

[Arduino 3] Communication par la voie série

[Arduino 301] Généralités sur la voie série

La communication... que ferait-on sans ! Le téléphone, Internet, la télévision, les journaux, la publicité... rien de tout cela n'existerait s'il n'y avait pas de communication. Évidemment, ce n'est pas de ces moyens là dont nous allons faire l'objet dans la partie présente. Non, nous allons voir un moyen de communication que possède la carte Arduino. Vous pourrez ainsi faire communiquer votre carte avec un ordinateur ou bien une autre carte Arduino ! Et oui ! Elle en a sous le capot cette petite carte ! 😊

Communiquer, pourquoi ?

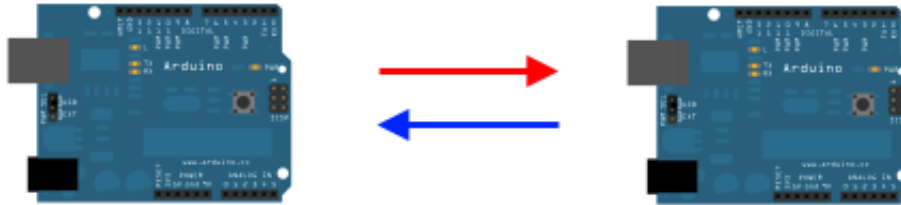
Nous avons vu dans la partie précédente où nous faisons nos premiers pas avec Arduino, comment utiliser la carte. Nous avons principalement utilisé des LED pour communiquer à l'utilisateur (donc vous, à priori) certaines informations. Cela pouvait être une LED ou un groupe de LED qui peut indiquer tout et n'importe quoi, ou bien un afficheur 7 segments qui affiche des chiffres ou certains caractères pouvant tout aussi bien indiquer quelque chose. Tout dépend de ce que vous voulez signaler avec les moyens que vous mettez à disposition. On peut très bien imaginer un ensemble de LED ayant chacune un nom, sigle ou autre marqueur pour indiquer, selon l'état d'une ou plusieurs d'entre-elles, un mode de fonctionnement ou bien une erreur ou panne d'un système. Cependant, cette solution reste tout de même précaire et demande à l'utilisateur d'être devant le système de signalisation. Aujourd'hui, avec l'avancée de la technologie et du "tout connecté", il serait fâcheux de ne pouvoir aller plus loin. Je vais donc vous présenter un nouveau moyen de **communication** grâce à la **voie série** (ou "liaison série"), qui va vous permettre de communiquer des informations à l'utilisateur par divers intermédiaires. A la fin de la partie, vous serez capable de transmettre des informations à un ordinateur ou une autre carte Arduino.

Transmettre des informations

Tel est le principal objectif de la communication. Mais comment transmettre des informations... et puis quelles informations ? Avec votre carte Arduino, vous aurez certainement besoin de transmettre des mesures de températures ou autres grandeurs (tension, luminosité, etc.). Ces informations pourront alimenter une base de donnée, servir dans un calcul, ou à autre chose. Tout dépendra de ce que vous en ferez.

Émetteur et récepteur

Lorsque l'on communique des informations, il faut nécessairement un **émetteur**, qui va transmettre les informations à communiquer, et un **récepteur**, qui va recevoir les informations pour les traiter.



Dans le cas présent, deux carte Arduino communiquent. L'une communique à l'autre tandis que l'autre réceptionne le message envoyé par la première.

Pourtant, il y a deux flèches sur ton dessin. L'autre aussi, qui réceptionne le message, peut envoyer des données ?

Absolument ! Cependant, tout dépend du type de communication.

La communication en trois cas

Pour parler, on peut par exemple différencier trois types de conversations. A chaque conversation, il n'y a que deux interlocuteurs. On ne peut effectivement pas en faire communiquer plus dans notre cas ! On dit que c'est une communication **point-à-point**.

- Le premier type serait lorsqu'un interlocuteur parle à son compère sans que celui-ci dise quoi que ce soit puisqu'il ne peut pas répondre. Il est muet et se contente d'écouter. C'est une communication à sens unilatérale, ou techniquement appelée communication **simplex**. L'un parle et l'autre écoute.
- Le deuxième type serait une conversation normale où chacun des interlocuteurs est poli et attend que l'autre est finie de parler pour parler à son tour. Il s'agit d'une communication **half-duplex**. Chaque interlocuteur parle **à tour de rôle**.
- Enfin, il y a la conversation du type "débat politique" (ce n'est évidemment pas son vrai nom 🗣️) où chaque interlocuteur parle en même temps que l'autre. Bon, cela dit, ce type de communication marche très bien (pas au sens politique, je parle au niveau technique !) et est très utilisé ! C'est une communication dite **full-duplex**.

A notre échelle, Arduino est capable de faire des communications de type full-duplex, puisqu'elle est capable de comprendre son interlocuteur tout en lui parlant en même temps.

Le récepteur

Qu'en est-il ? Eh bien il peut s'agir, comme je le sous-entendais plus tôt, d'une autre carte Arduino. Cela étant, n'importe quel autre appareil utilisant la voie série et son **protocole de communication** pourrait communiquer avec. Cela peut être notamment un ordinateur, c'est d'ailleurs le principal interlocuteur que nous mettrons en relation avec Arduino.

C'est quoi ça, un protocole de communication ?

C'est un ensemble de règles qui régissent la façon dont communiquent deux dispositifs entre eux. Cela définit par exemple le rythme de la conversation (le débit de parole des acteurs si vous préférez), l'ordre des informations envoyées (la grammaire en quelque

sorte), le nombre d'informations, etc... On peut analogiquement comparer à une phrase en français, qui place le sujet, le verbe puis le complément. C'est une forme de protocole. Si je mélange tout ça, en plaçant par exemple le sujet, le complément et le verbe, cela donnerait un style parlé de maître Yoda... bon c'est moins facilement compréhensible, mais ça le reste. En revanche, deux dispositifs qui communiquent avec un protocole différent ne se comprendront pas correctement et pourraient même interpréter des actions à effectuer qui seraient à l'opposé de ce qui est demandé. Ce serait en effet dommage que votre interlocuteur "donne le chat à manger" alors que vous lui avez demandé "donne à manger au chat" 🐱 Bref, si les dispositifs communiquant n'utilisent pas le bon protocole, cela risque de devenir un véritable capharnaüm !

La norme RS232

Des liaisons séries, il en existe un paquet ! Je peux en citer quelques unes : RS-232, Universal Serial Bus (USB), Serial ATA, SPI, ... Et pour dire, vous pouvez très bien inventer votre propre norme de communication pour la voie série que vous décidez de créer. L'inconvénient, bien que cela puisse être également un avantage, il n'y a que vous seul qui puissiez alors utiliser une telle communication.

Et nous, laquelle allons-nous voir parmi celles-là ? Il y en a des meilleurs que d'autres ? oO

D'abord, nous allons voir la voie série utilisant la norme RS-232. Ensuite, oui, il y en a qui ont des avantages par rapport à d'autres. On peut essentiellement noter le type d'utilisation que l'on veut en faire et la vitesse à laquelle les dispositifs peuvent communiquer avec.

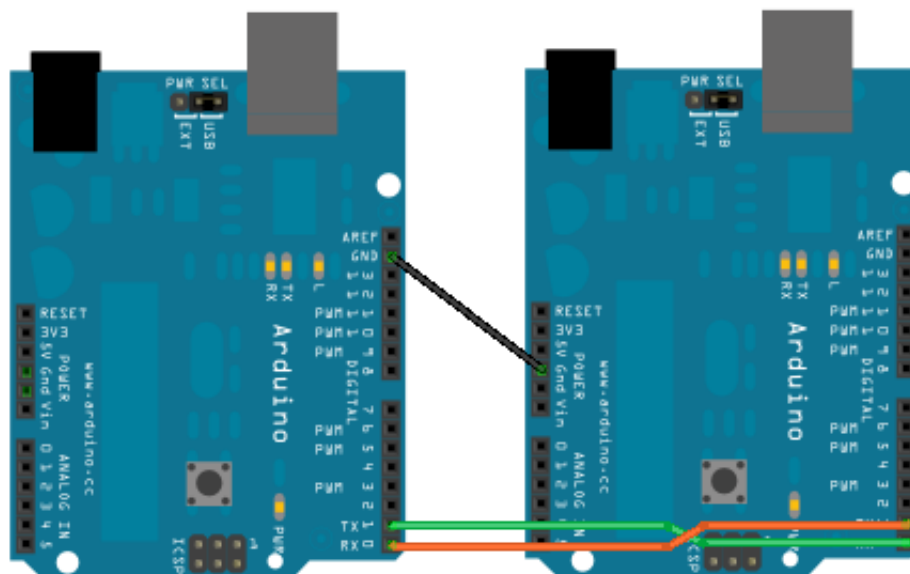
Applications de la norme

La **norme RS-232** s'applique sur trois champs d'une communication de type série. Elle définit le signal électrique, le protocole utilisé et tout ce qui est lié à la mécanique (la connectique, le câblage, etc...).

La mécanique

Pour communiquer via la voie série, deux dispositifs doivent avoir 3 câbles minimum.

- Le premier câble est la **référence électrique**, communément appelée **masse électrique**. Cela permet de prendre les mesures de tension en se fixant un même référentiel. Un peu lorsque vous vous mesurez : vous mesurez 1,7 mètre du sol au sommet de votre tête et non pas 4,4 mètre parce que vous êtes au premier étage et que vous vous basez par rapport au sol du rez-de-chaussée. Dans notre cas, on considérera que le 0V sera notre référentiel électrique commun.
- Les deux autres câbles permettent la transmission des données. L'un sert à l'envoi des données pour un émetteur, mais sert aussi pour la réception des données venant de l'autre émetteur. Idem pour l'autre câble. Il permet l'émission de l'un et la réception de l'autre.



Deux cartes Arduino reliées par 3 câbles :

- Le **noir** est la masse électrique commune
- Le **vert** est celui utilisé pour l'envoi des données de la première carte (à gauche), mais sert également à la réception des données envoyées pour la deuxième carte (à droite)
- Le **orange** est celui utilisé pour l'envoi des données de la deuxième carte (à droite), mais sert également à la réception des données envoyées pour la première carte (à gauche)

Cela, il s'agit du strict minimum utilisé. La norme n'interdit pas l'utilisation d'autres câbles qui servent à faire du contrôle de flux et de la gestion des erreurs.

Le signal électrique et le protocole

Avant tout, il faut savoir que pour communiquer, deux dispositifs électronique ou informatique utilisent des données sous forme de **bits**. Ces bits, je le rappel, sont des états logiques (vrai ou faux) qui peuvent être regroupés pour faire des ensembles de bits. Généralement, ces ensembles sont constitués de 8 bits qui forment alors un **octet**.

Les tensions utilisées

Ces bits sont en fait des niveaux de tension électrique. Et la norme RS-232 définit quelles tensions doivent être utilisées. On peut spécifier les niveaux de tension imposés par la norme dans un tableau, que voici :

	Niveau logique 0	Niveau logique 1
Tension électrique minimale	+3V	-3V
Tension électrique maximale	+25V	-25V

Ainsi, toutes les tensions au delà des valeurs imposées, donc entre -3V et +3V, au dessous de -25V et au dessus de +25V, sont hors normes. Pour les tensions trop élevées (aux extrêmes de + et -25V) elles pourraient endommager le matériel. Quand aux tensions comprises entre + et -3V, eh bien elles sont ignorées car c'est dans ces

zones là que se trouvent la plupart et même la quasi totalité des parasites. C'est un moyen permettant d'éviter un certain nombre d'erreurs de transmissions.

Les parasites dont je parle sont simplement des pics de tensions qui peuvent survenir à cause de différentes sources (interrupteur, téléviseur, micro-ondes, ...) et qui risquent alors de modifier des données lors d'une transmission effectuée grâce à la voie série.

Lorsqu'il n'y a pas de communication sur la voie série, il y a ce qu'on appelle un **état de repos**. C'est à dire un niveau logique toujours présent. Il s'agit du niveau logique 1. Soit une tension comprise entre -3V et -25V. Si cet état de repos n'est pas présent, c'est qu'il peut y avoir un problème de câblage.

Les données

Les données qui transitent par la voie série sont transmises sous une forme binaire. C'est à dire avec des niveaux logiques 0 et 1. Prenons une donnée que nous voudrions envoyer, par exemple la lettre "P" majuscule. Vous ne le saviez peut-être pas mais une lettre du clavier est codée sur un nombre de 8 bits, donc un octet. Réellement c'est en fait sur 7 bits qu'elle est codée, mais en rajoutant un 0 devant le codage, cela conserve sa valeur et permet d'avoir un codage de la lettre sur 8 bits. Ces codes sont définis selon la **table ASCII**. Ainsi, pour chaque caractère du clavier, on retrouve un codage sur 8 bits. Vous pouvez aller consulter cette table pour comprendre un peu comment elle fonctionne en [suivant ce lien](#). En haut à gauche de la table ASCII, on observe la ligne : "Code en base..." et là vous avez : 10, 8, 16, 2. Respectivement, ce sont les bases décimale (10), octale (8), hexadécimale (16) et binaire (2). Certaines ne vous sont donc pas inconnues puisque l'on en a vu. Nous, ce qui va nous intéresser, c'est la base binaire. Oui car le binaire est une succession de 0 et de 1, qui sont les états logiques 0 (LOW) et 1 (HIGH). En observant la table, on tombe sur la lettre "P" majuscule et l'on voit sa correspondance en binaire : **01010000**.

Je crois ne pas bien comprendre pourquoi on envoie une lettre... qui va la recevoir et pour quoi faire ? o_O

Il faut vous imaginer qu'il y a un destinataire. Dans notre cas, il s'agira avant tout de l'ordinateur avec lequel vous programmez votre carte. On va lui envoyer la lettre "P" mais cela pourrait être une autre lettre, une suite de lettres ou autres caractères, voir même des phrases complètes. Pour ne pas aller trop vite, nous resterons avec cette unique lettre. Lorsque l'on enverra la lettre à l'ordinateur, nous utiliserons un petit module intégré dans le logiciel Arduino pour visualiser le message réceptionné. C'est donc nous qui allons voir ce que l'on transmet via la voie série.

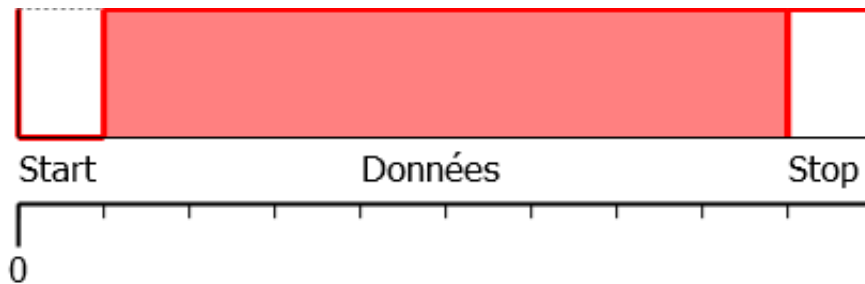
L'ordre et les délimiteurs

On va à présent voir comment est transmis un octet sur la voie série en envoyant notre exemple, la lettre "P". Analogiquement, je vais vous montrer que cette communication par la voie série se présente un peu comme un appel téléphonique :

1. Lorsque l'on passe un coup de fil, bien généralement on commence par dire "Bonjour" ou "Allo". Ce début de message permet de faire l'ouverture de la

conversation. En effet, si l'on reçoit un appel et que personne ne répond après avoir décroché, la conversation ne peut avoir lieu. Dans la norme RS-232, on va avoir une ouverture de la communication grâce à un **bit de départ**. C'est lui qui va engager la conversation avec son interlocuteur. Dans la norme RS-232, ce dernier est un état 0.

2. Ensuite, vous allez commencer à parler et donner les informations que vous souhaitez transmettre. Ce sera les **données**. L'élément principal de la conversation (ici notre lettre 'P').
3. Enfin, après avoir renseigné tout ce que vous aviez à dire, vous terminez la conversation par un "Au revoir" ou "Salut !", "A plus !" etc. Cela termine la conversation. Il y aura donc un **bit de fin** ou **bit de stop** qui fera de même sur la voie série. Dans la norme RS-232, c'est un état 1.

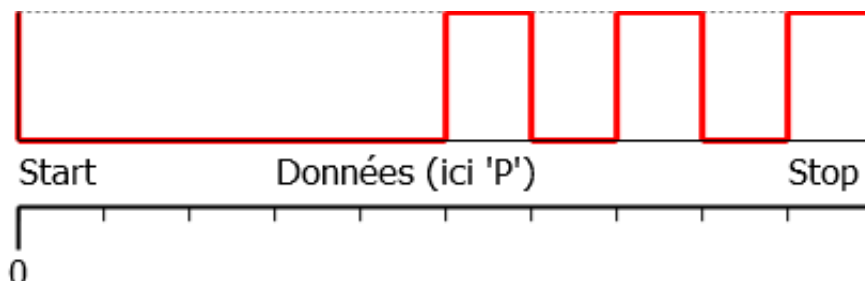


C'est de cette manière là que la communication série fonctionne. D'ailleurs, savez-vous pourquoi la voie série s'appelle ainsi ? En fait, c'est parce que les données à transmettre sont envoyées une par une. Si l'on veut, elles sont à la queue leu-leu. Exactement comme une conversation entre deux personnes : la personne qui parle ne peut pas dire plusieurs phrases en même temps, ni plusieurs mots ou sons. Chaque élément se suit selon un ordre logique. L'image précédente résume la communication que l'on vient d'avoir, il n'y a plus qu'à la compléter pour envoyer la lettre "P".

Ha, je vois. Donc il y a le bit de start, notre lettre P et le bit de stop. D'après ce qu'on a dit, cela donnerait, dans l'ordre, ceci : 0 (Start) 01010000 (Données) et 1 (Stop).



Eh bien... c'est presque ça. Sauf que les ~~petits-mains~~ ingénieurs qui ont inventé ce protocole ont eu la bonne idée de transmettre les données à l'envers... Par conséquent, la bonne réponse était : 000010101. Avec un chronogramme, on observerait ceci :



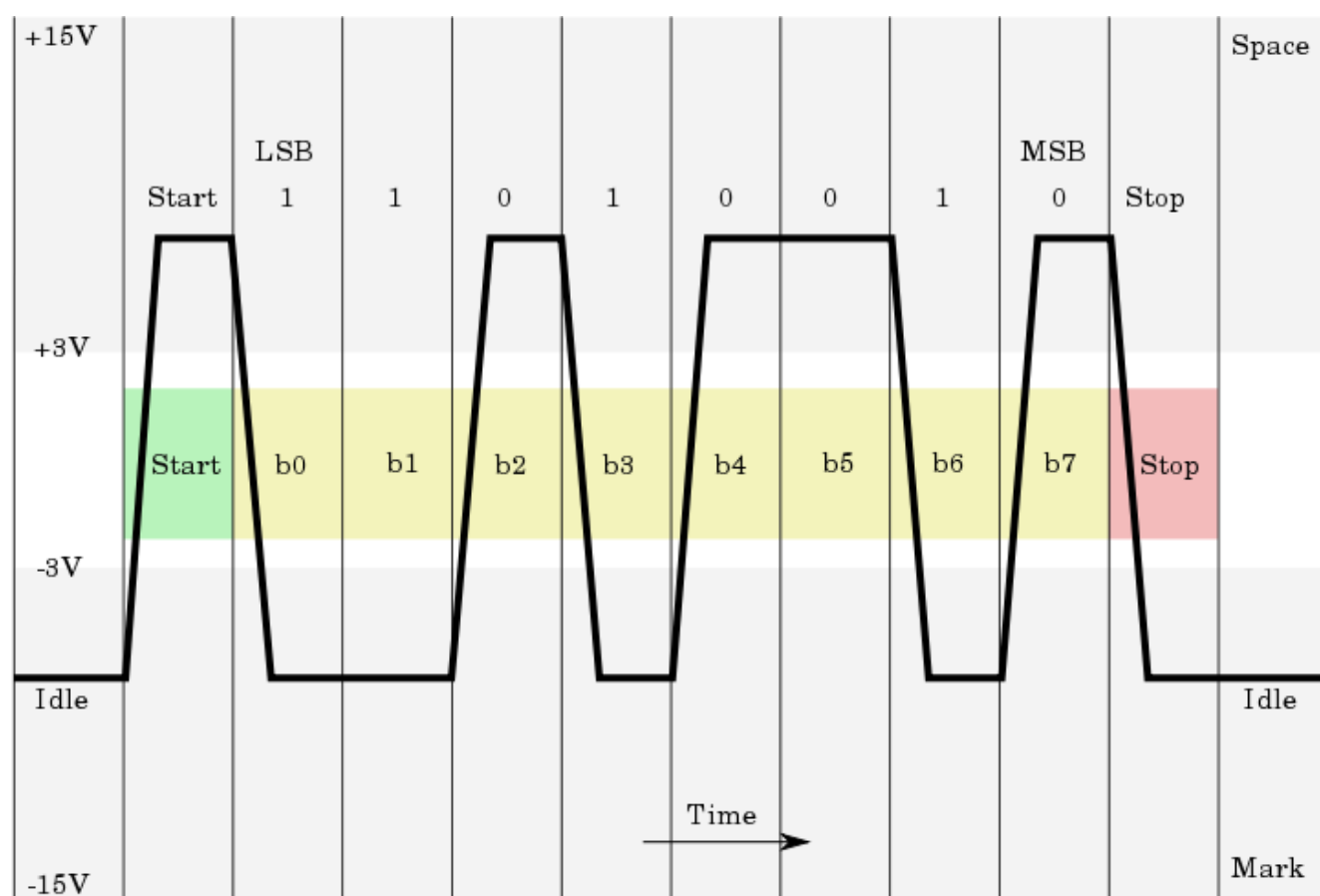
Un peu de vocabulaire

Avant de continuer à voir ce que compose le protocole RS-232, voyons un peu de vocabulaire, mais sans trop en abuser bien sûr ! 😊 Les données sont envoyées à l'envers, je le disais. Ce qu'il faut savoir c'est que le bit de donnée qui vient après le bit de start s'appelle le **bit de poids faible** ou **LSB** en anglais pour Less Significant Bit.

C'est un peu comme un nombre qui a des unités (tout à droite), des dizaines, des centaines, des milliers (à gauche), etc. Par exemple le nombre 6395 possède 5 unités (à droite), 9 dizaines, 3 centaines et 6 milliers (à gauche). On peut faire référence au bit de poids faible en binaire qui est donc à droite. Plus on s'éloigne et plus on monte vers... le bit de **poids fort** ou **MSB** en anglais pour Most Significant Bit. Et comme les données sont envoyées à l'envers sur la liaison série, on aura le bit de poids faible juste après le start, donc à gauche et le bit de poids fort à droite. Avec le nombre précédent, si l'on devait le lire à l'envers cela donnerait : 5396. Bit de poids faible à gauche et à droite le bit de poids fort.

Il est donc essentiel de savoir où est le bit de poids faible pour pouvoir lire les données à l'endroit. Sinon on se retrouve avec une donnée erronée !

Pour regrouper un peu tout ce que l'on a vu sur le protocole de la norme RS-232, voici une image extraite de [cette page Wikipédia](https://fr.wikipedia.org/wiki/S%C3%A9rie_(informatique)#Le_protocole_RS-232) :



Vous devrez être capable de trouver quel est le caractère envoyé sur cette trame... alors ? 😊 Indice : c'est une lettre... On lit les niveaux logiques de gauche à droite, soit 11010010 ; puis on les retourne soit 01001011 ; enfin on compare à la table ASCII et on trouve la lettre "K" majuscule. **Attention aux tensions négatives qui correspondent à l'état logique 1 et les tensions positives à l'état logique 0.**

La vitesse

La norme RS-232 définit la vitesse à laquelle sont envoyées les données. Elles sont exprimées en bit par seconde (bit/s). Elle préconise des vitesses inférieures à 20 000 bits/s. Sauf qu'en pratique, il est très courant d'utiliser des débits supérieurs pouvant

atteindre les 115 200 bits/s. Quand on va utiliser la voie série, on va définir la vitesse à laquelle sont transférées les données. Cette vitesse dépend de plusieurs contraintes que sont : la longueur du câble utilisé reliant les deux interlocuteurs et la vitesse à laquelle les deux interlocuteurs peuvent se comprendre. Pour vous donner un ordre d'idée, je reprend le tableau fourni sur la page Wikipédia sur la norme RS-232 :

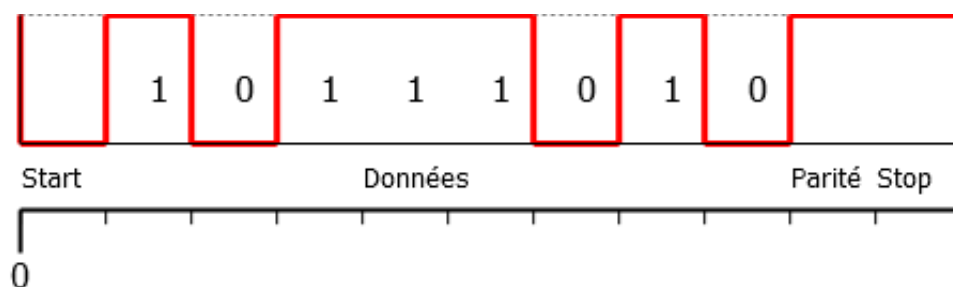
Débit en bit/s Longueur du câble en mètres (m)

2 400	900
4 800	300
9 600	150
19 200	15

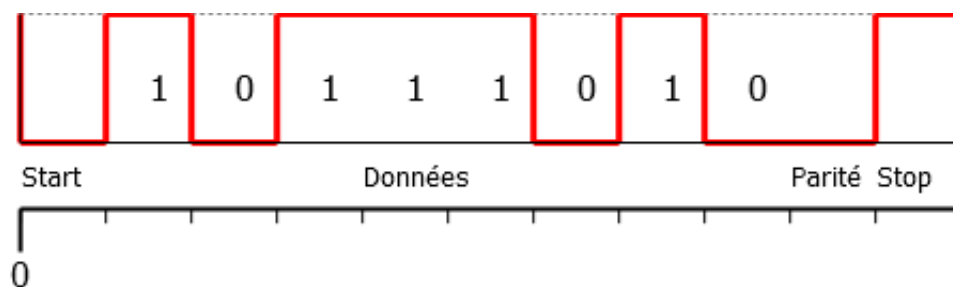
Plus le câble est court, plus le débit pourra être élevé car moins il y a d'affaiblissement des tensions et de risque de parasites. Tandis que si la distance séparant les deux interlocuteurs grandie, la vitesse de communication diminuera de façon effective.

La gestion des erreurs

Malgré les tensions imposées par la norme, il arrive qu'il y ai d'autres parasites et que des erreurs de transmission surviennent. Pour limiter ce risque, il existe une solution. Elle consiste à ajouter un **bit de parité**. Vous allez voir, c'est hyper simple ! 😊 Juste avant le bit de stop, on va ajouter un bit qui sera pair ou impair. Donc, respectivement, soit un 0 soit un 1. Lorsque l'on utilisera la voie série, si l'on choisi une parité paire, alors le nombre de niveaux logiques 1 dans les données plus le bit de parité doit donner un nombre paire. Donc, dans le cas ou il y a 5 niveaux logiques 1 sans le bit de parité, ce dernier devra prendre un niveau logique 1 pour que le nombre de 1 dans le signal soit paire. Soit 6 au total :



Dans le cas où l'on choisirait une parité impaire, alors dans le même signal où il y a 5 niveaux logiques 1, eh bien le bit de parité devra prendre la valeur qui garde un nombre impaire de 1 dans le signal. Soit un bit de parité égal à 0 dans notre cas :



Après, c'est le récepteur qui va vérifier si le nombre de niveaux logiques 1 est bien égale à ce que indique le bit de parité. Dans le cas où une erreur de transmissions

serait survenu, ce sera au récepteur de traiter le problème et de demander à son interlocuteur de répéter. Au fait, ne vous inquiétez pas, on aura l'occasion de voir tout ça plus tard dans les prochains chapitres. De quoi s'occuper en somme... 😡

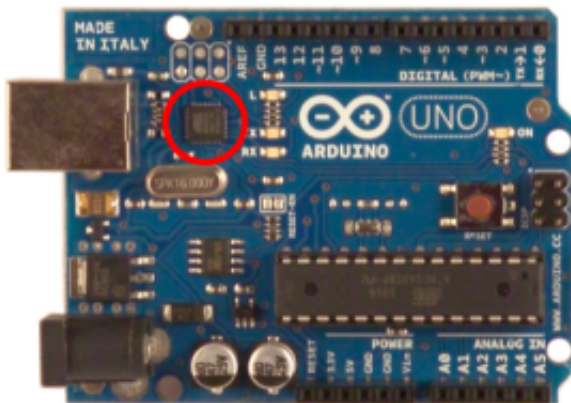
Connexion série entre Arduino et ...

Et on connecte quoi à où pour utiliser la voie série avec la carte Arduino et le PC ? C'est le même câblage ? Et on connecte où sur le PC ?

Là, on va avoir le choix...

Émulation du port série

Le premier objectif et le seul que nous mettrons en place dans le cours, va être de connecter et d'utiliser la voie série avec l'ordinateur. Pour cela, rien de plus simple, il n'y a que le câble USB à brancher entre la carte Arduino et le PC. En fait, la voie série va être **émulée** à travers l'USB. C'est une forme virtuelle de cette liaison. Elle n'existe pas réellement, mais elle fonctionne comme si c'était bien une vraie voie série. Tout ça va être géré par un petit composant présent sur votre carte Arduino et le gestionnaire de port USB et périphérique de votre ordinateur.



Le composant entouré en rouge gère l'émulation de la voie série

C'est la solution la plus simple et celle que nous allons utiliser pour vos débuts.

Arduino et un autre microcontrôleur

On a un peu abordé ce sujet, au début de la présentation sur la voie série. Mais, on va voir un peu plus de choses. Le but de connecter deux microcontrôleur ensemble est de pouvoir les faire communiquer entre eux pour qu'ils puissent s'échanger des données.

La tension des microcontrôleurs

Tension

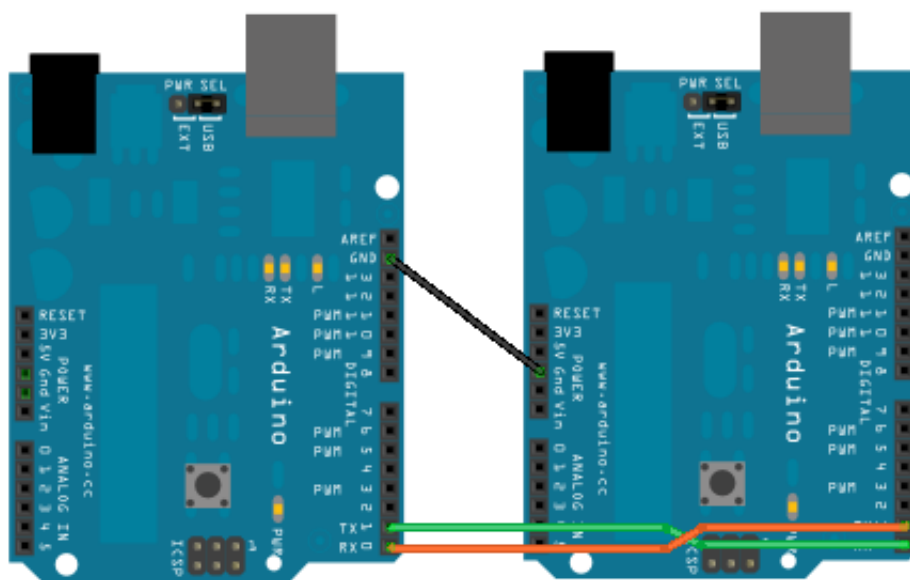
NL0 0V

NL1 +5V

Contrairement à ce qu'impose la norme RS-232, les microcontrôleurs ne peuvent pas utiliser des tensions négatives. Du coup, ils utilisent les seules et uniques tensions qu'ils peuvent utiliser, à savoir le 0V et le +5V. Il y a donc quelques petits changements au niveau de la transmission série. Un niveau logique 0 correspond à une tension de 0V et un niveau logique 1 correspond à une tension de +5V. (cf. tableau ci-contre) Fort heureusement, comme les microcontrôleurs utilisent quasiment tous cette norme, il n'y a aucun problème à connecter deux microcontrôleurs entre-eux. Cette norme s'appelle alors UART pour Universal Asynchronous Receiver Transmitter plutôt que RS232. Hormis les tensions électriques et le connecteur, c'est la même chose !

Croisement de données

Il va simplement falloir faire attention à bien croiser les fils. On connecte le Tx (broche de transmission) d'un microcontrôleur au Rx (broche de réception) de l'autre microcontrôleur. Et inversement, le Tx de l'autre au Rx du premier. Et bien sûr, la masse à la masse pour faire une référence commune. Exactement comme le premier schéma que je vous ai montré :



Tx -> Rx, fil vert Rx -> Tx, fil orange Masse -> Masse, fil noir

La couleur des fils importe peu, évidemment ! 😊

Arduino au PC

Le connecteur série (ou sortie DB9)

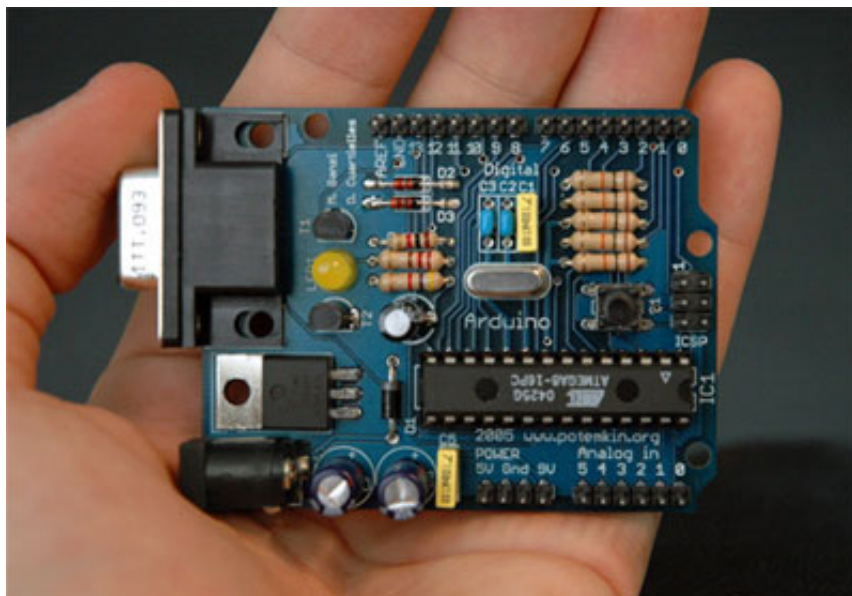
Alors là, les enfants, je vous parle d'un temps que les moins de vingt ans ne peuvent pas connaître... Bon on reprend ! Comme énoncé, je vous parle de quelque chose qui n'existe presque plus. Ou du moins, vous ne trouverez certainement plus cette "chose" sur la connectique de votre ordinateur. En effet, je vais vous parler du connecteur DB9 (ou DE9). Il y a quelques années, l'USB n'était pas si vélocité et surtout pas tant répandu. Beaucoup de matériels (surtout d'un point de vue industriel) utilisaient la voie série (et le font encore). A l'époque, les équipements se branchaient sur ce qu'on appelle une prise DB9 (9 car 9 broches). Sachez simplement que ce nom est attribué à un connecteur qui permet de relier divers matériels informatiques entre eux.



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

A quoi ça sert ?

Si je vous parle de ça dans le chapitre sur la voie série, c'est qu'il doit y avoir un lien, non ? o_O Juste, car la voie série (je parle là de la transmission des données) est véhiculée par ce connecteur dans la norme RS-232. Donc, notre ordinateur dispose d'un connecteur DB9, qui permet de relier, via un câble adapté, sa connexion série à un autre matériel. Avant, donc, lorsqu'il était très répandu, on utilisait beaucoup ce connecteur. D'ailleurs, la première version de la carte Arduino disposait d'un tel connecteur !



La première version de la carte Arduino, avec un connecteur DB9

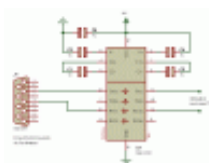
Aujourd'hui, le connecteur DB9 a déjà bien disparu mais reste présent sur les "vieux" ordinateurs ou sur d'autres appareils utilisant la voie série. C'est pourquoi, le jour où vous aurez besoin de communiquer avec un tel dispositif, il vous faudra faire un peu d'électronique...

Une petite histoire d'adaptation

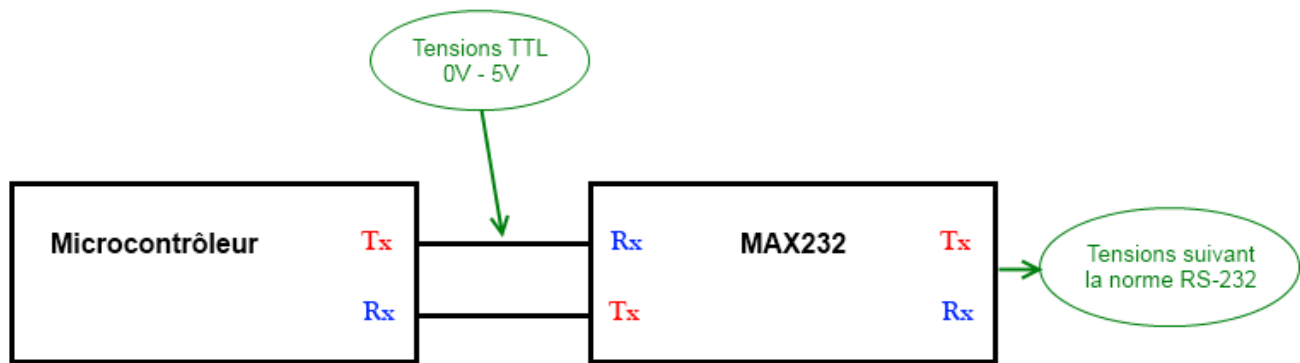
Si vous avez donc l'occasion de connecter votre carte Arduino à un quelconque dispositif utilisant la voie série, il va falloir faire attention aux tensions...oui, encore elles ! Je l'ai déjà dit, un microcontrôleur utilise des tensions de 0V et 5V, qu'on appelle TTL. Hors, la norme RS-232 impose des tensions positives et négatives comprises en +/-3V et +/-25V. Il va donc falloir adapter ces tensions. Pour cela, il existe un composant très courant et très utilisé dans ce type de cas, qu'est le MAX232.

[Datasheet du MAX232](#)

Je vous laisse regarder la datasheet et comprendre un peu le fonctionnement. Aussi, je vous mets un schéma, extrait du site internet sonelec-musique.com :



Le principe de ce composant, utilisé avec quelques condensateurs, est d'adapter les signaux de la voie série d'un microcontrôleur vers des tensions aux standards de la norme RS-232 et inversement. Ainsi, une fois le montage installé, vous n'avez plus à vous soucier de savoir quelle tension il faut, etc...



En revanche, n'utilisez jamais ce composant pour relier deux microcontrôleurs entre eux ! Vous risqueriez d'en griller un. Ou alors il faut utiliser deux fois ce composant (un pour TTL->RS232 et l'autre pour RS232->TTL 🤪), mais cela deviendrait alors peu utile.

Donc en sortie du MAX232, vous aurez les signaux Rx et Tx au standard RS-232. Elles dépendent de son alimentation et sont en générale centrées autour de +/-12V. Vous pourrez par exemple connecter un connecteur DB9 à la sortie du MAX232 et relier la carte Arduino à un dispositif utilisant lui aussi la voie série et un connecteur DB9. Ou même à un dispositif n'utilisant pas de connecteur DB9 mais un autre (dont il faudra connaître le brochage) et qui utilise la voie série.

Au delà d'Arduino avec la connexion série

Voici une petite annexe qui va vous présenter un peu l'utilisation du vrai port série. Je ne vous oblige pas à la lire, elle n'est pas indispensable et peut seulement servir si vous avez un jour besoin de communiquer avec un dispositif qui exploite cette voie série.

Le connecteur série (ou sortie DB9)

Le brochage au complet !

Oui, je veux savoir pourquoi il possède tant de broches puisque tu nous as dit que la voie série n'utilisait que 3 fils.

Eh bien, toutes ces broches ont une fonction bien précise. Je vais vous les décrire, ensuite on verra plus en détail ce que l'on peut faire avec :

1. **DCD** : Détection d'un signal sur la ligne. Utilisée uniquement pour la connexion de l'ordinateur à un modem ; détecte la porteuse
2. **RXD** : Broche de réception des données
3. **TXD** : Broche de transmission des données
4. **DTR** : Le support qui veut recevoir des données se déclare prêt à "écouter" l'autre
5. **GND** : Le référentiel électrique commun ; la masse
6. **DSR** : Le support voulant transmettre déclare avoir des choses à dire
7. **RTS** : Le support voulant transmettre des données indique qu'il voudrait communiquer
8. **CTS** : Invitation à émettre. Le support de réception attend des données
9. **RI** : Très peu utilisé, indiquait la sonnerie dans le cas des modems RS232

Vous voyez déjà un aperçu de ce que vous pouvez faire avec toutes ces broches. Mais parlons-en plus amplement.

Désolé, je suis occupé...

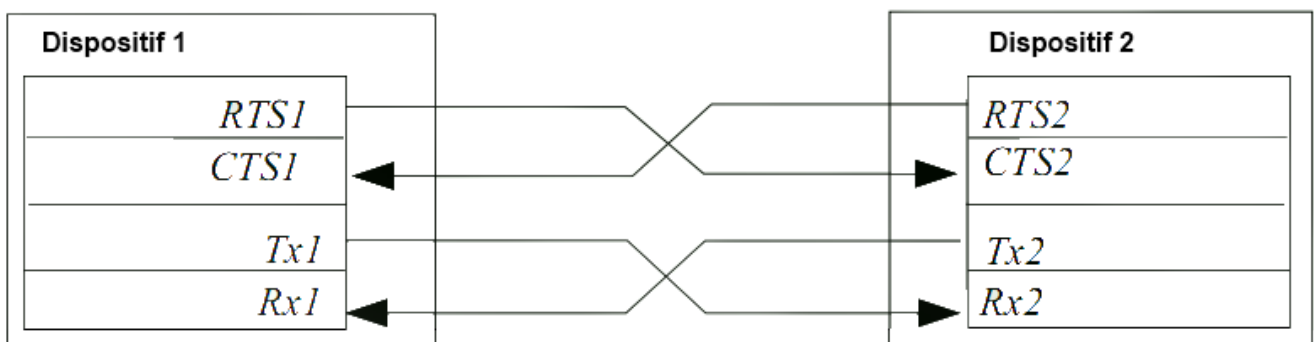
Dans certains cas, et il n'est pas rare, les dispositifs communicant entre eux par l'intermédiaire de la voie série ne traitent pas les données à la même vitesse. Tout comme lorsque l'on communique avec quelqu'un, il arrive parfois qu'il n'arrive plus à suivre ce que l'on dit car il en prend des notes. Il s'annonce alors indisponible à recevoir plus d'informations. Dès qu'il est à nouveau prêt, il nous le fait savoir. Il y a un moyen, mis en place grâce à certaines broches du connecteur pour effectuer ce genre d'opération que l'on appelle le **contrôle de flux**. Il y a deux manières d'utiliser un contrôle de flux, nous allons les voir tout de suite.

Contrôle de flux logiciel

Commençons par le contrôle de flux logiciel, plus simple à utiliser que le contrôle de flux matériel. En effet, il ne nécessite que trois fils : la masse, le Rx et le TX. Eh oui, ni plus ni moins, tout se passe logiciellement. Le fonctionnement très simple de ce contrôle de flux utilise des caractères de la table ASCII, le caractère 17 et 19, respectivement nommés **XON** et **XOFF**. Ceci se passe entre un équipement E, qui est l'émetteur, et un équipement R, qui est récepteur. Le récepteur reçoit des informations, il les traite et stockent celles qui continuent d'arriver en attendant de les traiter. Mais lorsqu'il ne peut plus stocker d'informations, le récepteur envoie le caractère XOFF pour indiquer à l'émetteur qu'il sature et qu'il n'est plus en mesure de recevoir d'autres informations. Lorsqu'il est à nouveau apte à traiter les informations, il envoie le caractère XON pour dire à l'émetteur qu'il est à nouveau prêt à écouter ce que l'émetteur a à lui dire.

Contrôle de flux matériel

On n'utilisera pas le contrôle de flux matériel avec Arduino car la carte n'en est pas équipée, mais il est bon pour vous que vous sachiez ce que c'est. Je ne parlerai en revanche que du contrôle matériel à 5 fils. Il en existe un autre qui utilise 9 fils. Le principe est le même que pour le contrôle logiciel. Cependant, on utilise certaines broches du connecteur DB9 dont je parlais plus haut. Ces broches sont **RTS** et **CTS**.



Voilà le branchement adéquat pour utiliser ce contrôle de flux matériel à 5 fils. Une transmission s'effectue de la manière suivante :

- Le dispositif 1, que je nommerais maintenant l'émetteur, met un état logique 0 sur

sa broche RTS1. Il demande donc au dispositif 2, le récepteur, pour émettre des données.

- Si le récepteur est prêt à recevoir des données, alors il met un niveau logique 0 sur sa broche RTS2.
- Les deux dispositifs sont prêts, l'émetteur peut donc envoyer les données qu'il a à transmettre.
- Une fois les données envoyées, l'émetteur passe à 1 l'état logique présent sur sa broche RTS1.
- Le récepteur voit ce changement d'état et sait donc que c'est la fin de la communication des données, il passe alors l'état logique de sa broche RTS2 à 1.

Ce contrôle n'est pas très compliqué et est utilisé lorsque le contrôle de flux logiciel ne l'est pas.

Avec ou sans horloge ?

Pour terminer, faisons une petite ouverture sur d'autres liaisons séries célèbres...

L'USB

On la côtoie tout les jours sans s'en soucier et pourtant elle nous entoure : C'est la liaison USB ! Comme son nom l'indique, Universal Serial Bus, il s'agit bien d'une voie série. Cette dernière existe en trois versions. La dernière, la 3.1, vient juste de sortir. Une des particularités de cette voie série est qu'elle se propose de livrer l'alimentation de l'équipement avec lequel elle communique. Par exemple votre ordinateur peut alimenter votre disque dur portable et en même temps lui demander des fichiers. Dans le cas de l'USB, la communication se fait de manière "maître-esclave". C'est l'hôte (l'ordinateur) qui demande des informations à l'esclave (le disque dur). Tant qu'il n'a pas été interrogé, ce dernier n'est pas censé parler. Afin de s'y retrouver, chaque périphérique se voit attribuer une adresse. La transmission électrique se fait grâce à un procédé "différentiel" entre deux fils, D+ et D-, afin de limiter les parasites.

L'I2C

L'I2C est un autre protocole de communication qui fut tout d'abord propriétaire (inventé par Philips) et né de la nécessité d'interfacer de plus en plus de microcontrôleurs. En effet, à ce moment là une voie série "classique" ne suffisait plus car elle ne pouvait relier que deux à deux les microcontrôleurs. La particularité de cette liaison est qu'elle transporte son propre signal d'horloge. Ainsi, la vitesse n'a pas besoin d'être connu d'avance. Les données sont transportées en même temps que l'horloge grâce à deux fils : SDA (Data) et SCL (Clock). Comme pour l'USB, la communication se fait sur un système de maître/esclave.

[Arduino 302] Envoyer et recevoir des données sur la voie série


Dans ce chapitre, nous allons apprendre à utiliser la voie série avec Arduino. Nous allons voir comment envoyer puis recevoir des informations avec l'ordinateur, enfin

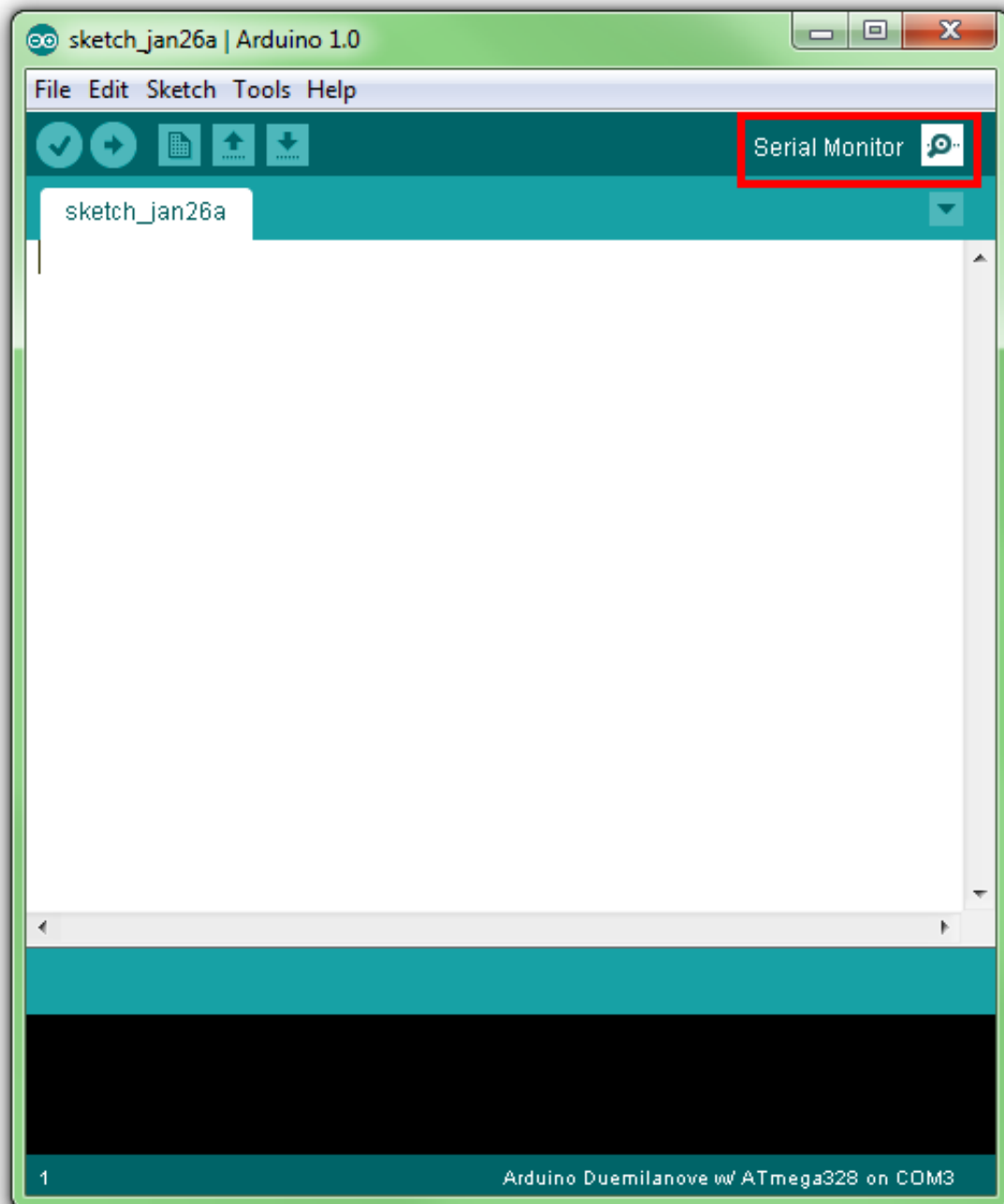
nous ferons quelques exercices pour vérifier que vous avez tout compris. 😊 Vous allez le découvrir bientôt, l'utilisation de la voie série avec Arduino est quasiment un jeu d'enfant, puisque tout est opaque aux yeux de l'utilisateur...

Préparer la voie série

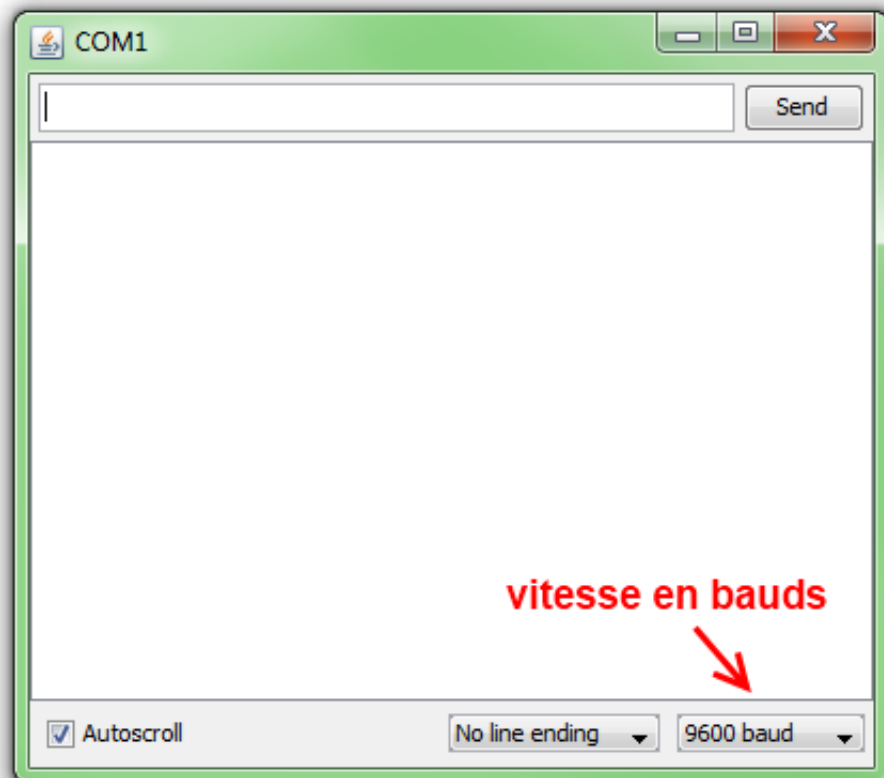
Notre objectif, pour le moment, est de communiquer des informations de la carte Arduino vers l'ordinateur et inversement. Pour ce faire, on va d'abord devoir préparer le terrain.

Du côté de l'ordinateur

Pour pouvoir utiliser la communication de l'ordinateur, rien de plus simple. En effet, L'environnement de développement Arduino propose de base un outil pour communiquer. Pour cela, il suffit de cliquer sur le bouton  (pour les versions antérieures à la version 1.0) dans la barre de menu pour démarrer l'outil. Pour la version 1.0, l'icône a changé et de place et de visuel :



Une nouvelle fenêtre s'ouvre : c'est le **terminal série** :



Dans cette fenêtre, vous allez pouvoir envoyer des messages sur la voie série de votre ordinateur (qui est émulée par l'Arduino) ; recevoir les messages que votre Arduino vous envoie ; et régler deux trois paramètres tels que la vitesse de communication avec l'Arduino et l'autoscroll qui fait défiler le texte automatiquement. On verra plus loin à quoi sert le dernier réglage.

Du côté du programme

L'objet Serial

Pour utiliser la voie série et communiquer avec notre ordinateur (par exemple), nous allons utiliser un *objet* (une sorte de variable mais plus évoluée) qui est intégré nativement dans l'ensemble Arduino : l'objet **Serial**.

Pour le moment, considérez qu'un objet est une variable évoluée qui peut exécuter plusieurs fonctions. On verra (beaucoup) plus loin ce que sont réellement des objets. On apprendra à en créer et à les utiliser lorsque l'on abordera le logiciel [Processing](#).

Cet objet rassemble des informations (vitesse, bits de données, etc.) et des fonctions (envoi, lecture de réception,...) sur ce qu'est une voie série pour Arduino. Ainsi, pas besoin pour le programmeur de recréer tout le protocole (sinon on aurait dû écrire nous même TOUT le protocole, tel que "Ecrire un bit haut pendant 1 ms, puis 1 bit bas pendant 1 ms, puis le caractère 'a' en 8 ms...), bref, on gagne un temps fou et on évite les bugs !

Le setup

Pour commencer, nous allons donc initialiser l'objet Serial. Ce code sera à copier à

chaque fois que vous allez créer un programme qui utilise la voie série. Le logiciel Arduino a prévu, dans sa *bibliothèque Serial*, tout un tas de fonctions qui vont nous être très utiles, voir même indispensables afin de bien utiliser la voie série. Ces fonctions, je vous les laisse découvrir par vous même si vous le souhaitez, elles se trouvent sur [cette page](#). Dans le but de créer une communication entre votre ordinateur et votre carte Arduino, il faut déclarer cette nouvelle communication et définir la vitesse à laquelle ces deux dispositifs vont communiquer. Et oui, si la vitesse est différente, l'Arduino ne comprendra pas ce que veut lui transmettre l'ordinateur et vice versa ! Ce réglage va donc se faire dans la fonction setup, en utilisant la fonction `begin()` de l'objet Serial.

Lors d'une communication informatique, une vitesse s'exprime en bits par seconde ou **bauds**. Ainsi, pour une vitesse de 9600 bauds on enverra jusqu'à 9600 '0' ou '1' en une seule seconde. Les vitesses les plus courantes sont 9600, 19200 et 115200 bits par seconde.

```
1 void setup()  
2 {  
3     //on démarre la liaison en la réglant à une vitesse de 9600 bits par seconde.  
4     Serial.begin(9600);  
5 }
```

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

Envoyer des données

Le titre est piégeur, en effet, cela peut être l'Arduino qui envoie des données ou l'ordinateur. Bon, on est pas non plus dénué d'une certaine logique puisque pour envoyer des données à partir de l'ordinateur vers la carte Arduino il suffit d'ouvrir le terminal série et de taper le texte dedans ! 😊 Donc, on va bien programmer et voir comment faire pour que votre carte Arduino envoie des données à l'ordinateur.

Et ces données, elles proviennent d'où ?

Eh bien de la carte Arduino... En fait, lorsque l'on utilise la voie série pour transmettre de l'information, c'est qu'on en a de l'information à envoyer, sinon cela ne sert à rien. Ces informations proviennent généralement de capteurs connectés à la carte ou de son programme (par exemple la valeur d'une variable). La carte Arduino traite les informations provenant de ces capteurs, s'il faut elle adapte ces informations, puis elle les transmet. On aura l'occasion de faire ça dans la partie dédiée aux capteurs, comme afficher la température sur son écran, l'heure, le passage d'une personne, etc.

Appréhender l'objet Serial

Dans un premier temps, nous allons utiliser l'objet Serial pour tester quelques envois de données. Puis nous nous attèlerons à un petit exercice que vous ferez seul ou presque, du moins vous aurez eu auparavant assez d'informations pour pouvoir le réaliser (ben oui, sinon c'est plus un exercice !).

Phrase ? Caractère ?

On va commencer par envoyer un caractère et une phrase. À ce propos, savez-vous quelle est la correspondance entre un caractère et une phrase ? Une phrase est constituée de caractères les uns à la suite des autres. En programmation, on parle plutôt de **chaîne caractères** pour désigner une phrase.

- Un caractère seul s'écrit entre guillemets simples : 'A', 'a', '2', '!', ...
- Une phrase est une suite de caractère et s'écrit entre guillemets doubles : "Salut tout le monde", "J'ai 42 ans", "Vive Zozor !"

Pour vous garantir un succès dans le monde de l'informatique, essayez d'y penser et de respecter cette convention, écrire 'A' ce n'est pas pareil qu'écrire "A" !

println()

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

- `print()` : cette fonction permet d'envoyer des données sur la voie série. On peut par exemple envoyer un caractère, une chaîne de caractère ou d'autres données dont je ne vous ai pas encore parlé.
- `println()` : c'est la même fonction que la précédente, elle permet simplement un retour à la ligne à la fin du message envoyé.

Pour utiliser ces fonctions, rien de plus simple :

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

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

```
1 void setup()  
2 {  
3     //création de l'objet Serial (=établissement d'une nouvelle communication série)  
4     Serial.begin(9600);  
5     //envoi de la chaîne "Salut les zéros !" sur la voie série  
6     Serial.print("Salut les zéros !");  
7 }
```

Cet objet, parlons-en. Pour vous aider à représenter de façon plus concise ce qu'est l'objet Serial, je vous propose cette petite illustration de mon propre chef :

Programme

Objet Serial

fonctions :

print()
println()
write()
read()
...

appel des
fonctions



Comme je vous le présente, l'objet Serial est muni d'un panel de fonctions qui lui sont propres. Cet objet est capable de réaliser ces fonctions selon ce que le programme lui ordonne de faire. Donc, par exemple, quand j'écris : `print()` en lui passant pour paramètre la chaîne de caractère : "Salut les zéros !". On peut compléter le code précédent comme ceci :

```
1 void setup()  
2 {  
3     Serial.begin(9600);  
4  
5     //l'objet exécute une première fonction  
6     Serial.print("Salut les zéros ! ");  
7     //puis une deuxième fonction, différente cette fois-ci  
8     Serial.println("Vive Zozor !");  
9     //et exécute à nouveau la même  
10    Serial.println("Cette phrase passe en dessous des deux précédentes");  
11 }
```

Sur le terminal série, on verra ceci :

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

La fonction `print()` en détail

Après cette courte prise en main de l'objet Serial, je vous propose de découvrir plus en profondeur les surprises que nous réserve la fonction `print()`.

Petite précision, je vais utiliser de préférence `print()`.

Résumons un peu ce que nous venons d'apprendre : on sait maintenant envoyer des caractères sur la voie série et des phrases. C'est déjà bien, mais ce n'est qu'un très bref aperçu de ce que l'on peut faire avec cette fonction.

Envoyer des nombres

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

```
1 void setup()
2 {
3     Serial.begin(9600);
4
5     Serial.println(9);           //chiffre
6     Serial.println(42);          //nombre
7     Serial.println(32768);       //nombre
8     Serial.print(3.1415926535);  //nombre à virgule
9 }
```

```
1 9
2 42
3 32768
4 3.14
```

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

Rassurez-vous, ce n'est ni un bug, ni un oubli inopiné de ma part. 🤖 En fait, pour les nombres décimaux, la fonction `print()` affiche par défaut seulement deux chiffres après la virgule. C'est la valeur par défaut et heureusement elle est modifiable. Il suffit de rajouter le nombre de décimales que l'on veut afficher :

```
1 void setup()
2 {
3     Serial.begin(9600);
4
5     Serial.println(3.1415926535, 0);
6     Serial.println(3.1415926535, 2); //valeur par défaut
7     Serial.println(3.1415926535, 4);
8     Serial.println(3.1415926535, 10);
9 }
```

```
1 3
2 3.14
3 3.1415
4 3.1415926535
```

Envoyer la valeur d'une variable

Là encore, on utilise toujours la même fonction (qu'est-ce qu'elle polyvalente !). Ici aucune surprise. Au lieu de mettre un caractère ou un nombre, il suffit de passer la

variable en paramètre pour qu'elle soit ensuite affichée à l'écran :

```
1 int variable = 512;
2 char lettre = 'a';
3
4 void setup()
5 {
6     Serial.begin(9600);
7
8     Serial.println(variable);
9     Serial.print(lettre);
10 }
```

```
1 512
2 a
```

Trop facile n'est-ce pas ?

Envoyer d'autres données

Ce n'est pas fini, on va terminer notre petit tour avec les types de variables que l'on peut transmettre grâce à cette fonction `print()` sur la voie série. Prenons l'exemple d'un nombre choisi judicieusement : 65.

Pourquoi ce nombre en particulier ? Et pourquoi pas 12 ou 900 ?

Eh bien, c'est relatif à la **table ASCII** que nous allons utiliser dans un instant.

Tout d'abord, petit cours de prononciation, ASCII se prononce comme si on disait "A ski", on a donc : "la table à ski" en prononciation phonétique.

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

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

En somme, c'est un tableau de valeurs codées sur 8bits qui à chaque valeur associent un caractère. Ces caractères sont les lettres de l'alphabet en minuscule et majuscule, les chiffres, des caractères spéciaux et des symboles bizarres. Dans cette table, il y a plusieurs colonnes avec la valeur décimale, la valeur hexadécimale, la valeur binaire et la valeur octale parfois. Nous n'aurons pas besoin de tout ça, donc je vous donne une table ASCII "allégée".

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

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
128	80	Ç	160	A0	á	192	C0	Ł	224	E0	α
129	81	ü	161	A1	í	193	C1	ł	225	E1	β
130	82	é	162	A2	ó	194	C2	Ť	226	E2	Γ
131	83	â	163	A3	ú	195	C3	ť	227	E3	π
132	84	ä	164	A4	ñ	196	C4	—	228	E4	Σ
133	85	à	165	A5	Ñ	197	C5	†	229	E5	σ
134	86	å	166	A6	*	198	C6	‡	230	E6	μ
135	87	ç	167	A7	°	199	C7	‡	231	E7	ι
136	88	ê	168	A8	ˆ	200	C8	£	232	E8	φ
137	89	ë	169	A9	˜	201	C9	₣	233	E9	θ
138	8A	è	170	AA	˘	202	CA	₤	234	EA	Ω
139	8B	ï	171	AB	½	203	CB	₥	235	EB	δ
140	8C	î	172	AC	¼	204	CC	₦	236	EC	∞
141	8D	ì	173	AD		205	CD	=	237	ED	ψ
142	8E	Ä	174	AE	«	206	CE	₧	238	EE	ε
143	8F	Å	175	AF	»	207	CF	₨	239	EF	∩
144	90	É	176	B0	⋮	208	D0	₪	240	FO	≡
145	91	æ	177	B1	⋮	209	D1	₫	241	F1	±
146	92	Æ	178	B2	⋮	210	D2	€	242	F2	≥
147	93	ø	179	B3	⋮	211	D3	₭	243	F3	≤
148	94	ö	180	B4	⋮	212	D4	₮	244	F4	∫
149	95	ò	181	B5	⋮	213	D5	₯	245	F5	
150	96	ó	182	B6	⋮	214	D6	₰	246	F6	÷
151	97	û	183	B7	⋮	215	D7	₱	247	F7	≈
152	98	ÿ	184	B8	⋮	216	D8	₲	248	F8	∞
153	99	Û	185	B9	⋮	217	D9	₳	249	F9	·
154	9A	Ü	186	BA	⋮	218	DA	₴	250	FA	˙
155	9B	φ	187	BB	⋮	219	DB	₵	251	FB	√
156	9C	£	188	BC	⋮	220	DC	₶	252	FC	ˆ
157	9D	¥	189	BD	⋮	221	DD	₷	253	FD	²
158	9E	₣	190	BE	⋮	222	DE	₸	254	FE	■
159	9F	₤	191	BF	⋮	223	DF	₹	255	FF	

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

Voici une deuxième table avec les caractères et symboles affichés :

0	32	64	Q	96	'	128	Ç	160	á	192	Ł	224	Ó
1	☐	33	!	65	À	97	a	129	ü	161	í	193	Ɔ
2	☐	34	"	66	B	98	b	130	é	162	ó	194	Ô
3	♥	35	#	67	C	99	c	131	â	163	ú	195	Ɔ
4	♦	36	\$	68	D	100	d	132	ä	164	ñ	196	õ
5	♣	37	%	69	E	101	e	133	à	165	Ñ	197	Õ
6	♠	38	&	70	F	102	f	134	ã	166	•	198	μ
7	•	39	'	71	G	103	g	135	ç	167	•	199	Ɔ
8	◼	40	(72	H	104	h	136	ê	168	ı	200	Ɔ
9	◊	41)	73	I	105	i	137	ë	169	®	201	Ú
10	◼	42	*	74	J	106	j	138	è	170	¬	202	Û
11	♂	43	+	75	K	107	k	139	ï	171	½	203	Ù
12	♀	44	,	76	L	108	l	140	î	172	¼	204	Ú
13	♯	45	_	77	M	109	m	141	ì	173	ı	205	Ý
14	♯	46	.	78	N	110	n	142	Ä	174	«	206	'
15	✱	47	/	79	O	111	o	143	Å	175	»	207	'
16	▶	48	0	80	P	112	p	144	É	176	▤	208	-
17	◀	49	1	81	Q	113	q	145	æ	177	▥	209	±
18	↕	50	2	82	R	114	r	146	ff	178	▦	210	=
19	!!	51	3	83	S	115	s	147	ô	179		211	¾
20	¶	52	4	84	T	116	t	148	ö	180	†	212	¶
21	§	53	5	85	U	117	u	149	ò	181	Á	213	§
22	—	54	6	86	V	118	v	150	û	182	Â	214	÷
23	±	55	7	87	W	119	w	151	ù	183	À	215	˘
24	↑	56	8	88	X	120	x	152	ÿ	184	©	216	•
25	↓	57	9	89	Y	121	y	153	ö	185	¶	217	ˆ
26	→	58	:	90	Z	122	z	154	ü	186		218	˙
27	←	59	;	91	[123	{	155	ø	187	¶	219	ı
28	└	60	<	92	\	124	!	156	£	188	¶	220	3
29	•	61	=	93]	125	}	157	Ø	189	¢	221	2
30	▲	62	>	94	^	126	~	158	×	190	¥	222	■
31	▼	63	?	95	_	127	△	159	f	191	ł	223	■

Source de cette table : http://www.lyceedupaysdesoule.fr/informatique/divers/table_ascii.htm

Revenons à notre exemple, le nombre 65. C'est en effet grâce à la table ASCII que l'on sait passer d'un nombre à un caractère, car rappelons-le, dans l'ordinateur tout est traité

sous forme de nombre en base 2 (binaire). Donc lorsque l'on code :

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

Si vous faites ensuite :

```
1 maVariable = maVariable + 1; //la valeur stockée passe à 66 (= 65 + 1)
2
3 //à l'écran, on verra s'afficher la lettre "B"
```

Au début, on trouvait une seule table ASCII, qui allait de 0 à 127 (codée sur 7bits) et représentait l'alphabet, les chiffres arabes et quelques signes de ponctuation. Depuis, de nombreuses tables dites "étendues" sont apparues et vont de 0 à 255 caractères (valeurs maximales codables sur un type *char* qui fait 8 bits).

Et que fait-on avec la fonction `print()` et cette table ?

Là est tout l'intérêt de la table, on peut envoyer des données, avec la fonction `print()`, de tous types ! En binaire, en hexadécimal, en octal et en décimal.

```
1 void setup()
2 {
3     Serial.begin(9600);
4
5     Serial.println(65, BIN); //envoie la valeur 1000001
6     Serial.println(65, DEC); //envoie la valeur 65
7     Serial.println(65, OCT); //envoie la valeur 101 (ce n'est pas du binaire !)
8     Serial.println(65, HEX); //envoie la valeur 41
9 }
```

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

Exercice : Envoyer l'alphabet

Objectif

Nous allons maintenant faire un petit exercice, histoire de s'entraîner à envoyer des données. Le but, tout simple, est d'envoyer l'ensemble des lettres de l'alphabet de manière *la plus intelligente* possible, autrement dit, sans écrire 26 fois "`print()`"... La fonction `setup` restera la même que celle vue précédemment. Un délai de 250 ms est attendu entre chaque envoi de lettre et un delay de 5 secondes est attendu entre l'envoi de deux alphabets.

Bon courage !

Correction

Bon j'espère que tout c'est bien passé et que vous n'avez pas joué au roi du copier/coller en me mettant 26 `print`...

```
1 void loop()
2 {
3     char i = 0;
```

```

4 char lettre = 'a'; // ou 'A' pour envoyer en majuscule
5
6 Serial.println("----- L'alphabet des Zéros -----"); //petit message d'accuei
7
8 //on commence les envois
9 for(i=0; i<26; i++)
10 {
11     Serial.print(lettre); //on envoie la lettre
12     lettre = lettre + 1; //on passe à la lettre suivante
13     delay(250); //on attend 250ms avant de réenvoyer
14 }
15 Serial.println(""); //on fait un retour à la ligne
16
17 delay(5000); //on attend 5 secondes avant de renvoyer l'alphabet
18 }

```

Si l'exercice vous a paru trop simple, vous pouvez essayer d'envoyer l'alphabet à l'envers, ou l'alphabet minuscule ET majuscule ET les chiffres de 0 à 9... Amusez-vous bien ! 😊

Recevoir des données

Cette fois, il s'agit de l'Arduino qui reçoit les données que nous, utilisateur, allons transmettre à travers le terminal série. Je vais prendre un exemple courant : une communication téléphonique. En règle générale, on dit "Hallo" pour dire à l'interlocuteur que l'on est prêt à écouter le message. Tant que la personne qui appelle n'a pas cette confirmation, elle ne dit rien (ou dans ce cas elle fait un monologue 🗣️). Pareillement à cette conversation, l'objet Serial dispose d'une fonction pour "écouter" la voie série afin de savoir si oui ou non il y a une communication de données.

Réception de données

On m'a parlé ?

Pour vérifier si on a reçu des données, on va régulièrement interroger la carte pour lui demander si des données sont disponibles dans son **buffer de réception**. Un buffer est une zone mémoire permettant de stocker des données sur un court instant. Dans notre situation, cette mémoire est dédiée à la réception sur la voie série. Il en existe un aussi pour l'envoi de donnée, qui met à la queue leu leu les données à envoyer et les envoie dès que possible. En résumé, un buffer est une sorte de salle d'attente pour les données. Je disais donc, nous allons régulièrement vérifier si des données sont arrivées. Pour cela, on utilise la fonction `available()` (de l'anglais "disponible") de l'objet Serial. Cette fonction renvoie le nombre de caractères dans le buffer de réception de la voie série. Voici un exemple de traitement :

```

1 void loop()
2 {
3     int donneesALire = Serial.available(); //lecture du nombre de caractères disponib
4     if(donneesALire > 0) //si le buffer n'est pas vide
5     {
6         //Il y a des données, on les lit et on fait du traitement
7     }
8     //on a fini de traiter la réception ou il n'y a rien à lire
9 }

```

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

Lire les données reçues

Une fois que l'on sait qu'il y a des données, il faut aller les lire pour éventuellement en faire quelque chose. La lecture se fera tout simplement avec la fonction... `read()` ! Cette fonction renverra le premier caractère arrivé non traité (comme un supermarché traite la première personne arrivée dans la file d'attente de la caisse avant de passer au suivant). On accède donc caractère par caractère aux données reçues. Ce type de fonctionnement est appelé FIFO (First In First Out, premier arrivé, premier traité). Si jamais rien n'est à lire (personne dans la file d'attente), je le disais, la fonction renverra -1 pour le signaler.

```
1 void loop()
2 {
3     //on lit le premier caractère non traité du buffer
4     char choseLue = Serial.read();
5
6     if(choseLue == -1) //si le buffer est vide
7     {
8         //Rien à lire, rien lu
9     }
10    else //le buffer n'est pas vide
11    {
12        //On a lu un caractère
13    }
14 }
```

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

Le serialEvent

Si vous voulez éviter de mettre le test de présence de données sur la voie série dans votre code, Arduino a rajouté une fonction qui s'exécute de manière régulière. Cette dernière se lance régulièrement avant chaque redémarrage de la loop. Ainsi, si vous n'avez pas besoin de traiter les données de la voie série à un moment précis, il vous suffit de rajouter cette fonction. Pour l'implémenter c'est très simple, il suffit de mettre du code dans une fonction nommée "serialEvent()" (attention à la casse) qui sera rajoutée en dehors du setup et du loop. Le reste du traitement de texte se fait normalement, avec `Serial.read` par exemple. Voici un exemple de squelette possible :

```
1 const int maLed = 11; //on met une LED sur la broche 11
2
3 void setup()
4 {
5     pinMode(maLed, OUTPUT); //la LED est une sortie
6     digitalWrite(maLed, HIGH); //on éteint la LED
7     Serial.begin(9600); //on démarre la voie série
8 }
9
10 void loop()
11 {
12     delay(500); //fait une petite pause
```



```

13 //on ne fait rien dans la loop
14 digitalWrite(maLed, HIGH); //on éteint la LED
15
16 }
17
18 void serialEvent() //déclaration de la fonction d'interruption sur la voie série
19 {
20     //lit toutes les données (vide le buffer de réception)
21     while(Serial.read() != -1);
22
23     //puis on allume la LED
24     digitalWrite(maLed, LOW);
25 }

```

Exemple de code complet

Voici maintenant un exemple de code complet qui va aller lire les caractères présents dans le buffer de réception s'il y en a et les renvoyer tels quels à l'expéditeur (mécanisme d'écho).

```

1 void setup()
2 {
3     Serial.begin(9600);
4 }
5
6 void loop()
7 {
8     char carlu = 0; //variable contenant le caractère à lire
9     int cardispo = 0; //variable contenant le nombre de caractère disponibles dans le buffer
10
11     cardispo = Serial.available();
12
13     while(cardispo > 0) //tant qu'il y a des caractères à lire
14     {
15         carlu = Serial.read(); //on lit le caractère
16         Serial.print(carlu); //puis on le renvoi à l'expéditeur tel quel
17         cardispo = Serial.available(); //on relit le nombre de caractères dispo
18     }
19     //fin du programme
20 }

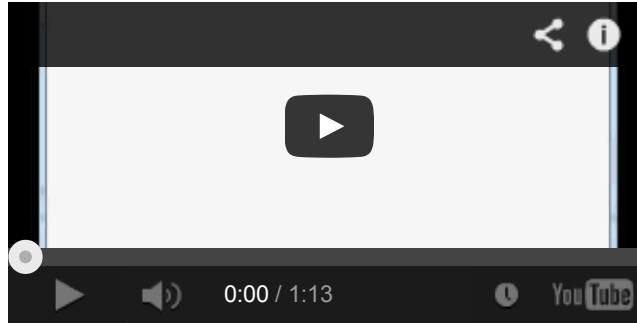
```

Avouez que tout cela n'était pas bien difficile. Je vais donc en profiter pour prendre des vacances et vous laisser faire un exercice qui demande un peu de réflexion. 😊

[Exercice] Attention à la casse !

Consigne

Le but de cet exercice est très simple. L'utilisateur saisit un caractère à partir de l'ordinateur et si ce caractère est minuscule, il est renvoyé en majuscule ; s'il est majuscule il est renvoyé en minuscule. Enfin, si le caractère n'est pas une lettre on se contente de le renvoyer normalement, tel qu'il est. Voilà le résultat de mon programme :



Correction

Je suppose que grâce au superbe tutoriel qui précède vous avez déjà fini sans problème, n'est-ce pas ? 😊

La fonction setup() et les variables utiles

Une fois n'est pas coutume, on va commencer par énumérer les variables utiles et le contenu de la fonction setup(). Pour ce qui est des variables globales, on n'en retrouve qu'une seule, "carlu". Cette variable de type int sert à stocker le caractère lu sur le buffer de la carte Arduino. Puis on démarre une nouvelle voie série à 9600bauds :

```
1 int carlu; //stock le caractère lu sur la voie série
2
3 void setup()
4 {
5     Serial.begin(9600);
6 }
```

Le programme

Le programme principal n'est pas très difficile non plus. Il va se faire en trois temps.

- Tout d'abord, on boucle jusqu'à recevoir un caractère sur la voie série
- Lorsqu'on a reçu un caractère, on regarde si c'est une lettre
- Si c'est une lettre, on renvoie son acolyte majuscule ; sinon on renvoie simplement le caractère lu

Voici le programme décrivant ce comportement :

```
1 void loop()
2 {
3     //on commence par vérifier si un caractère est disponible dans le buffer
4     if(Serial.available() > 0)
5     {
6         carlu = Serial.read(); //lecture du premier caractère disponible
7
8         if(carlu >= 'a' && carlu <= 'z') //Est-ce que c'est un caractère minuscule ?
9         {
10             carlu = carlu - 'a'; //on garde juste le "numéro de lettre"
11             carlu = carlu + 'A'; //on passe en majuscule
12         }
13         else if(carlu >= 'A' && carlu <= 'Z') //Est-ce que c'est un caractère MAJUSC
14         {
15             carlu = carlu - 'A'; //on garde juste le "numéro de lettre"
```

```

16     carlu = carlu + 'a'; //on passe en minuscule
17     }
18     //ni l'un ni l'autre on renvoie en tant que BYTE ou alors on renvoie le caractere
19     Serial.write(carlu);
20 }
21 }

```

Je vais maintenant vous expliquer les parties importantes de ce code. Comme vu dans le cours, la ligne 4 va nous servir à attendre un caractère sur la voie série. Tant qu'on ne reçoit rien, on ne fait rien ! Sitôt que l'on reçoit un caractère, on va chercher à savoir si c'est une lettre. Pour cela, on va faire deux tests. L'un est à la ligne 8 et l'autre à la ligne 13. Ils se présentent de la même façon : **SI** le caractère lu à une valeur supérieure ou égale à la lettre 'a' (ou 'A') **ET** inférieure ou égale à la lettre 'z' ('Z'), alors on est en présence d'une lettre. Sinon, c'est autre chose, donc on se contente de passer au renvoi du caractère lu ligne 21. Une fois que l'on a détecté une lettre, on effectue quelques transformations afin de changer sa casse. Voici les explications à travers un exemple :

Description	Opération (lettre)	Opération (nombre)	Valeur de carlu
On récupère la lettre 'e'	e	101	'e'
On isole son numéro de lettre en lui enlevant la valeur de 'a'	e-a	101-97	4
On ajoute ce nombre à la lettre 'A'	A + (e-a)	65 + (101-97) = 69	'E'
Il ne suffit plus qu'à retourner cette lettre	'E'	69	E

On effectuera sensiblement les mêmes opérations lors du passage de majuscule à minuscule.

A la ligne 19, j'utilise la fonction `write()` qui envoie le caractère en tant que variable de type *byte*, signifiant que l'on renvoie l'information sous la forme d'un seul octet. Sinon Arduino enverrait le caractère en tant que 'int', ce qui donnerait des problèmes lors de l'affichage.

Vous savez maintenant lire et écrire sur la voie série de l'Arduino ! Grâce à cette nouvelle corde à votre arc, vous allez pouvoir ajouter une touche d'interactivité supplémentaire à vos programmes.

[Arduino 303] [TP] Baignade interdite !

Afin d'appliquer vos connaissances acquises durant la lecture de ce tutoriel, nous allons maintenant faire un **gros TP**. Il regroupera tout ce que vous êtes censé savoir en terme de matériel (LED, boutons, voie série et bien entendu Arduino) et je vous fais aussi confiance pour utiliser au mieux vos connaissances en terme de "savoir coder" (variables, fonctions, tableaux...). Bon courage et, le plus important : Amusez-vous bien !

Sujet du TP sur la voie série

Contexte

Imaginez-vous au bord de la plage. Le ciel est bleu, la mer aussi... Ahhh le rêve. Puis, tout un coup le drapeau rouge se lève ! “Requiiiiinn” crie un nageur... L’application que je vous propose de développer ici correspond à ce genre de situation. Vous êtes au QG de la zPlage, le nouvel endroit branché pour les vacances. Votre mission si vous l’acceptez est d’afficher en temps réel un indicateur de qualité de la plage et de ses flots. Pour cela, vous devez informer les zTouristes par l’affichage d’un code de 3 couleurs. Des zSurveillants sont là pour vous prévenir que tout est rentré dans l’ordre si un incident survient.

Objectif

Comme expliqué ci-dessus, l’affichage de qualité se fera au travers de 3 couleurs qui seront représentées par des LEDs :

- **Rouge** : Danger, ne pas se baigner
- **Orange** : Baignade risquée pour les novices
- **Vert** : Tout baigne !

La zPlage est équipée de deux boutons. L’un servira à déclencher un SOS (si quelqu’un voit un nageur en difficulté par exemple). La lumière passe alors au rouge clignotant jusqu’à ce qu’un sauveteur ait appuyé sur l’autre bouton signalant “**Problème réglé, tout revient à la situation précédente**”. Enfin, dernier point mais pas des moindres, le QG (vous) reçoit des informations météorologiques et provenant des marins au large. Ces messages sont retransmis sous forme de textos (symbolisés par la voie série) aux sauveteurs sur la plage pour qu’ils changent les couleurs en temps réel. Voici les mots-clés et leurs impacts :

- **meduse, tempete, requin** : Des animaux dangereux ou la météo rendent la zPlage dangereuse. Baignade interdite
- **vague** : La natation est réservée aux bons nageurs
- **surveillant, calme** : Tout baigne, les zSauveteurs sont là et la mer est cool

Conseil

Voici quelques conseils pour mener à bien votre objectif.

Réalisation

- Une fois n’est pas coutume, **nommez bien vos variables** ! Vous verrez que dès qu’une application prend du volume il est agréable de ne pas avoir à chercher qui sert à quoi. – N’hésitez pas à **décomposer votre code en fonction**. Par exemple les fonctions `changerDeCouleur()` peuvent-être les bienvenues. 😊

Précision sur les chaines de caractères

Lorsque l'on écrit une phrase, on a l'habitude de la finir par un point. En informatique c'est pareil mais à l'échelle du mot ! Je m'explique. Une chaîne de caractères (un mot) est, comme l'indique son nom, une suite de caractères. Généralement on la déclare de la façon suivante :

```
1 char mot[20] = "coucou"
```

Lorsque vous faites ça, vous ne le voyez pas, l'ordinateur rajoute juste après le dernier caractère (ici 'u') un caractère invisible qui s'écrit '\0' (antislash-zéro). Ce caractère signifie "fin de la chaîne". En mémoire, on a donc :

```
mot[0] 'c'  
mot[1] 'o'  
mot[2] 'u'  
mot[3] 'c'  
mot[4] 'o'  
mot[5] 'u'  
mot[6] '\0'
```

Ce caractère est **très important** pour la suite car je vais vous donner un petit coup de pouce pour le traitement des mots reçus.

Une bibliothèque, nommée "string" (chaîne en anglais) et présente nativement dans votre logiciel Arduino, permet de traiter des chaînes de caractères. Vous pourrez ainsi plus facilement comparer deux chaînes avec la fonction `strcmp(chaine1, chaine2)`. Cette fonction vous renverra 0 si les deux chaînes sont identiques. Vous pouvez par exemple l'utiliser de la manière suivante :

```
1 int resultat = strcmp(motRecu, "requin"); //utilisation de la fonction strcmp(chaine1  
2  
3 if(resultat == 0)  
4     Serial.print("Les chaines sont identiques");  
5 else  
6     Serial.print("Les chaines sont différentes");
```

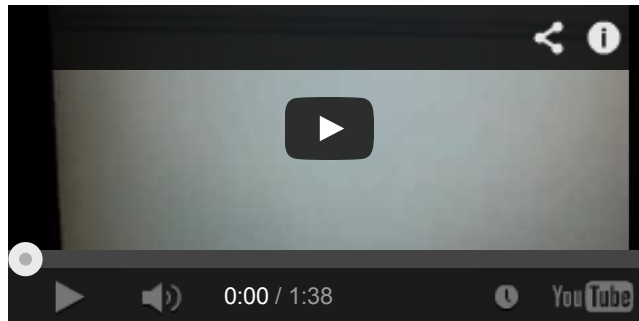
Le truc, c'est que cette fonction compare *caractère par caractère* les chaînes, or celle de droite : "requin" possède ce fameux '\0' après le 'n'. Pour que le résultat soit identique, il faut donc que les deux chaînes soient parfaitement identiques ! Donc, avant d'envoyer la chaîne tapée sur la voie série, il faut lui rajouter ce fameux '\0'.

Je comprends que ce point soit délicat à comprendre, je ne vous taperais donc pas sur les doigts si vous avez des difficultés lors de la comparaison des chaînes et que vous allez vous balader sur la solution... Mais essayez tout de même, c'est tellement plus sympa de réussir en réfléchissant et en essayant ! 😊

Résultat

Prenez votre temps, faites-moi quelque chose de beau et amusez-vous bien ! Je vous laisse aussi choisir comment et où brancher les composants sur votre carte Arduino.

:ninja: Voici une photo d'illustration du montage ainsi qu'une vidéo du montage en action.



Bon Courage !

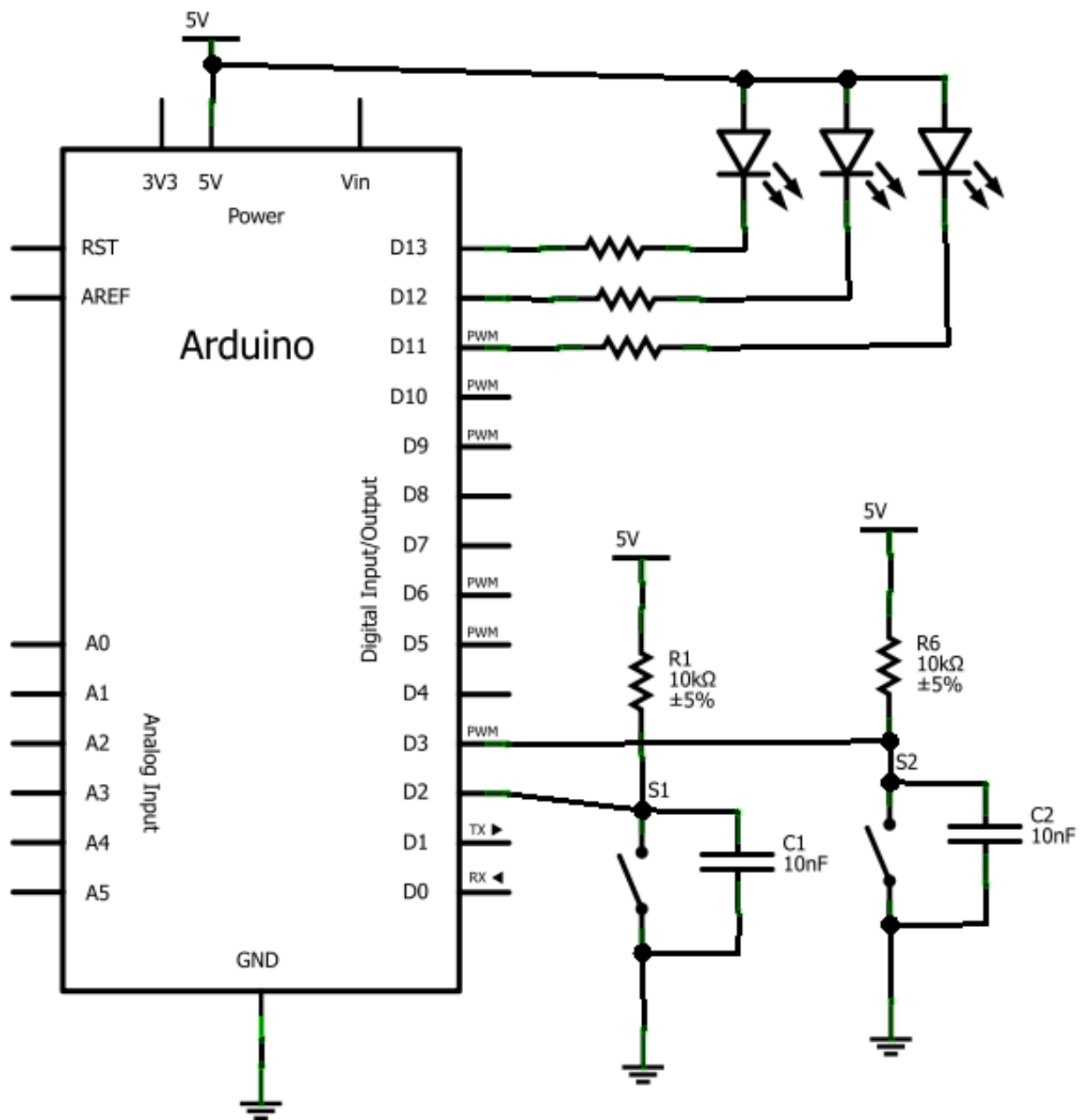
Correction !

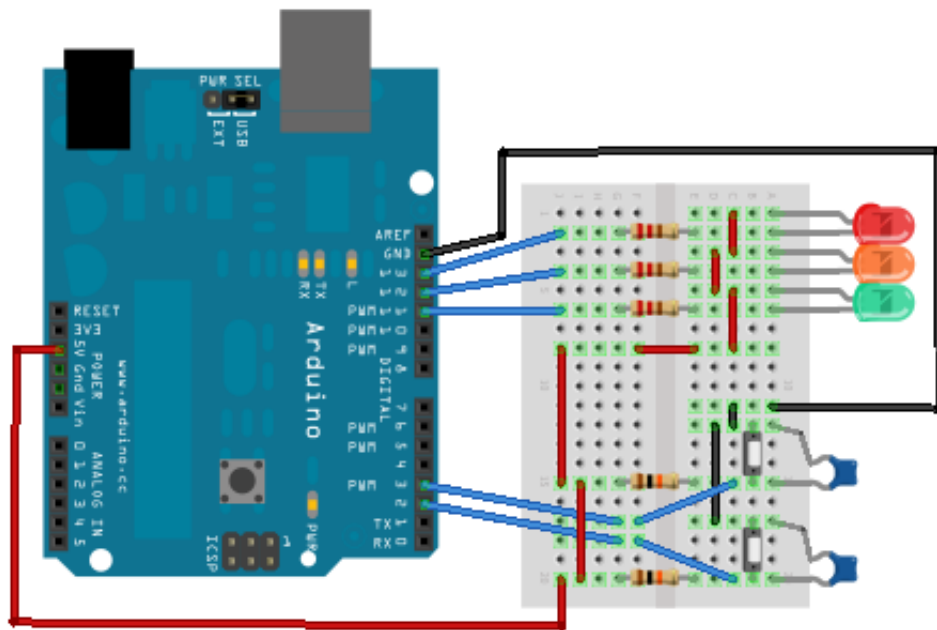
On corrige ?

J'espère que vous avez réussi à avoir un bout de solution ou une solution complète et que vous vous êtes amusé. Si vous êtes énervé sans avoir trouvé de solutions mais que vous avez cherché, ce n'est pas grave, regardez la correction et essayez de comprendre où et pourquoi vous avez fait une erreur. 😊

Le schéma électronique

Commençons par le schéma électronique, voici le mien, entre vous et moi, seules les entrées/sorties ne sont probablement pas les mêmes. En effet, il est difficile de faire autrement que comme ceci :





Quelles raisons nous ont poussés à faire ces branchements ? Eh bien :

- On utilise la voie série, donc il ne faut pas brancher de boutons ou de LED sur les broches 0 ou 1 (broche de transmission/réception)
- On utilisera les LED à l'état bas, pour éviter que la carte Arduino délivre du courant
- Les rebonds des boutons sont filtrés par des condensateurs (au passage, les boutons sont actifs à l'état bas)

Les variables globales et la fonction setup()

Poursuivons notre explication avec les variables que nous allons utiliser dans le programme et les paramètres à déclarer dans la fonction setup().

Les variables globales

```

1 #define VERT 0
2 #define ORANGE 1
3 #define ROUGE 2
4
5 int etat = 0; //stock l'état de la situation (vert = 0, orange = 1, rouge = 2)
6 char mot[20]; //le mot lu sur la voie série
7
8 //numéro des broches utilisées
9 const int btn_SOS = 2;
10 const int btn_OK = 3;
11 const int leds[3] = {11,12,13}; //tableau de 3 éléments contenant les numéros de bro

```

Afin d'appliquer le cours, on se servira ici d'un tableau pour contenir les numéros des broches des LED. Cela nous évite de mettre trois fois "int leds_xxx" (vert, orange ou rouge). Bien entendu, dans notre cas, l'intérêt est faible, mais ça suffira pour l'exercice.

Et c'est quoi ça "#define" ?

Le “#define” est ce que l’on appelle une **directive de préprocesseur**. Lorsque le logiciel Arduino va compiler votre programme, il va remplacer le terme défini par la valeur qui le suit. Par exemple, chaque fois que le compilateur verra le terme VERT (en majuscule), il mettra la valeur 0 à la place. Tout simplement ! C’est exactement la même chose que d’écrire : `const int btn_SOS = 2;`

La fonction `setup()`

Rien de particulier dans la fonction `setup()` par rapport à ce que vous avez vu précédemment, on initialise les variables

```
1 void setup()
2 {
3     Serial.begin(9600); //On démarre la voie série avec une vitesse de 9600 bits/sec
4
5     //réglage des entrées/sorties
6     //les entrées (2 boutons)
7     pinMode(btn_SOS, INPUT);
8     pinMode(btn_OK, INPUT);
9
10    //les sorties (3 LED) éteintes
11    for(int i=0; i<3; i++)
12    {
13        pinMode(leds[i], OUTPUT);
14        digitalWrite(leds[i], HIGH);
15    }
16 }
```

Dans le code précédent, l’astuce mise en œuvre est celle d’utiliser une boucle for pour initialiser les broches en tant que sorties et les mettre à l’état haut en même temps ! Sans cette astuce, le code d’initialisation (lignes 11 à 15) aurait été comme ceci :

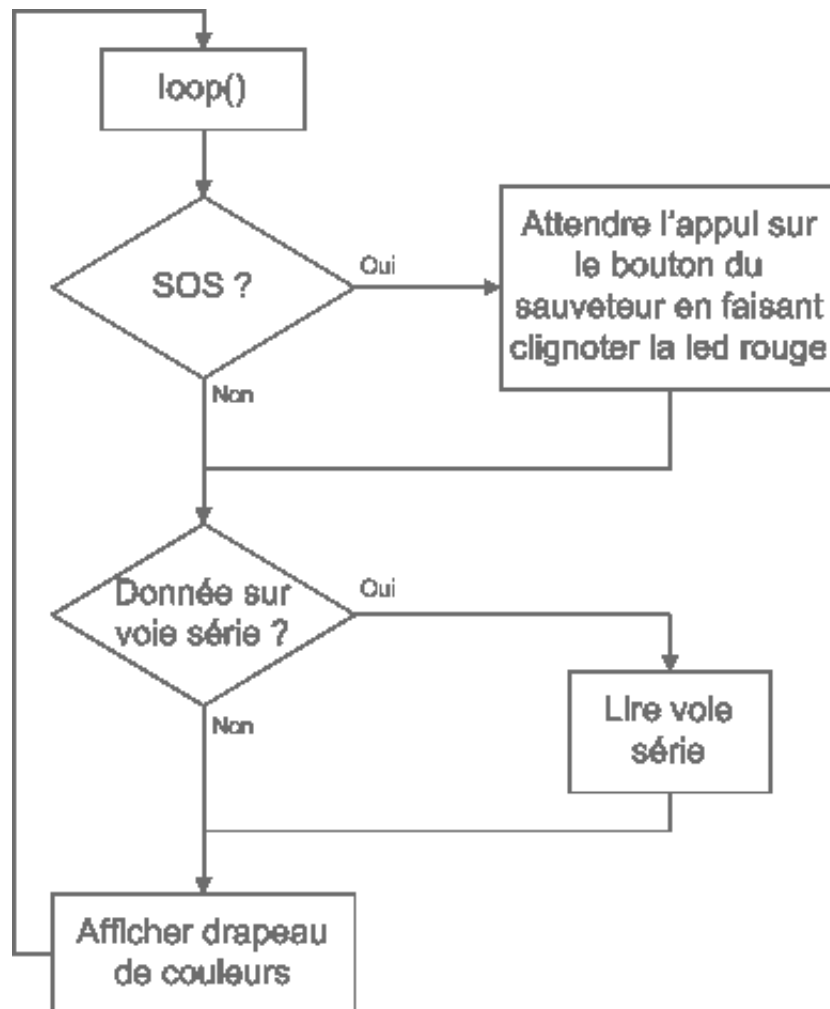
```
1 //on définit les broches, où les LED sont connectées, en sortie
2 pinMode(led_vert, OUTPUT);
3 pinMode(led_rouge, OUTPUT);
4 pinMode(led_orange, OUTPUT);
5
6 //On éteint les LED
7 digitalWrite(led_vert, HIGH);
8 digitalWrite(led_orange, HIGH);
9 digitalWrite(led_rouge, HIGH);
```

Si vous n’utilisez pas cette astuce dans notre cas, ce n’est pas dramatique. En fait, cela est utilisé lorsque vous avez 20 ou même 100 LED et broches à initialiser ! C’est moins fatigant comme ça... Qui a dit programmeur ? o_O

La fonction principale et les autres

Algorithme

Prenez l’habitude de toujours rédiger un brouillon de type algorithme ou quelque chose qui y ressemble avant de commencer à coder, cela vous permettra de mieux vous repérer dans l’endroit où vous en êtes sur l’avancement de votre programme. Voilà l’organigramme que j’ai fait lorsque j’ai commencé ce TP :



Et voilà en quelques mots la lecture de cet organigramme:

- On démarre la fonction `loop`
- Si on a un appui sur le bouton SOS :
 - On commence par faire clignoter la led rouge pour signaler l'alarme
 - Et on clignote tant que le sauveteur n'a pas appuyé sur le second bouton
- Sinon (ou si l'évènement est fini) on vérifie la présence d'un mot sur la voie série
 - S'il y a quelque chose à lire on va le récupérer
 - Sinon on continue dans le programme
- Enfin, on met à jour les drapeaux
- Puis on repart au début et refaisons le même traitement

Fort de cet outil, nous allons pouvoir coder proprement notre fonction `loop()` puis tout un tas de fonctions utiles tout autour.

Fonction `loop()`

Voici dès maintenant la fonction `loop()`, qui va exécuter l'algorithme présenté ci-dessus. Vous voyez qu'il est assez "léger" car je fais appel à de nombreuses fonctions que j'ai créées. Nous verrons ensuite le rôle de ces différentes fonctions. Cependant, j'ai fait en sorte qu'elles aient toutes un nom explicite pour que le programme soit facilement compréhensible sans même connaître le code qu'elles contiennent.

```

1 void loop()
2 {
3     //on regarde si le bouton SOS est appuyé
4     if(digitalRead(btn_SOS) == LOW)
5     {
6         //si oui, on émet l'alerte en appelant la fonction prévue à cet effet
7         alerte();
8     }
9
10    //puis on continue en vérifiant la présence de caractère sur la voie série
11    //s'il y a des données disponibles sur la voie série (Serial.available() renvoi
12    if(Serial.available())
13    {
14        //alors on va lire le contenu de la réception
15        lireVoieSerie();
16        //on entre dans une variable la valeur retournée par la fonction comparerMot
17        etat = comparerMot(mot);
18    }
19    //Puis on met à jour l'état des LED
20    allumerDrapeau(etat);
21 }

```

Lecture des données sur la voie série

Afin de garder la fonction loop “légère”, nous avons rajouté quelques fonctions annexes. La première sera celle de lecture de la voie série. Son job consiste à aller lire les informations contenues dans le buffer de réception du micro-contrôleur. On va lire les caractères en les stockant dans le tableau global “mot[]” déclaré plus tôt. La lecture s’arrête sous deux conditions :

- Soit on a trop de caractère et donc on risque d’inscrire des caractères dans des variables n’existant pas (ici tableau limité à 20 caractères)
- Soit on a rencontré le caractère symbolisant la fin de ligne. Ce caractère est ‘\n’.

Voici maintenant le code de cette fonction :

```

1 //lit un mot sur la voie série (lit jusqu'à rencontrer le caractère '\n')
2 void lireVoieSerie(void)
3 {
4     int i = 0; //variable locale pour l'incréméntation des données du tableau
5
6     //on lit les caractères tant qu'il y en a
7     //OU si jamais le nombre de caractères lus atteint 19 (limite du tableau stockan
8     while(Serial.available() > 0 && i <= 19)
9     {
10        //on enregistre le caractère lu
11        mot[i] = Serial.read();
12        //laisse un peu de temps entre chaque accès a la mémoire
13        delay(10);
14        //on passe à l'indice suivant
15        i++;
16    }
17    //on supprime le caractère '\n' et on le remplace par celui de fin de chaîne '\0
18    mot[i] = '\0';
19 }

```

Allumer les drapeaux

Voilà un titre à en rendre fou plus d'un ! Vous pouvez ranger vos briquets, on en aura pas besoin. 🚒 Une deuxième fonction est celle permettant d'allumer et d'éteindre les LED. Elle est assez simple et prend un paramètre : le numéro de la LED à allumer. Dans notre cas : 0, 1 ou 2 correspondant respectivement à vert, orange, rouge. En passant le paramètre -1, on éteint toutes les LED.

```
1  /*
2  Rappel du fonctionnement du code qui précède celui-ci :
3  > lit un mot sur la voie série (lit jusqu'à rencontrer le caractère '\n')
4  Fonction allumerDrapeau() :
5  > Allume un des trois drapeaux
6  > paramètre : le numéro du drapeau à allumer (note : si le paramètre est -1, on éteint
7  */
8
9  void allumerDrapeau(int numLed)
10 {
11     //On commence par éteindre les trois LED
12     for(int j=0; j<3; j++)
13     {
14         digitalWrite(leds[j], HIGH);
15     }
16     //puis on allume une seule LED si besoin
17     if(numLed != -1)
18     {
19         digitalWrite(leds[numLed], LOW);
20     }
21
22     /* Note : vous pourrez améliorer cette fonction en
23     vérifiant par exemple que le paramètre ne
24     dépasse pas le nombre présent de LED
25     */
26 }
```

Vous pouvez voir ici un autre intérêt du tableau utilisé dans la fonction setup() pour initialiser les LED. Une seule ligne permet de faire l'allumage de la LED concernée !

Faire clignoter la LED rouge

Lorsque quelqu'un appui sur le bouton d'alerte, il faut immédiatement avertir les sauveteurs sur la zPlage. Dans le programme principal, on va détecter l'appui sur le bouton SOS. Ensuite, on passera dans la fonction alerte() codée ci-dessous. Cette fonction est assez simple. Elle va tout d'abord relever le temps à laquelle elle est au moment même (nombre de millisecondes écoulées depuis le démarrage). Ensuite, on va éteindre toutes les LED. Enfin, et c'est là le plus important, on va attendre du sauveteur un appui sur le bouton. TANT QUE cet appui n'est pas fait, on change l'état de la LED rouge toute les 250 millisecondes (choix arbitraire modifiable selon votre humeur). Une fois que l'appui du Sauveteur a été réalisé, on va repartir dans la boucle principale et continuer l'exécution du programme.

```
1  //Éteint les LED et fais clignoter la LED rouge en attendant l'appui du bouton "sauv
2
3  void alerte(void)
4  {
5      long temps = millis();
6      boolean clignotant = false;
7      allumerDrapeau(-1); //on éteint toutes les LED
```

```

8
9 //tant que le bouton de sauveteur n'est pas appuyé on fait clignoté la LED rouge
10 while(digitalRead(btn_OK) != LOW)
11 {
12     //S'il s'est écoulé 250 ms ou plus depuis la dernière vérification
13     if(millis() - temps > 250)
14     {
15         //on change l'état de la LED rouge
16         clignotant = !clignotant; //si clignotant était FALSE, il devient TRUE e
17         //la LED est allumée au gré de la variable clignotant
18         digitalWrite(leds[ROUGE], clignotant);
19         //on se rappelle de la date de dernier passage
20         temps = millis();
21     }
22 }
23 }

```

Comparer les mots

Et voici maintenant le plus dur pour la fin, enfin j'exagère un peu. En effet, il ne vous reste plus qu'à comparer le mot reçu sur la voie série avec la banque de données de mots possible. Nous allons donc effectuer cette vérification dans la fonction `comparerMot()`. Cette fonction recevra en paramètre la chaîne de caractères représentant le mot qui doit être vérifié et comparé. Elle renverra ensuite "l'état" (vert (0), orange (1) ou rouge (2)) qui en résulte. Si aucun mot n'a été reconnu, on renvoie "ORANGE" car incertitude.

```

1 int comparerMot(char mot[])
2 {
3     //on compare les mots "VERT" (surveillant, calme)
4     if(strcmp(mot, "surveillant") == 0)
5     {
6         return VERT;
7     }
8     if(strcmp(mot, "calme") == 0)
9     {
10        return VERT;
11    }
12    //on compare les mots "ORANGE" (vague)
13    if(strcmp(mot, "vague") == 0)
14    {
15        return ORANGE;
16    }
17    //on compare les mots "ROUGE" (meduse, tempete, requin)
18    if(strcmp(mot, "meduse") == 0)
19    {
20        return ROUGE;
21    }
22    if(strcmp(mot, "tempete") == 0)
23    {
24        return ROUGE;
25    }
26    if(strcmp(mot, "requin") == 0)
27    {
28        return ROUGE;
29    }
30
31    //si on a rien reconnu on renvoi ORANGE
32    return ORANGE;

```

Code complet

Comme vous avez été sage jusqu'à présent, j'ai rassemblé pour vous le code complet de ce TP. Bien entendu, il va de pair avec le bon câblage des LED, placées sur les bonnes broches, ainsi que les boutons et le reste... Je vous fais cependant confiance pour changer les valeurs des variables si les broches utilisées sont différentes.

```

1  #define VERT 0
2  #define ORANGE 1
3  #define ROUGE 2
4
5  int etat = 0; //stock l'état de la situation (vert = 0, orange = 1, rouge = 2)
6  char mot[20]; //le mot lu sur la voie série
7
8  //numéro des broches utilisées
9  const int btn_SOS = 2;
10 const int btn_OK = 3;
11
12 //tableau de 3 éléments contenant les numéros de broches des LED
13 const int leds[3] = {11,12,13};
14
15 void setup()
16 {
17     //On démarre la voie série avec une vitesse de 9600 bits/seconde
18     Serial.begin(9600);
19
20     //réglage des entrées/sorties
21     //les entrées (2 boutons)
22     pinMode(btn_SOS, INPUT);
23     pinMode(btn_OK, INPUT);
24
25     //les sorties (3 LED) éteintes
26     for(int i=0; i<3; i++)
27     {
28         pinMode(leds[i], OUTPUT);
29         digitalWrite(leds[i], HIGH);
30     }
31 }
32
33
34 void loop()
35 {
36     //on regarde si le bouton SOS est appuyé
37     if(digitalRead(btn_SOS) == LOW)
38     {
39         //si oui, on émet l'alerte en appelant la fonction prévue à cet effet
40         alerte();
41     }
42
43     //puis on continue en vérifiant la présence de caractère sur la voie série
44     //s'il y a des données disponibles sur la voie série (Serial.available() renvoi
45     if(Serial.available())
46     {
47         //alors on va lire le contenu de la réception
48         lireVoieSerie();
49         //on entre dans une variable la valeur retournée par la fonction comparerMo
50         etat = comparerMot(mot);

```

```

51     }
52     //Puis on met à jour l'état des LED
53     allumerDrapeau(etat);
54 }
55
56
57 //lit un mot sur la voie série (lit jusqu'à rencontrer le caractère '\n')
58 void lireVoieSerie(void)
59 {
60     int i = 0; //variable locale pour l'incrémentation des données du tableau
61
62     //on lit les caractères tant qu'il y en a
63     //OU si jamais le nombre de caractères lus atteint 19 (limite du tableau stocka
64     while(Serial.available() > 0 && i <= 19)
65     {
66         mot[i] = Serial.read(); //on enregistre le caractère lu
67         delay(10); //laisse un peu de temps entre chaque accès a la mémoire
68         i++; //on passe à l'indice suivant
69     }
70     mot[i] = '\0'; //on supprime le caractère '\n' et on le remplace par celui de f
71 }
72
73
74 /*
75 Rappel du fonctionnement du code qui précède celui-ci :
76 > lit un mot sur la voie série (lit jusqu'à rencontrer le caractère '\n')
77 Fonction allumerDrapeau() :
78 > Allume un des trois drapeaux
79 > paramètre : le numéro du drapeau à allumer (note : si le paramètre est -1, on éte
80 */
81
82 void allumerDrapeau(int numLed)
83 {
84     //On commence par éteindre les trois LED
85     for(int j=0; j<3; j++)
86     {
87         digitalWrite(leds[j], HIGH);
88     }
89     //puis on allume une seule LED si besoin
90     if(numLed != -1)
91     {
92         digitalWrite(leds[numLed], LOW);
93     }
94
95     /* Note : vous pourrez améliorer cette fonction en
96 vérifiant par exemple que le paramètre ne
97 dépasse pas le nombre présent de LED
98 */
99 }
100
101
102 //Éteint les LED et fais clignoter la LED rouge en attendant l'appui du bouton "sau
103
104 void alerte(void)
105 {
106     long temps = millis();
107     boolean clignotant = false;
108     allumerDrapeau(-1); //on éteint toutes les LED
109
110     //tant que le bouton de sauveteur n'est pas appuyé on fait clignoté la LED rouge
111     while(digitalRead(btn_OK) != LOW)

```



```

112     {
113         //S'il s'est écoulé 250 ms ou plus depuis la dernière vérification
114         if(millis() - temps > 250)
115         {
116             //on change l'état de la LED rouge
117             clignotant = !clignotant; //si clignotant était FALSE, il devient TRUE
118             //la LED est allumée au gré de la variable clignotant
119             digitalWrite(leds[ROUGE], clignotant);
120             //on se rappelle de la date de dernier passage
121             temps = millis();
122         }
123     }
124 }
125
126
127 int comparerMot(char mot[])
128 {
129     //on compare les mots "VERT" (surveillant, calme)
130     if(strcmp(mot, "surveillant") == 0)
131     {
132         return VERT;
133     }
134     if(strcmp(mot, "calme") == 0)
135     {
136         return VERT;
137     }
138     //on compare les mots "ORANGE" (vague)
139     if(strcmp(mot, "vague") == 0)
140     {
141         return ORANGE;
142     }
143     //on compare les mots "ROUGE" (meduse, tempete, requin)
144     if(strcmp(mot, "meduse") == 0)
145     {
146         return ROUGE;
147     }
148     if(strcmp(mot, "tempete") == 0)
149     {
150         return ROUGE;
151     }
152     if(strcmp(mot, "requin") == 0)
153     {
154         return ROUGE;
155     }
156
157     //si on a rien reconnu on renvoi ORANGE
158     return ORANGE;
159 }

```

Je rappelle que si vous n'avez pas réussi à faire fonctionner complètement votre programme, aidez-vous de celui-ci pour comprendre le pourquoi du comment qui empêche votre programme de fonctionner correctement ! A bons entendeurs. 😊

Améliorations

Je peux vous proposer quelques idées d'améliorations que je n'ai pas mises en œuvre, mais qui me sont passées par la tête au moment où j'écrivais ces lignes :

Améliorations logicielles

Avec la nouvelle version d'Arduino, la version 1.0,; il existe une fonction `SerialEvent()` qui est exécutée dès qu'il y a un événement sur la voie série du micro-contrôleur. Je vous laisse le soin de chercher à comprendre comment elle fonctionne et s'utilise, sur [cette page](#).

Améliorations matérielles

- On peut par exemple automatiser le changement d'un drapeau en utilisant un système mécanique avec un ou plusieurs moteurs électriques. Ce serait dans le cas d'utilisation réelle de ce montage, c'est-à-dire sur une plage...
- Une liaison filaire entre un PC et une carte Arduino, ce n'est pas toujours la joie. Et puis bon, ce n'est pas toujours facile d'avoir un PC sous la main pour commander ce genre de montage. Alors pourquoi ne pas rendre la connexion sans-fil en utilisant par exemple des modules XBee ? Ces petits modules permettent une connexion sans-fil utilisant la voie série pour communiquer. Ainsi, d'un côté vous avez la télécommande (à base d'Arduino et d'un module XBee) de l'autre vous avez le récepteur, toujours avec un module XBee et une Arduino, puis le montage de ce TP avec l'amélioration précédente.

Sérieusement si ce montage venait à être réalité avec les améliorations que je vous ai données, prévenez-moi par MP et faites en une vidéo pour que l'on puisse l'ajouter en lien ici même ! 📺

Voilà une grosse tâche de terminée ! J'espère qu'elle vous a plu même si vous avez pu rencontrer des difficultés. Souvenez-vous, "à vaincre sans difficulté on triomphe sans gloire", donc tant mieux si vous avez passé quelques heures dessus et, surtout, j'espère que vous avez appris des choses et pris du plaisir à faire votre montage, le dompter et le faire fonctionner comme vous le souhaitiez !

[Arduino 304] [Annexe] Votre ordinateur et sa voie série dans un autre langage de programmation

Maintenant que vous savez comment utiliser la voie série avec Arduino, il peut être bon de savoir comment visualiser les données envoyées avec vos propres programmes (l'émulateur terminal Windows ou le moniteur série Arduino ne comptent pas 😊). Cette annexe a donc pour but de vous montrer comment utiliser la voie série avec quelques langages de programmation. Les langages utilisés ci-dessous ont été choisis arbitrairement en fonction de mes connaissances, car je ne connais pas tous les langages possibles et une fois vu quelques exemples, il ne devrait pas être trop dur de l'utiliser avec un autre langage. Nous allons donc travailler avec :

Afin de se concentrer sur la partie "Informatique", nous allons reprendre un programme travaillé précédemment dans le cours. Ce sera celui de l'exercice : [Attention à la casse](#). Pensez donc à le charger dans votre carte Arduino avant de faire les tests. 😊

En C++ avec Qt

Avant de commencer cette sous-partie, il est indispensable de connaître la programmation en C++ et savoir utiliser le framework Qt. Si vous ne connaissez pas tout cela, vous pouvez toujours aller vous renseigner avec le [tutoriel C++](#) !

Le C++, OK, mais pourquoi Qt ?

J'ai choisi de vous faire travailler avec Qt pour plusieurs raisons d'ordres pratiques.

- Qt est multiplateforme, donc les réfractaires à Linux (ou à Windows) pourront quand même travailler.
- Dans le même ordre d'idée, nous allons utiliser une librairie tierce pour nous occuper de la voie série. Ainsi, aucun problème pour interfacer notre matériel que l'on soit sur un système ou un autre !
- Enfin, j'aime beaucoup Qt et donc je vais vous en faire profiter 😊

En fait, sachez que chaque système d'exploitation à sa manière de communiquer avec les périphériques matériels. L'utilisation d'une librairie tierce nous permet donc de faire abstraction de tout cela. Sinon il m'aurait fallu faire un tutoriel par OS, ce qui, on l'imagine facilement, serait une perte de temps (écrire trois fois *environ* les mêmes choses) et vraiment galère à maintenir.

Installer QextSerialPort

QextSerialPort est une librairie tierce réalisée par un membre de la communauté Qt. Pour utiliser cette librairie, il faut soit la compiler, soit utiliser les sources directement dans votre projet.

1ère étape : télécharger les sources

Le début de tout cela commence donc par récupérer les sources de la librairie. Pour cela, rendez-vous sur [la page google code](#) du projet. A partir d'ici vous avez plusieurs choix. Soit vous récupérez les sources en utilisant le gestionnaire de source mercurial (Hg). Il suffit de faire un clone du dépôt avec la commande suivante :

```
1 hg clone https://code.google.com/p/qextserialport/
```

Sinon, vous pouvez [récupérer les fichiers un par un](#) (une dizaine). C'est plus contraignant mais ça marche aussi si vous n'avez jamais utilisé de gestionnaire de sources (mais c'est vraiment plus contraignant !)

Cette dernière méthode est vraiment **déconseillée**. En effet, vous vous retrouverez avec le strict minimum (fichiers sources sans exemples ou docs).

La manipulation est la même sous Windows ou Linux !

Compiler la librairie

Maintenant que nous avons tous nos fichiers, nous allons pouvoir compiler la librairie. Pour cela, nous allons laisser Qt travailler à notre place.

- Démarrez QtCreator et ouvrez le fichier .pro de QextSerialPort
- Compilez...
- C'est fini !

Normalement vous avez un nouveau dossier à côté de celui des sources qui contient des exemples, ainsi que les **librairies** QextSerialPort.

Installer la librairie : Sous Linux

Une fois que vous avez compilé votre nouvelle librairie, vous allez devoir placer les fichiers aux bons endroits pour les utiliser. Les librairies, qui sont apparues dans le dossier "build" qui vient d'être créé, vont être déplacées vers le dossier /usr/lib. Les fichiers sources qui étaient avec le fichier ".pro" pour la compilation sont à copier dans un sous-dossier "QextSerialPort" dans le répertoire de travail de votre projet courant.

A priori il y aurait un bug avec la compilation en mode release (la librairie générée ne fonctionnerait pas correctement). Je vous invite donc à compiler aussi la debug et travailler avec.

Installer la librairie : Sous Windows

Ce point est en cours de rédaction, merci de patienter avant sa mise en ligne. 😊

Infos à rajouter dans le .pro

Dans votre nouveau projet Qt pour traiter avec la voie série, vous aller rajouter les lignes suivantes à votre .pro :

```
1 INCLUDEPATH += QextSerialPort
2
3 CONFIG(debug, debug|release):LIBS += -lqextserialportd
4 else:LIBS += -lqextserialport
```

La ligne "INCLUDEPATH" représente le dossier où vous avez mis les fichiers sources de QextSerialPort. Les deux autres lignes font le lien vers les librairies copiées plus tôt (les .so ou les .dll selon votre OS).

Les trucs utiles

L'interface utilisée

Comme expliqué dans l'introduction, nous allons toujours travailler sur le même exercice et juste changer le langage étudié. Voici donc l'interface sur laquelle nous allons travailler, et quels sont les noms et les types d'objets instanciés :



Cette interface possède deux parties importantes : La **gestion de la connexion** (en haut) et **l'échange de résultat** (milieu -> émission, bas -> réception). Dans la partie supérieure, nous allons choisir le port de l'ordinateur sur lequel communiquer ainsi que la vitesse de cette communication. Ensuite, deux boîtes de texte sont présentes. L'une pour écrire du texte à émettre, et l'autre affichant le texte reçu. Voici les noms que j'utiliserai dans mon code :

Widget	Nom	Rôle
QComboBox	comboPort	Permet de choisir le port série
QComboBox	comboVitesse	Permet de choisir la vitesse de communication
QPushButton	btnconnexion	(Dé)Connecte la voie série (bouton "checkable")
QTextEdit	boxEmission	Nous écrivons ici le texte à envoyer
QTextEdit	boxReception	Ici apparaîtra le texte à recevoir

Lister les liaisons séries

Avant de créer et d'utiliser l'objet pour gérer la voie série, nous allons en voir quelques-uns pouvant être utiles. Tout d'abord, nous allons apprendre à obtenir la liste des ports série présents sur notre machine. Pour cela, un objet a été créé spécialement, il s'agit de `QextPortInfo`. Voici un exemple de code leur permettant de fonctionner ensemble :

```
1 //L'objet mentionnant les infos
2 QextSerialEnumerator enumerateur;
3 //on met ces infos dans une liste
4 QList<QextPortInfo> ports = enumerateur.getPorts();
```

Une fois que nous avons récupéré une énumération de tous les ports, nous allons pouvoir les ajouter au combobox qui est censé les afficher (comboPort). Pour cela on va parcourir la liste construite précédemment et ajouter à chaque fois une item dans le

menu déroulant :

```
1 //on parcourt la liste des ports
2 for(int i=0; i<ports.size(); i++)
3     ui->ComboPort->addItem(ports.at(i).physName);
```

Les ports sont nommés différemment sous Windows et Linux, ne soyez donc pas surpris avec mes captures d'écrans, elles viennent toutes de Linux.

Une fois que la liste des ports est faite (attention, certains ports ne sont connectés à rien), on va construire la liste des vitesses, pour se laisser le choix le jour où l'on voudra faire une application à une vitesse différente. Cette opération n'est pas très compliquée puisqu'elle consiste simplement à ajouter des items dans la liste déroulante "comboVitesse".

```
1 ui->comboVitesse->addItem("300");
2 ui->comboVitesse->addItem("1200");
3 ui->comboVitesse->addItem("2400");
4 ui->comboVitesse->addItem("4800");
5 ui->comboVitesse->addItem("9600");
6 ui->comboVitesse->addItem("14400");
7 ui->comboVitesse->addItem("19200");
8 ui->comboVitesse->addItem("38400");
9 ui->comboVitesse->addItem("57600");
10 ui->comboVitesse->addItem("115200");
```

Votre interface est maintenant prête. En la démarrant maintenant vous devriez être en mesure de voir s'afficher les noms des ports séries existant sur l'ordinateur ainsi que les vitesses. Un clic sur le bouton ne fera évidemment rien puisque son comportement n'est pas encore implémenté.

Gérer une connexion

Lorsque tous les détails concernant l'interface sont terminés, nous pouvons passer au cœur de l'application : la communication série. La première étape pour pouvoir faire une communication est de se connecter (tout comme vous vous connectez sur une borne WiFi avant de communiquer et d'échanger des données avec cette dernière). C'est le rôle de notre bouton de connexion. A partir du système de slot automatique, nous allons créer une fonction qui va recevoir le clic de l'utilisateur. Cette fonctioninstanciera un objet QextSerialPort pour créer la communication, réglera cet objet et enfin ouvrira le canal. Dans le cas où le bouton était déjà coché (puisque'il sera "checkable" rappelons-le) nous ferons la déconnexion, puis la destruction de l'objet QextSerialPort créé auparavant. Pour commencer nous allons donc déclarer les objets et méthodes utiles dans le .h de la classe avec laquelle nous travaillons :

```
1 private:
2     //l'objet représentant le port
3     QextSerialPort * port;
4
5     //une fonction utile que j'expliquerais après
6     BaudRateType getBaudRateFromString(QString baudRate);
7
8     private slots:
9     //le slot automatique du bouton de connexion
10    void on_btnconnexion_clicked();
```

Ensuite, il nous faudra instancier le slot du bouton afin de traduire un comportement. Pour rappel, il devra :

- Créer l'objet "port" de type `QextSerialPort`
- Le régler avec les bons paramètres
- Ouvrir la voie série

Dans le cas où la voie série est déjà ouverte (le bouton est déjà appuyé) on devra la fermer et détruire l'objet. Voici le code commenté permettant l'ouverture de la voie série (quelques précisions viennent ensuite) :

```
1 //Slot pour le click sur le bouton de connexion
2 void Fenetre::on_btnconnexion_clicked() {
3     //deux cas de figures à gérer, soit on coche (connecte), soit on décoche (déconn
4
5     //on coche -> connexion
6     if(ui->btnconnexion->isChecked()) {
7         //on essaie de faire la connexion avec la carte Arduino
8         //on commence par créer l'objet port série
9         port = new QextSerialPort();
10        //on règle le port utilisé (sélectionné dans la liste déroulante)
11        port->setPortName(ui->ComboPort->currentText());
12        //on règle la vitesse utilisée
13        port->setBaudRate(getBaudRateFromString(ui->comboVitesse->currentText()));
14        //quelques réglages pour que tout marche bien
15        port->setParity(PAR_NONE); //parité
16        port->setStopBits(STOP_1); //nombre de bits de stop
17        port->setDataBits(DATA_8); //nombre de bits de données
18        port->setFlowControl(FLOW_OFF); //pas de contrôle de flux
19        //on démarre !
20        port->open(QextSerialPort::ReadWrite);
21        //change le message du bouton
22        ui->btnconnexion->setText("Disconnecter");
23
24        //on fait la connexion pour pouvoir obtenir les évènements
25        connect(port, SIGNAL(readyRead()), this, SLOT(readData()));
26        connect(ui->boxEmission, SIGNAL(textChanged()), this, SLOT(sendData()));
27    }
28    else {
29        //on se déconnecte de la carte Arduino
30        port->close();
31        //puis on détruit l'objet port série devenu inutile
32        delete port;
33        ui->btnconnexion->setText("Connecter");
34    }
35 }
```

Ce code n'est pas très compliqué à comprendre. Cependant quelques points méritent votre attention. Pour commencer, pour régler la vitesse du port série on fait appel à la fonction "setBaudRate". Cette fonction prend un paramètre de type `BaudRateType` qui fait partie d'une énumération de `QextSerialPort`. Afin de faire le lien entre le comboBox qui possède des chaînes et le type particulier attendu, on crée et utilise la fonction "getBaudRateFromString". A partir d'un simple `BaudRateType`.

```
1 BaudRateType Fenetre::getBaudRateFromString(QString baudRate) {
2     int vitesse = baudRate.toInt();
3     switch(vitesse) {
4         case(300):return BAUD300;
```



```

5         case(1200):return BAUD1200;
6         case(2400):return BAUD2400;
7         case(4800):return BAUD4800;
8         case(9600):return BAUD9600;
9         case(14400):return BAUD14400;
10        case(19200):return BAUD19200;
11        case(38400):return BAUD38400;
12        case(57600):return BAUD57600;
13        case(115200):return BAUD115200;
14        default:return BAUD9600;
15    }
16 }

```

Un autre point important à regarder est l'utilisation de la fonction `open()` de l'objet `QextSerialPort`. En effet, il existe plusieurs façons d'ouvrir un port série :

- En lecture seule -> `QextSerialPort::ReadOnly`
- En écriture seule -> `QextSerialPort::WriteOnly`
- En lecture/écriture -> `QextSerialPort::ReadWrite`

Ensuite, on connecte simplement les signaux émis par la voie série et par la boîte de texte servant à l'émission (que l'on verra juste après). Enfin, lorsque l'utilisateur re-clic sur le bouton, on passe dans le `NULL`).

Ce code présente le principe et n'est pas parfait ! Il faudrait par exemple s'assurer que le port est bien ouvert avant d'envoyer des données (faire un test `if(port->isOpen())` par exemple).

Émettre et recevoir des données

Maintenant que la connexion est établie, nous allons pouvoir envoyer et recevoir des données. Ce sera le rôle de deux slots qui ont été brièvement évoqués dans la fonction `connect()` du code de connexion précédent.

Émettre des données

L'émission des données se fera dans le slot "sendData". Ce slot sera appelé à chaque fois qu'il y aura une modification du contenu de la boîte de texte "boxEmission". Pour l'application concernée (l'envoi d'un seul caractère), il nous suffit de chercher le dernier caractère tapé. On récupère donc le dernier caractère du texte contenu dans la boîte avant de l'envoyer sur la voie série. L'envoi de texte se fait à partir de la fonction `toAscii()` et on peut donc les utiliser directement. Voici le code qui illustre toutes ces explications (ne pas oublier de mettre les déclarations des slots dans le .h) :

```

1 void Fenetre::sendData() {
2     //On récupère le dernier caractère tapé
3     QString caractere = ui->boxEmission->toPlainText().right(1);
4     //si le port est instancié (donc ouvert a priori)
5     if(port != NULL)
6         port->write(caractere.toAscii());
7 }

```

Recevoir des données

Le programme étudié est censé nous répondre en renvoyant le caractère émis mais dans une casse opposée (majuscule contre minuscule et vice versa). En temps normal, deux politiques différentes s'appliquent pour savoir si des données sont arrivées. La première est d'aller voir de manière régulière (ou pas) si des caractères sont présents dans le tampon de réception de la voie série. Cette méthode dite de *Polling* n'est pas très fréquemment utilisée. La seconde est de déclencher un évènement lorsque des données arrivent sur la voie série. C'est la forme qui est utilisée par défaut par l'objet `readyRead()` est émis par l'objet et peut donc être connecté à un slot. Pour changer le mode de fonctionnement, il faut utiliser la méthode `QextSerialPort::EventDriven` pour la seconde (par défaut). Comme la connexion entre le signal et le slot est créée dans la fonction de connexion, il ne nous reste qu'à écrire le comportement du slot de réception lorsqu'une donnée arrive. Le travail est simple et se résume en deux étapes :

- Lire le caractère reçu grâce à la fonction `QextSerialPort`
- Le copier dans la boîte de texte "réception"

```
1 void Fenetre::readData() {  
2     QByteArray array = port->readAll();  
3     ui->boxReception->insertPlainText(array);  
4 }
```

Et voilà, vous êtes maintenant capable de travailler avec la voie série dans vos programmes Qt en C++. Au risque de me répéter, je suis conscient qu'il y a des lacunes en terme de "sécurité" et d'efficacité. Ce code a pour but de vous montrer les bases de la classe pour que vous puissiez continuer ensuite votre apprentissage. En effet, la programmation C++/Qt n'est pas le sujet de ce tutoriel. :ninja: Nous vous serons donc reconnaissants de

ne pas nous harceler de commentaires relatifs au tuto pour nous dire "bwaaaa c'est mal codééééé". Merci ! 😊

En C# (.Net)

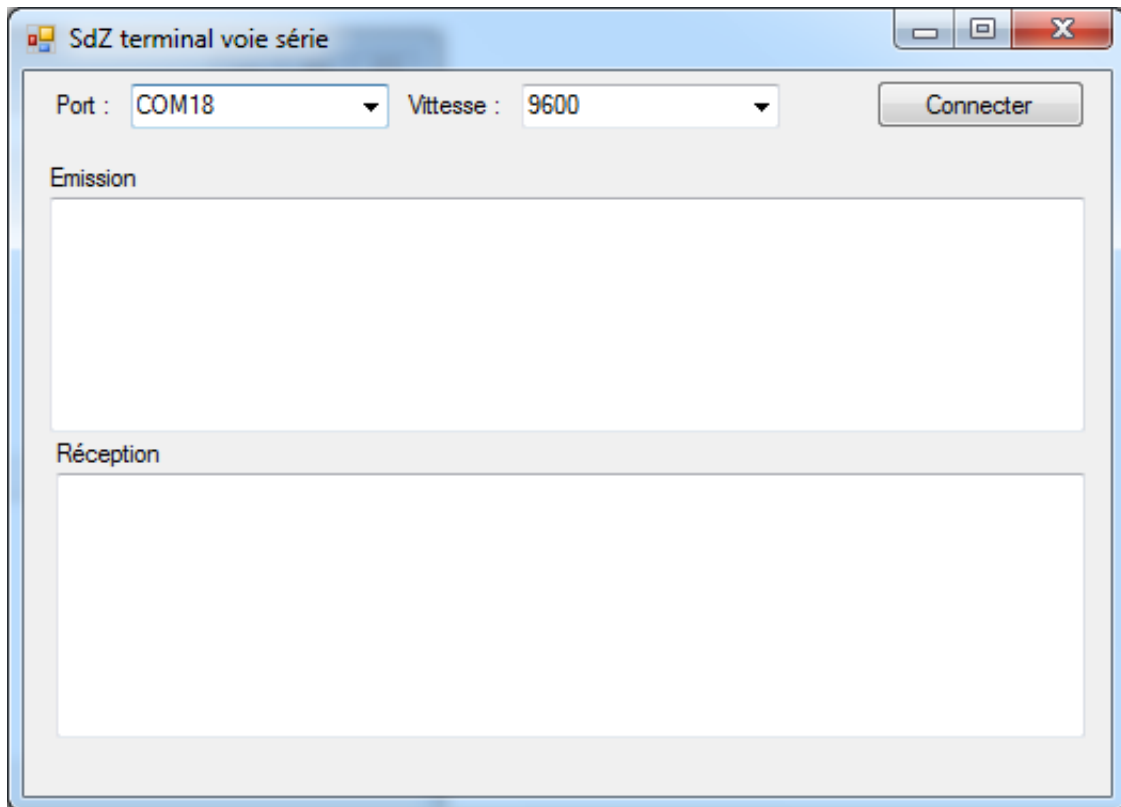
Dans cette partie (comme dans les précédentes) je pars du principe que vous connaissez le langage et avez déjà dessiné des interfaces et créé des actions sur des boutons par exemple. Cette sous-partie n'est pas là pour vous apprendre le C# !

Là encore je vais reprendre la même structure que les précédentes sous-parties.

Les trucs utiles

L'interface et les imports

Voici tout de suite l'interface utilisée ! Je vous donnerai juste après le nom que j'utilise pour chacun des composants (et tant qu'à faire je vous donnerai aussi leurs types).



Comme cette interface est la même pour tout ce chapitre, nous retrouvons comme d'habitude le bandeau pour gérer la connexion ainsi que les deux boîtes de texte pour l'émission et la réception des données. Voici les types d'objets et leurs noms pour le bandeau de connexion :

Composant	Nom	Rôle
System.Windows.Forms.ComboBox	comboPort	Permet de choisir le port série
System.Windows.Forms.ComboBox	comboVitesse	Permet de choisir la vitesse de communication
System.Windows.Forms.Button	btnConnexion	(Dé)Connecte la voie série (bouton "checkable")
System.Windows.Forms.TextBox	boxEmission	Nous écrirons ici le texte à envoyer
System.Windows.Forms.TextBox	boxReception	Ici apparaîtra le texte à recevoir

Avant de commencer les choses marrantes, nous allons d'abord devoir ajouter une librairie : celle des liaisons séries. Elle se nomme simplement `using System.IO.Ports;`. Nous allons en profiter pour rajouter une variable membre de la classe de type `SerialPort` que j'appellerai "port". Cette variable représentera, vous l'avez deviné, notre port série !

```
1 SerialPort port
```

Maintenant que tous les outils sont prêts, nous pouvons commencer !

Lister les liaisons séries

La première étape sera de lister l'ensemble des liaisons séries sur l'ordinateur. Pour cela nous allons nous servir d'une fonction statique de la classe `String`. Chaque case

du tableau sera une chaîne de caractère comportant le nom d'une voie série. Une fois que nous avons ce tableau, nous allons l'ajouter sur l'interface, dans la liste déroulante prévue à cet effet pour pouvoir laisser le choix à l'utilisateur au démarrage de l'application. Dans le même élan, on va peupler la liste déroulante des vitesses avec quelques-unes des vitesses les plus courantes. Voici le code de cet ensemble. Personnellement je l'ai ajouté dans la méthode `InitializeComponent()` qui charge les composants.

```
1 private void Form1_Load(object sender, EventArgs e)
2 {
3     //on commence par lister les voies séries présentes
4     String[] ports = SerialPort.GetPortNames(); //fonction statique
5     //on ajoute les ports au combo box
6     foreach (String s in ports)
7         comboPort.Items.Add(s);
8
9     //on ajoute les vitesses au combo des vitesses
10    comboVitesse.Items.Add("300");
11    comboVitesse.Items.Add("1200");
12    comboVitesse.Items.Add("2400");
13    comboVitesse.Items.Add("4800");
14    comboVitesse.Items.Add("9600");
15    comboVitesse.Items.Add("14400");
16    comboVitesse.Items.Add("19200");
17    comboVitesse.Items.Add("38400");
18    comboVitesse.Items.Add("57600");
19    comboVitesse.Items.Add("115200");
20 }
```

Si vous lancez votre programme maintenant avec la carte Arduino connectée, vous devriez avoir le choix des vitesses mais aussi d'au moins un port série. Si ce n'est pas le cas, il faut trouver pourquoi avant de passer à la suite (Vérifiez que la carte est bien connectée par exemple).

Gérer une connexion

Une fois que la carte est reconnue et que l'on voit bien son port dans la liste déroulante, nous allons pouvoir ouvrir le port pour établir le canal de communication entre Arduino et l'ordinateur. Comme vous vous en doutez sûrement, la fonction que nous allons écrire est celle du clic sur le bouton. Lorsque nous cliquons sur le bouton de connexion, deux actions peuvent être effectuées selon l'état précédent. Soit nous nous connectons, soit nous nous déconnectons. Les deux cas seront gérés en regardant le texte contenu dans le bouton ("Connecter" ou "Déconnecter"). Dans le cas de la déconnexion, il suffit de fermer le port à l'aide de la méthode `close()`. Dans le cas de la connexion, plusieurs choses sont à faire. Dans l'ordre, nous allons commencer par instancier un nouvel objet de type `BaudRate` et ainsi de suite. Voici le code commenté pour faire tout cela. Il y a cependant un dernier point évoqué rapidement juste après et sur lequel nous reviendrons plus tard.

```
1 private void btnConnexion_Click(object sender, EventArgs e)
2 {
3     //on gère la connexion/déconnexion
4     if (btnConnexion.Text == "Connecter") //alors on connecte
5     {
6         //crée un nouvel objet voie série
7         port = new SerialPort();
```

```

8 //règle la voie série
9 //parse en int le combo des vitesses
10 port.BaudRate = int.Parse(comboVitesse.SelectedItem.ToString());
11 port.DataBits = 8;
12 port.StopBits = StopBits.One;
13 port.Parity = Parity.None;
14 //récupère le nom sélectionné
15 port.PortName = comboPort.SelectedItem.ToString();
16
17 //ajoute un gestionnaire de réception pour la réception de donnée sur la voie
18 port.DataReceived += new SerialDataReceivedEventHandler(DataReceivedHandler)
19
20 port.Open(); //ouvre la voie série
21
22 btnConnexion.Text = "Deconnecter";
23 }
24 else //sinon on déconnecte
25 {
26     port.Close(); //ferme la voie série
27     btnConnexion.Text = "Connecter";
28 }
29 }

```

Le point qui peut paraître étrange est la ligne 16, avec la propriété `Handler()` qui devra être appelée lorsque des données arriveront. Je vais vous demander d'être patient, nous en reparlerons plus tard lorsque nous verrons la réception de données. A ce stade du développement, lorsque vous lancez votre application vous devriez pouvoir sélectionner une voie série, une vitesse, et cliquer sur "Connecter" et "Déconnecter" sans aucun bug.

Émettre et recevoir des données

La voie série est prête à être utilisée ! La connexion est bonne, il ne nous reste plus qu'à envoyer les données et espérer avoir quelque chose en retour. 😊

Envoyer des données

Pour envoyer des données, une fonction toute prête existe pour les objets `char` qui serait envoyé un par un. Dans notre cas d'utilisation, c'est ce deuxième cas qui nous intéresse. Nous allons donc implémenter la méthode `TextChanged` du composant "boxEmission" afin de détecter chaque caractère entré par l'utilisateur. Ainsi, nous enverrons chaque nouveau caractère sur la voie série, un par un. Le code suivant, commenté, vous montre la voie à suivre.

```

1 //lors d'un envoi de caractère
2 private void boxEmission_TextChanged(object sender, EventArgs e)
3 {
4     //met le dernier caractère dans un tableau avec une seule case le contenant
5     char[] car = new char[] {boxEmission.Text[boxEmission.TextLength-1]};
6     //on s'assure que le port est existant et ouvert
7     if(port!=null && port.IsOpen)
8         //envoie le tableau de caractère, depuis la position 0, et envoie 1 seul élém
9         port.Write(car,0,1);
10 }

```

Recevoir des données

La dernière étape pour pouvoir gérer de manière complète notre voie série est de pouvoir afficher les caractères reçus. Cette étape est un petit peu plus compliquée. Tout d'abord, revenons à l'explication commencée un peu plus tôt. Lorsque nous démarrons la connexion et créons l'objet `boxReception`. Dans l'idéal nous aimerions faire de la façon suivante :

```
1 //gestionnaire de la réception de caractère
2 private void DataReceivedHandler(object sender, SerialDataReceivedEventArgs e)
3 {
4     String texte = port.ReadExisting();
5     boxReception.Text += texte;
6 }
```

Cependant, les choses ne sont pas aussi simples cette fois-ci. En effet, pour des raisons de sécurité sur les processus, C# interdit que le texte d'un composant (`boxReception`) soit modifié de manière asynchrone, quand les données arrivent. Pour contourner cela, nous devons créer une méthode "déléguée" à qui on passera notre texte à afficher et qui se chargera d'afficher le texte quand l'interface sera prête. Pour créer cette déléguée, nous allons commencer par rajouter une méthode dite de *callback* pour gérer la mise à jour du texte. La ligne suivante est donc à ajouter dans la classe, comme membre :

```
1 //une déléguée pour pouvoir mettre à jour le texte de la boîte de réception de manière
2 delegate void SetTextCallback(string text);
```

Le code de la réception devient alors le suivant :

```
1 //gestionnaire de la réception de caractère
2 private void DataReceivedHandler(object sender, SerialDataReceivedEventArgs e)
3 {
4     String texte = port.ReadExisting();
5     //boxReception.Text += texte;
6     SetText(texte);
7 }
8
9 private void SetText(string text)
10 {
11     if (boxReception.InvokeRequired)
12     {
13         SetTextCallback d = new SetTextCallback(SetText);
14         boxReception.Invoke(d, new object[] { text });
15     }
16     else
17     {
18         boxReception.Text += text;
19     }
20 }
```

Je suis désolé si mes informations sont confuses. Je ne suis malheureusement pas un maître dans l'art des threads UI de C#. Cependant, un tas de documentation mieux expliquée existe sur internet si vous voulez plus de détails.

Une fois tout cela instancié, vous devriez avoir un terminal voie série tout beau fait par

vous même ! Libre à vous maintenant toutes les cartes en main pour créer des applications qui communiqueront avec votre Arduino et feront des échanges d'informations avec.

En Python

Comme ce langage à l'air d'être en vogue, je me suis un peu penché dessus pour vous fournir une approche de comment utiliser python pour se servir de la voie série. Mon niveau en python étant équivalent à "grand débutant", je vais vous proposer un code simple reprenant les fonctions utiles à savoir le tout sans interface graphique. Nul doute que les pythonistes chevronnés sauront creuser plus loin 😊

Comme pour les exemples dans les autres langages, on utilisera l'exercice "Attention à la casse" dans l'Arduino pour tester notre programme.

Pour communiquer avec la voie série, nous allons utiliser une librairie externe qui s'appelle [pySerial](#).

Installation Ubuntu

Pour installer pySerial sur votre machine Ubuntu c'est très simple, il suffit de lancer une seule commande :

```
1 sudo apt-get install python3-serial
```

Vous pouvez aussi l'installer à partir des sources à l'adresse suivante : <https://pypi.python.org/pypi/pyserial>. Ensuite, décompresser l'archive et exécuter la commande : (pour python 2)

```
1 python setup.py install
```

(pour python 3)

```
1 python3 setup.py install
```

Windows

Si vous utilisez Windows, il vous faudra un logiciel capable de décompresser les archives de types tar.gz (comme 7-zip par exemple). Ensuite vous devrez récupérer les sources à la même adresse que pour Linux : <https://pypi.python.org/pypi/pyserial> Enfin, comme pour Linux encore il vous suffira d'exécuter la commande qui va bien :

```
1 python setup.py install
```

Utiliser la librairie

Pour utiliser la librairie, il vous faudra tout d'abord l'importer. Pour cela, on utilise la commande import :


```
1 import serial
```

mais comme seule une partie du module nous intéresse vraiment (Serial) on peut restreindre :

```
1 from serial import Serial
```

(notez l'importance des majuscules/minuscules)

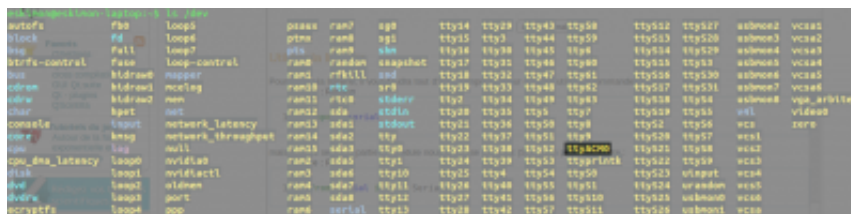
Ouvrir un port série

Maintenant que le module est bien chargé, nous allons pouvoir commencer à l'utiliser. La première chose importante à faire est de connaître le port série à utiliser. On peut obtenir une liste de ces derniers grâce à la commande :

```
1 python -m serial.tools.list_ports
```

Si comme chez moi cela ne fonctionne pas, vous pouvez utiliser d'autres méthodes.

- Sous Windows : en allant dans le gestionnaire de périphériques pour trouver le port série concerné (COMx)
- Sous Linux : en utilisant la commande `ls /dev`, vous pourrez trouver le nom du port série sous le nom "ttyACMx" par exemple



Lorsque le port USB est identifié, on peut créer un objet de type Serial. Le constructeur que l'on va utiliser prend deux paramètres, le nom du port série et la vitesse à utiliser (les autres paramètres (parité...) conviennent par défaut).

```
1 port = Serial('/dev/ttyACM0', 9600)
```

Une fois cet objet créé, la connexion peut-être ouverte avec la fonction open()

```
1 port.open()
```

Pour vérifier que la voie série est bien ouverte, on utilisera la méthode "isOpen()" qui retourne un booléen vrai si la connexion est établie.

Envoyer des données

Maintenant que la voie série est ouverte, nous allons pouvoir lui envoyer des données à traiter. Pour le bien de l'exercice, il nous faut récupérer un (des) caractère(s) à envoyer et retourner avec la casse inversée. Nous allons donc commencer par récupérer une chaîne de caractère de l'utilisateur :

```
1 chaine = input("Que voulez vous transformer ? ")
```

Puis nous allons simplement l'envoyer avec la fonction "write". Cette fonction prend en paramètre un tableau de bytes. Nous allons donc transformer notre chaîne pour convenir à ce format avec la fonction python "bytes()" qui prend en paramètres la chaîne de caractères et le format d'encodage.

```
1 bytes(chaine, 'UTF-8')
```

Ce tableau peut directement être envoyé dans la fonction write() :

```
1 port.write(bytes(chaine, 'UTF-8'))
```

Recevoir des données

La suite logique des choses voudrait que l'on réussisse à recevoir des données. C'est ce que nous allons voir maintenant. 😊 Nous allons tout d'abord vérifier que des données sont arrivées sur la voie série via la méthode inWaiting(). Cette dernière nous renvoie le nombre de caractères dans le buffer de réception de la voie série. S'il est différent de 0, cela signifie qu'il y a des données à lire. S'il y a des caractères, nous allons utiliser la fonction "read()" pour les récupérer. Cette fonction retourne les caractères (byte) un par un dans l'ordre où il sont arrivés. Un exemple de récupération de caractère pourrait-être :

```
1 while(port.inWaiting() != 0):
2     car = port.read() #on lit un caractère
3     print(car) #on l'affiche
```

Vous en savez maintenant presque autant que moi sur la voie série en python 😊 ! Je suis conscient que c'est assez maigre comparé aux autres langages, mais je ne vais pas non plus apprendre tout les langages du monde 😊

Code exemple complet et commenté

```
1 #!/usr/bin/python3
2 # -*-coding:Utf-8 -*
3
4 from serial import Serial
5 import time
6
7 port = Serial('/dev/ttyACM0', 9600)
8
9 port.open()
10
11 if (port.isOpen()): #test que le port est ouvert
12     chaine = input("Que voulez vous transformer ? ") #demande de la chaîne à envoyer
13     port.write(bytes(chaine, 'UTF-8')) #on écrit la chaîne en question
14     while(port.inWaiting() == 0): #attend que des données soit revenues
15         #on attend 0.5 seconde pour que les données arrive
16         time.sleep(0.5)
17
18     #si on arrive là, des données sont arrivées
19     while(port.inWaiting() != 0): #il y a des données !
20         car = port.read() #on lit un caractère
21         print(car) #on l'affiche
22 else:
23     print ("Le port n'a pas pu être ouvert !")
```

Cette annexe vous aura permis de comprendre un peu comment utiliser la voie série en général avec un ordinateur. Avec vos connaissances vous êtes dorénavant capable de créer des interfaces graphiques pour communiquer avec votre arduino. De grandes possibilités s'offrent à vous, et de plus grandes vous attendent avec les parties qui suivent...