *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" :




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.

### *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.

### *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

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 ! 

# [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 !

## 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 !

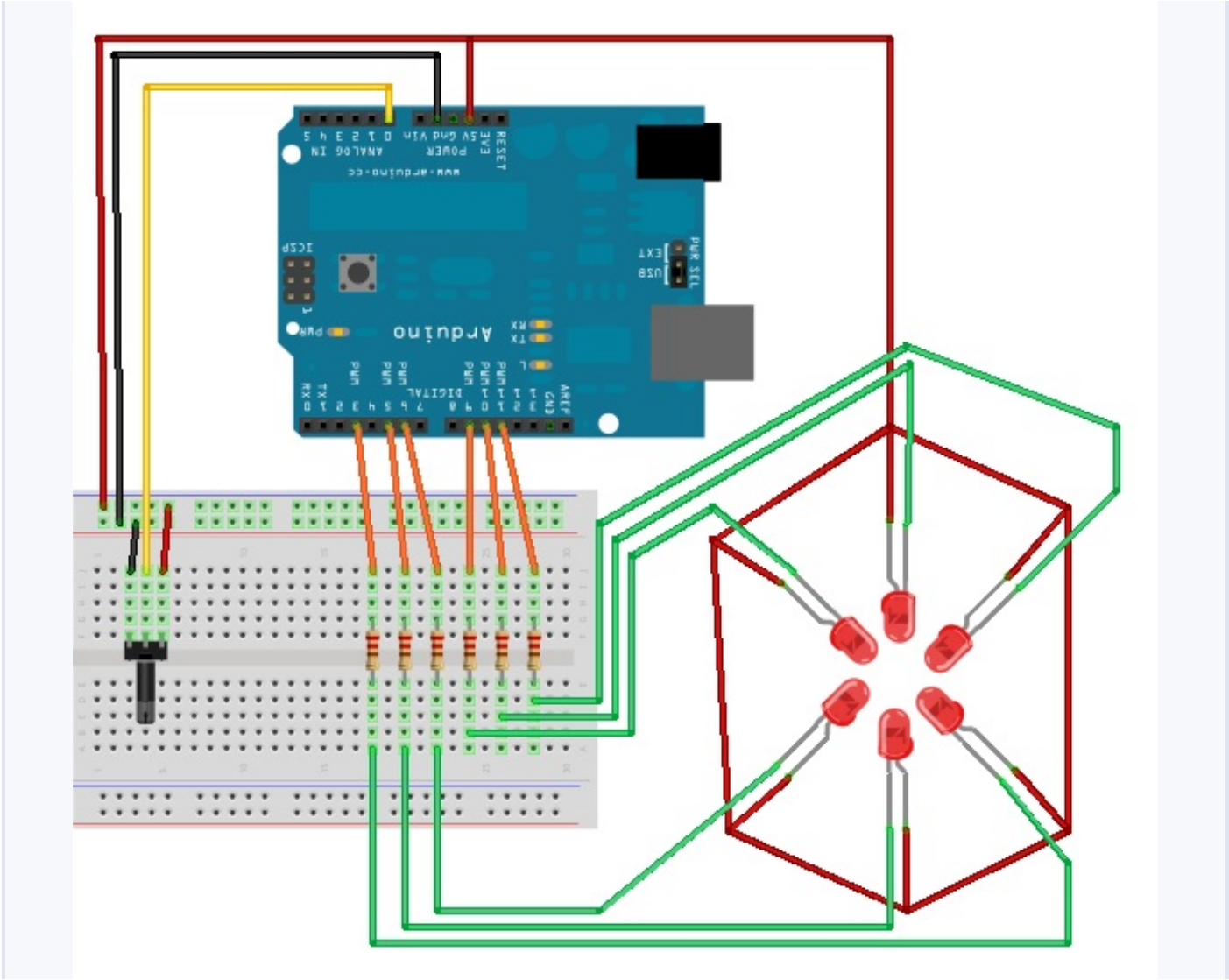
## Solution

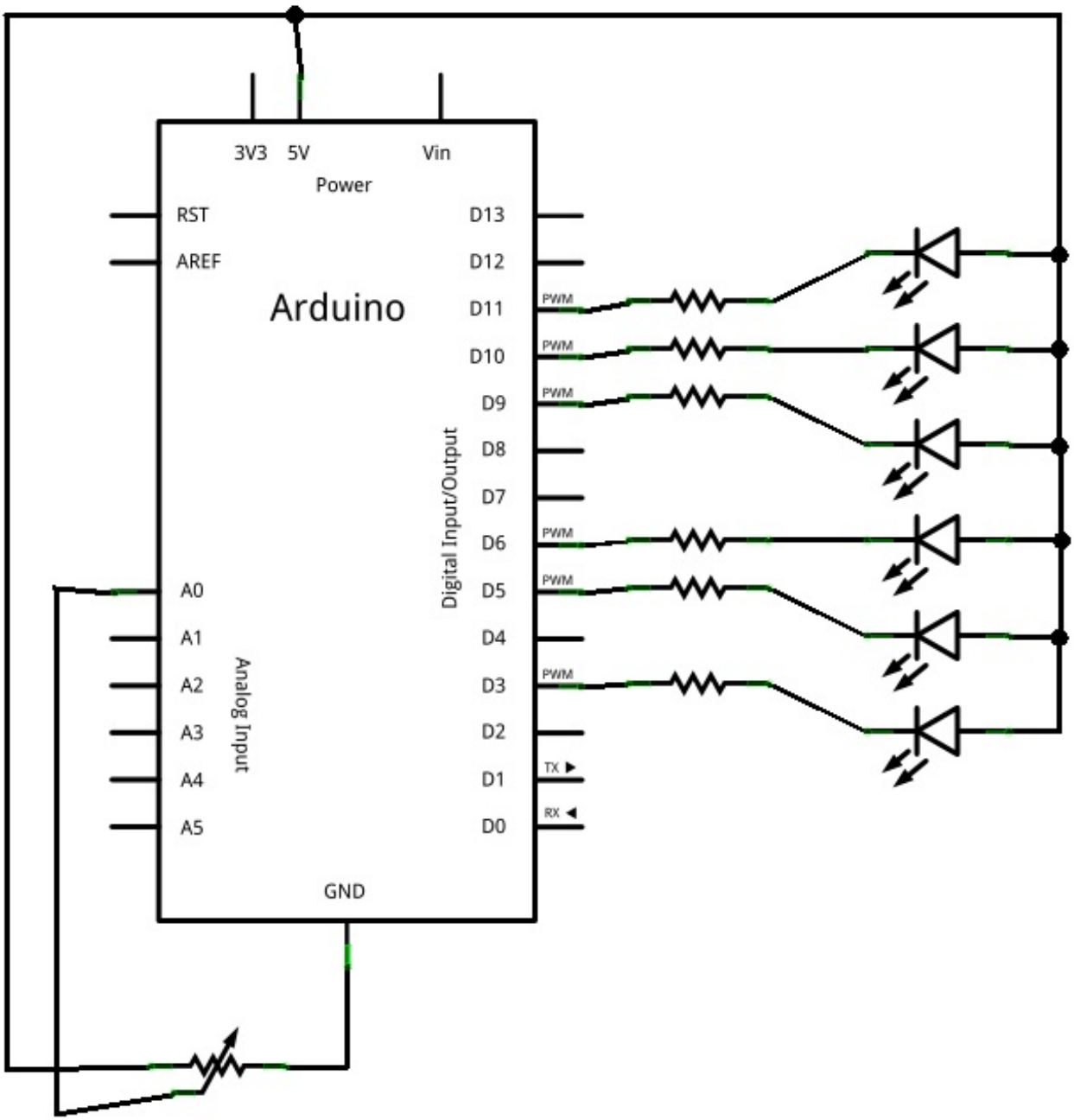
### Le schéma

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).

**Secret** (cliquez pour afficher)






## 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...

### *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 :

**Secret** (cliquez pour afficher)

**Code : C**

```
const int LED[6] = {3,5,6,9,10,11}; //sortie LEDs
const char pwm[6] = {255,210,160,200,220,240}; //niveaux de
luminosité utilisé
const int potar = 0; //potentiometre sur la broche 0
```

### *Le setup*

Personne ne devrais 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 ).

**Secret** (cliquez pour afficher)

**Code : C**

```
void setup()
{
 pinMode(potar, INPUT); //le potentiomètre en entrée
 //les LEDs en sorties
 for(int i=0; i<6; i++)
  pinMode(LED[i], OUTPUT);
}
```

### *La loop*

Passons au cœur du programme, la boucle `loop()` ! Je vais vous la divulguer dès maintenant puis l'expliquer ensuite :

**Secret** (cliquez pour afficher)

**Code : C**

```
void loop()
{
 for(int i=0; i<6; i++) //étape de l'animation
 {
  for(int n=0; n<6; n++) //mise à jour des LEDs
  {
   analogWrite(LED[n], pwm[(n+i)%6]);
  }
  int temps = analogRead(potar); //récupère le temps
  delay(temps/6 + 20); //tmax = 190ms, tmin = 20ms
 }
}
```

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.

### *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 (puisqu'on passerait directement à l'étape suivante sans attendre), j'ajoute 20 à ce résultat. Le temps final sera donc compris dans l'intervalle : $20ms \leq temps \leq 190ms$.

### *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 5 (nombre maximale de l'index pour un tableau de 6 cases). En utilisant le modulo, nous prenons le résultat de la division de 8/5 soit 3. Il nous faudra donc utiliser la case numéro 3 du tableau pwm[] pour cette utilisation. *Tout simplement* 

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...

### *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).

**Secret** (cliquez pour afficher)

**Code : C**

```
const int LED[6] = {3,5,6,9,10,11}; //sortie LEDs
const char pwm[6] = {255,210,160,200,220,240}; //niveaux de
luminosité utilisé
const int potar = 0; //potentiometre sur la broche 0

void setup()
{
 pinMode(potar, INPUT);
 for(int i=0; i<6; i++)
  pinMode(LED[i], OUTPUT);
}
```

```
void loop()
{
 for(int i=0; i<6; i++) //étape de l'animation
 {
  for(int n=0; n<6; n++) //mise à jour des LEDs
  {
   analogWrite(LED[n], pwm[(n+i)%6]);
  }
  int temps = analogRead(potar);
  delay(temps/6 + 20); //tmax = 190ms, tmin = 20ms
 }
}
```

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... 

Vous venez de terminer une des parties essentiels, alors je vous fait savoir que dorénavant, vous pouvez parcourir la suite du cours dans l'ordre que vous voulez !

Si vous avez envie d'en apprendre plus sur la communication entre votre ordinateur et votre carte Arduino, alors allez jeter un coup d'œil à la partie traitant du logiciel Processing.

Si en revanche votre but est de créer un robot, consultez les deux prochaines parties.

Vous voulez afficher du texte sur un petit écran LCD, alors dirigez-vous vers la partie traitant de ce sujet.

Bon voyage ! 

# Partie 5 : [Pratique] L'affichage

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 d'une certaine catégorie pour pouvoir réaliser vos projet les plus fous.

Il est courant d'utiliser ces écrans permettant l'affichage du texte en domotique, robotique, voir même pour déboguer un programme !

Avec eux, vos projet n'aurons plus la même allure !

---> **Matériel nécessaire :** dans la balise secret pour la partie 7.

## 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 !

### 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 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 maitrisée 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.

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 : ceux 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 néons. 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 si ne disposent pas de rétro-éclairage et ne sont ni des écrans LCD, ni des 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.

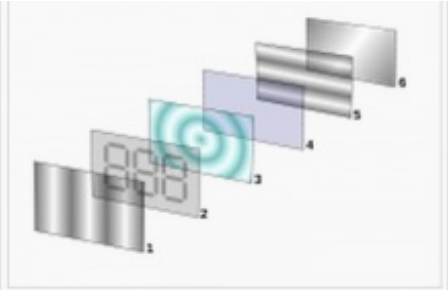
#### *Fonctionnement de l'écran*

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 ca 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 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). Chaque pixel est un cristal liquide. Lorsque 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.

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).



Afficheur 3 chiffres
1 et 5 : filtres polarisants ;
2 : électrodes avant ;
4 : électrode arrière ;
3 : cristaux liquides ;
6 : miroir.

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.

## 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 7*5 pixels. Ce qui fait qu'un écran de 16 colonnes et 2 lignes représente un total de 16*2*7*5 = 1120 pixels ! Heureusement pour nous, des ingénieurs sont passés par la et nous ont simplifié la tâche.

### *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'achat 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 'zéros' 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 | V0 | 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 | D0 à D7 | Bits de données |
| 15 | A | Anode du rétroéclairage (+5V) |
| 16 | K | Cathode du rétroéclairage (masse) |

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 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 :


## Quel écran choisir ?
## Les caractéristiques

### *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 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 zéros. 

| Afficheur alphanumérique | Afficheur graphique (monochrome) | Afficheur graphique (couleur) |
|---|---|---|
|  |  |  |

### *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 plait 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) !

### *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 !)

## Communication avec l'écran

### *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 : Registre 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, comme 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.

Ne vous inquiétez pas à l'idée de tout cela. Pour la suite du chapitre nous utiliserons une libraire 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".

#### *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).

#### *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).

### Comment on s'en sert ?

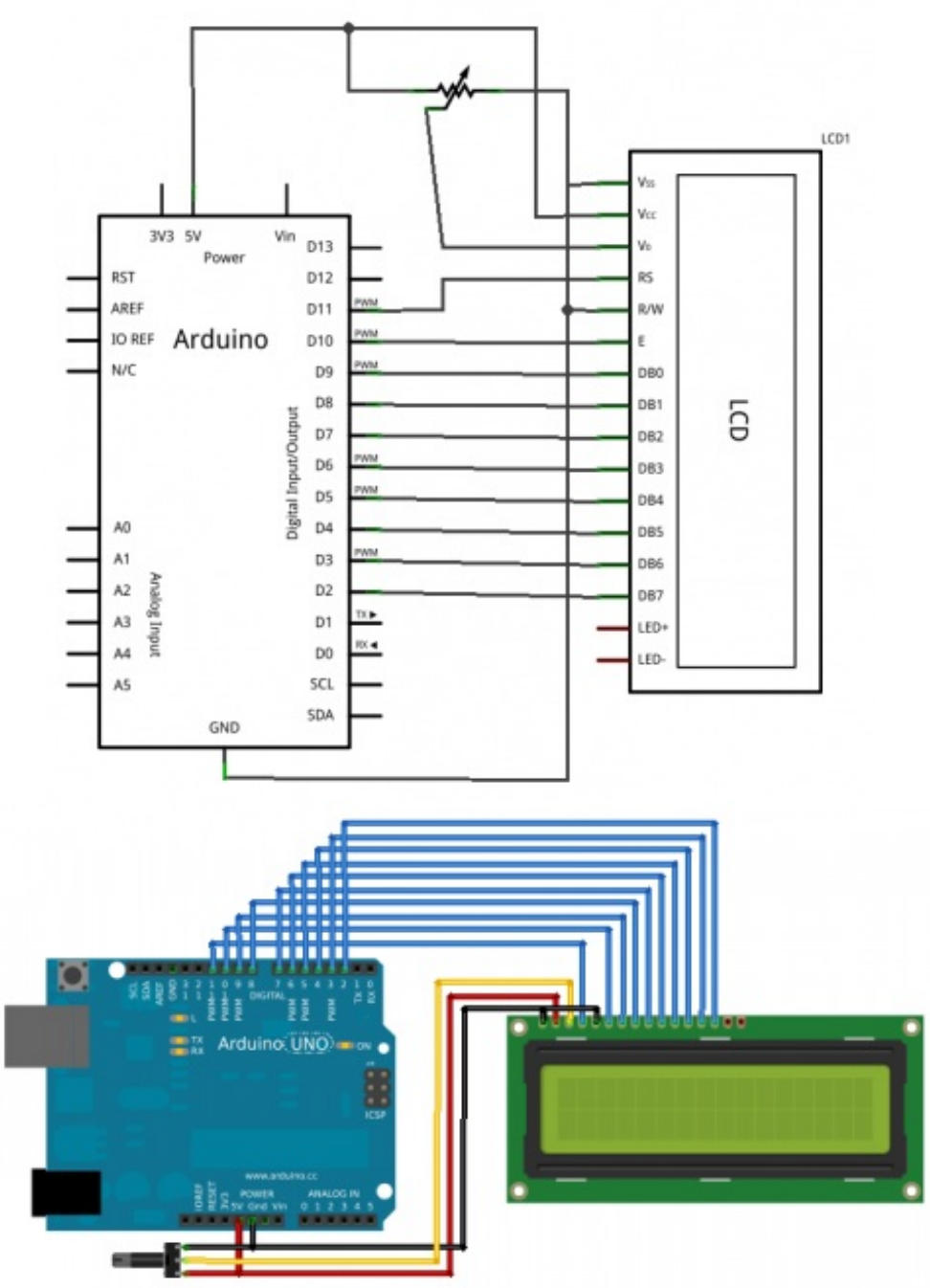
Comme expliqué précédemment, je vous propose de travailler avec un écran dont seulement quatre broches de données sont utilisées. Pour le bien de tous je vais présenter ici les deux montages, mais ne soyez pas surpris si dans les autres montages ou les vidéos vous voyez seulement un des deux.

## 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.

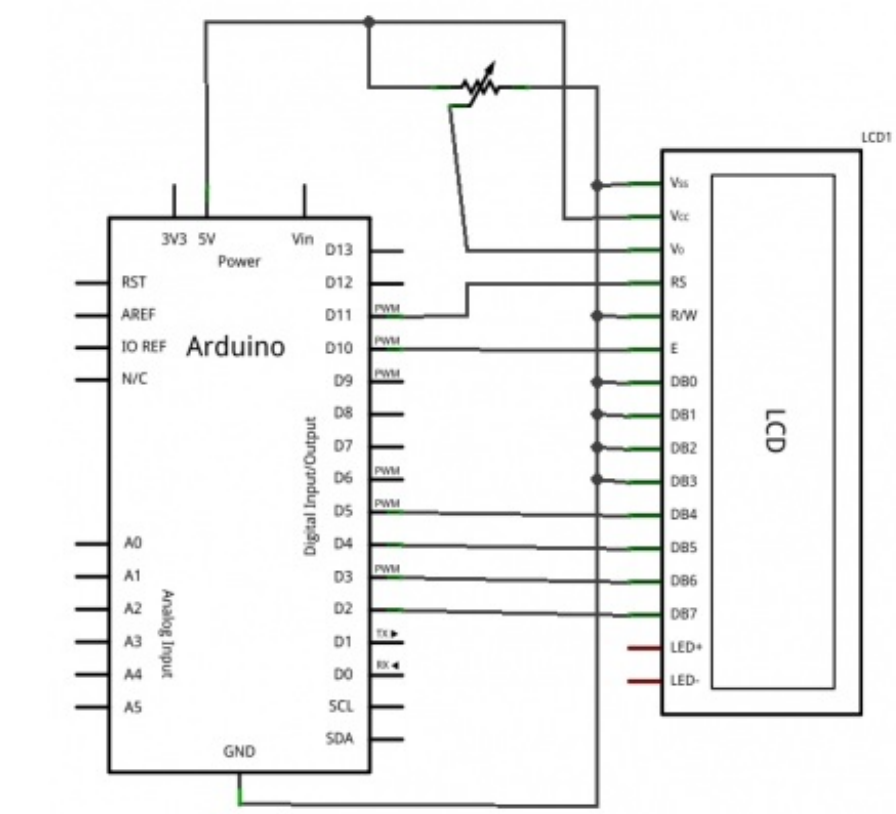
#### *Le montage à 8 broches de données*

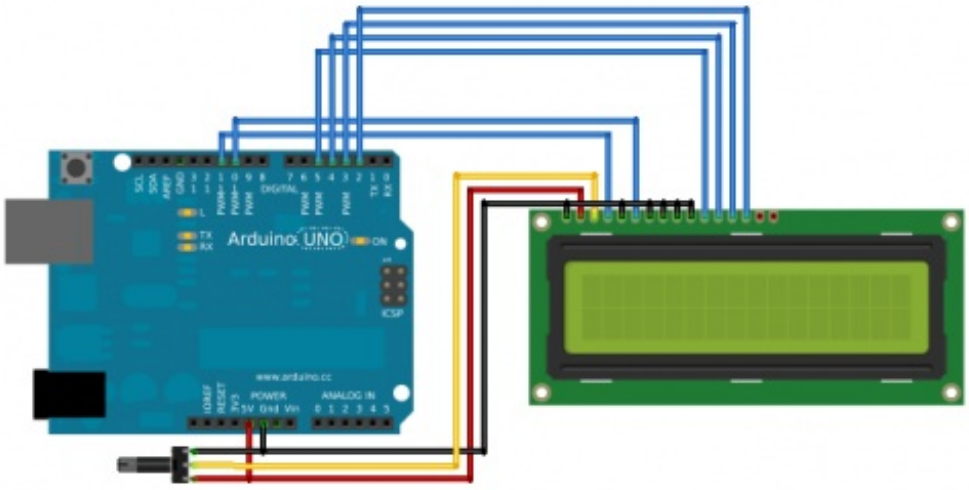






*Le montage à 4 broches de données*




## 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 `#include "LiquidCrystal.h"` doit apparaitre 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 commentaires 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) :

**Code : C**

```
#include "LiquidCrystal.h" //ajout de la librairie

//Vérifier 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 les Zer0s !"); //petit test pour vérifier que
tout marche
}

void loop() {}
```

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 :


Si jamais rien ne s'affiche, essayez de tourner votre potentiomètre de contraste. Si cela ne marche toujours pas, vérifier 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 !

# Votre premier texte !

Ça y est, on va pouvoir commencer à apprendre des trucks avec notre écran. 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.

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 !

## Ecrire du texte

## Afficher du texte

Vous vous rappelez comme 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 ce 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 zéros de la terre nous aurons juste à écrire :

**Code : C**

```
lcd.print("Salut les Zer0s!");
```

et pour code complet avec les déclarations on obtient :

**Code : C**

```
#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() {
  // set up the LCD's number of columns and rows:
  lcd.begin(16, 2);
  lcd.print("Salut les Zer0s!");
}

void loop() {
}
```

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 !

## 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 suspens 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 !

**Code : C**

```
int mavariable = 42;
lcd.print(mavariable);
```

## Combo ! Afficher du texte ET une variable

Bon vous aurez remarqué que notre code possède une certaine faiblesse... On n'affiche au choix un texte ou un nombre, mais pas les deux en même temps ! Nous allons donc voir maintenant une manière d'y remédier.

### *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 connaitre, ou connaitre sa cousine "printf".

Cette fonction est un peu particulière car elle ne prend pas un nombre d'argument fini. En effet, si vous voulez afficher 2 variables vous ne lui donnerez pas autant d'arguments que pour en afficher 4 (ce qui parait 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 chaine de caractère. Une fois que nous aurons écrit dedans, il nous suffira de l'envoyer sur l'écran en utilisant... print() !

### *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 chaine de caractère et une chaine à écrire. Un exemple simple serait d'écrire :

**Code : C**

```
char message[16] = "";
sprintf(message,"J'ai 42 ans");
```

Au début, le tableau message ne contient rien. Après la fonction `sprintf()`, il possédera le texte "J'ai 42 ans". Simple non ?

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 !

**Code : C**

```
char message[16] = "";
int nbA = 3;
int nbB = 5;
```

```
sprintf(message,"%d + %d = %d", nbA, nbB, nbA+nbB);
```

Cela affichera :

**Code : Console**

```
3 + 5 = 8
```

### *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 désirer 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. Ce paramètre prendra donc toujours autant de place sur l'écran (utile pour maitriser la disposition des caractères). Exemple :

**Code : C**

```
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);
```

À l'écran, on aura un texte tel que :

**Code : Console**

```
Ben:42,Luc: 5
```

On note l'espace avant le 5 grâce au forçage de l'écriture de la variable sur 2 caractères induit par *%2d*.

## Exercice, faire une horloge

### *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 vois pas pourquoi je pose la question puisque de toute manière vous n'avez pas le choix ! 

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 !

### *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). 

**Secret** (cliquez pour afficher)

**Code : C**

```
#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 minutes 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++;
  }
```

## Se déplacer sur l'écran

Bon, autant vous prévenir d'avance, 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...

## 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 ré-afficher 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 :

**Code : C**

```
#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("Hello World !");
}

void loop() {
  lcd.noDisplay();
  delay(500);
  lcd.display();
  delay(500);
}
```

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 ré-afficher il faudra le renvoyer à l'afficheur. Cette fonction au nom évident est : `clear()`.

Le code suivant vous permettra ainsi d'afficher un texte puis, au bout de 2 secondes, il disparaitra (pas de loop(), pas nécessaire) :

**Code : C**

```
#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("Hello World !");
  delay(2000);
  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.

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 disparaitra et ainsi de suite.

## Gérer le curseur

### *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). 

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.

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 horizontaux 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 chaines de caractères, donc des tableaux, qui eux sont adressés à partir de la case 0. Il parait 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 :

**Code : C**

```
#include <LiquidCrystal.h>

// 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
}
```

### *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()`.

**Code : C**

```
#include <LiquidCrystal.h>

// 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
}
```

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.

## 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.

### *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 !

**Code : C**

```
#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

// 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() {
  //règlage 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);

  //paramètrage du LCD
  lcd.begin(16, 2); // règle la taille du LCD
  lcd.print("Hello les Zeros !");
}

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 appelé par l'interruption du deuxième bouton
void aDroite() {
  lcd.scrollDisplayRight(); //on va à droite !
}
```

### *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 apparaitre et se "pousser" les uns après les autres :

**Code : C**

```
#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 déplacer
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("}");
}
```

## 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 !

### *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 réalités divisés en grille de 5x8 (5 en largeur et 8 en hauteur). C'est parce que le contrôleur de l'écran connait 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 :

**Code : C**

```
byte smiley[8] = {
  B00000,
  B10001,
  B00000,
  B00000,
  B10001,
  B01110,
  B00000,
  B00000
};
```

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.

Oh le joli smiley !

### *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 connaitre, 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, 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 :

**Code : C**

```
#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
}
```

Désormais, vous savez l'essentiel sur les LCD alphanumériques, vous êtes donc aptes pour passer au TP. 

# [TP] Supervision

Chers zéros, 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 !**

## 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 sur) 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 donc le signal passe de 1 à 0 lorsque un objet passe devant.
- Deux potentiomètres. Un sert de "consigne" et est réglé par l'utilisateur. L'autre représentera un capteur (mais comme vous n'avez pas forcément lu la partie sur les capteurs (et qu'elle n'est pas rédigée à l'heure de la validation de cette partie), ce capteur sera représenté par un autre potentiomètre). A 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.

### *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 toute les informations de manière lisible en même temps. On se propose donc de faire un affichage alterné entre deux interface. Chaque interface sera affiché pendant cinq secondes à tour de rôle.

La première affichera l'état des boutons. On pourra par exemple lire :

**Code : Autre**

```
Bouton G : ON
Bouton D : OFF
```

La seconde interface affichera la valeur de la consigne et celle du capteur. On aura par exemple :

**Code : Autre**

```
Consigne : 287
Capteur  : 115
```

(Sur la vidéo vous verrez "gauche / droite" pour symboliser les deux potentiomètres, chacun fait comme il veut).

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 !

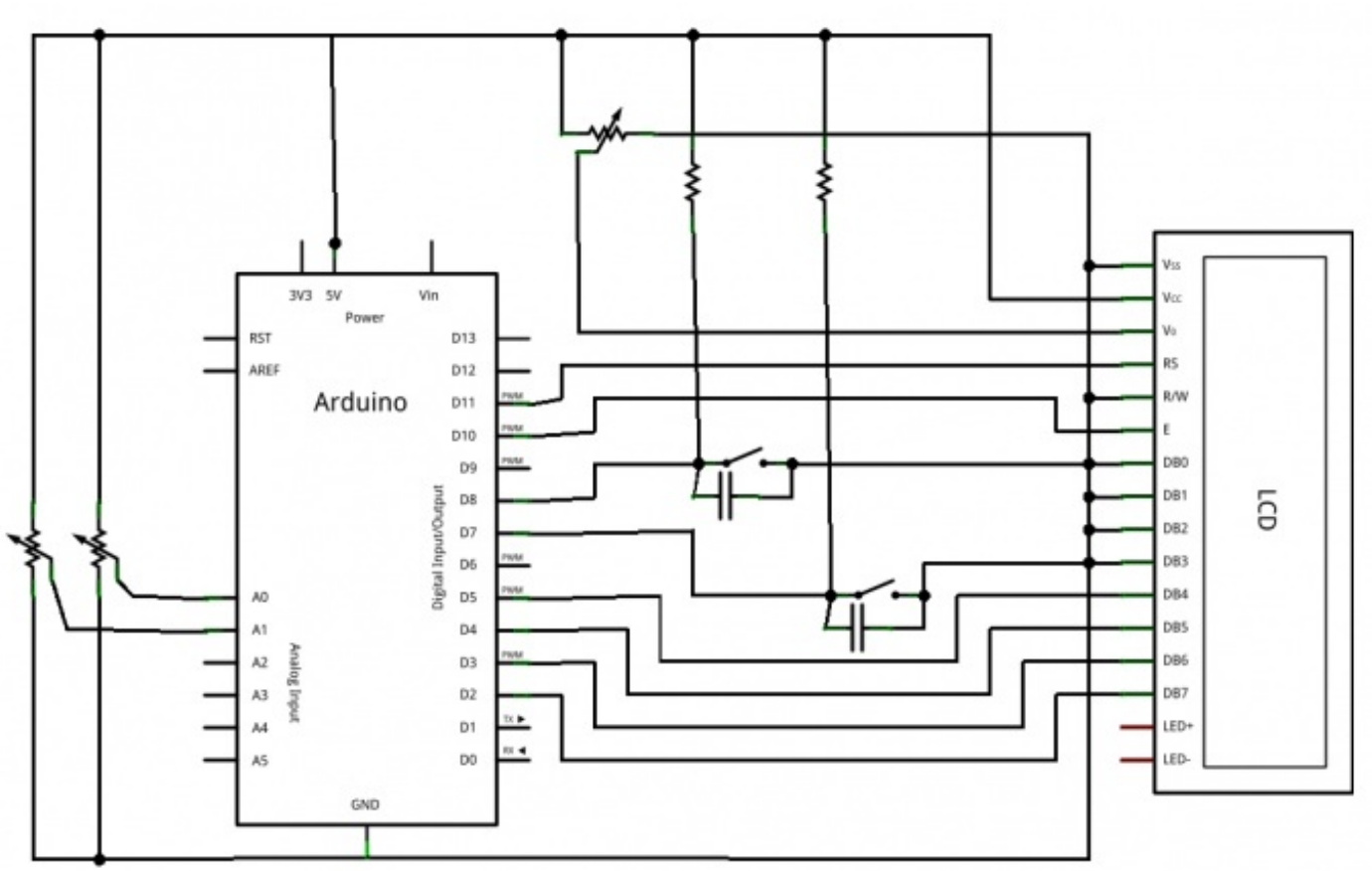
Je pense avoir fait le tour de mes attentes !
Je vous souhaite un bon courage, prenez votre temps, faites un beau schéma/montage/code et à bientôt pour la correction !

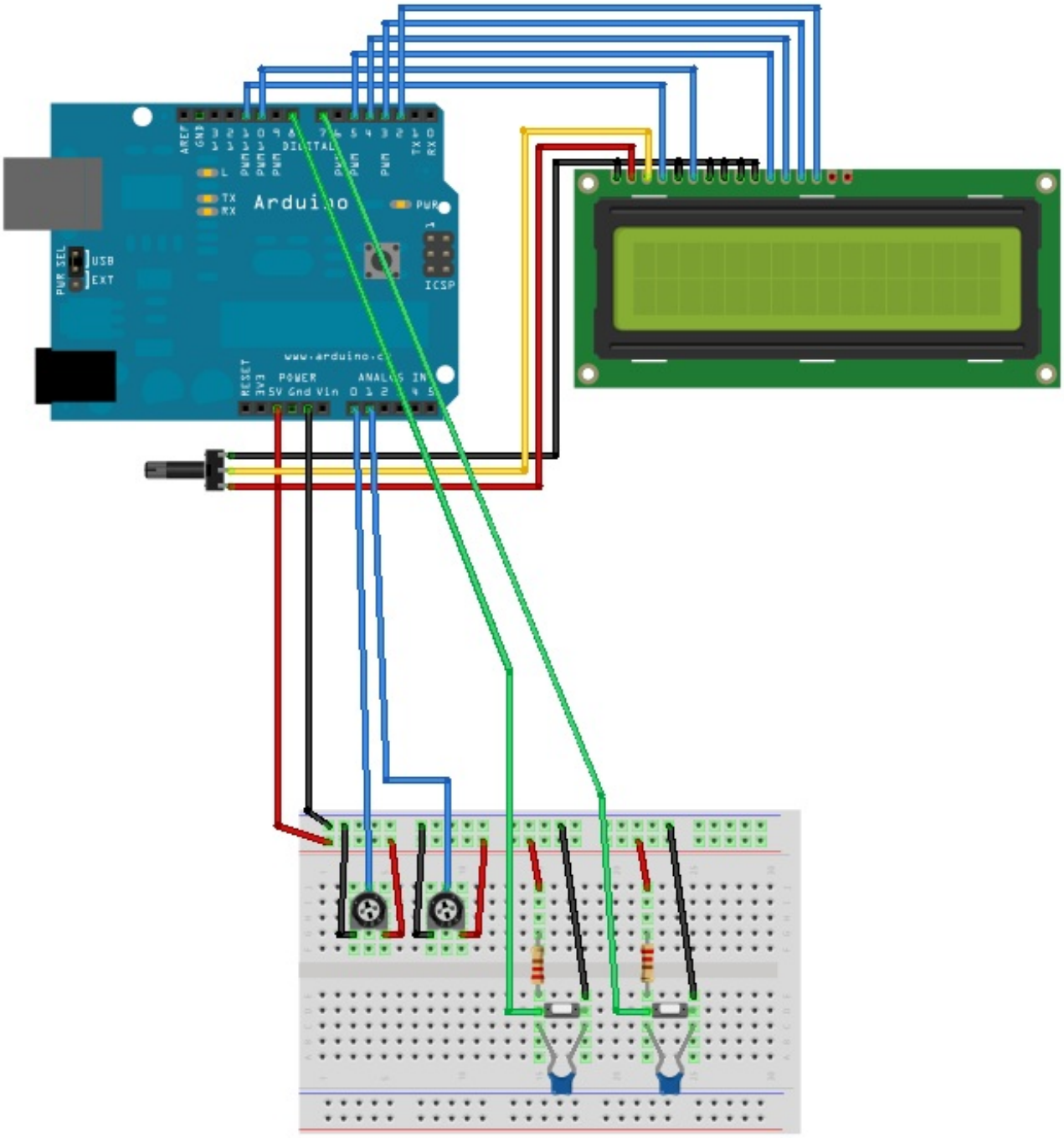
## Correction !
## Le montage

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 sur lesquelles 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) !


3V3
5V
Vin
Power
RST
AREF
Arduino
D13
D12
D11
D10
D9
D8
D7
D6
D5
D4
D3
D2
D1
D0
PWM
Digital Input/Output
A0
A1
A2
A3
A4
A5
Analog Input
GND
Vss
Vcc
Vo
RS
R/W
E
DB0
DB1
DB2
DB3
DB4
DB5
DB6
DB7
LED+
LED-
LCD



## 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.

### *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 quel "interface" nous sommes en train d'écrire. Voici un petit tableau résumant tout cela ainsi que le type des variables.

**Secret** (cliquez pour afficher)

| 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 |

### *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.

**Secret** (cliquez pour afficher)

**Code : C**

```
void setup() {
  //règlage des entrées/sorties
  pinMode(boutonGauche, INPUT);
  pinMode(boutonDroite, INPUT);
  pinMode(ledAlarme, OUTPUT);

  //paramètrage 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
}
```

### *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 toute 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.

**Secret** (cliquez pour afficher)

**Code : C**

```
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);

  delay(2); //pour s'assurer que les conversions analogiques sont
terminées avant de passer à la suite
}
```

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 envoi 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 stable (je rappel que le convertisseur offre plus ou moins deux bits de précision, soit 20mV de précision otale), 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 renvoi `true` sinon `false`.

**Secret** (cliquez pour afficher)

**Code : C**

```
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 entre ces deux
valeurs de plus de 2 unités, alors on met à jour
  if(abs(niveauGauche_old-niveauGauche) > 2 ||
abs(niveauDroite_old-niveauDroite) > 2)
    return true;
  else
    return false;
}
```

Une dernière fonction nous servira à faire la mise à jour de l'écran. Elle va préparer les deux chaines de caractères (celle du haut et celle du bas) et va ensuite les envoyer successivement sur l'écran. Pour écrire dans les chaines, 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 pourrait se chevaucher (essayer d'écrire un OFF puis un ON, sans clear(), cela vous fera un "ONF" à la fin).


**Secret** (cliquez pour afficher)

**Code : C**

```
void updateEcran()
{
  if(ecran)
  {
    //prépare les chaines 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 chaines à mettre sur l'écran : potentiomètres
    sprintf(messageHaut,"gauche = %4d", niveauGauche);
    sprintf(messageBas,"droite = %4d", niveauDroite);
  }

  //on envoie le texte
  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print(messageHaut);
  lcd.setCursor(0,1);
  lcd.print(messageBas);
}
```

### *La boucle principale*

Nous voici enfin au cœur du programme, la boucle principale. Cette dernière est relativement légère, grâce aux fonctions permettant de repartir le code en unité logique. La boucle principale n'a plus qu'à les utiliser à bon escient et dans le bon ordre (  ) 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 !


Aller, comme vous êtes sages, voici le code de cette boucle (qui va de paire avec les fonctions expliquées précédemment) :

**Secret** (cliquez pour afficher)

**Code : C**

```
void loop() {
```

```
  recupererDonnees(); //commence par récupérer les données des
boutons et capteurs

  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)
    digitalWrite(ledAlarme, LOW); //RAPPEL : piloté à l'état bas
donc on allume !
  else
    digitalWrite(ledAlarme, HIGH);

  if(millis() - temps > 5000) //si ça fait 5s qu'on affiche la
même donnée
  {
    ecran = ~ecran;
    lcd.clear();
    updateEcran();
    temps = millis();
  }
}
```

### *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).

**Secret** (cliquez pour afficher)

**Code : C**

```
#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
sur l'entrée analogique 0
const int potentiometreDroite = 1; //le potentiomètre de droite
sur l'entrée analogique 1
const int ledAlarme = 2; //la LED est branché sur la sortie 2

// initialise l'écran avec les bonne broche
// 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 et gérer les séquences
boolean ecran = LOW; //pour 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 memes 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() {

  recupererDonnees(); //commence par récupérer les données des
boutons et capteurs

  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)
    digitalWrite(ledAlarme, LOW); //RAPPEL : piloté à l'état bas
donc on allume !
  else
    digitalWrite(ledAlarme, HIGH);

  if(millis() - temps > 5000) //si ca fait 5s qu'on affiche la
même donnée
  {
    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(potentiometreDroite);

  delay(1); //pour s'assurer que les conversions analogiques sont
terminées avant de passer à la suite
}

boolean boutonsChanged()
{
  if(etatGauche_old != etatGauche || etatDroite_old != etatDroite)
    return true;
  else
    return false;
}

boolean potarChanged()
{
  //si un potentiomètre affiche une différence entre ces deux
valeurs de plus de 2 unités, 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,"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 chaines à mettre sur l'écran
    sprintf(messageHaut,"gauche = %4d", niveauGauche);
    sprintf(messageBas,"droite = %4d", niveauDroite);
  }

  //on envoie le texte
  lcd.clear();
  lcd.setCursor(0,0);
  lcd.print(messageHaut);
  lcd.setCursor(0,1);
  lcd.print(messageBas);
}
```

Que diriez-vous si je vous proposais d'utiliser des écrans LCD graphique ? Mmm ?

Ce cours n'en est qu'à ses débuts, il y a encore plein de chapitres en préparation. Soyez patient, les mises à jour se font régulièrement.

**Pour connaitre l'avancement du cours, cliquez-ici.**

En tous cas j'espère qu'il vous a plu et qu'il vous a donner envie de vous mettre à Arduino pour réaliser vos projets les plus fous

en toute facilité ! Je vous invite à laisser des commentaires sur les chapitres que vous avez lu, on essaye de prendre en compte vos messages afin de rendre le cours encore plus abouti qu'il ne l'est déjà.

Merci à tous et à Xababafr pour avoir soutenu le cours dès ses débuts et les corrections orthographiques et les quelques images qu'il a apportées au cours !

**Vous avez des questions ? Des commentaires ? Des suggestions ?**
**Alors postez un message ici : forum du cours Arduino.**

***

**Vous avez besoin d'aide pour un projet ? Besoin de conseils ?**
**Alors lisez les règles avant de poster sur le forum.**

**Usez des forums qui sont là pour vous aider et ne m'envoyez pas de MP je n'y répondrai plus lorsqu'il s'agira de demande d'aide ou de conseil. Pensez à tous ceux qui auront leur réponse grâce à vos questions !**

Bonne continuation !