

[Arduino 4] Les grandeurs analogiques

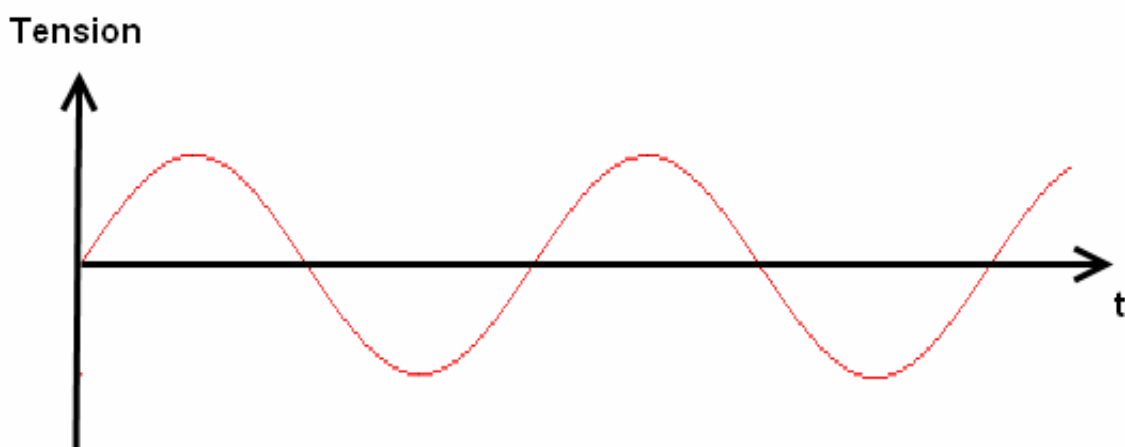
Dans cette partie, je vais introduire la notion de grandeur analogique qui sont opposées aux grandeurs logiques. Grâce à ce chapitre, vous serez ensuite capable d'utiliser des capteurs pour interagir avec l'environnement autour de votre carte Arduino, ou bien des actionneurs tels que les moteurs.

[Arduino 401] 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 :

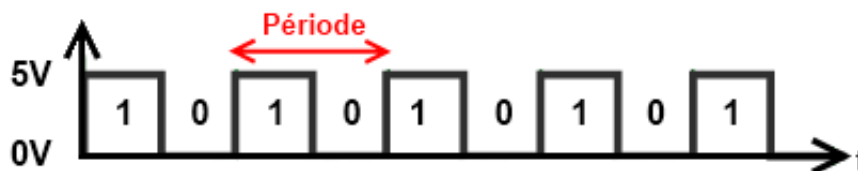


On retiendra que l'on ne s'occupe que de la tension et non du courant, en fonction du temps.

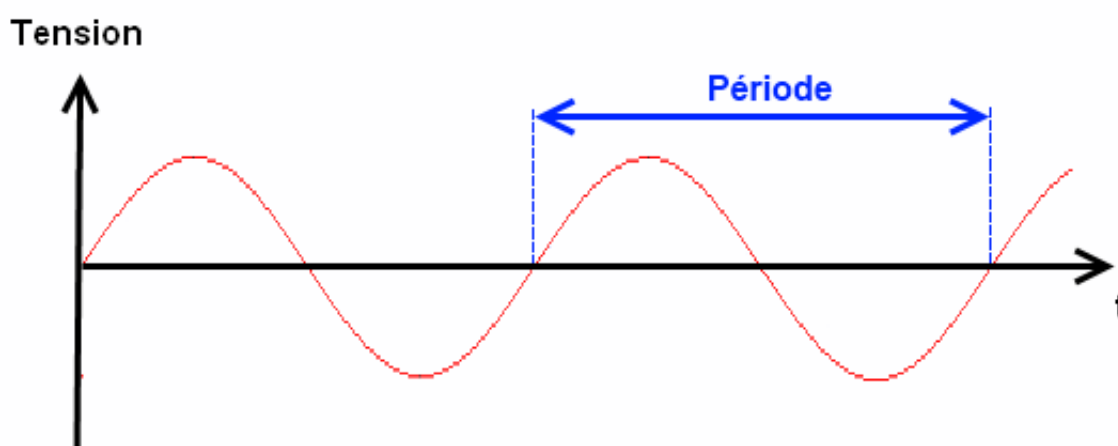
Si on essaye de mettre ce signal (ce que je vous déconseille de faire) sur une entrée numérique de l'Arduino et qu'on lit les valeurs avec `digitalRead()`, on ne lira que 0 ou 1. Les broches numériques de l'Arduino étant incapable de lire les valeurs d'un signal analogique.

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 motif de ce signal qui se répète et qui donne ainsi la forme du signal. Prenons l'exemple d'un signal binaire qui prend un niveau logique 1 puis un 0, puis un 1, puis un 0, ...



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 court laps de temps entre deux valeurs extrêmes. En même temps qu'il produit cette tension, il compte. Lorsque la tension d'entrée du convertisseur devient égale à la tension générée par ce dernier, alors le convertisseur arrête de compter. Et pour terminer, avec la valeur du compteur, il détermine la valeur de la grandeur d'entrée. Malgré sa bonne précision, sa conversion reste assez lente et dépend de la grandeur à mesurer. Il est, de ce fait, peu utilisé.
- **Convertisseur flash** : ce type de convertisseur génère lui aussi des tensions analogiques. Pour être précis, il en génère plusieurs, chacune ayant une valeur plus grande que la précédente (par exemple 2V, 2.1V, 2.2V, 2.3V, etc.) et compare la grandeur d'entrée à chacun de ces paliers de tension. Ainsi, il sait entre quelle et quelle valeur se trouve la tension mesurée. Ce n'est pas très précis comme mesure, mais il a l'avantage d'être rapide et malheureusement cher.
- **Convertisseur à approximations successives** : Pour terminer, c'est ce convertisseur que nous allons étudier...

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 connaît pas ce nombre et doit le trouver. La méthode la plus rapide pour que le joueur 2 puisse trouver quel est le nombre choisi par le joueur 1 est :

1	Joueur 1 dit : "quel est le nombre mystère ?"
2	> 40
3	
4	Joueur 1 dit : "Ce nombre est plus grand"
5	> 80
6	
7	Joueur 1 dit : "Ce nombre est plus petit"
8	> 60
9	
10	Joueur 1 dit : "Ce nombre est plus grand"
11	> 70
12	
13	Joueur 1 dit : "Ce nombre est plus grand"
14	> 75
15	
16	Joueur 1 dit : "Ce nombre est plus petit"
17	> 72
18	
19	Bravo, Joueur 2 a trouvé le nombre mystère !

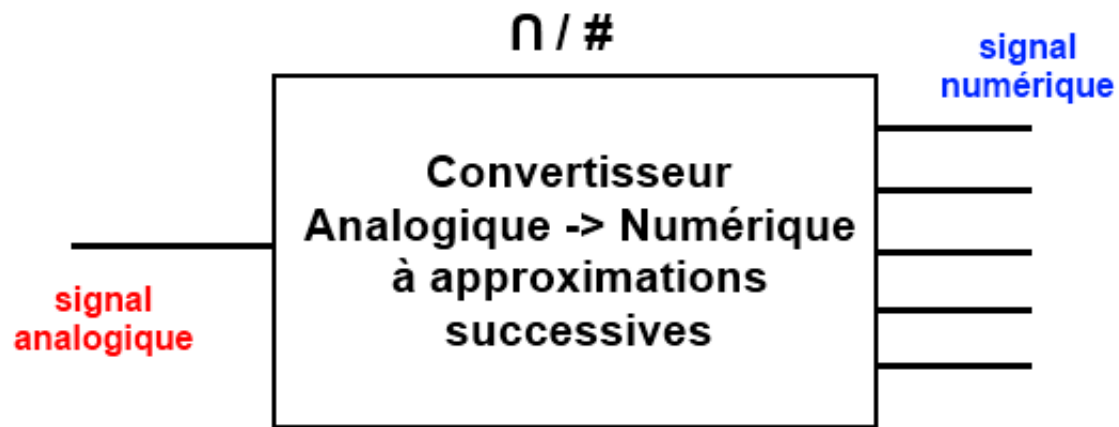
Je le disais, le joueur 2, pour arriver le plus rapidement au résultat, doit choisir une méthode rapide. Cette méthode, vous l'aurez deviné, consiste à couper en deux l'espace de recherche. Au début, cet espace allait de 0 à 100, puis au deuxième essai de 40 à 100, au troisième essai de 40 à 80, etc.

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'abrègerai 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 #.

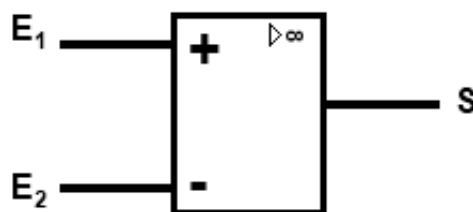
Cette **tension analogique** de 3.36V va rentrer dans le CAN et va ressortir sous **forme numérique** (avec des 0 et 1). Mais que se passe-t-il à l'intérieur pour arriver à un tel résultat ? Pour que vous puissiez comprendre correctement comment fonctionne ce type de CAN, je vais être obligé de vous 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

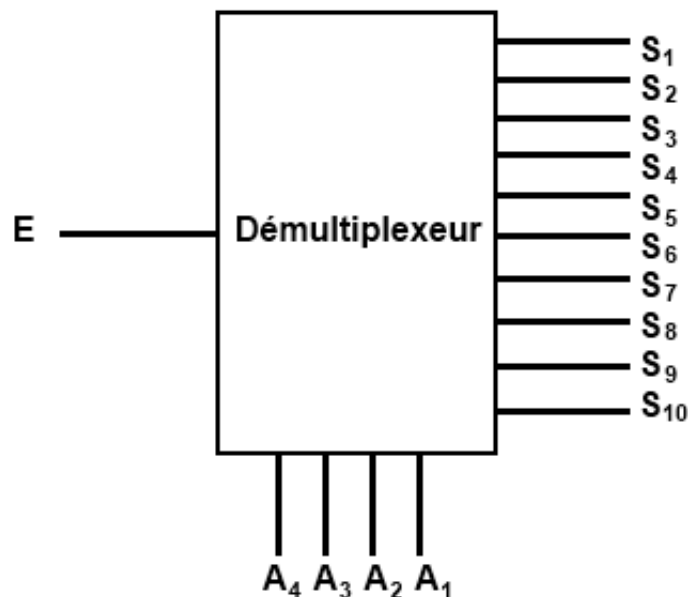
l'alimenter entre 0V et +5V. Comme cela, sa sortie sera soit égale à 0V, soit égale à +5V. Ainsi, on rentre dans le domaine des tensions acceptées par les micro-contrôleurs et de plus il verra soit un état logique BAS, soit un état logique HAUT. On peut réécrire les conditions précédemment énoncées comme ceci :

- $E_1 > E_2$ alors $S = 1$
- $E_1 < E_2$ alors $S = 0$
- $E_1 = E_2$, alors $S = \textit{indéfini}$

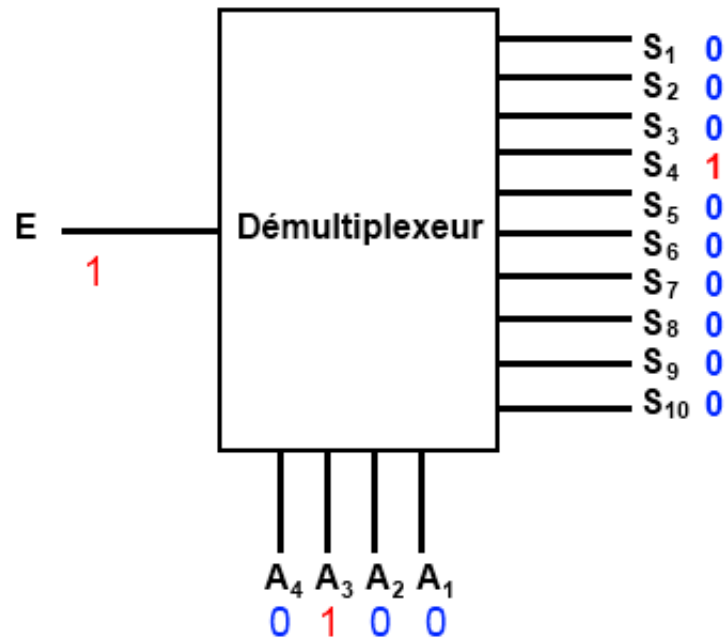
Simple n'est-ce pas ?

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

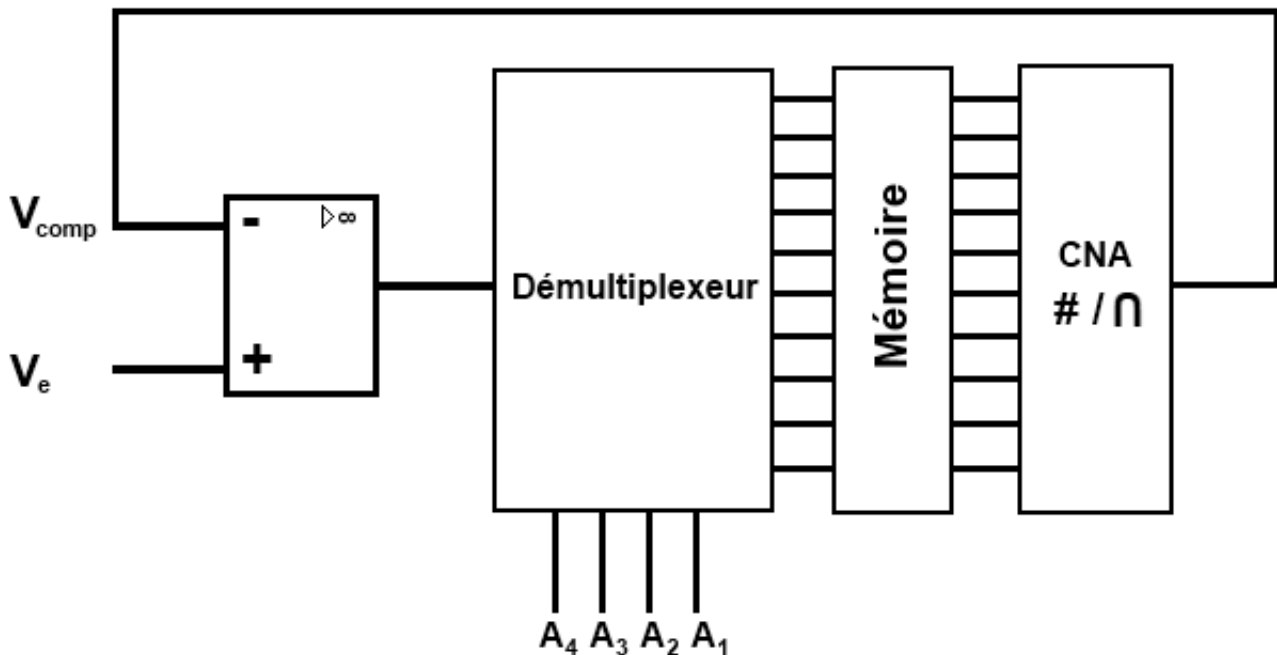
Ce composant électronique sert simplement à stocker des données sous forme binaire.

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

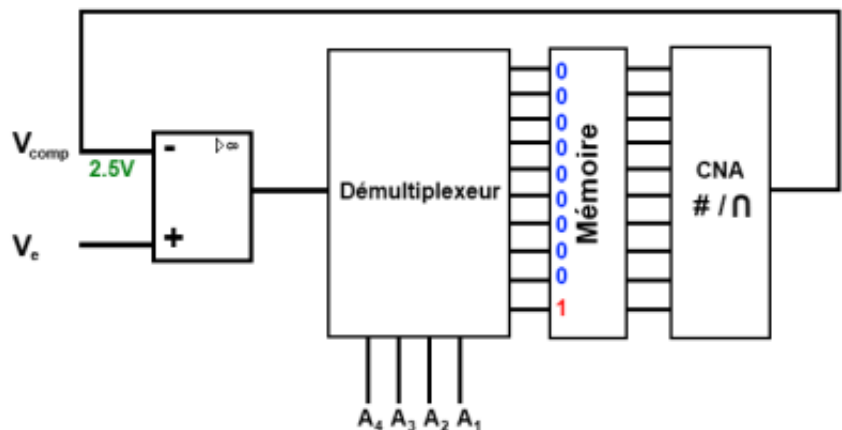
Rentrons dans les explications du fonctionnement d'un CAN à approximations successives. Je vous ai fait un petit schéma rassemblant les éléments précédemment présentés :



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 \rightarrow 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.

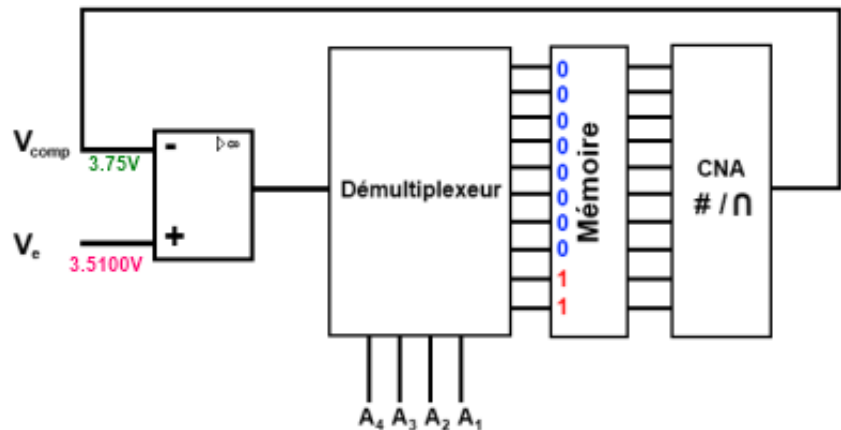
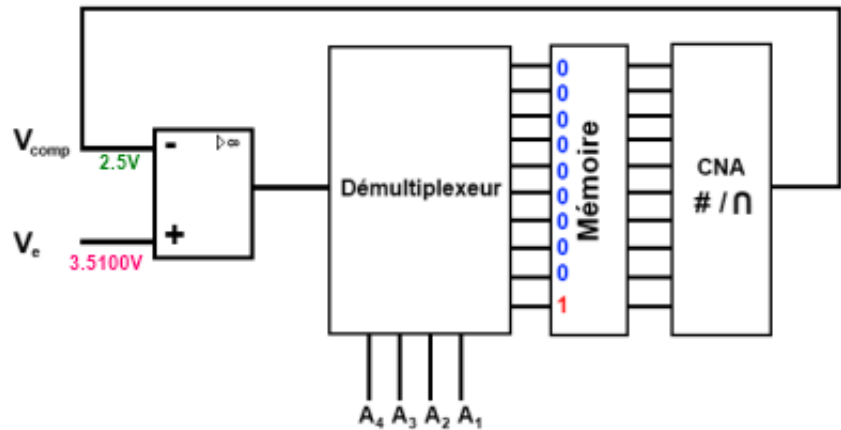


Suivons le fonctionnement étape par étape : **Étape 1 :**

- 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 jeu. Avec ses signaux de sélections, il va sélectionner la sortie ayant le poids le plus élevé, soit S_{10} .
- La mémoire va alors enregistrer le niveau logique présent sur la broche S_{10} , dans notre cas c'est **1**.

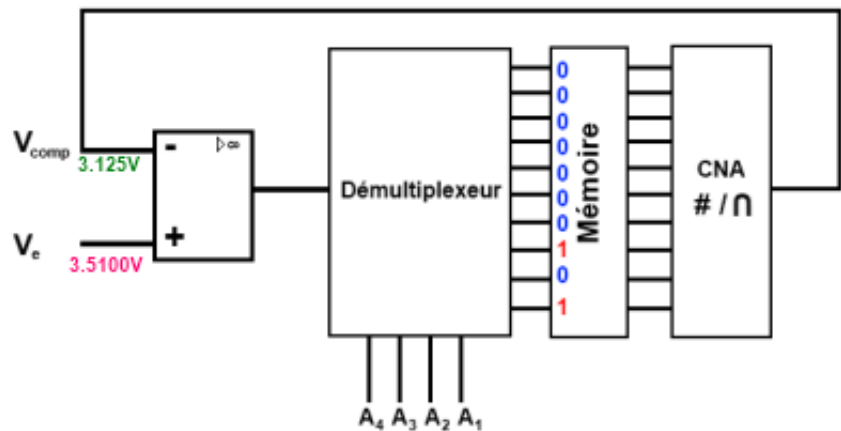
Étape 2 :

- 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 trois principaux :

- **la plage de tension d'entrée** : le convertisseur analogique de l'Arduino ne peut recevoir à son entrée que des tensions comprises entre 0V et +5V. On verra plus loin comment améliorer la précision du CAN.
- **la précision** : la précision du convertisseur est très bonne sauf pour les deux derniers bits de poids faible. On dit alors que la précision est de $\pm 2LSB$ (à cause du pas de calcul que je viens de vous expliquer).
- **la vitesse de conversion** : le convertisseur N/A de la carte Arduino n'a pas une

très grande vitesse de conversion par rapport à un signal audio par exemple. Ainsi , si l'on convertit un signal audio analogique en numérique grâce à la carte Arduino, on ne pourra entendre que les fréquences en dessous de 10kHz. Dans bien des cas cela peut être suffisant, mais d'en d'autre il faudra utiliser un convertisseur A/N externe (un composant en plus) qui sera plus rapide afin d'obtenir le spectre audio complet d'un signal sonore.

Lecture analogique, on y vient...

Lire la tension sur une broche analogique

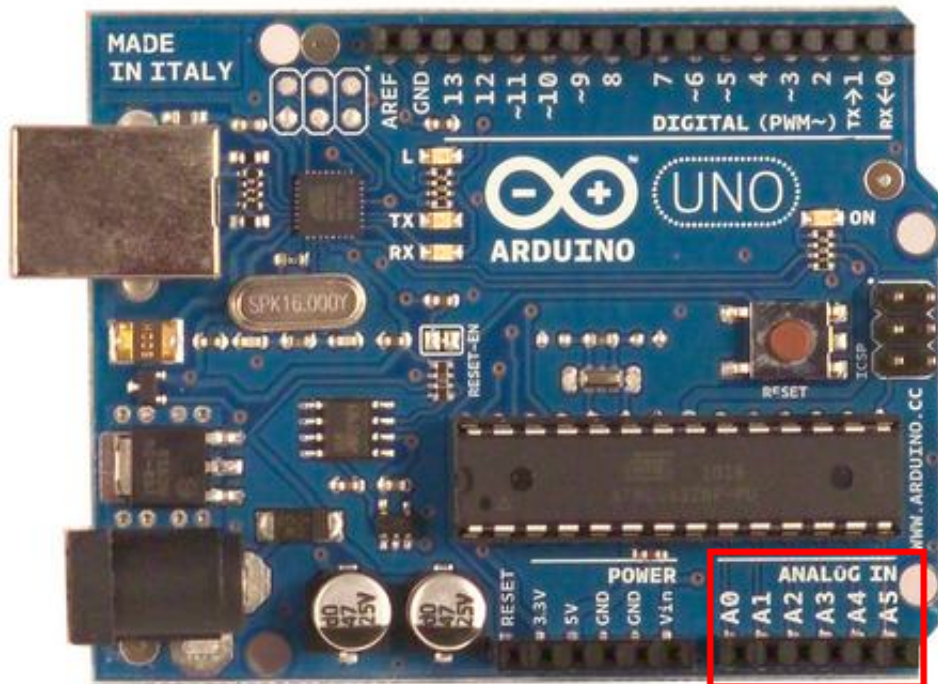
Un truc très sympa avec Arduino, c'est que c'est facile à prendre en main. Et ça se voit une fois de plus avec l'utilisation des convertisseurs numérique – > analogique ! En effet, vous n'avez qu'une seule nouvelle fonction à retenir : `analogRead()` !

`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 :

```
1 //broche analogique 3 OU broche numérique 3
2 const int monCapteur = 3;
3
4 //la valeur lue sera comprise entre 0 et 1023
5 int valeurLue = 0;
6
7 //fonction setup()
8 {
9
10 }
11
12 void loop()
13 {
14     //on mesure la tension du capteur sur la broche analogique 3
15     valeurLue = analogRead(monCapteur);
16
17     //du code et encore du code ...
18 }
```

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 a 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 :

```
1 //variable stockant la valeur lue sur le CAN
2 int valeurLue;
3 //résultat stockant la conversion de valeurLue en Volts
4 float tension;
5
6 void loop()
7 {
8     valeurLue = analogRead(uneBrocheAvecUnCapteur);
9     //produit en croix, ATTENTION, donne un résultat en mV !
10    tension = valeurLue * 4.88;
11    //formule à aspect "physique", donne un résultat en mV !
12    tension = valeurLue * (5 / 1024);
13 }
```

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 :

```
1 sortie = map(valeur_d_entree,
2     valeur_extreme_basse_d_entree,
3     valeur_extreme_haute_d_entree,
4     valeur_extreme_basse_de_sortie,
5     valeur_extreme_haute_de_sortie
6 );
7 //cette fonction retourne la valeur calculée équivalente entre les deux intervalles d
```

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 :

```
1 tension = map(valeurLue, 0, 1023, 0, 5000); //conversion de la valeur lue en tension
```

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 :

```
1 tension = map(valeurLue, 0, 1023, 0, 5000); //conversion de la valeur lue en tension
2 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) :

```
1 valeurLue = 458
2 tension = 2.290V
```

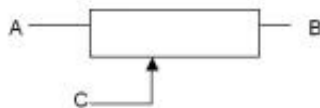
On est moins précis que la tension calculée plus haut, mais on peut jouer en précision en modifiant les valeurs de sortie de la fonction `map()`. Ou bien garder le calcul théorique et le placer dans une “fonction maison”.

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](#). 😊

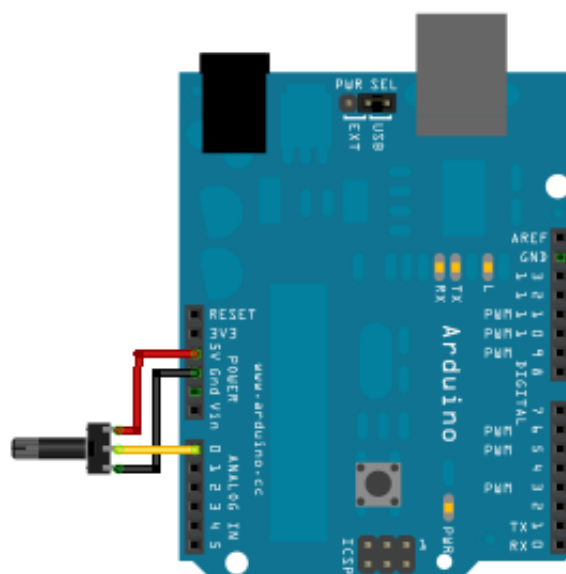
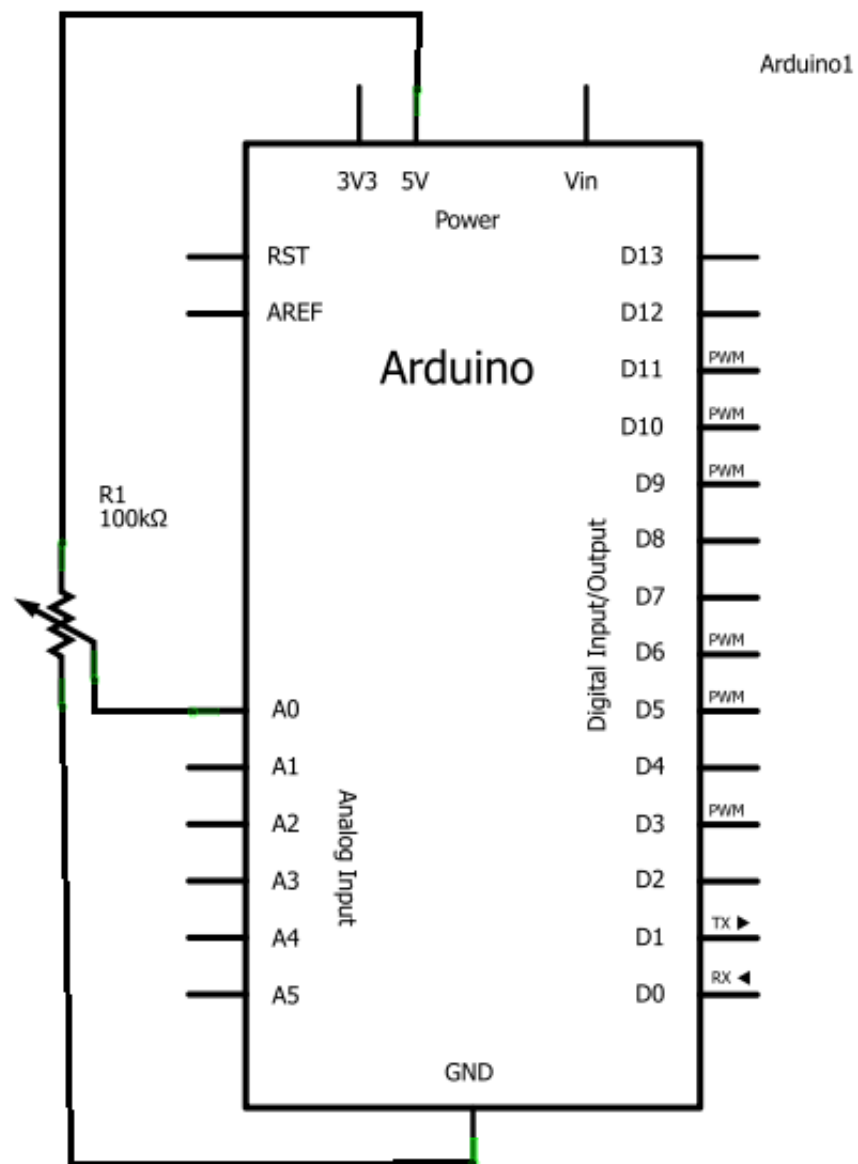
Cas n°2 : la résistance variable

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 donc 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 :

```
1 // le potentiomètre, branché sur la broche analogique 0
2 const int potar = 0;
3 //variable pour stocker la valeur lue après conversion
4 int valeurLue;
5 //on convertit cette valeur en une tension
6 float tension;
7
8 void setup()
9 {
10     //on se contente de démarrer la liaison série
11     Serial.begin(9600);
12 }
13
14 void loop()
15 {
16     //on convertit en nombre binaire la tension lue en sortie du potentiomètre
17     valeurLue = analogRead(potar);
18
19     //on traduit la valeur brute en tension (produit en croix)
20     tension = valeurLue * 5.0 / 1024;
21
22     //on affiche la valeur lue sur la liaison série
23     Serial.print("valeurLue = ");
24     Serial.println(valeurLue);
25
26     //on affiche la tension calculée
27     Serial.print("Tension = ");
28     Serial.print(tension, 2);
29     Serial.println(" V");
30
31     //on saute une ligne entre deux affichages
32     Serial.println();
33     //on attend une demi-seconde pour que l'affichage ne soit pas trop rapide
34     delay(500);
35 }
```

Vous venez de créer votre premier Voltmètre ! 😊

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.

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 1V et jamais au delà. Par défaut, la mesure se fera entre 0 et 5V sur 1024 niveaux (soit une précision de 4.88 mV). Ce qui veut dire qu'on aura seulement les 204 premiers niveaux d'utilité puisque tout le reste correspondra à plus d'un volt. Pour améliorer la précision de lecture, on va réduire la plage de mesure d'entrée du convertisseur analogique – > numérique en réduisant la tension de référence utilisée (initialement 5V). Pour cela, rien de matériel, tout se passe au niveau du programme puisqu'il y a une fonction qui existe : [analogReference\(\)](#). Cette fonction prend en paramètre le nom de la référence à utiliser :

- DEFAULT : La référence de 5V par défaut (ou 3,3V pour les cartes Arduino fonctionnant sous cette tension, telle la Due)
- INTERNAL : Une référence interne de 1.1V (pour la Arduino Uno)
- INTERNAL1V1 : Comme ci-dessus mais pour la Arduino Mega
- INTERNAL2V56 : Une référence de 2.56V (uniquement pour la Mega)
- EXTERNAL : La référence sera celle appliquée sur la broche ARef

Dans notre cas, le plus intéressant sera de prendre la valeur INTERNAL pour pouvoir faire des mesures entre 0 et 1.1V. Ainsi, on aura 1024 niveaux ce qui nous fera une précision de 1.07mV. C'est bien meilleur ! Le code est à placer dans la fonction setup() de votre programme :

```
1 void setup()  
2 {  
3     //permet de choisir une tension de référence de 1.1V  
4     analogReference(INTERNAL);  
5 }
```

Tension de référence externe

Maintenant que se passe-t'il si notre mesure devait être faite entre 0 et 3V ? On ne pourrait plus utiliser INTERNAL1V1 puisqu'on dépasse les 1.1V. On risquerait alors de griller le comparateur. Dans le cas d'une Arduino Mega, on ne peut pas non plus utiliser INTERNAL2V56 puisqu'on dépasse les 2.56V. Nous allons donc ruser en prenant une référence externe à l'aide de la valeur EXTERNAL comme ceci :

```
1 void setup()  
2 {  
3     //permet de choisir une tension de référence externe à la carte  
4     analogReference(EXTERNAL);  
5 }
```

Ensuite, il ne restera plus qu'à apporter une tension de référence supérieure à 3V sur la broche ARef de la carte pour obtenir notre nouvelle référence.

Astuce : la carte Arduino produit une tension de 3.3V (à côté de la tension 5V). Vous pouvez donc utiliser cette tension directement pour la tension de référence du convertisseur. 😊 Il suffit pour cela de relier avec un fil la sortie indiquée 3.3V à l'entrée AREF.

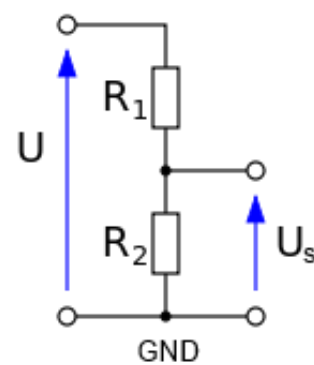
Attention cependant, la tension maximale de référence **ne peut être supérieure à +5V** et la **minimale inférieure à 0V**. En revanche, toutes les tensions comprises entre ces deux valeurs sont acceptables.

Mais, si je veux que ma tension d'entrée puisse varier au-delà de +5V, comment je fais ? Y a-t-il un moyen d'y parvenir ? o_O

Oui, voyez ce qui suit...

Solution 2 : utiliser un pont diviseur de tension

Nous avons vu cela à la partie précédente avec le potentiomètre. Il s'agit en fait de diviser votre signal par un certain ratio afin de l'adapter aux contraintes d'entrées du convertisseur. Par exemple, imaginons que nous ayons une mesure à faire sur un appareil qui délivre une tension comprise entre 0 et 10V. Cela ne nous arrange pas puisque nous ne sommes capable de lire des valeurs qu'entre 0 et 5V. Nous allons donc diviser cette mesure par deux afin de pouvoir la lire sans risque. Pour cela, on utilisera deux résistances de valeur identique (2 fois $10k\Omega$ par exemple) pour faire un pont diviseur de tension. La tension à mesurer rentre dans le pont (U sur le schéma ci-contre) et la tension mesurable sort au milieu du pont (tension U_s). Il ne reste plus qu'à connecter la sortie du pont à une entrée analogique de la carte Arduino et lire la valeur de la tension de sortie.



Libre à vous de modifier les valeurs des résistances du pont diviseur de tension pour faire rentrer des tensions différentes (même au delà de 10V !). Attention cependant à ne pas dépasser les +5V en sortie du pont !

Solution 3 : utiliser un CAN externe

La deuxième solution consisterait simplement en l'utilisation d'un convertisseur analogique –> numérique externe. A vous de choisir le bon. Il en existe beaucoup, ce qu'il faut principalement regarder c'est :

- la **précision** (10 bits pour Arduino, existe d'autre en 12bits, 16bits, ...)
- la **vitesse d'acquisition** (celui d'Arduino est à une vitesse de $100\mu s$)
- le **mode de transfert** de la lecture (liaison série, I²C, ...)
- le **nombre d'entrées** (6 sur Arduino Uno)
- la **tension d'entrée** maximale et minimale (max +5V et min 0V)

Au programme :

- Le prochain chapitre est un TP faisant usage de ces voies analogiques
- Le chapitre qui le suit est un chapitre qui vous permettra de créer des tensions analogiques avec votre carte Arduino, idéal pour mettre en œuvre la deuxième solution d'amélioration de la précision de lecture 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.

[Arduino 402] [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

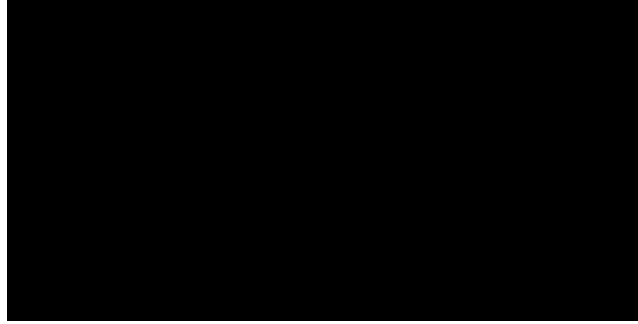
Vu-mètre, ça vous parle ?

Dans ce TP, nous allons réaliser un **vu-mètre**. Même si le nom ne vous dit rien, je suis sûr que vous en avez déjà rencontré. Par exemple, sur une chaîne hi-fi ou sur une table de mixage on voit souvent des loupottes s'allumer en fonction du volume de la note joué. Et bien c'est ça un vu-mètre, c'est un système d'affichage sur plusieurs LED, disposées en ligne, qui permettent d'avoir un retour visuel sur une information analogique (dans l'exemple, ce sera le volume).

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 !

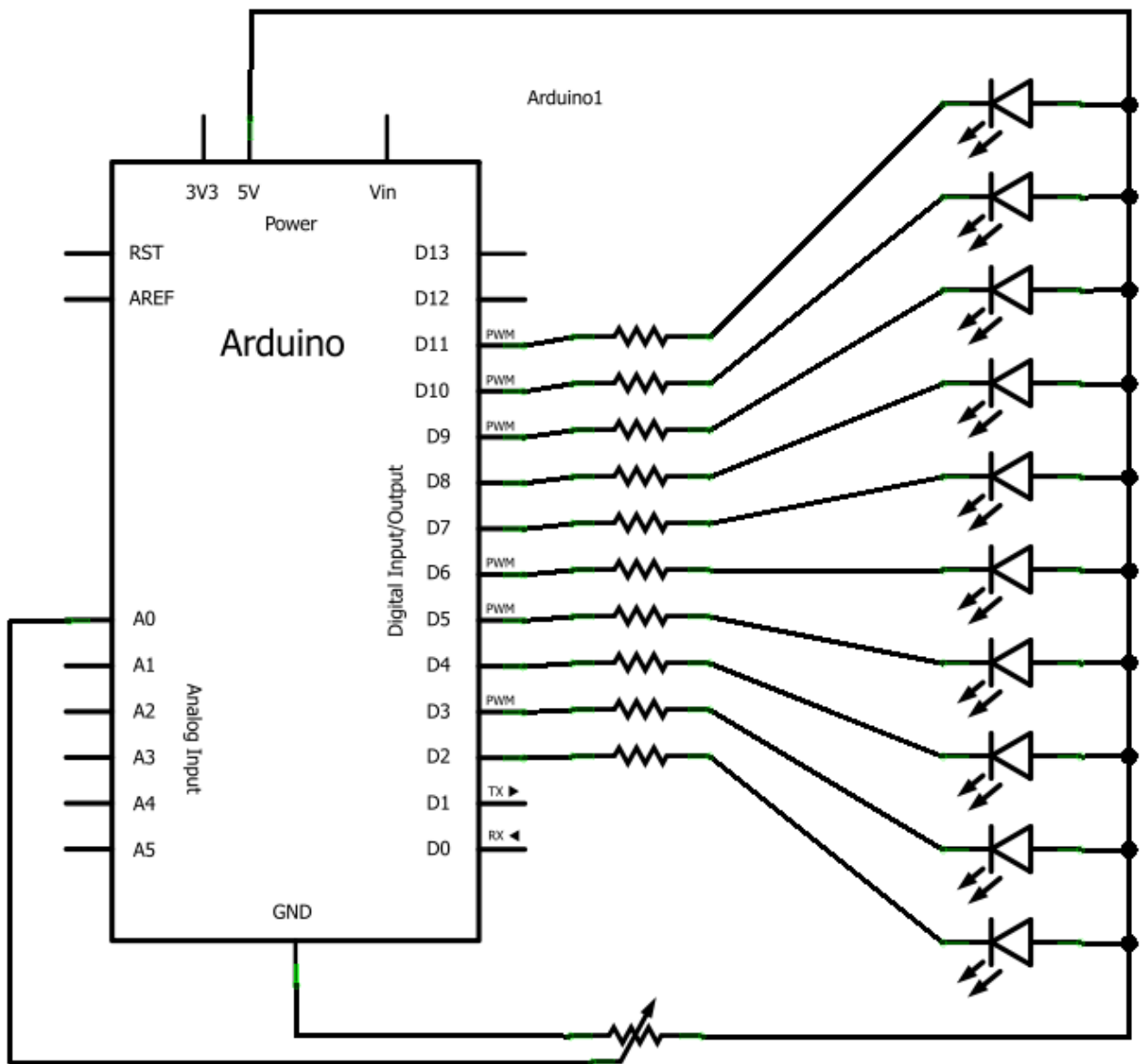


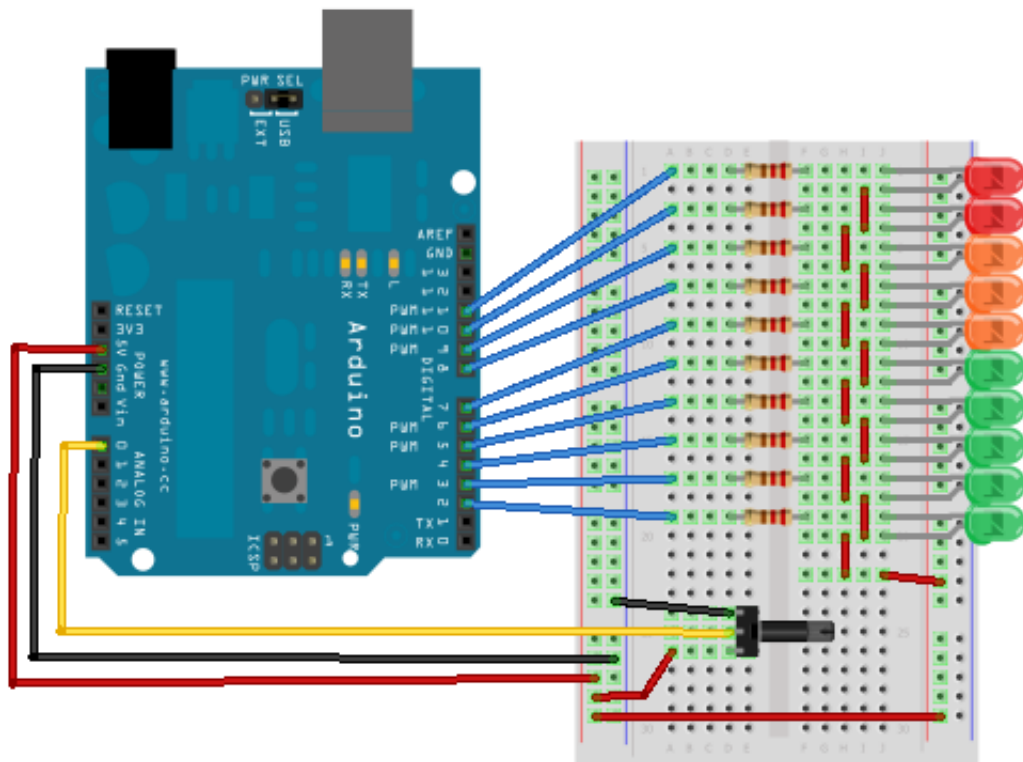
Correction !

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 :





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 !

```
1 // Déclaration et remplissage du tableau...
2 // ...représentant les broches des LEDs
3 const int leds[10] = {2,3,4,5,6,7,8,9,10,11};
4 //le potentiomètre sera branché sur la broche analogique 0
5 const int potar = 0;
6 //variable stockant la tension mesurée
7 int tension = 0;
```

Une fois que l'on a fait ces déclarations, il ne nous reste plus qu'à déclarer les broches en sortie et à les mettre à l'état HAUT pour éteindre les LEDs. Pour faire cela de manière simple (au lieu de 10 copier/coller), nous allons utiliser une boucle `for` pour effectuer l'opération 10 fois (afin d'utiliser la puissance du tableau).

```
1 void setup()
2 {
3     int i = 0;
```

```

4   for(i = 0; i < 10; i++)
5   {
6       //déclaration de la broche en sortie
7       pinMode(leds[i], OUTPUT);
8       //mise à l'état haut
9       digitalWrite(leds[i], HIGH);
10  }
11 }

```

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 :

```

1 void loop()
2 {
3     //on récupère la valeur de la tension du potentiomètre
4     tension = analogRead(potar);
5     //et on affiche sur les LEDs cette tension
6     afficher(tension);
7 }

```

Encore plus fort, la même écriture mais en une seule ligne !

```

1 void loop()
2 {
3     //la même chose qu'avant même en une seule ligne !
4     afficher(analogRead(potar));
5 }

```

Fonction d'affichage

Alors certes la fonction principale est très légère, mais ce n'est pas une raison pour ne pas avoir un peu de code autre part. En effet, le gros du traitement va se faire dans la fonction d'affichage, qui, comme son nom et ses arguments l'indiquent, va servir à afficher sur les LEDs la tension mesurée. Le but de cette dernière sera d'allumer les LEDs de manière proportionnelle à la tension mesurée. Par exemple, si la tension mesurée vaut 2,5V (sur 5V max) on allumera 5 LEDs (sur 10). Si la tension vaut 5V, on les allumera toutes. Je vais maintenant vous montrer une astuce toute simple qui va tirer pleinement parti du tableau de broches créé tout au début. Tout d'abord, mettons-nous d'accord. Lorsque l'on fait une mesure analogique, la valeur retournée est comprise entre 0 et 1023. Ce que je vous propose, c'est donc d'allumer une LED par tranche de 100 unités. Par exemple, si la valeur est comprise entre 0 et 100, une seule LED est allumée. Ensuite, entre 100 et 200, on allume une LED supplémentaire, etc. Pour une valeur entre 700 et 800 on allumera donc... 8 LEDs, bravo à ceux qui suivent ! Ce comportement va donc s'écrire simplement avec une boucle for, qui va incrémenter une variable i de 0 à 10. Dans cette boucle, nous allons tester si la valeur (image de la tension) est inférieure à i multiplié par 100 (ce qui représentera nos différents pas). Si le test vaut VRAI, on allume la LED i, sinon on l'éteint. Démonstration :

```

1 void afficher(int valeur)
2 {

```



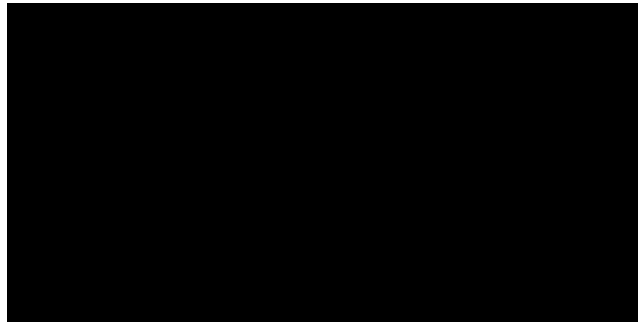
```

3   int i;
4   for(i=0; i < 10; i++)
5   {
6       if(valeur < (i*100))
7           digitalWrite(leds[i], LOW); //on allume la LED
8       else
9           digitalWrite(leds[i], HIGH); //ou on éteint la LED
10  }
11 }

```

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.



```

1  //les broches du décodeur 7 segments
2  const int bit_A = 2;
3  const int bit_B = 3;
4  const int bit_C = 4;
5  const int bit_D = 5;
6
7  //les broches des transistors pour l'afficheur des dizaines et celui des unités
8  const int alim_dizaine = 6;
9  const int alim_unite = 7;
10
11 //la broche du potar
12 const int potar = 0;
13
14 float tension = 0.0; //tension mise en forme
15 int val = 0; //tension brute lue (0 à 1023)
16 bool afficheur = false;
17 long temps;
18
19 void setup()
20 {
21     //Les broches sont toutes des sorties (sauf les boutons)
22     pinMode(bit_A, OUTPUT);
23     pinMode(bit_B, OUTPUT);
24     pinMode(bit_C, OUTPUT);
25     pinMode(bit_D, OUTPUT);
26     pinMode(alim_dizaine, OUTPUT);

```

```

27     pinMode(alim_unite, OUTPUT);
28
29     //Les broches sont toutes mise à l'état bas (sauf led rouge éteinte)
30     digitalWrite(bit_A, LOW);
31     digitalWrite(bit_B, LOW);
32     digitalWrite(bit_C, LOW);
33     digitalWrite(bit_D, LOW);
34     digitalWrite(alim_dizaine, LOW);
35     digitalWrite(alim_unite, LOW);
36     temps = millis(); //enregistre "l'heure"
37 }
38
39 void loop()
40 {
41     //on fait la lecture analogique
42     val = analogRead(potar);
43     //mise en forme de la valeur lue
44     tension = val * 5; //simple relation de trois pour la conversion ( *5/1023)
45     tension = tension / 1023;
46     //à ce stade on a une valeur de type 3.452 Volts... que l'on va multiplier par :
47     tension = tension*10;
48
49     //si ca fait plus de 10 ms qu'on affiche, on change de 7 segments
50     if((millis() - temps) > 10)
51     {
52         //on inverse la valeur de "afficheur" pour changer d'afficheur (unité ou di
53         afficheur = !afficheur;
54         //on affiche
55         afficher_nombre(tension, afficheur);
56         temps = millis(); //on met à jour le temps
57     }
58 }
59
60 //fonction permettant d'afficher un nombre
61 void afficher_nombre(float nombre, bool afficheur)
62 {
63     long temps;
64     char unite = 0, dizaine = 0;
65     if(nombre > 9)
66         dizaine = nombre / 10; //on recupere les dizaines
67     unite = nombre - (dizaine*10); //on recupere les unités
68
69     if(afficheur)
70     {
71         //on affiche les dizaines
72         digitalWrite(alim_unite, LOW);
73         digitalWrite(alim_dizaine, HIGH);
74         afficher(dizaine);
75     }
76     else
77     {
78         //on affiche les unités
79         digitalWrite(alim_dizaine, LOW);
80         digitalWrite(alim_unite, HIGH);
81         afficher(unite);
82     }
83 }
84
85 //fonction écrivant sur un seul afficheur
86 void afficher(char chiffre)
87 {

```

```

88 //on commence par écrire 0, donc tout à l'état bas
89 digitalWrite(bit_A, LOW);
90 digitalWrite(bit_B, LOW);
91 digitalWrite(bit_C, LOW);
92 digitalWrite(bit_D, LOW);
93
94 if(chiffre >= 8)
95 {
96     digitalWrite(bit_D, HIGH);
97     chiffre = chiffre - 8;
98 }
99 if(chiffre >= 4)
100 {
101     digitalWrite(bit_C, HIGH);
102     chiffre = chiffre - 4;
103 }
104 if(chiffre >= 2)
105 {
106     digitalWrite(bit_B, HIGH);
107     chiffre = chiffre - 2;
108 }
109 if(chiffre >= 1)
110 {
111     digitalWrite(bit_A, HIGH);
112     chiffre = chiffre - 1;
113 }
114 //Et voilà !!
115 }

```

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...

[Arduino 403] 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 a 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 entrée analogique 🤖). Tiens, en voilà un projet intéressant ! Je vais me le garder sous la main... :ninja:

Alors ! alors ! alors !! Comment on fait !? 😊

Serait-ce un léger soupçon de curiosité que je perçois dans vos yeux fréillants ? 🤖 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

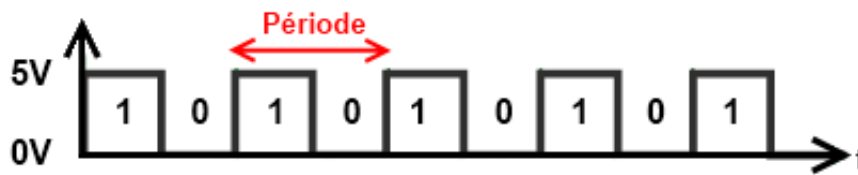
N'ayez point peur, je vais vous expliquer ce que c'est au lieu de vous donner une définition tordue comme on peut en trouver parfois dans les dictionnaires. 😊 D'abord, la PWM se veut dire : **Pulse Width Modulation** et en français cela donne **Modulation à Largeur d'Impulsion** (MLI). La PWM est en 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 ? o_O

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 :



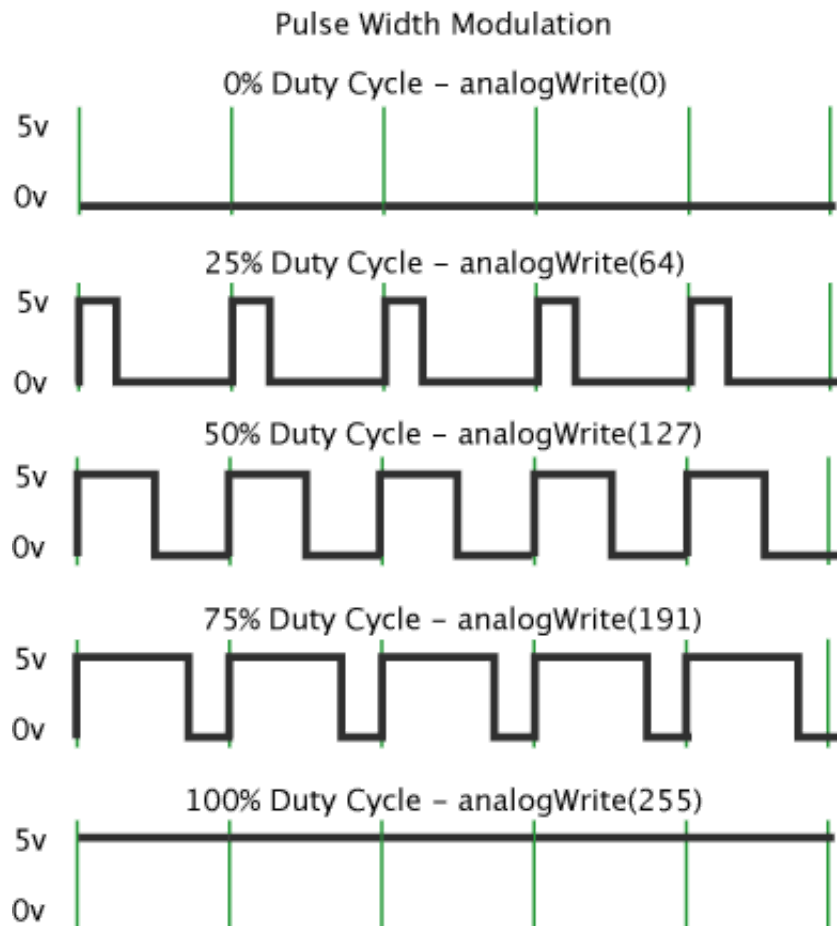
Ensuite, lorsque l'on aura mesuré cette période, on va pouvoir calculer sa fréquence (le nombre de périodes en une seconde) grâce à la formule suivante :

$$F = \frac{1}{T}$$

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 :

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 :

```
1 //une sortie analogique sur la broche 6
2 const int sortieAnalogique = 6;
3
4 void setup()
5 {
6     pinMode(sortieAnalogique, OUTPUT);
7 }
8
9 void loop()
10 {
11     //on met un rapport cyclique de 107/255 = 42 %
12     analogWrite(sortieAnalogique, 107);
13 }
```

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.

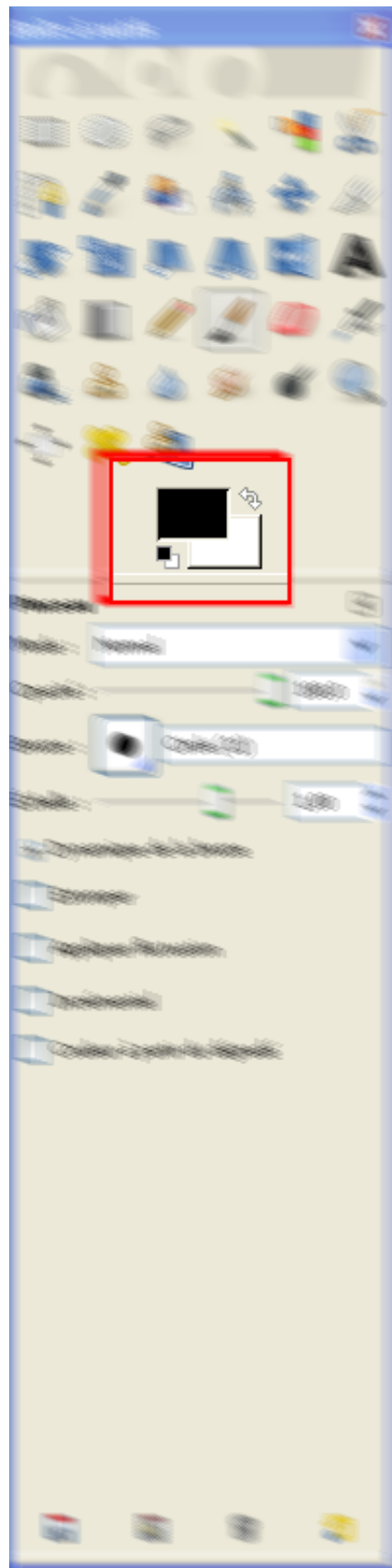
Mixer les couleurs

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 :

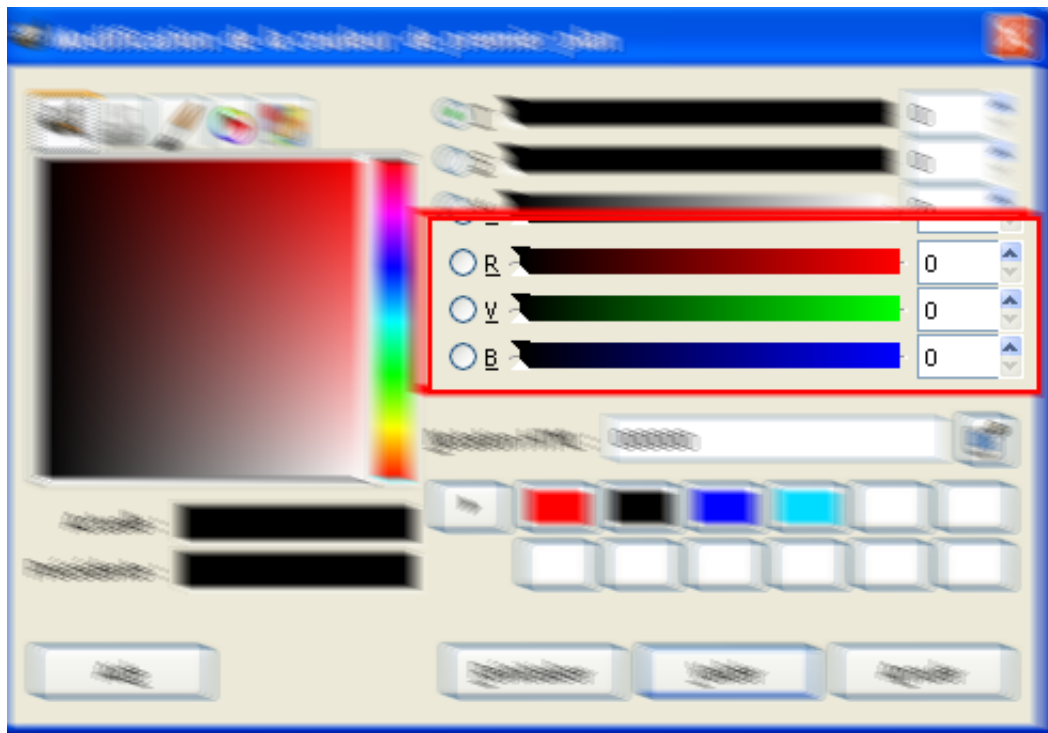
- **Le rouge**
- **Le vert**
- **Le bleu**

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 :

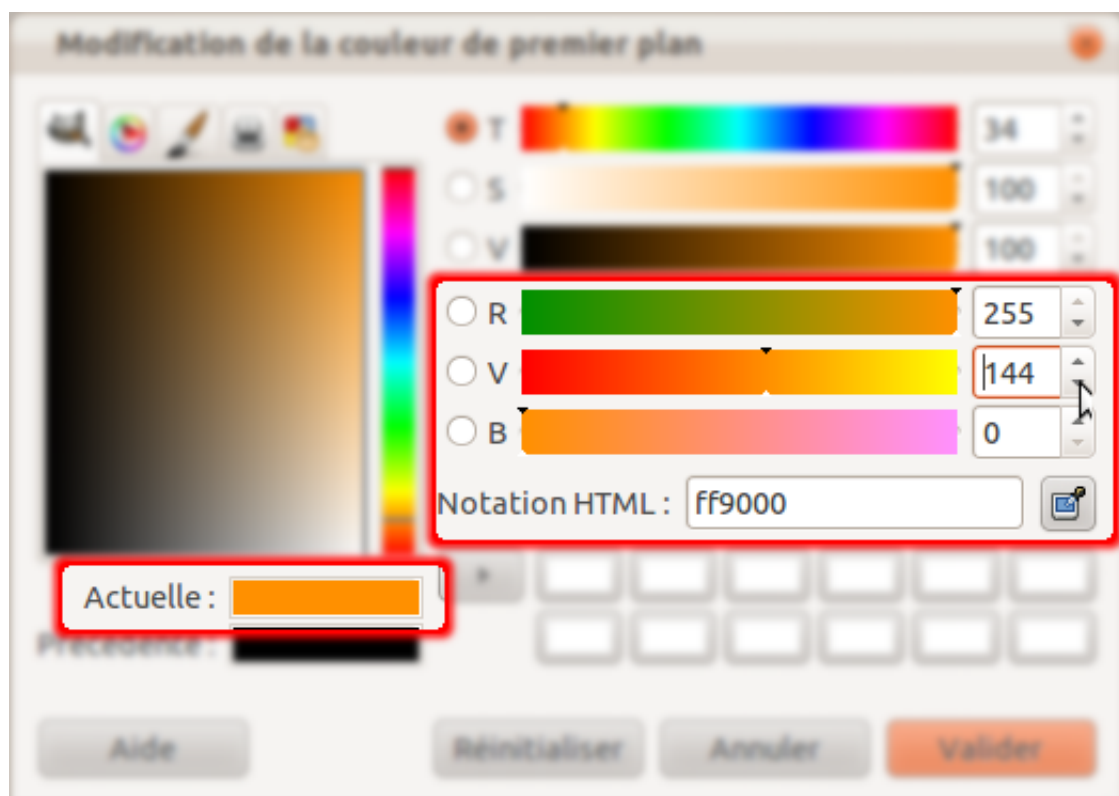
1. Tout d'abord on sélectionne la "boîte à couleurs" dans la boîte à outils
2. Ensuite, en jouant sur les valeurs R, G et B on peut voir la couleur obtenue



→



Afin de faire des jolies couleurs, nous utiliserons `analogWrite()` trois fois (une pour chaque LED). Prenons tout de suite un exemple avec du **orange** et regardons sa composition sous Gimp :



À partir de cette image nous pouvons voir qu'il faut :

- 100 % de rouge (255)
- 56 % de vert (144)
- 0% de bleu (0)

Nous allons donc pouvoir simplement utiliser ces valeurs pour faire une jolie couleur sur notre LED RGB :

```
1  const int ledRouge = 11;
2  const int ledVerte = 9;
3  const int ledBleue = 10;
4
5  void setup()
6  {
7      //on déclare les broches en sorties
8      pinMode(ledRouge, OUTPUT);
9      pinMode(ledVerte, OUTPUT);
10     pinMode(ledBleue, OUTPUT);
11
12     //on met la valeur de chaque couleur
13     analogWrite(ledRouge, 255);
14     analogWrite(ledVerte, 144);
15     analogWrite(ledBleue, 0);
16 }
17
18 void loop()
19 {
20     //on ne change pas la couleur donc rien à faire dans la boucle principale
21 }
```

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 :

```
1  const int ledRouge = 11;
2  const int ledVerte = 9;
3  const int ledBleue = 10;
4
5  void setup()
6  {
7      //on déclare les broches en sorties
8      pinMode(ledRouge, OUTPUT);
9      pinMode(ledVerte, OUTPUT);
10     pinMode(ledBleue, OUTPUT);
11
12     //on met la valeur de chaque couleur
13     analogWrite(ledRouge, 255-255);
14     analogWrite(ledVerte, 255-144);
15     analogWrite(ledBleue, 255-0);
16 }
```

On en a fini avec les rappels, on va pouvoir commencer un petit exercice.

À vos claviers, prêt... programmez !
L'objectif

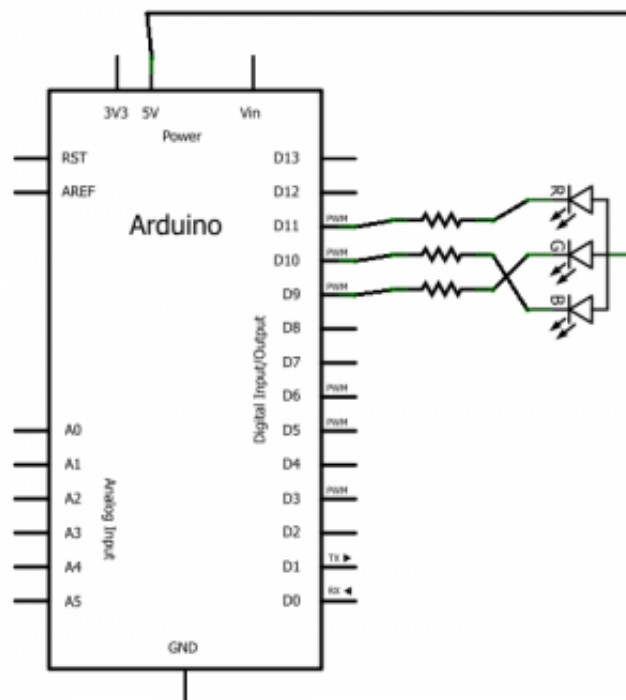
L'objectif est assez simple, vous allez générer trois PWM différentes (une pour chaque LED de couleur) et créer 7 couleurs (le noir ne compte pas ! 😊) distinctes qui sont les suivantes :

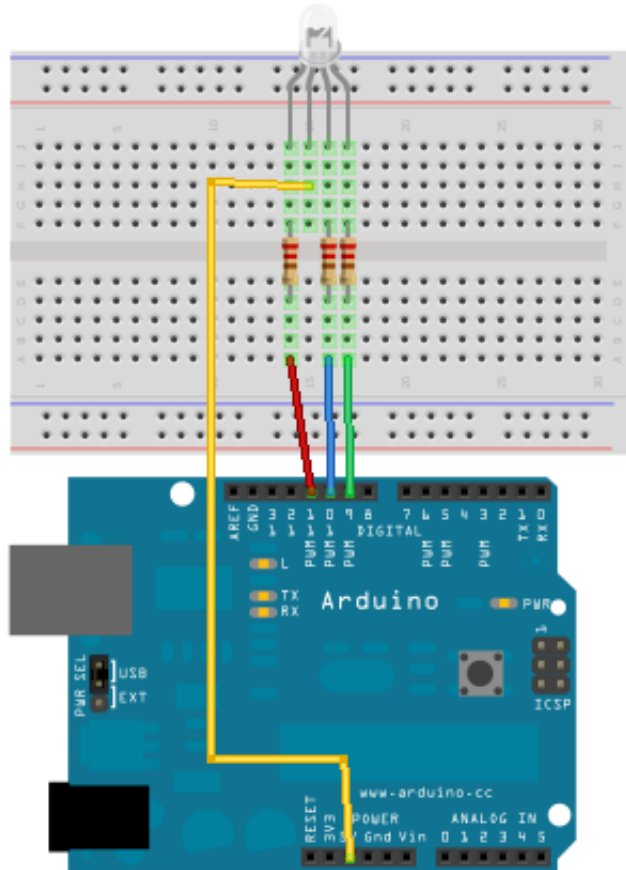
- rouge
- vert
- bleu
- jaune
- bleu ciel
- violet
- blanc

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.





C'est parti ! 😊

Correction

Voilà le petit programme que j'ai fait pour répondre à l'objectif demandé :

```

1 //définition des broches utilisée (vous êtes libre de les changer)
2 const int led_verte = 9;
3 const int led_bleue = 10;
4 const int led_rouge = 11;
5
6 int compteur_defilement = 0; //variable permettant de changer de couleur
7
8
9 void setup()
10 {
11     //définition des broches en sortie
12     pinMode(led_rouge, OUTPUT);
13     pinMode(led_verte, OUTPUT);
14     pinMode(led_bleue, OUTPUT);
15 }
16
17 void loop()
18 {
19     couleur(compteur_defilement); //appel de la fonction d'affichage
20     compteur_defilement++; //incrémentatation de la couleur à afficher
21     if(compteur_defilement > 6) compteur_defilement = 0; //si le compteur dépasse 6 c
22
23     delay(500);
24 }
25
26 void couleur(int numeroCouleur)
27 {

```

```

28     switch(numeroCouleur)
29     {
30     case 0 : //rouge
31         analogWrite(led_rouge, 0); //rapport cyclique au minimum pour une meilleure
32         //qui je le rappel est commandée en "inverse"
33         //(0 -> LED allumée ; 255 -> LED éteinte)
34         analogWrite(led_verte, 255);
35         analogWrite(led_bleue, 255);
36         break;
37     case 1 : //vert
38         analogWrite(led_rouge, 255);
39         analogWrite(led_verte, 0);
40         analogWrite(led_bleue, 255);
41         break;
42     case 2 : //bleu
43         analogWrite(led_rouge, 255);
44         analogWrite(led_verte, 255);
45         analogWrite(led_bleue, 0);
46         break;
47     case 3 : //jaune
48         analogWrite(led_rouge, 0);
49         analogWrite(led_verte, 0);
50         analogWrite(led_bleue, 255);
51         break;
52     case 4 : //violet
53         analogWrite(led_rouge, 0);
54         analogWrite(led_verte, 255);
55         analogWrite(led_bleue, 0);
56         break;
57     case 5 : //bleu ciel
58         analogWrite(led_rouge, 255);
59         analogWrite(led_verte, 0);
60         analogWrite(led_bleue, 0);
61         break;
62     case 6 : //blanc
63         analogWrite(led_rouge, 0);
64         analogWrite(led_verte, 0);
65         analogWrite(led_bleue, 0);
66         break;
67     default : //"noir"
68         analogWrite(led_rouge, 255);
69         analogWrite(led_verte, 255);
70         analogWrite(led_bleue, 255);
71         break;
72     }
73 }

```

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.

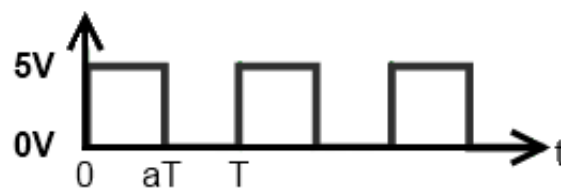
Cependant, avant de continuer, je tiens à vous informer que l'on va aborder des notions plus profondes en électronique et que vous n'êtes pas obligé de lire cette sous-partie si vous ne vous en sentez pas capable. Revenez plus tard si vous le voulez. Pour ceux qui cela intéresserait vraiment, je ne peux 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 un 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 apparaître les temps. On observe donc que du temps 0 (l'origine) au temps T , on a une période du signal. aT correspond au moment où le signal change d'état. En somme, il s'agit du temps de l'état haut, qui donne aussi le temps à l'état bas et finalement permet de calculer le rapport cyclique du signal. Donnons quelques valeurs numériques à titre d'exemple :

- $T = 1ms$
- $a = 0.5$ (correspond à un rapport cyclique de 50%)

La formule permettant de calculer la valeur moyenne de cette période est la suivante :

$$= \frac{U_1 \times aT + U_2 \times (T - aT)}{T}$$

La valeur moyenne d'un signal se note avec des chevrons $\langle \rangle$ autour de la lettre indiquant de quelle grandeur physique il s'agit.

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 : $= a \times U_1 + U_2 - a \times U_2$. Et cela se simplifie encore en :

$$= a \times (U_1 - U_2) + U_2 \quad [/latex]$$

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 : $= a \times U_1$, 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 maîtrisée, on va pouvoir créer un signal analogique de la forme qu'on le souhaite, compris entre 0 et 5V, en extrayant la valeur moyenne du signal. On retiendra également que, dans cette formule uniquement, le temps n'a pas d'importance.

Extraire cette valeur moyenne

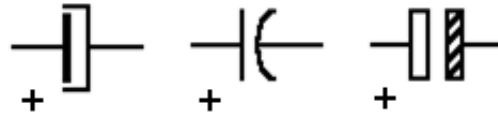
Alors, mais comment faire pour extraire la valeur moyenne du signal de la PWM, me direz-vous. Eh bien on va utiliser les propriétés d'un certain couple de composants très connu : le **couple RC** ou **résistance-condensateur**.

La résistance on connaît, mais, le condensateur... tu nous avais pas dit qu'il servait à supprimer les parasites ? o_O

Si, bien sûr, mais il possède plein de caractéristiques intéressantes. C'est pour cela que c'est un des composants les plus utilisés en électronique. Cette fois, je vais vous montrer une de ses caractéristiques qui va nous permettre d'extraire cette fameuse valeur moyenne.

Le condensateur

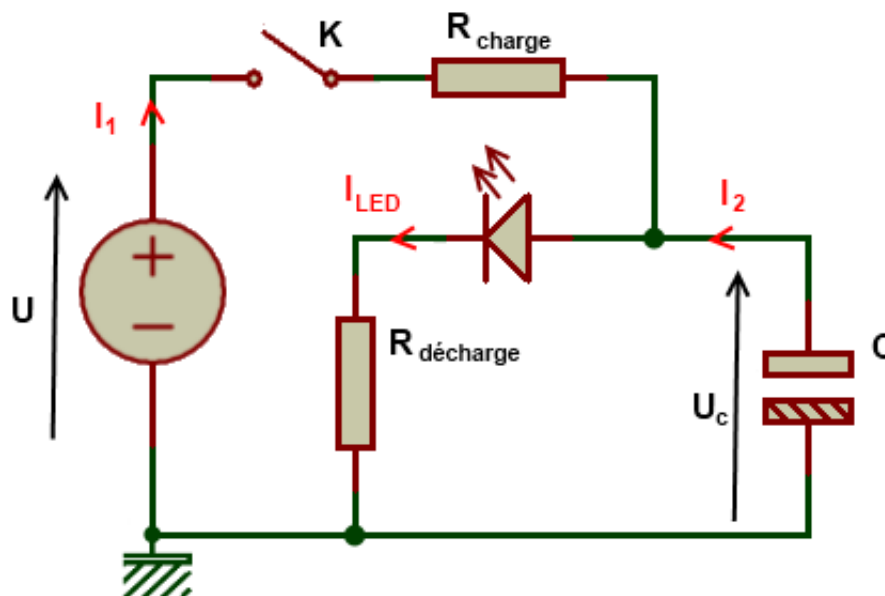
Je vous ai déjà parlé de la résistance, vous savez qu'elle limite le courant suivant la loi d'Ohm. Je vous ai aussi parlé du condensateur, je vous disais qu'il absorbait les parasites créés lors d'un appui sur un bouton poussoir. À présent, on va voir un peu plus en profondeur son fonctionnement car on est loin d'avoir tout vu ! Le condensateur,



je rappelle ses symboles : est constitué de deux plaques métalliques, des **armatures**, posées face à face et isolées par... un isolant ! 😊
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 ?

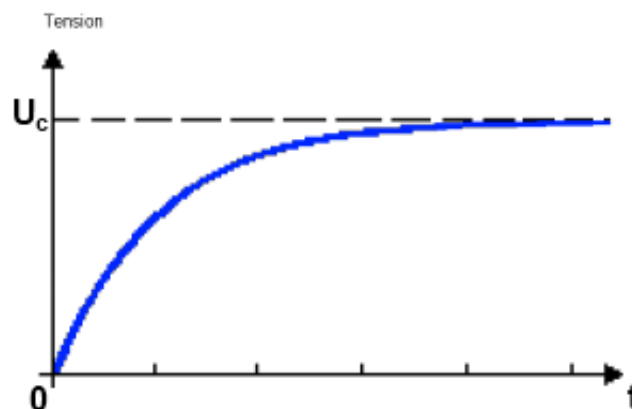
Lorsque vous fermez l'interrupteur, le courant peut s'établir dans le circuit. Il va donc aller allumer la LED. Ceci fait abstraction du condensateur. Mais, justement, dans ce montage il y a un condensateur. Qu'observez-vous ? La LED ne s'allume pas immédiatement et met un peu de temps avant d'être complètement 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 ? 😊

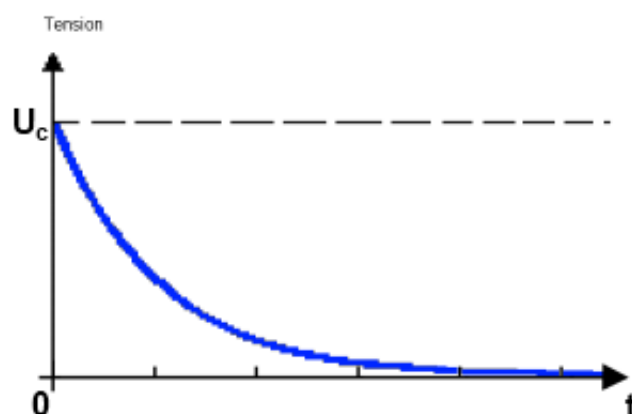
Vous pouvez réitérer l'expérience en changeant la valeur des composants, sans jamais descendre en dessous de 220 Ohm pour la résistance de décharge.

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 croît, moins il aime ça et plus il met du temps à l'accepter. Si on regarde la tension aux bornes de ce pauvre condensateur, on peut observer ceci :



La tension augmente 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 :

- τ : (prononcez "to") temps de charge/décharge en secondes (s)
- R : valeur de la résistance en Ohm (Ω)
- C : valeur de la capacité du condensateur en Farad (F)

Cette formule donne le temps τ qui correspond à 63% de la charge à la tension appliquée au condensateur. On considère que le condensateur est complètement chargé à partir de 3τ (soit 95% de la tension de charge) ou 5τ (99% de la tension de charge).

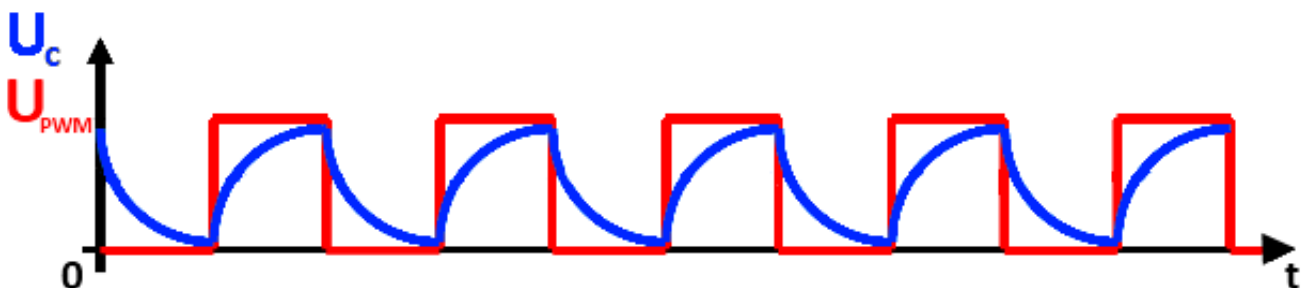
Imposons notre PWM !

Bon, très bien, mais quel est le rapport avec la PWM ?

Ha, haa ! Alors, pour commencer, vous connaissez la réponse.

Depuis quand ? o_O

Depuis que je vous ai donné les explications précédentes. Dès que l'on aura imposé notre PWM au couple RC, il va se passer quelque chose. Quelque chose que je viens de vous expliquer. À chaque fois que le signal de la PWM sera au NL 1, 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 :

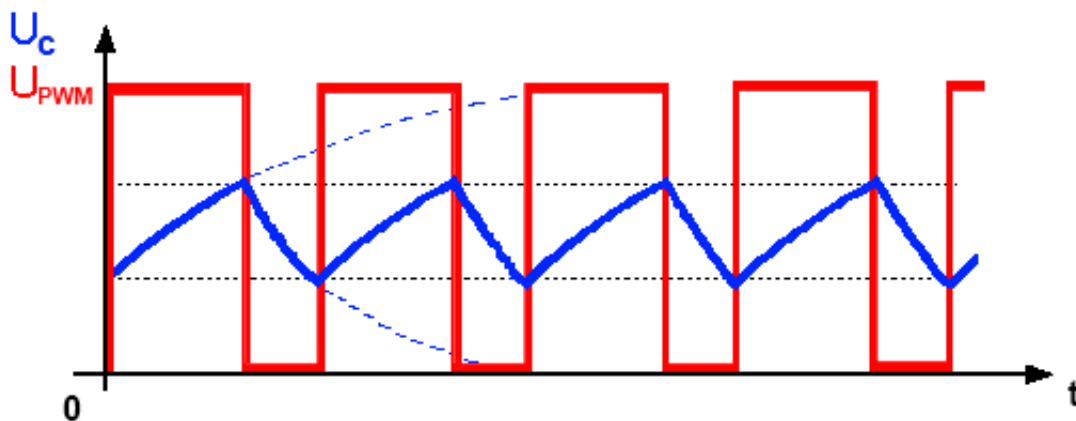


Qu'y a-t-il de nouveau par rapport au signal carré, à part sa forme bizarroïde !?

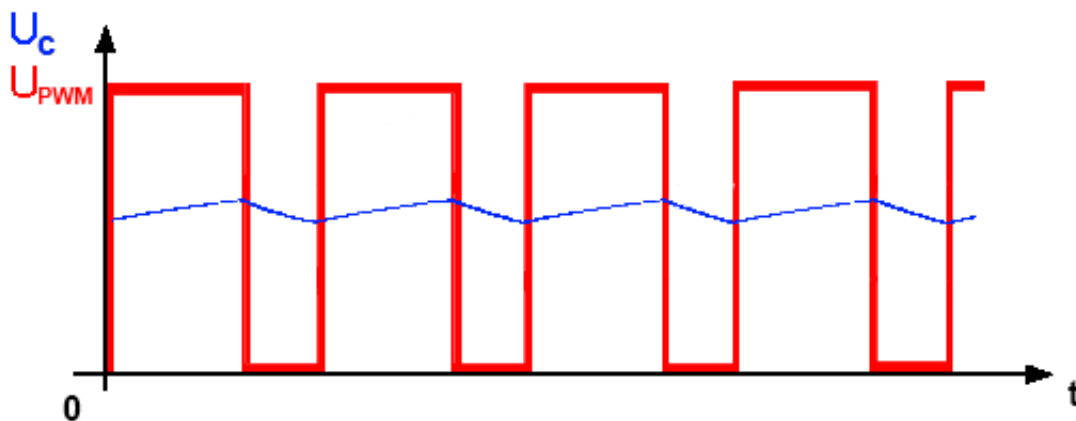
Dans ce cas, rien de plus, si on calcule la valeur moyenne du signal bleu, on trouvera la même valeur que pour le signal rouge. (Ne me demandez pas pourquoi, c'est comme ça, c'est une formule très compliquée qui le dit 😊). Précisons que dans ce cas, encore une fois, le temps de charge/décharge 3τ du condensateur est choisi de façon à ce qu'il soit égal à une demi-période du signal. Que se passera-t-il si on choisit un temps de charge/décharge plus petit ou plus grand ?

Constante de temps τ supérieure à la période

Voilà le chronogramme lorsque la constante de temps de charge/décharge du condensateur est plus grande que la période du signal :



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 !!

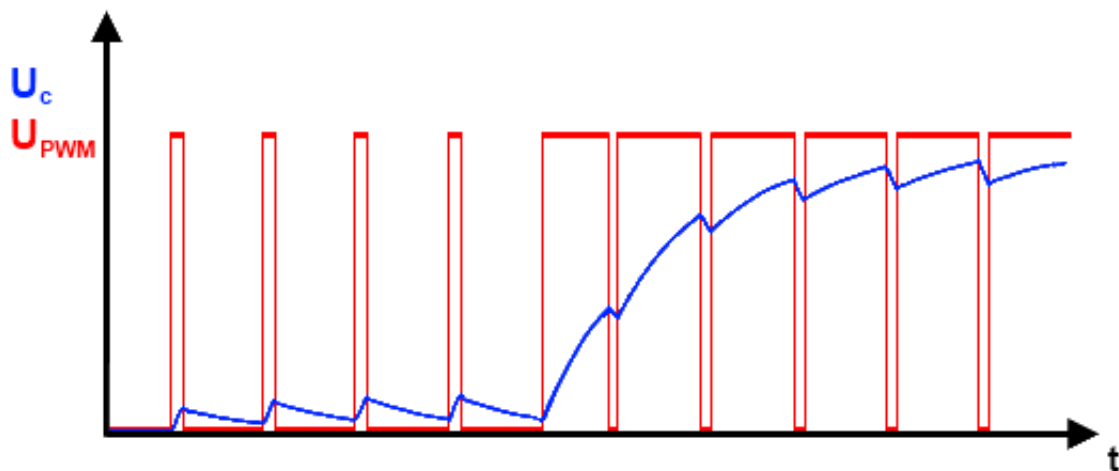


Calibrer correctement la constante RC

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.

Enfin, 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

Modifier la fréquence de la PWM

On l'a vu, avec la fréquence actuelle de la PWM en sortie de l'Arduino, on va ne pouvoir créer que des signaux "lents". Lorsque vous aurez besoin d'aller plus vite, vous vous confronterez à ce problème. C'est pourquoi je vais vous expliquer comment modifier la

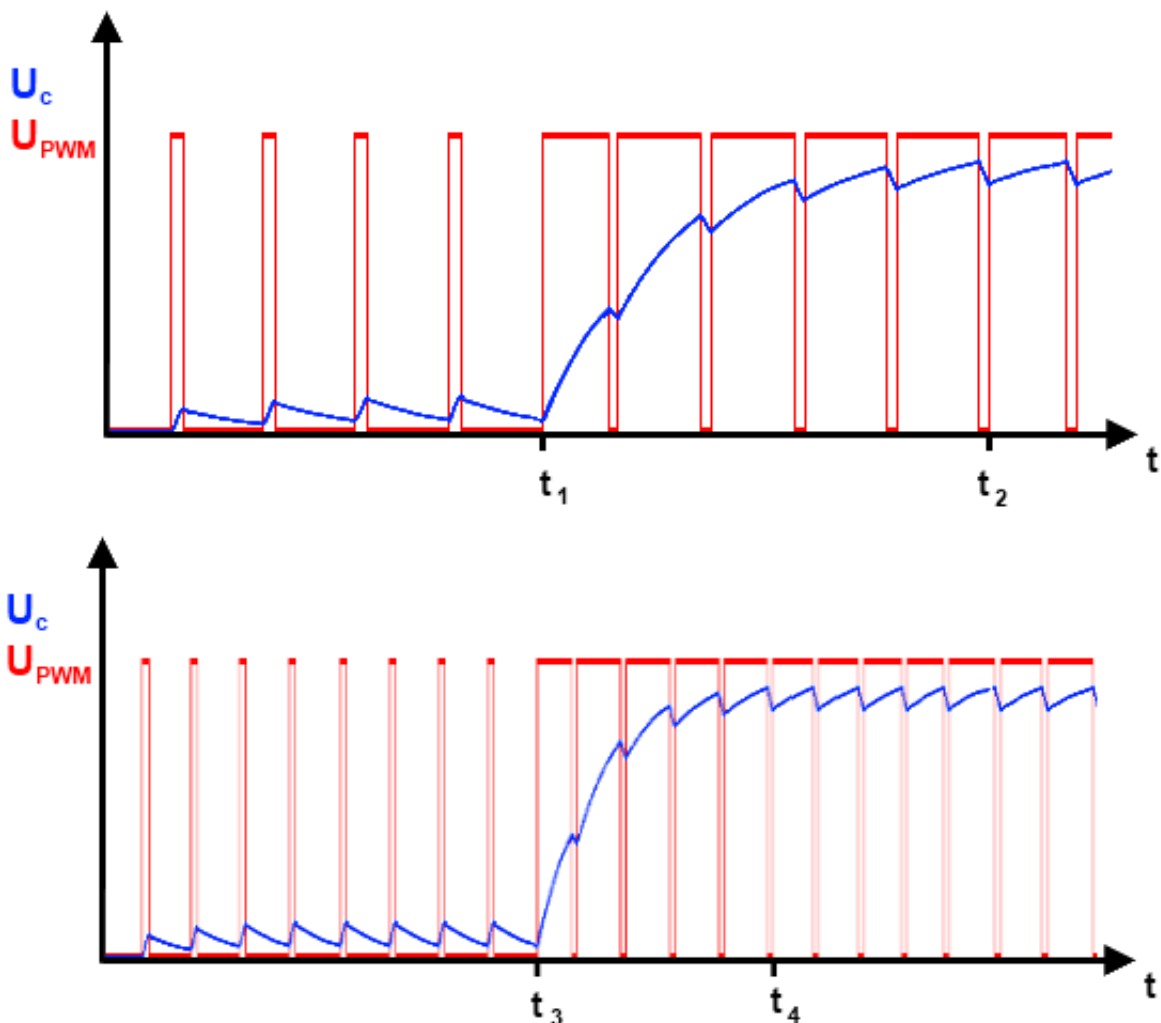
fréquence de cette PWM.

Nouveau message d'avertissement : cette fois, on va directement toucher aux registres du microcontrôleur, donc si vous comprenez pas tout, ce n'est pas très grave car cela requiert un niveau encore plus élevé que celui que vous avez actuellement.

Commençons cette très courte sous-partie.

Pourquoi changer la fréquence de la PWM ?

Oui, pourquoi ? Tout simplement pour essayer de créer un signal qui se rapproche le plus de la valeur moyenne de la PWM à chaque instant. L'objectif est de pouvoir maximiser l'avantage de la structure ayant une faible constante de temps tout en éliminant au mieux son désavantage. Vous verrez peut-être mieux avec des chronogrammes. En voici deux, le premier est celui où la fréquence de la PWM est celle fournie d'origine par l'Arduino, le second est la PWM à une fréquence deux fois plus élevée, après modification du programme :



Pour une constante de temps identique pour chaque courbe réalisée, on relève que le temps de stabilisation du signal est plus rapide sur le chronogramme où la fréquence est deux fois plus élevée qu'avec la fréquence standard d'Arduino. Ici donc, on a : $t_2 - t_1 = 2 \times (t_4 - t_3)$. En effet car le temps (T) est proportionnel à la fréquence (F)

selon cette formule : $F = \frac{1}{T}$ Avec quelques mots pour expliquer cela, le temps de charge du condensateur, pour se stabiliser au nouveau palier de tension, est plus rapide avec une fréquence plus élevée. À comparaison, pour le premier signal, le temps de charge est deux fois plus grand que celui pour le deuxième signal où la fréquence est deux fois plus élevée.

Mes dessins ne sont pas très rigoureux, mais mes talents de graphistes me limitent à ça. Soyez indulgent à mon égard. 😊 Quoi qu'il en soit, il s'agissait, ici, simplement d'illustrer mes propos et donner un exemple concret.

Utilisation du registre

Bigre ! Je viens de comprendre pourquoi on avait besoin de changer la fréquence de la PWM. 😊 Mais euh... comment on fait ? C'est quoi les registres ? 😊

Les registres..... eh bien..... c'est compliqué ! :ninja: Non, je n'entrerai pas dans le détail en expliquant ce qu'est un registre, de plus c'est un sujet que je ne maîtrise pas bien et qui vous sera certainement inutile dans le cas présent. Disons pour l'instant que le registre est une variable très spéciale.

Code de modification de la fréquence de la PWM

Alors, pour modifier la fréquence de la PWM de l'Arduino on doit utiliser le code suivant :

```
1 //on définit une variable de type byte
2 //qui contiendra l'octet à donner au registre pour diviser la fréquence de la PWM
3
4 //division par : 1, 8, 64, 256, 1024
5 byte division_frequence=0x01;
6 //fréquence : 62500Hz, 7692Hz, ...
7 //temps de la période : 16µs, 130µs, ...
8
9 void setup()
10 {
11     pinMode(6, OUTPUT); //broche de sortie
12
13     //TCCR0B c'est le registre, on opère un masquage sur lui même
14     TCCR0B = TCCR0B & 0b11111000 | division_frequence;
15     //ce qui permet de modifier la fréquence de la PWM
16 }
17
18 void loop ()
19 {
20     //on écrit simplement la valeur de 0 à 255 du rapport cyclique du signal
21     analogWrite(6, 128);
22     //qui est à la nouvelle fréquence choisit
23 }
```

Vous remarquerez que les nombres binaires avec Arduino s'écrivent avec les caractères **0b** devant.

Cette sous-partie peut éventuellement être prise pour un *truc et astuce*. C'est

quelque peu le cas, malheureusement, mais pour éviter que cela ne le soit complètement, je vais vous expliquer des notions supplémentaires, par exemple la ligne 12 du code.

Traduction s'il vous plait !

Je le disais donc, on va voir ensemble comment fonctionne la ligne 12 du programme :

```
1 TCCR0B = TCCR0B & 0b11111000 | division_frequence;
```

Très simplement, **TCCR0B** est le nom du registre utilisé. Cette ligne est donc là pour modifier le registre puisqu'on fait une opération avec et le résultat est inscrit dans le registre. Cette opération est, il faudra l'avouer, peu commune. On effectue, ce que l'on appelle en programmation, un **masquage**. Le masquage est une opération logique que l'on utilise avec des données binaires. On peut faire différents masquages en utilisant les opérations logiques ET, OU, OU exclusif, ... Dans le cas présent, on a la variable TCCR0B qui est sous forme binaire et on effectue une opération **ET** (symbole **&**) puis une opération **OU** (symbole **|**). Les opérations ET et OU sont définies par une **table de vérité**. Avec deux entrées, on a une sortie qui donne le résultat de l'opération effectuée avec les deux niveaux d'entrée.

Entrées Sortie ET Sortie OU

A	B	A ET B	A OU B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

Les opérations de type ET et OU ont un niveau de priorité comme la multiplication et l'addition. On commence toujours par effectuer l'opération ET, puis on termine avec l'opération OU. On pourrait donc écrire ceci :

```
1 TCCR0B = (TCCR0B & 0b11111000) | division_frequence;
```

Prenons maintenant un exemple où la variable spéciale TCCR0B serait un nombre binaire égal à :

```
1 TCCR0B = 0b10011101; //valeur du registre
```

À présent, on lui fait un masquage de type ET :

```
1 TCCR0B = 0b10011101;
2
3 TCCR0B = TCCR0B & 0b11111000; //masquage de type ET
```

On fait l'opération à l'aide de la table de vérité du ET (voir tableau ci-dessus) et on trouve le résultat :

```
1 TCCR0B = 0b10011101;
2
3 TCCR0B = TCCR0B & 0b11111000;
```



```
4
5 //TCCR0B vaut maintenant : 0b10011000
```

En somme, on conclut que l'on a gardé la valeur des 5 premiers bits, mais l'on a effacé la valeur des 3 derniers bits pour les mettre à zéro. Ainsi, quelle que soit la valeur binaire de TCCR0B, on met les bits que l'on veut à l'état bas. Ceci va ensuite nous permettre de changer l'état des bits mis à l'état bas en effectuant l'opération OU :

```
1 byte division_frequence = 0x01; //nombre haxadécimal qui vaut 0b00000001
2
3 TCCR0B = 0b10011101;
4
5 TCCR0B = TCCR0B & 0b11111000;
6 //TCCR0B vaut maintenant : 0b10011000
7
8 TCCR0B = TCCR0B | division_frequence;
9 //TCCR0B vaut maintenant : 0b10011001
```

D'après la table de vérité du OU logique, on a modifié la valeur de TCCR0B en ne changeant que le ou les bits que l'on voulait.

La valeur de TCCR0B que je vous ai donnée est bidon. C'est un exemple qui vous permet de comprendre comment fonctionne un masquage.

Ce qu'il faut retenir, pour changer la fréquence de la PWM, c'est que pour la variable `division_frequence`, il faut lui donner les valeurs hexadécimales suivantes :

```
1 0x01 //la fréquence vaut 62500Hz (fréquence maximale fournie par la PWM => provient d
2 //effectue une division par 1 de la fréquence max
3
4 0x02 //f = 7692Hz (division par 8 de la fréquence maximale)
5 0x03 //f = 976Hz, division par 64
6 0x04 //f = 244Hz, division par 256
7 0x05 //f = 61Hz, division par 1024
```

Vous trouverez plus de détails sur [cette page](#) (en anglais).

Test de vérification

Pour vérifier que la fréquence a bien changé, on peut reprendre le montage que l'on a fait plus haut et enlever l'interrupteur en le remplaçant par un fil. On ne met plus un générateur de tension continue, mais on branche une sortie PWM de l'arduino avec le programme qui va bien. Pour deux fréquences différentes, on devrait voir la LED s'allumer plus ou moins rapidement. On compare le temps à l'état au lorsque l'on écrit 1000 fois un niveau de PWM à 255 à celui mis par le même programme avec une fréquence de PWM différente, grâce à une LED.

À partir de maintenant, vous allez pouvoir faire des choses amusantes avec la PWM. Cela va nous servir pour les moteurs pour ne citer qu'eux. Mais avant, car on en est pas encore là, je vous propose un petit TP assez sympa. Rendez-vous au prochain chapitre ! 😊

[Arduino 404] [Exercice] Une animation “YouTube”

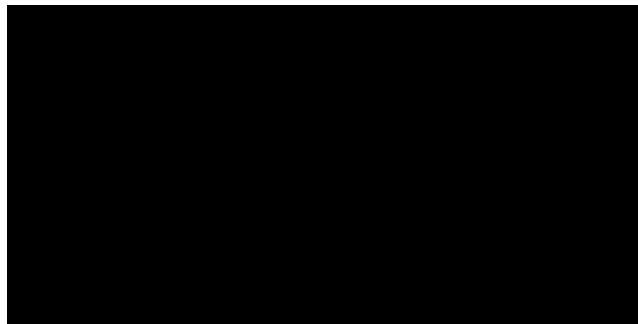
Dans ce petit exercice, je vous propose de faire une animation que vous avez tous vu au moins une fois dans votre vie : le .gif de chargement YouTube ! Pour ceux qui se posent des questions, nous n'allons pas faire de Photoshop ou quoi que ce soit de ce genre. Non, nous (vous en fait 😊) allons le faire ... 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 !

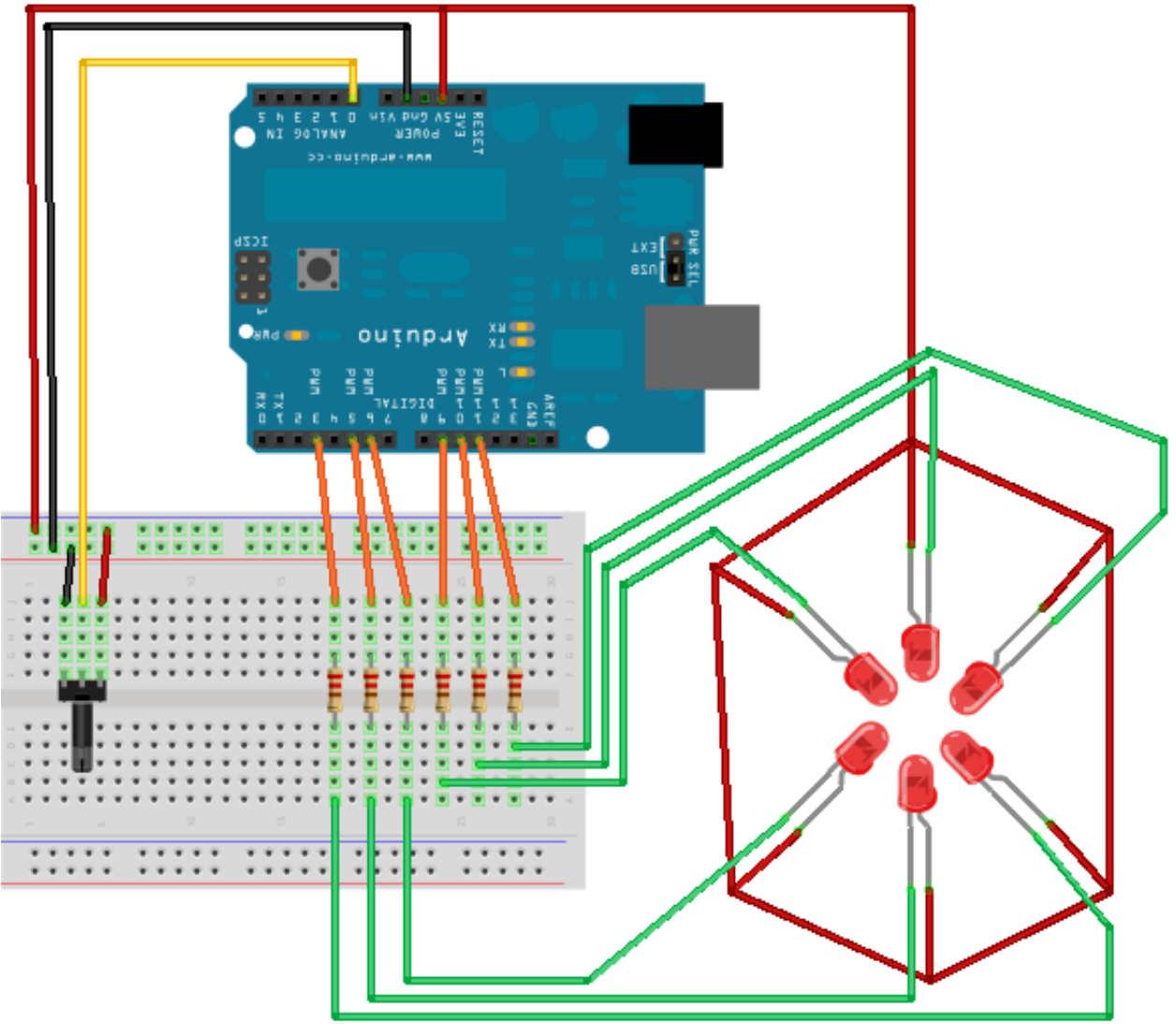
- 6 LED + leurs résistances de limitation de courant
- Un potentiomètre
- Une Arduino, une breadboard et des fils !

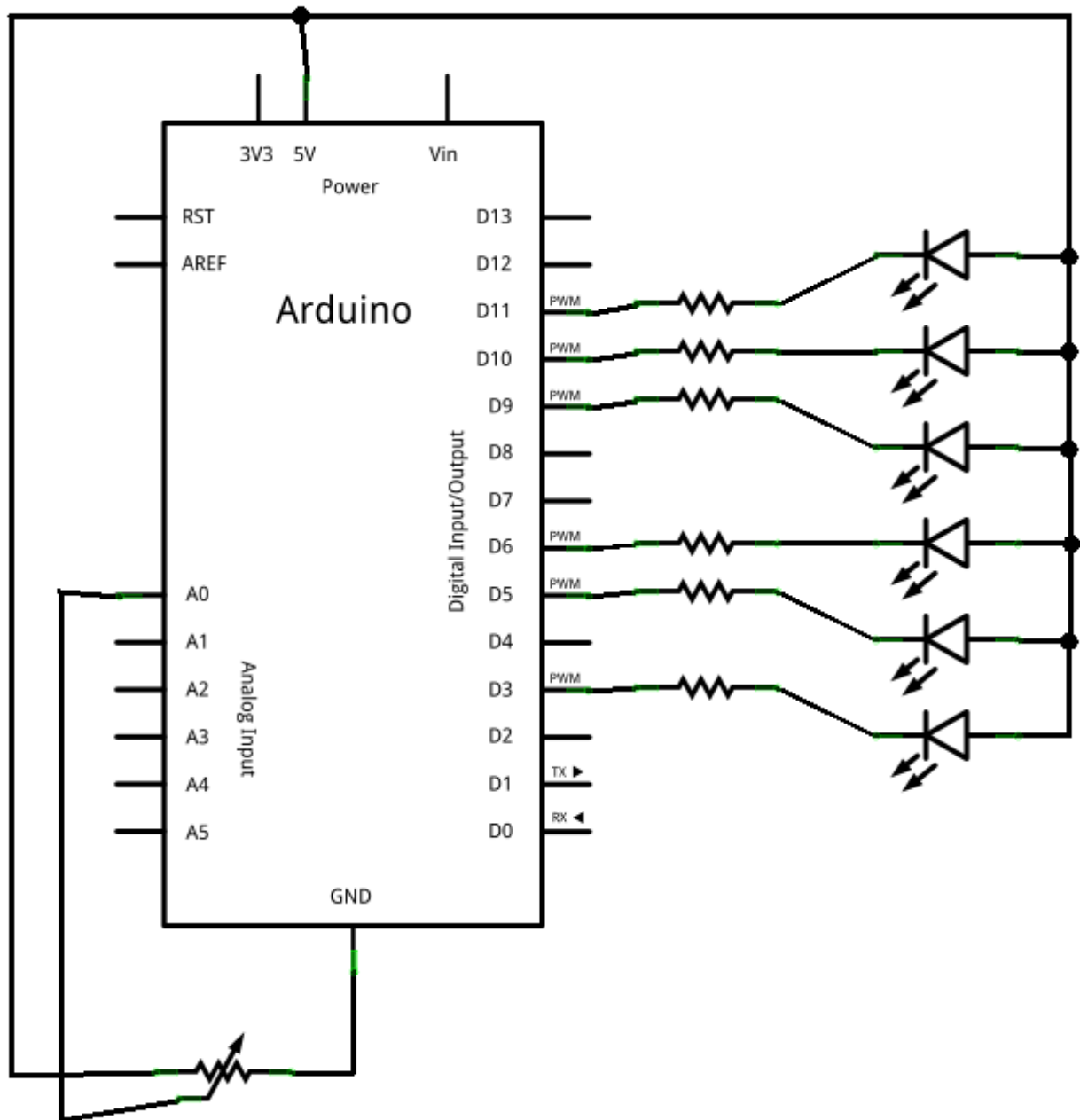


Solution

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).





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 :

```
1 //sortie LEDs
2 const int LED[6] = {3,5,6,9,10,11};
3 //niveaux de luminosité utilisé
4 const char pwm[6] = {255,210,160,200,220,240};
5 //potentiometre sur la broche 0
6 const int potar = 0;
```

Le setup

Personne ne devrait se tromper dans cette fonction, on est dans le domaine du connu, vu et revu ! Il nous suffit juste de mettre en entrée le potentiomètre sur son convertisseur analogique et en sortie mettre les LED (une simple boucle `for` suffit grâce au tableau 😊).

```
1 void setup()
2 {
3     //le potentiomètre en entrée
4     pinMode(potar, INPUT);
5     //les LEDs en sorties
6     for(int i=0; i<6; i++)
7         pinMode(LED[i], OUTPUT);
8 }
```

La loop

Passons au cœur du programme, la boucle `loop()` ! Je vais vous la divulguer dès maintenant puis l'expliquer ensuite :

```
1 void loop()
2 {
3     //étape de l'animation
4     for(int i=0; i<6; i++)
5     {
6         //mise à jour des LEDs
7         for(int n=0; n<6; n++)
8         {
9             analogWrite(LED[n], pwm[(n+i)%6]);
10        }
11        //récupère le temps
12        int temps = analogRead(potar);
13        //tmax = 190ms, tmin = 20ms
14        delay(temps/6 + 20);
15    }
16 }
```

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 (puisque l'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 rappellent 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 cases dans mon tableau. Mais comment obtenir la 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 à 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 demandent 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).

```
1 //sortie LEDs
2 const int LED[6] = {3,5,6,9,10,11};
3 //niveaux de luminosité utilisé
4 const char pwm[6] = {255,210,160,200,220,240};
5 //potentiometre sur la broche 0
6 const int potar = 0;
7
8 void setup()
9 {
10     pinMode(potar, INPUT);
11     for(int i=0; i<6; i++)
12         pinMode(LED[i], OUTPUT);
13 }
14
15 void loop()
16 {
17     //étape de l'animation
18     for(int i=0; i<6; i++)
19     {
20         //mise à jour des LEDs
21         for(int n=0; n<6; n++)
22         {
23             analogWrite(LED[n], pwm[(n+i)%6]);
24         }
25         //récupère le temps
26         int temps = analogRead(potar);
27         //tmax = 190ms, tmin = 20ms
28         delay(temps/6 + 20);
29     }
30 }
```

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... 😊