Eskimon

Zeste de savoir

Ajouter des sorties numériques à l'Arduino, le 74HC595

21 décembre 2018

Table des matières

1. Introduction

Dans ce petit tutoriel, nous allons découvrir comment ajouter des sorties numériques à une carte Arduino. En effet, pour vos projets les plus fous, vous serez certainement amené à avoir besoin d'un grand nombre de sorties. Deux choix s'offrent alors à vous : le premier serait d'opter pour une carte Arduino qui dispose de plus de sorties, telle que la Arduino Mega ; mais dans le cas où vous aurez besoin d'un énorme nombre de sorties, même la Mega ne pourrait suffire. Le deuxième choix c'est donc... de lire ce tuto !

Ce que vous allez découvrir se révélera fort utile, soyez-en certains.

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

2. Présentation du 74HC595

2.1. Principe

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

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

2.2. Le composant

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

2.2.1. Brochage

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

PINNING

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

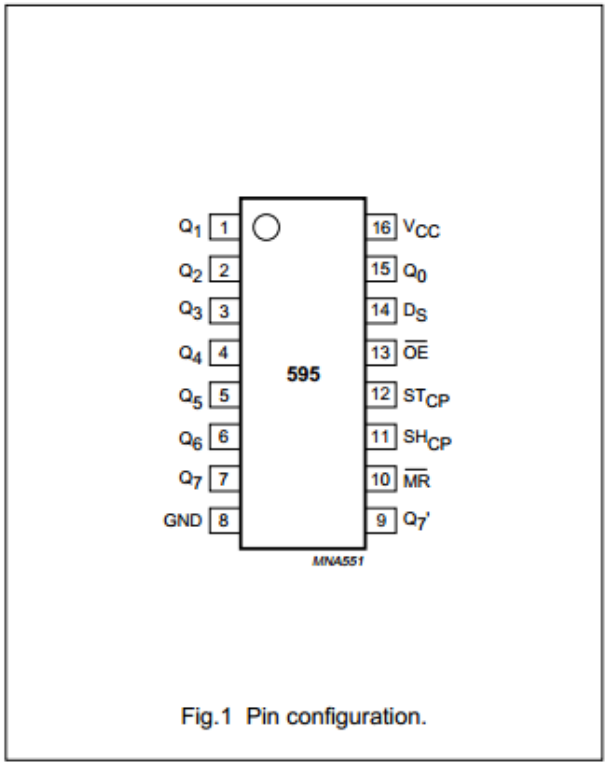

Fig.1 Pin configuration.

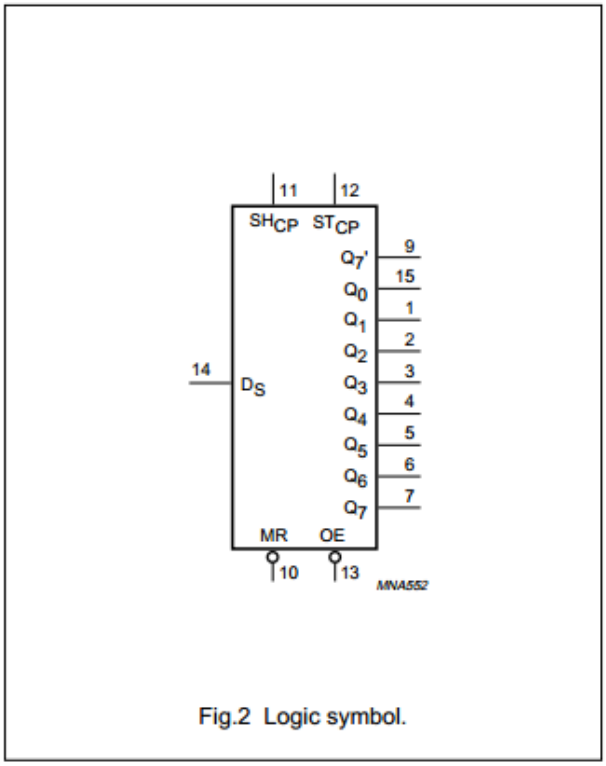

Fig.2 Logic symbol.

FIGURE 2. – Brochage du 74HC595

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

2.2.2. Fonctionnement

Comme tout composant électronique, il faut commencer par l'alimenter pour le faire fonctionner. Le tableau que nous avons vu juste au-dessus nous indique que les broches d'alimentation sont la broche 16 (VCC) et la broche 8 (masse). Quelques pages plus loin dans la datasheet, page 7 précisément, nous voyons la tension à appliquer pour l'alimenter : entre 2V et 5.5V (et idéalement 5.0V).

Une fois que ce dernier est alimenté, il faut se renseigner sur le rôle des broches pour savoir comment l'utiliser correctement. Pour cela il faut revenir sur le tableau précédent et la table de vérité qui le suit. On découvre donc que les sorties sont les broches de 1 à 7 et la broche 15 (*Qn*) ; l'entrée des données série, qui va commander les sorties du composant, se trouve sur la broche 14 (*serial data input*) ; une sortie particulière est disponible sur la broche 9 (*serial data output*, nous y reviendrons à la fin de ce chapitre). Sur la broche 10 on trouve le *Master Reset*, pour mettre à zéro toutes les sorties. Elle est active à l'état BAS. Vous ferez alors attention, dans le cas où vous utiliseriez cette sortie, de la forcer à un état logique HAUT, en la reliant par exemple au +5V ou bien à une broche de l'Arduino que vous ne mettrez à l'état BAS que lorsque vous voudrez mettre toutes les sorties du 74HC595 à l'état bas. Nous, nous mettrons cette sortie sur le +5V.

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

La n°11 et la n°12. Ce sont des "horloges". Nous allons expliquer quelles fonctions elles remplissent. Lorsque nous envoyons un ordre au 74HC595, nous envoyons cet ordre sous forme d'états logiques qui se suivent. Par exemple l'ordre 01100011. Cet ordre est composé de 8 états logiques, ou bits, et forme un octet (une suite de 8 bits). Cet ordre va précisément définir l'état de sortie de chacune des sorties du 74HC595. Le problème c'est que ce composant ne peut pas dissocier chaque bit qui arrive...

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

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

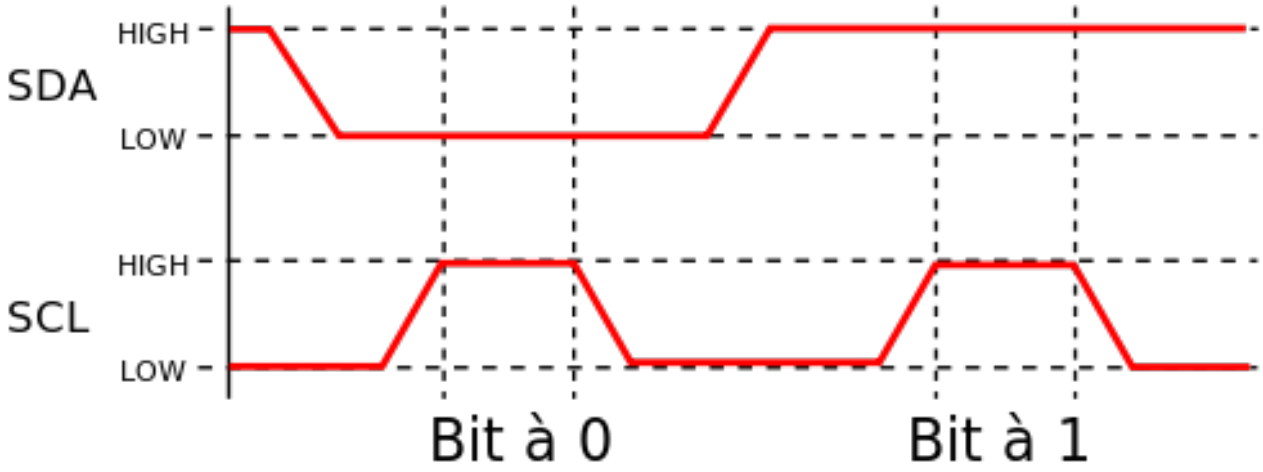

FIGURE 2. – Signaux

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

Pour câbler cette horloge, il faudra connecter une broche de l'Arduino à la broche numéro 11 du 74HC595. Ce signal travaillera donc en corrélation avec le signal de données relié sur la broche 14 du composant. La seconde horloge pourrait aussi s'appeler "verrou". Elle sert à déterminer si le composant doit mettre à jour les états de ses sorties ou non, en fonction de l'ordre qui est transmis. Lorsque ce signal passe de l'état BAS à l'état HAUT, le composant change les niveaux logiques de ses sorties en fonction des bits de données reçues. En clair, il copie les huit derniers bits transmis sur ses sorties. Ce verrou se présente sur la broche 12.

2.2.3. Montage

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

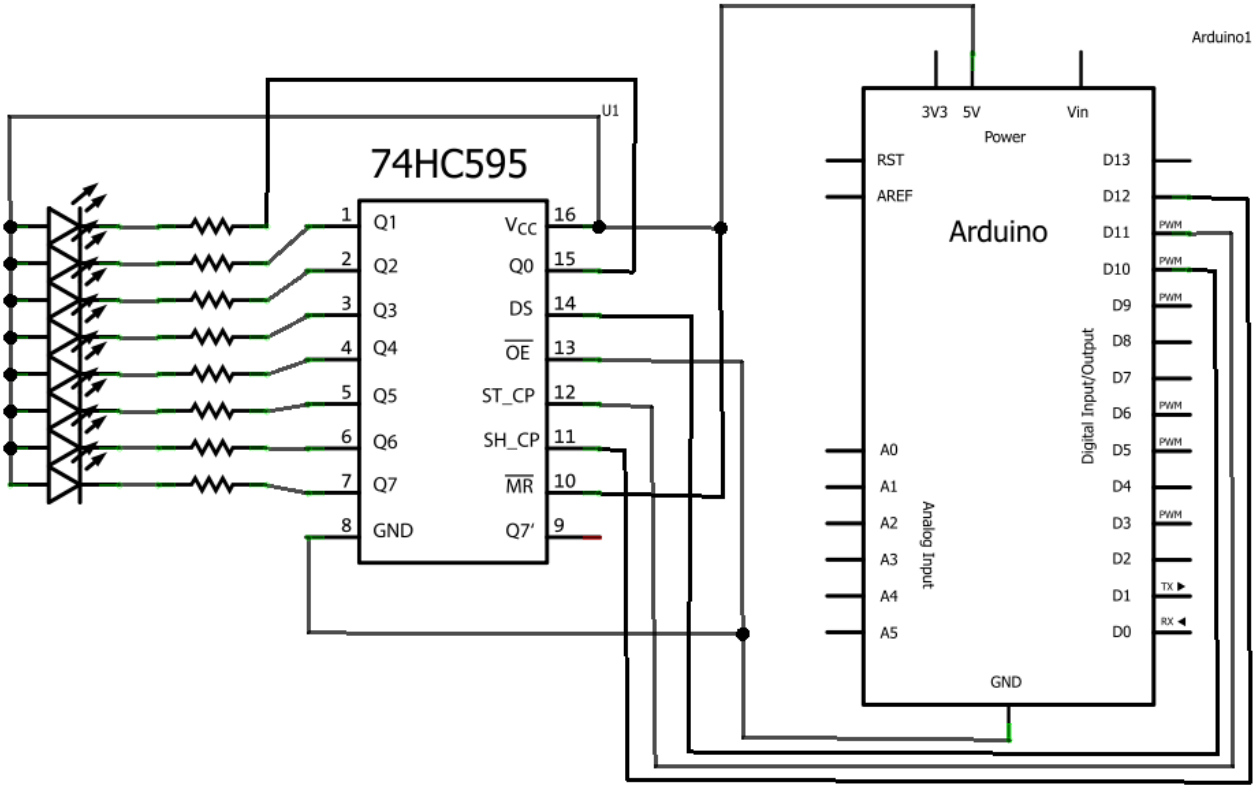

FIGURE 2. – Utilisation du 74HC595 - schéma

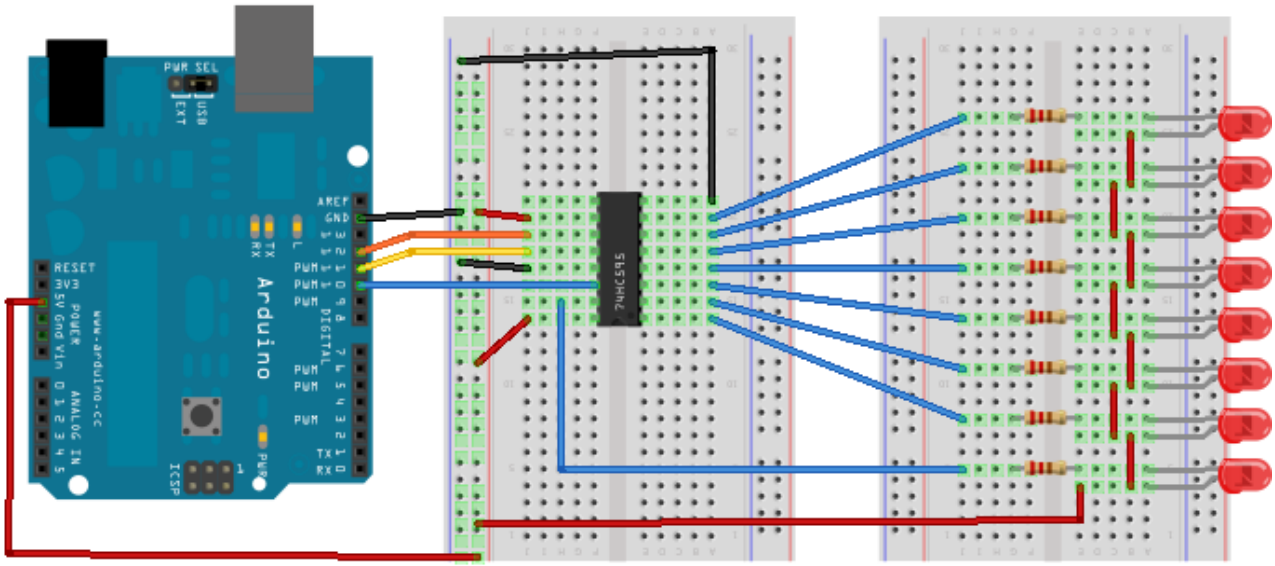

FIGURE 2. – Utilisation du 74HC595 - montage

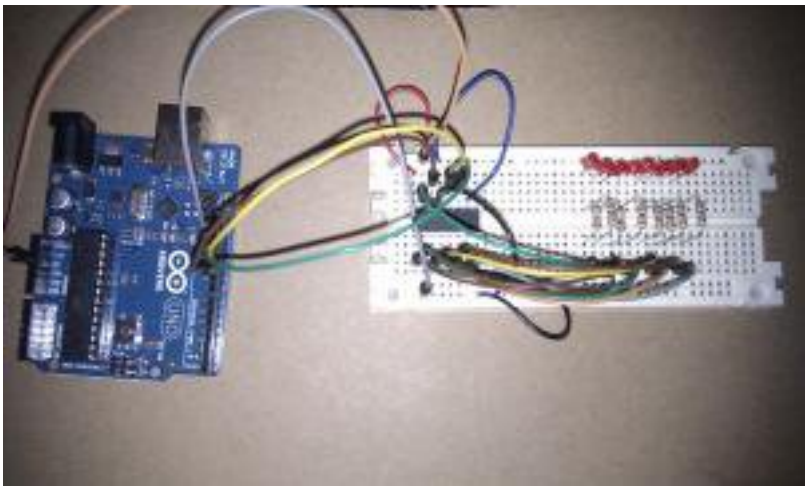

FIGURE 2. – Utilisation du 74HC595 et 8 LEDs

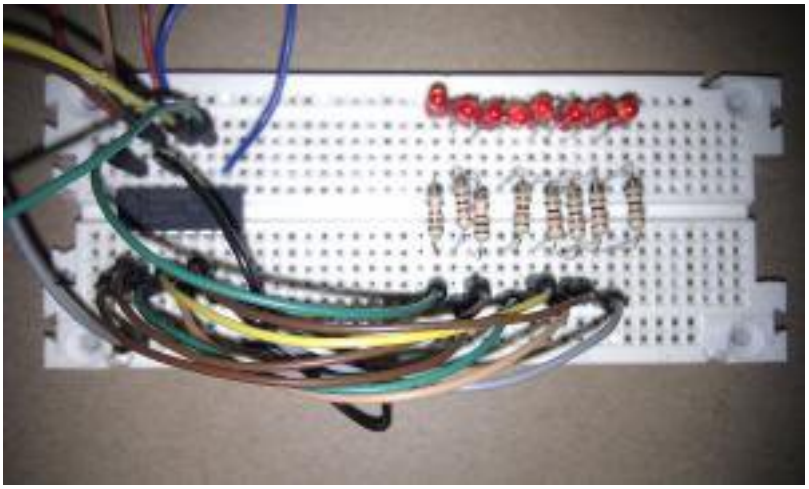

FIGURE 2. – Utilisation du 74HC595 et 8 LEDs (zoom)

3. Programmons pour utiliser ce composant

3.1. Envoyer un ordre au 74HC595

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

3.1.1. Le protocole

Nous le verrons dans le chapitre sur la liaison série plus en détail, le protocole est en fait un moyen qui permet de faire communiquer deux dispositifs. C'est une sorte de convention qui établit des règles de langage.

Par exemple, si deux personnes parlent deux langues différentes, elles vont avoir un mal fou à se comprendre l'une de l'autre. Et bien le protocole sert à imposer un langage qui leur permettra

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

Nous l'avons vu tout à l'heure, pour envoyer un ordre au composant, il faut lui transmettre une série de bits. Autrement dit, il faut envoyer des bits les uns après les autres sur la même broche d'entrée. Cette broche sera nommée "data". Ensuite, rappelez-vous, le composant a besoin de savoir quand lire la donnée, quand est-ce qu'un nouveau bit est arrivé ? C'est donc le rôle de l'horloge, ce que je vous expliquais plus haut. On pourrait s'imaginer qu'elle dit au composant : " Top ! tu peux lire la valeur car c'est un autre bit qui arrive sur ton entrée ! ". Enfin, une troisième broche où l'on va amener l'horloge de verrou sert à dire au composant : " Nous sommes en train de mettre à jour la valeur de tes sorties, alors le temps de la mise à jour, garde chaque sortie à son état actuel ". Quand elle changera d'état, en passant du niveau BAS au niveau HAUT (front montant), cela donnera le "top" au composant pour qu'il puisse mettre à jour ses sorties avec les nouvelles valeurs.

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

3.1.2. Création de la fonction d'envoi

Passons à la création de la fonction d'envoi des données. C'est avec cette fonction que nous enverrons les ordres au 74HC595, pour lui dire par exemple d'allumer une LED sur sa sortie 4. On va donc faire un peu de programmation, aller zou ! Commençons par nommer judicieusement cette fonction : `envoi_ordre()`. Cette fonction va prendre quatre paramètres. Le premier sera le numéro de la broche de données. Nous l'appellerons "dataPin". Le second sera similaire puisque ce sera le numéro de la broche d'horloge. Nous l'appellerons "clockPin". Le troisième sera le "sens" d'envoi des données, je reviendrai là-dessus ensuite. Enfin le dernier paramètre sera la donnée elle-même, donc un char (sur 8 bits, exactement comme l'ordre qui est à envoyer), que nous appellerons "donnee". Le prototype de la fonction sera alors le suivant :

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

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

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

Listing 1 – Boucle pour envoyer un octet

3.1.3. Envoyer un `char` en tant que donnée binaire

Maintenant que l'on a défini une partie de la fonction `envoi_ordre()`, il va nous rester un léger problème à régler : envoyer une donnée de type `char` en tant que suite de bit (ou donnée binaire). Prenons un exemple : le nombre 231 s'écrit aussi sous la forme 11100111 en base 2 (et oui, c'est le moment de se rappeler les conversions décimales/binaires). Seulement, en voulant envoyer ce nombre sur la broche de donnée pour commander le 74HC595, cela ne marchera pas d'écrire :

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

En faisant de cette façon, la carte Arduino va simplement comprendre qu'il faut mettre un état HAUT (car 231 est différent de 0) sur sa broche de sortie que l'on a nommée `dataPin`. Pour pouvoir donc envoyer ce nombre sous forme binaire, il va falloir ajouter à la fonction que l'on a créée un morceau de code supplémentaire. Ce que nous allons va faire va être une vraie boucherie : on va découper ce nombre en huit tranches et envoyer chaque morceau un par un sur la sortie `dataPin`. :mrgreen : Pour découper ce nombre, ça va pas être de la tarte... euh... je m'égare. :roll : On va utiliser une technique qui se nomme, tenez-vous bien, le **masquage**. On va en fait utiliser un masque qui va cacher la véritable valeur du nombre 231. Bon bon, je vous explique. Tout d'abord, on va considérer que le nombre 231 est vu sous sa forme binaire, qui je le rappel est 11100111, par votre carte Arduino. Donc, lorsque l'on va passer en paramètre `donnee` le nombre 231, le programme verra la suite de 1 et de 0 : 11100111. Jusque-là, rien de bien sorcier. Voilà donc notre suite de 1 et de 0 que l'on va devoir découper. Alors, il n'existe pas de fonction toute prête spécialement conçue pour découper un nombre binaire. Non, ça va être à nous de faire cela. Et c'est pourquoi je vous parlais du masquage. Cette technique ne porte pas son nom par hasard, en effet, nous allons réellement utiliser un **masque**. Quelques précisions s'imposent, je le sens bien. Reprenons notre suite binaire :

FIGURE 3. – Une suite binaire

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

FIGURE 3. – La suite masquée

Ce qui, au final, donnera :

FIGURE 3. – Résultat

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

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

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

FIGURE 3. – Envoi des 8 bits

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

FIGURE 3. – Gestion du masque

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

3.1.4. Les masques en programmation

Maintenant que vous connaissez cela, nous allons pouvoir voir comment isoler chacun des bits pour les envoyer un par un. En programmation, il est évident que l'on ne peut pas mettre un masque papier sur les bits pour les cacher. Il existe donc un moyen de les cacher. Cela va faire appel à la **logique binaire**. Nous n'entrerons pas dans le détail, mais sachez que nous allons employer des **opérateurs logiques**. Il en existe plusieurs, dont deux très utilisés, même dans la vie courante, l'opérateur **ET** et **OU**. Commençons par l'opérateur logique ET (je vous laisse regarder le OU tout seul, nous n'en aurons pas besoin ici). Il s'utilise avec le symbole **&** que vous trouverez sous la touche 1 au-dessus de la lettre "a" sur un clavier azerty. Pour envoyer le premier bit de notre donnée, nous allons effectuer le masquage avec cet opérateur logique dont la table de vérité se trouve être la suivante :

Bit 2	Bit 1	Résultat
0	0	0
0	1	0
1	0	0
1	1	1

Table de vérité du ET

? Je ne comprends pas trop où tu veux en venir ? :roll :

Je vais vous expliquer. Pour faire le masquage, on va faire une opération avec ce fameux ET logique. Il s'agit de la même chose que si l'on additionnait deux nombres ensemble, ou si on les multipliait. Dans notre cas l'opération est "un peu bizarre". Disons que c'est une opération évoluée. Cette opération va utiliser deux nombres : le premier on le connaît bien, il s'agit de la suite logique 11100111, quant au second, il s'agira du masque. Pour l'instant, vous ne connaissez pas la valeur du masque, qui sera lui aussi sous forme binaire. Pour définir cette valeur, on va utiliser la table de vérité précédente. Afin que vous ne vous perdiez pas dans mes explications, on va prendre pour objectif d'envoyer le bit de poids faible de notre nombre 11100111 (celui tout à droite). Le code qui suit est un pseudo-code, mis sous forme d'une opération mathématique telle que l'on en ferait à l'école :

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

Pour comprendre ce qui vient de se passer, il faut se référer à la table de vérité de l'opérateur ET : on sait que lorsque l'on fait 1 et 0 le résultat est 0. Donc, pour cacher tous les bits du nombre à masquer, il n'y a qu'à mettre que des 0 dans le masque. Là, l'Arduino ne verra que le bit 0 puisque le masque aura caché au complet le nombre du départ. On sait aussi que 1 ET 1 donne 1. Donc, lorsque l'on voudra montrer un bit à l'Arduino, on va mettre un 1 dans le masque, à l'emplacement du bit qui doit être montré. Pour monter ensuite le bit supérieur au bit de poids faible, on procède de la même manière :

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

Pour le quatrième bit en partant de la droite :

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

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

```
  11100111 (donnée à transmettre)
& 01000100 (on veut envoyer le septième et troisième bit)
= --------
  01000100 (donnée à transmettre au final) -> soit 68 en base
      décimale
```

3.1.5. L'évolution du masque

Ce titre pourrait être apparenté à celui d'un film d'horreur, mais n'indique finalement que nous allons faire évoluer le masque automatiquement à chaque fois que l'on aura envoyé un bit. Cette fois, cela va être un peu plus simple, car nous n'avons qu'à rajouter un opérateur spécialisé dans le décalage. Si l'on veut déplacer le 1 du masque (qui permet de montrer un bit à l'Arduino) de la droite vers la gauche (pour le LSBFirst) ou dans l'autre sens (pour le MSBFirst), nous avons la possibilité d'utiliser l'opérateur « pour décaler vers la gauche ou » pour décaler vers la droite. Par exemple :

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

Et dans le sens opposé :

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

Avouez que ce n'est pas très compliqué maintenant que vous maîtrisez un peu les masques. On va donc pouvoir isoler un par un chacun des bits pour les envoyer au 74HC595. Comme le sens dépend d'un paramètre de la fonction, nous rajoutons un test pour décaler soit vers la droite, soit vers la gauche. Voici la fonction que nous obtenons à la fin :

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

Listing 2 – La fonction `envoi_ordre`

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

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

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

3.1.6. Un petit programme d'essai

Je vous propose maintenant d'essayer notre belle fonction. Pour cela, quelques détails sont à préciser/rajouter. Pour commencer, il nous faut déclarer les broches utilisées. Il y en a trois : verrou, horloge et data. Pour ma part elles sont branchées respectivement sur les broches 11, 12 et 10. Il faudra donc aussi les déclarer en sortie dans le setup(). Si vous faites de même, vous devriez obtenir le code suivant :

```
1  // Broche connectée au ST_CP du 74HC595
2  const int verrou = 11;
3  // Broche connectée au SH_CP du 74HC595
4  const int horloge = 12;
5  // Broche connectée au DS du 74HC595
6  const int data = 10;
7  void setup()
8  {
9      // On met les broches en sortie
10     pinMode(verrou, OUTPUT);
11     pinMode(horloge, OUTPUT);
12     pinMode(data, OUTPUT);
13 }
```

Listing 3 – Code de setup de nos expériences

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

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

Listing 4 – Mécanisme de verrou

Et voici le code complet que vous aurez sûrement deviné :

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

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

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

void envoi_ordre(int dataPin, int clockPin, boolean sens, char
    donnee)
{
    // on va parcourir chaque bit de l'octet
```

```
35      for(int i=0; i<8; i++)
36      {
37          // on met l'horloge à l'état bas
38          digitalWrite(clockPin, LOW);
39          // on met le bit de donnée courante en place
40          if(sens)
41          {
42              digitalWrite(dataPin, donnee & 0x01 << i);
43          }
44          else
45          {
46              digitalWrite(dataPin, donnee & 0x80 >> i);
47          }
48          // enfin on remet l'horloge à l'état haut pour
49          // faire prendre en compte cette dernière
50          digitalWrite(clockPin, HIGH);
51      }
52  }
```

Listing 5 – Code complet d'envoi d'un octet

Et voilà le travail ! :

Élément externe (vidéo) —
Consultez cet élément à l'adresse (.

---

https ://www.youtube.com/embed/if2xNvj7DYo)[Video][]

Et une petite démonstration avec le simulateur interactif :!(https ://www.tinkercad.com/embed/5JnMmA7XnqH)

3.2. La fonction magique, ShiftOut

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

?

*#@"e !! :mad :

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

(donc a été testée par de nombreux développeurs, ne laissant pas beaucoup de place pour les bugs !). Cette fonction prend quatre paramètres :

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

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

```
1  void loop()
2  {
3      // on affiche les nombres de 0 à 255 en binaire
4      for (int i = 0; i < 256; i++)
5      {
6          // On active le verrou le temps de transférer les données
7          digitalWrite(verrou, LOW);
8          // on envoi toutes les données grâce à shiftOut
9          // (octet inversée avec '~' pour piloter les LED à l'état
               bas)
10         shiftOut(data, horloge, LSBFIRST, ~i);
11         // et enfin on relache le verrou
12         digitalWrite(verrou, HIGH);
13         // une petite pause pour constater l'affichage
14         delay(1000);
15     }
16 }
```

Listing 6 – Utilisation de la fonction `shiftOut`

Exemple interactif :!(https ://www.tinkercad.com/embed/hUcu806J9Sn)

4. Exercices : des chenillards !

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

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

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

4.1. ”J’avance et repars !”

4.1.1. Consigne

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

Élément externe (vidéo) —
Consultez cet élément à l’adresse (.

https ://www.youtube.com/embed/uMiJnwISEFA)[Video][]

4.1.2. Correction

Contenu masqué n°1

4.2. ”J’avance et reviens !”

4.2.1. Consigne

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

Élément externe (vidéo) —
Consultez cet élément à l’adresse (.

https ://www.youtube.com/embed/kYtlfWnNC34)[Video][]

4.2.2. Correction

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

Contenu masqué n°2

4.3. Un dernier pour la route !

4.3.1. Consigne

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

Élément externe (vidéo) —
Consultez cet élément à l'adresse (.

---

https ://www.youtube.com/embed/fjev82HNJaQ)[Video][]

4.3.2. Correction

Contenu masqué n°3

4.4. Exo bonus

4.4.1. Consigne

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

Élément externe (vidéo) —
Consultez cet élément à l'adresse (.

---

https ://www.youtube.com/embed/1h0kh57GVps)[Video][]

4.4.2. Correction

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

Contenu masqué n°4

5. Pas assez ? Augmentez encore !

Si jamais 8 nouvelles sorties ne vous suffisent pas (bien que cela n'en face que 5 au total puisque trois sont prises pour communiquer avec le composant), les ingénieurs ont déjà tout prévu.

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

5.1. Branchement

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

i

Les images proviennent d'une explication du site Arduino . Dans ce schéma les LEDs sont branchées en cathode commune.

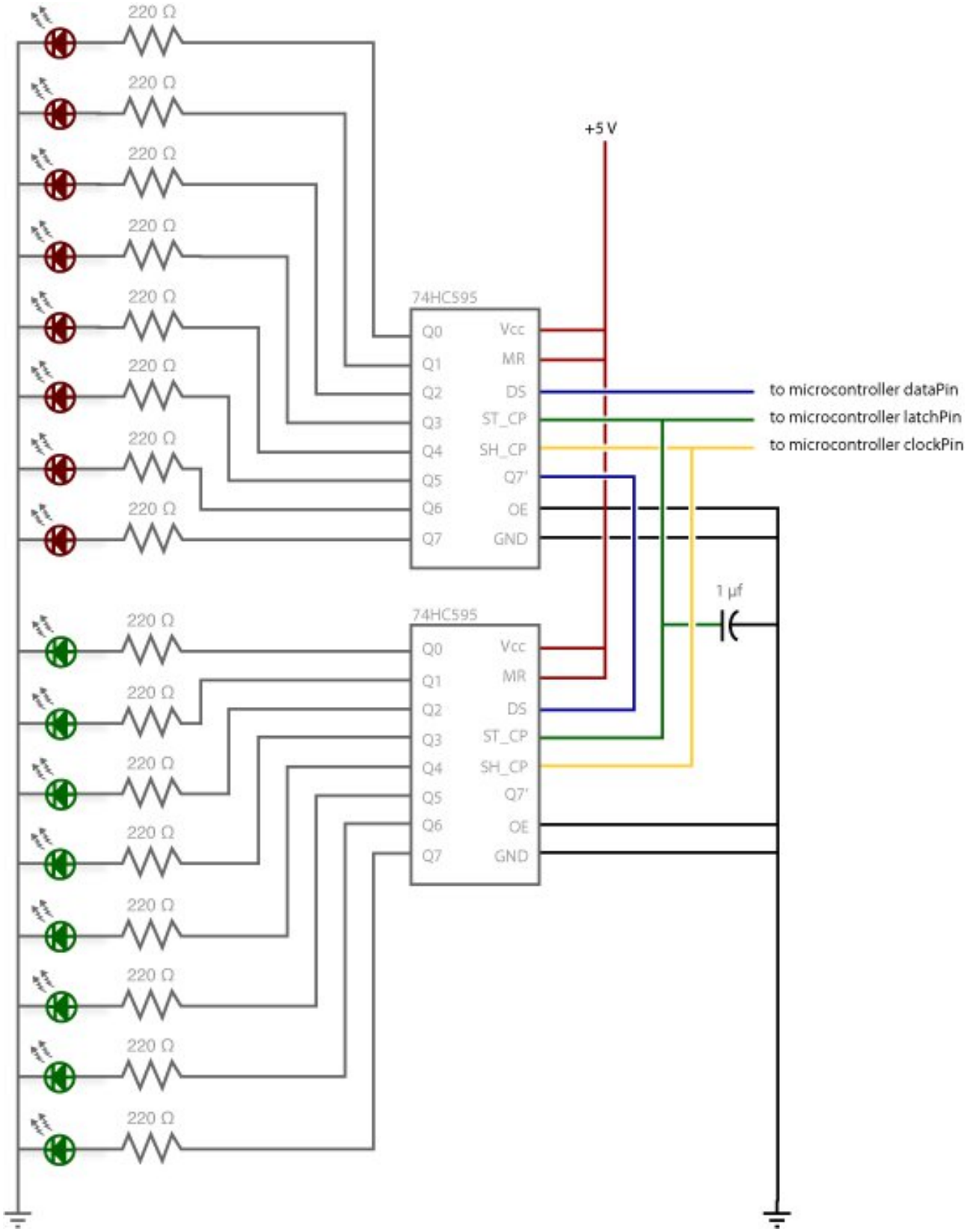

FIGURE 5. – Deux 74HC595 en cascade, schéma

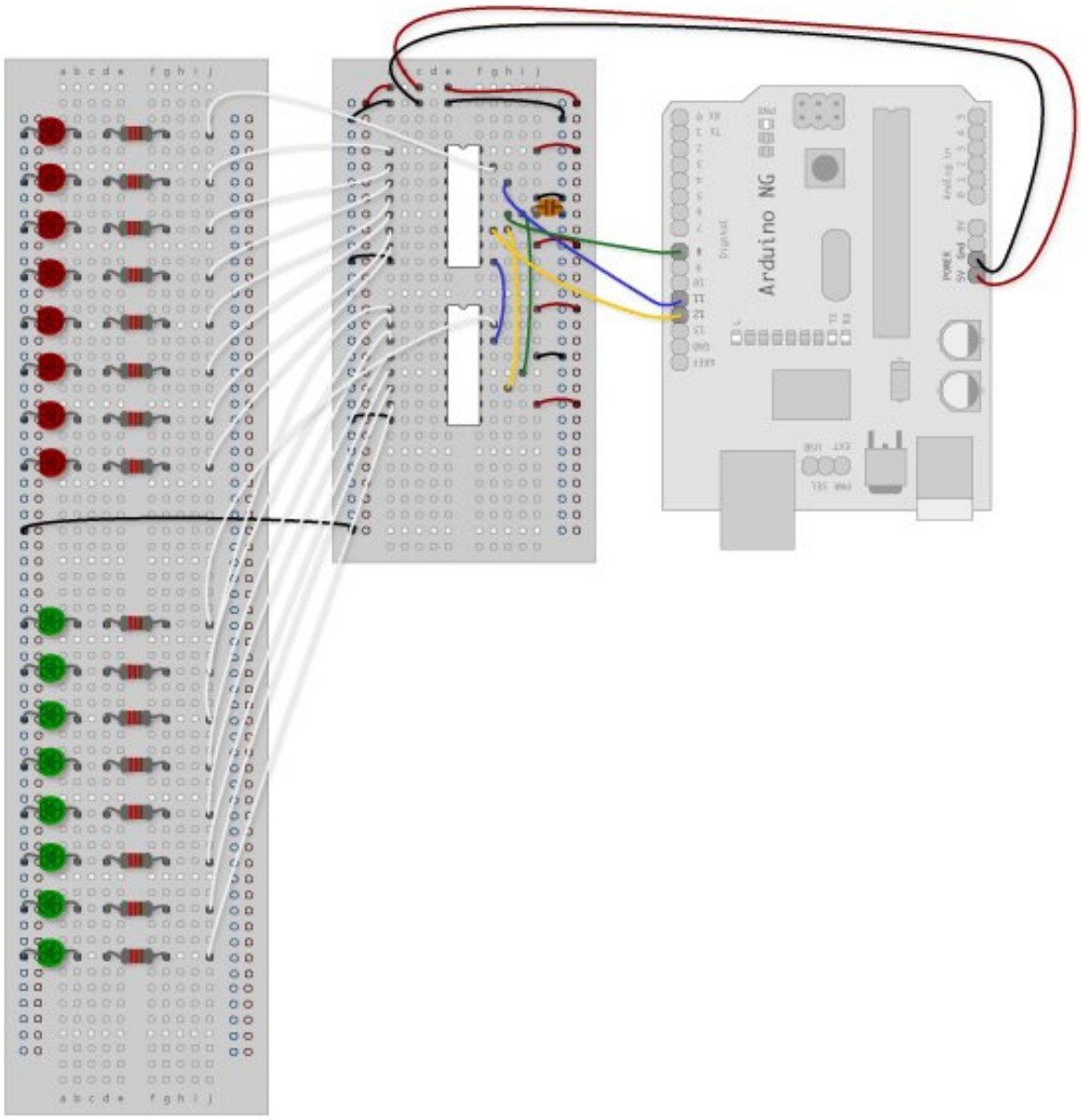

FIGURE 5. – Deux 74HC595 en cascade, montage

5.2. Exemple d'un affichage simple

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

```
1 const int verrou = 11;
2 const int horloge = 12;
3 const int data = 10;
4
```

5. Pas assez ? Augmentez encore !

```
5  char premier = 8; // en binaire : 00001000
6  char second = 35; // en binaire : 00100011
7
8  void setup()
9  {
10     // on déclare les broches en sortie
11     pinMode(verrou, OUTPUT);
12     pinMode(data, OUTPUT);
13     pinMode(horloge, OUTPUT);
14     // puis on envoie les données juste une fois
15     // on commence par mettre le verrou
16     digitalWrite(verrou, LOW);
17     // on envoie la seconde donnée d'abord
18     // les LEDs vertes du montage
19     shiftOut(data, horloge, LSBFIRST, ~second);
20     // on envoie la première donnée
21     // les LEDs rouges du montage
22     shiftOut(data, horloge, LSBFIRST, ~premier);
23     // et on relache le verrou pour mettre à jour les données
24     digitalWrite(verrou, HIGH);
25 }
26
27 void loop()
28 {
29     // rien à faire
30 }
```

Listing 7 – Envoi de deux octets consécutifs avec deux 74HC595

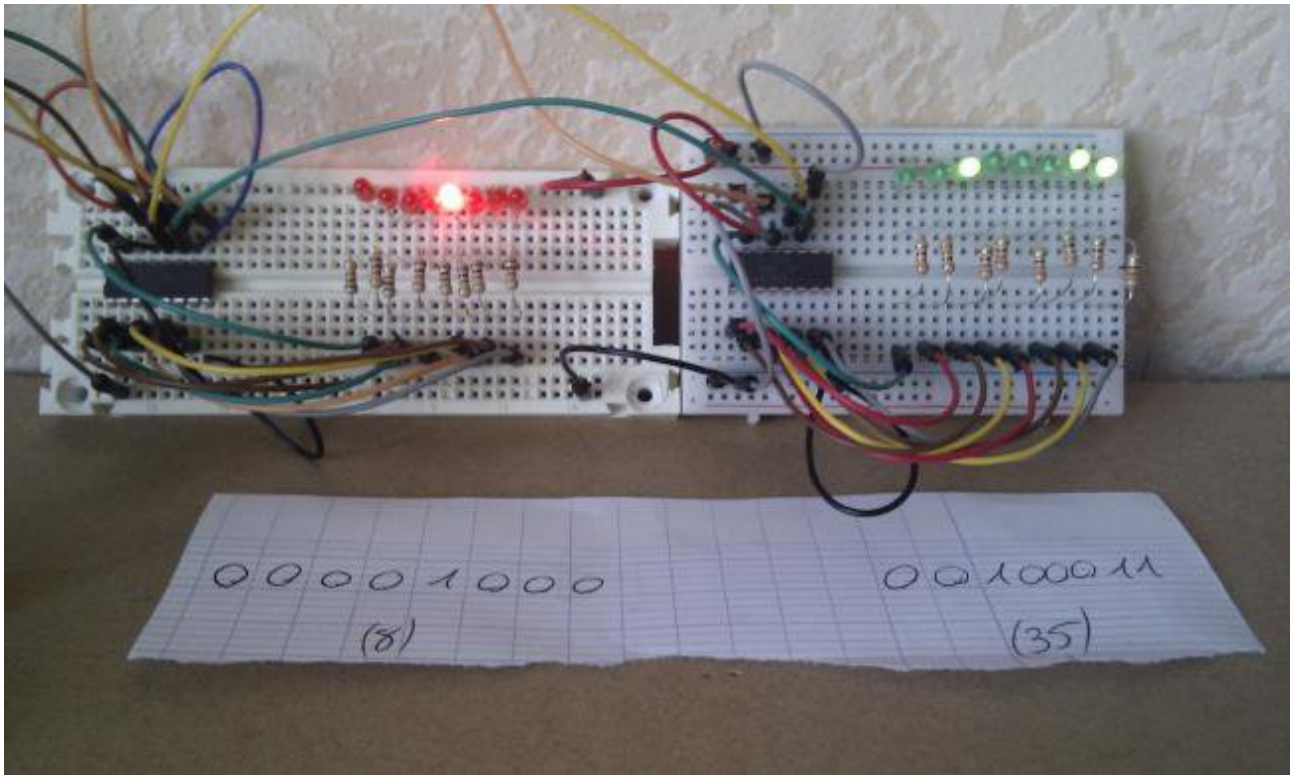

FIGURE 5. – Exemple de deux 74HC595 en cascade

5.3. Exemple d'un chenillard

Voici maintenant un petit exemple pour faire un chenillard sur 16 LEDs. Pour cela, j'utiliserai un int qui sera transformé en char au moment de l'envoi. Il faudra donc le décaler vers la droite de 8 bits pour pouvoir afficher ses 8 bits de poids fort. Voici une loop pour illustrer mes propos (le setup étant toujours le même). (Attention cependant, contrairement au montage précédent en cathode commune, j'utilise pour ma part un montage à anode commune et donc toutes les sorties sont inversé).

```
1  void loop()
2  {
3      int masque = 0;
4      for(int i=0; i<16; i++)
5      {
6          // on décale d'un cran le masque
7          masque = 0x01 << i;
8          // on commence par mettre le verrou
9          digitalWrite(verrou, LOW);
10         // on envoie la seconde donnée d'abord
11         // on envoie les 8 premiers bits
12         shiftOut(data, horloge, LSBFIRST, ~(masque & 0x00FF));
13         // on envoie la première donnée
14         // on envoie les 8 derniers bits
```

```
15          shiftOut(data, horloge, LSBFIRST, ~((masque & 0xFF00) >>
                8));
16          // et on relache le verrou pour mettre à jour les données
17          digitalWrite(verrou, HIGH);
18          delay(500);
19      }
20  }
```

Listing 8 – Démonstration de l'utilisation de deux 74HC595

Élément externe (vidéo) —
Consultez cet élément à l'adresse (.

https ://www.youtube.com/embed/in$_G$IqS5CU4)[*Video*][]
Et voici là encore la démonstration sur simulateur interactif :
!(https ://www.tinkercad.com/embed/ibDMHV4H3Cl)

6. Conclusion

Ce composant peut vous paraître un peu superflu, mais il existe en fait de très nombreuses applications avec. Par exemple, si vous voulez réaliser un cube de LED (disons 4x4x4 pour commencer gentiment). Si vous vouliez donner une broche par LED vous seriez bloqué puisque Arduino n'en possède pas autant (il vous en faudrait 64). Ici le composant vous permet donc de gérer plus de sorties que vous ne le pourriez initialement.

Contenu masqué

Contenu masqué n°1

```
1  void loop()
2  {
3      for (int i = 0; i < 8; i++)
4      {
5          // On active le verrou le temps de transférer les données
6          digitalWrite(verrou, LOW);
7          // on envoie la donnée
8          // ici, c'est assez simple.
9          // On va décaler l'octet 00000001 i fois puis l'envoyer
10         shiftOut(data, horloge, LSBFIRST, ~(0x01 << i));
11         // et enfin on relache le verrou
12         digitalWrite(verrou, HIGH);
13         // une petite pause pour constater l'affichage
```

```
14          delay(250);
15      }
16  }
```

Listing 9 – Exercice chenillard "J'avance et repars !"

Retourner au texte.

Contenu masqué n°2

```
1  char sens = MSBFIRST;
2  // on commence à aller de droite vers gauche
3  void loop()
4  {
5      // on ne fait la boucle que 7 fois pour
6      // ne pas se répéter au début et à la fin
7      for (int i = 0; i < 7; i++)
8      {
9          // On active le verrou le temps de transférer les données
10         digitalWrite(verrou, LOW);
11         // on envoie la donnée
12         // On va décaler l'octet 00000001 i fois puis l'envoyer
13         shiftOut(data, horloge, sens, ~(0x01 << i));
14         // et enfin on relache le verrou
15         digitalWrite(verrou, HIGH);
16         // une petite pause pour constater l'affichage
17         delay(250);
18     }
19     // on inverse le sens d'affichage pour la prochaine fois
20     sens = !sens;
21 }
```

Listing 10 – Exercice chenillard "J'avance et reviens !"

Retourner au texte.

Contenu masqué n°3

```
1  void loop()
2  {
3    char donnee = 0;
4    for (int i = 0; i < 8; i++)
5    {
6      // on saute la boucle si i vaut 4
7      // (pour une histoire de fluidité de l'animation, testez sans
           et comparez)
8      if(i == 4)
```

```
      continue;
    // calcule la donnée à envoyer
    donnee = 0;
    //  on calcule l'image du balayage dans un sens
    donnee = donnee | (0x01 << i);
    // et on ajoute aussi l'image du balayage dans l'autre sens
    donnee = donnee | (0x80 >> i);
    // On active le verrou le temps de transférer les données
    digitalWrite(verrou, LOW);
    // on envoie la donnée
    shiftOut(data, horloge, LSBFIRST, ~donnee);
    // et enfin on relache le verrou
    digitalWrite(verrou, HIGH);
    // une petite pause pour constater l'affichage
    delay(250);
  }
}
```

Listing 11 – Exercice chenillard "Un dernier pour la route !"

Retourner au texte.

Contenu masqué n°4

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

void loop()
{
    // on démarre à 0 ou 1 selon...
    char donnee = extinction;
    for (int i = 0; i < 8; i++)
    {
        // On active le verrou le temps de transférer les données
        digitalWrite(verrou, LOW);
        // si on est en train d'éteindre
        if(extinction)
            // on envoie la donnée inversé
            shiftOut(data, horloge, MSBFIRST, ~donnee);
        else // sinon
            // on envoie la donnée normale
            shiftOut(data, horloge, LSBFIRST, donnee);

        // et enfin on relache le verrou
        digitalWrite(verrou, HIGH);
        // une petite pause pour constater l'affichage
        delay(250);
        // et on met à jour la donnée en cumulant les décalages
```

Contenu masqué

```
25          donnee = donnee | (0x01 << i);
26      }
27      // permet d'inverser "MSBFIRST  LSBFIRST" comme dans l'exercice
            2
28      extinction = !extinction;
29  }
```

Listing 12 – Exercice ”effet de chargement / déchargement”

Retourner au texte.