Contenu

[[ Arduino 301] Généralités sur la voie série 4](#_Toc473877260)

[Communiquer, pourquoi ? 4](#_Toc473877261)

[Transmettre des informations 4](#_Toc473877262)

[Le récepteur 5](#_Toc473877263)

[La norme RS232 6](#_Toc473877264)

[La mécanique 6](#_Toc473877265)

[Le signal électrique et le protocole 7](#_Toc473877266)

[Connexion série entre Arduino et … 11](#_Toc473877267)

[Émulation du port série 12](#_Toc473877268)

[Arduino et un autre microcontrôleur 12](#_Toc473877269)

[Arduino au PC 13](#_Toc473877270)

[Au delà d’Arduino avec la connexion série 15](#_Toc473877271)

[Le connecteur série (ou sortie DB9) 15](#_Toc473877272)

[Désolé, je suis occupé… 16](#_Toc473877273)

[Avec ou sans horloge ? 17](#_Toc473877274)

[RS-232 18](#_Toc473877275)

[Prepare the breadboard 18](#_Toc473877276)

[Cables 20](#_Toc473877277)

[Program the Arduino 22](#_Toc473877278)

[La communication série avec arduino 26](#_Toc473877279)

[Présentation 26](#_Toc473877280)

[Pré-requis 27](#_Toc473877281)

[Logiciels et bibliothèques à installer 27](#_Toc473877282)

[Vocabulaire 27](#_Toc473877283)

[Communiquer avec le moniteur série 27](#_Toc473877284)

[Communication la plus simple possible 28](#_Toc473877285)

[Echo 29](#_Toc473877286)

[Lire des entiers 31](#_Toc473877287)

[Lecture basique d'un entier 31](#_Toc473877288)

[Faire des maths avec deux entiers 31](#_Toc473877289)

[Communication entre un programme et la carte 33](#_Toc473877290)

[Lecture 33](#_Toc473877291)

[Écriture 34](#_Toc473877292)

[Influence du nombre de bauds 34](#_Toc473877293)

[Conclusion 35](#_Toc473877294)

[En résumé 35](#_Toc473877295)

[Quelques problèmes fréquents 35](#_Toc473877296)

[Problème de téléchargement 35](#_Toc473877297)

[Lecture de données de taille indéterminées 35](#_Toc473877298)

[Téléchargements 35](#_Toc473877299)

[Shield Sparkfun RS232 v2 DEV-13029 37](#_Toc473877300)

[RS-232 (by Arduino.cc) 39](#_Toc473877301)

[Prepare the breadboard 39](#_Toc473877302)

[Cables 41](#_Toc473877303)

[Program the Arduino 43](#_Toc473877304)

[Shield Aliexpress MoreSuns 15€ 47](#_Toc473877305)

<http://eskimon.fr/102-arduino-301-generalites-voie-serie>

<https://www.arduino.cc/en/Tutorial/ArduinoSoftwareRS232>

<https://www.jujens.eu/posts/2014/May/05/Communication-serie/>

<https://www.sparkfun.com/products/13029>

<https://www.arduino.cc/en/Tutorial/ArduinoSoftwareRS232>

Shield Aliexpress <https://www.aliexpress.com/store/1548016>

# [ Arduino 301] Généralités sur la voie série

Posté par [Eskimon](http://eskimon.fr/author/eskimon) le [10 décembre 2013, 8:03](http://eskimon.fr/102-arduino-301-generalites-voie-serie)

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 !

**Sommaire**

* [Communiquer, pourquoi ?](http://eskimon.fr/102-arduino-301-generalites-voie-serie#770226)
* [La norme RS232](http://eskimon.fr/102-arduino-301-generalites-voie-serie#770227)
* [Connexion série entre Arduino et …](http://eskimon.fr/102-arduino-301-generalites-voie-serie#771592)
* [Au delà d’Arduino avec la connexion série](http://eskimon.fr/102-arduino-301-generalites-voie-serie#775484)

## Communiquer, pourquoi ?

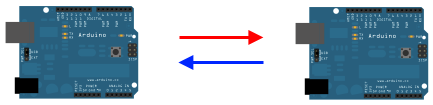
Nous avons vu dans la partie précédente où nous faisions nos premiers pas avec Arduino, comment utiliser la carte. Nous avons principalement utilisé des LED pour communiquer à l’utilisateur (donc vous, à priori) certaines informations. Cela pouvait être une LED ou un groupe de LED qui peut indiquer tout 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.

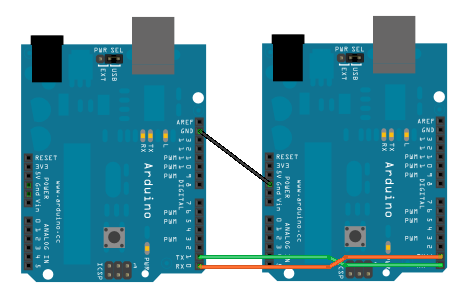
#### Applications de la norme

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-chaussé. Dans notre cas, on considérera que le 0V sera notre référentiel électrique commun.
* Les deux autres câbles permettent la transmission des données. L’un sert à l’envoi des données pour un émetteur, mais sert aussi pour la réception des données venant de l’autre émetteur. Idem pour l’autre câble. Il permet l’émission de l’un et la réception de l’autre.

  
Deux cartes Arduino reliées par 3 câbles :

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

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

### 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 codé 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](http://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange#Table_des_128_caract.C3.A8res_ASCII). En haut à gauche de la table ASCII, on observe la ligne : « Code en base… » et là vous avez : 10, 8, 16, 2. Respectivement, ce sont les bases décimale (10), octale (8), hexadécimale (16) et binaire (2). 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.

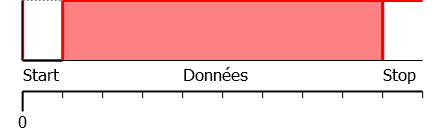
Je crois ne pas bien comprendre pourquoi on envoie une lettre… qui va la recevoir et pour quoi faire ? o\_O

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, voire même des phrases complètes. Pour ne pas aller trop vite, nous resterons avec cette unique lettre. Lorsque l’on enverra la lettre à l’ordinateur, nous utiliserons un petit module intégré dans le logiciel Arduino pour visualiser le message réceptionné. C’est donc nous qui allons voir ce que l’on transmet via la voie série.

#### L’ordre et les délimiteurs

On va à présent voir comment est transmit 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 :

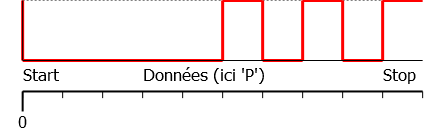
1. Lorsque l’on passe un coup de fil, bien généralement on commence par dire « Bonjour » ou « Allo ». Ce début de message permet de faire l’ouverture de la conversation. En effet, si l’on reçoit un appel et que personne ne répond après avoir décroché, la conversation ne peut avoir lieu. Dans la norme 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 ingénieurs qui ont inventé ce protocole ont eu la bonne idée de transmettre les données à l’envers… Par conséquent, la bonne réponse était : 0000010101. Avec un chronogramme, on observerait ceci :

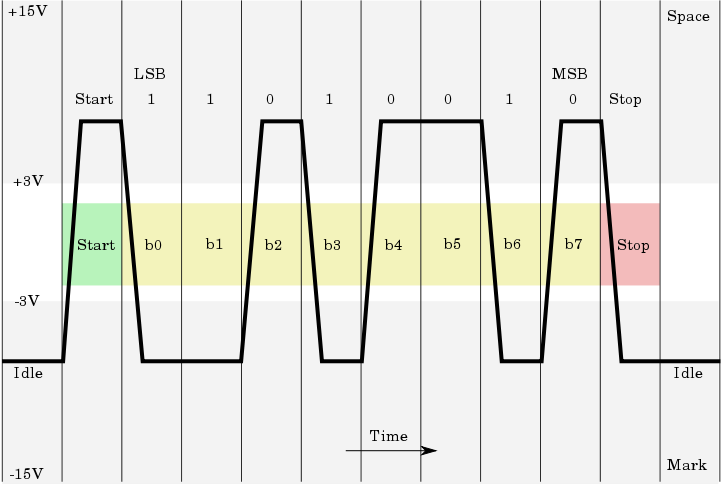


#### Un peu de vocabulaire

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 liaisons série, on aura le bit de poids faible juste après le start, donc à gauche et le bit de poids fort à droite. Avec le nombre précédent, si l’on devait le lire à l’envers cela donnerait : 5936. Bit de poids faible à gauche et à droite le bit de poids fort.

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](http://fr.wikipedia.org/wiki/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

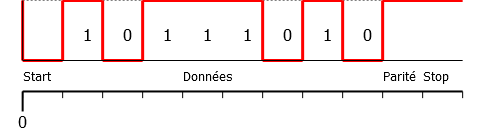
La norme RS-232 définit la vitesse à laquelle sont envoyée les données. Elles sont exprimés en bit par seconde (bit/s). Elle préconise des vitesse inférieures à 20 000 bits/s. Sauf qu’en pratique, il est très courant d’utiliser des débits supérieurs pouvant atteindre les 115 200 bits/s. Quand on va utiliser 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 |

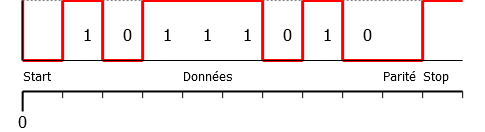
Plus le câble est court, plus le débit pourra être élevé car moins il y a d’affaiblissement des tensions et de risque de parasites. Tandis que si la distance séparant les deux interlocuteurs grandie, la vitesse de communication diminuera de façon effective.

#### 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…

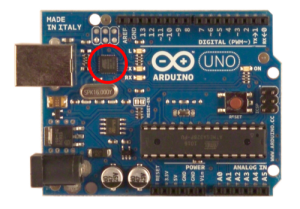
## Connexion série entre Arduino et …

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.

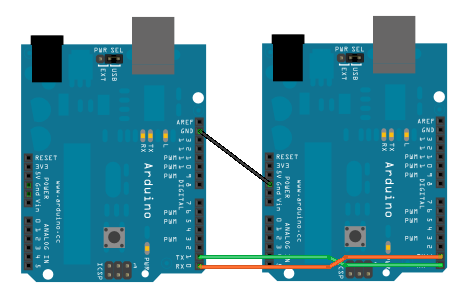
#### La tension des microcontrôleurs

|  |  |
| --- | --- |
|  | **Tension** |
| **NL0** | 0V |
| **NL1** | +5V |

Contrairement à ce qu’impose la norme RS-232, les microcontrôleur ne peuvent pas utiliser des tensions négatives. Du coup, ils utilisent les seuls et uniques tensions qu’ils peuvent utiliser, à savoir le 0V et le +5V. Il y a donc quelques petits changement au niveau de la transmission série. Un niveau logique 0 correspond à une tension de 0V et un niveau logique 1 correspond à une tension de +5V. (cf. tableau ci-contre) Fort heureusement, comme les microcontrôleurs utilisent quasiment tous cette norme, il n’y a aucun problème à connecter deux microcontrôleurs entre-eux. Cette norme s’appelle alors UART pour 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

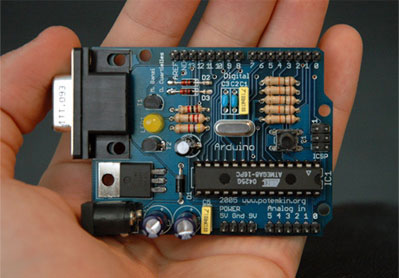
#### Le connecteur série (ou sortie DB9)

Alors là, les enfants, je vous parle d’un temps que les moins de vingt ans ne peuvent pas connaittttrrreuhhh… Bon on reprend ! Comme énoncé, je vous parle de quelque chose qui n’existe presque plus. Ou du moins, vous ne trouverez certainement plus cette « chose » sur la connectique de votre ordinateur. En effet, je vais vous parler du connecteur DB9 (ou DE9). Il y a quelques années, l’USB n’était pas si véloce et surtout pas tant répandu. Beaucoup de matériels (surtout d’un point de vue industriel) utilisaient la voie série (et le font encore). 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

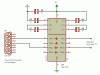
Aujourd’hui, le connecteur DB9 a déjà bien disparu mais reste présent sur les « vieux » ordinateurs ou sur d’autres appareils utilisant la voie série. C’est pourquoi, le jour où vous aurez besoin de communiquer avec un tel dispositif, il vous faudra faire un peu d’électronique…

#### Une petite histoire d’adaptation

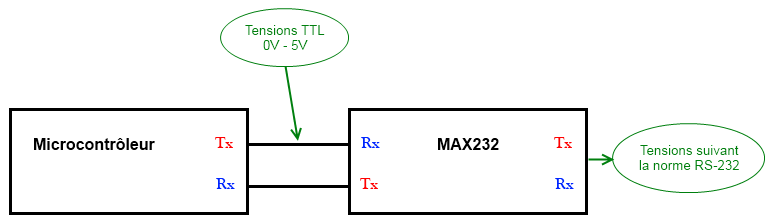
Si vous avez donc l’occasion de connecter votre carte Arduino à un quelconque dispositif utilisant la voie série, il va falloir faire attention aux tensions…oui, encore elles ! Je l’ai déjà dis, un microcontrôleur utilise des tensions de 0V et 5V, qu’on appel TTL. Hors, la norme RS-232 impose des tensions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

[**Datasheet du MAX232**](http://www.ti.com/lit/ds/symlink/max232.pdf)

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



Le principe de ce composant, utilisé avec quelques condensateur, 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 connaitre 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 peu 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…

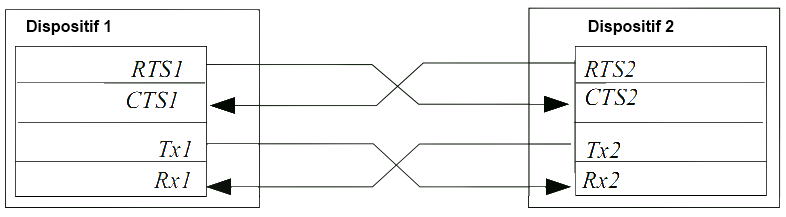
Dans certains cas, et il n’est pas rare, les dispositifs communicant entre eux par l’intermédiaire de la voie série ne traitent pas les données à la même vitesse. Tout comme lorsque l’on communique avec quelqu’un, il arrive parfois qu’il n’arrive plus à suivre ce que l’on dit car il en prend des notes. 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 à a 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 utilise ce contrôle de flux matériel à 5 fils. Une transmission s’effectue de la manière suivante :

* Le dispositif 1, que je nommerais maintenant l’émetteur, met un état logique 0 sur sa broche RTS1. Il demande donc au dispositif 2, le récepteur, pour émettre des données.
* Si le récepteur est prêt à recevoir des données, alors il met un niveau logique 0 sur sa broche RTS2.
* Les deux dispositifs sont prêts, l’émetteur peut donc envoyer les données qu’il a à transmettre.
* Une fois les données envoyées, l’émetteur passe à 1 l’état logique présent sur sa broche RTS1.
* Le récepteur voit ce changement d’état et sait donc que c’est la fin de la communication des données, il passe alors l’état logique de sa broche RTS2 à 1.

Ce contrôle n’est 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.

<https://www.arduino.cc/en/Tutorial/ArduinoSoftwareRS232>

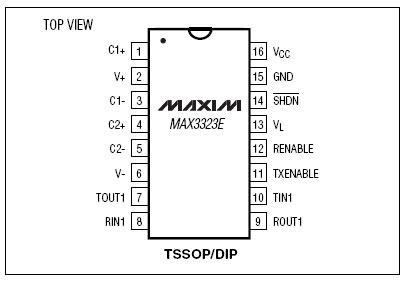
# RS-232

In this tutorial you will learn how to communicate with a computer using a MAX3323 single channel RS-232 driver/receiver and a software serial connection on the Arduino. A general purpose software serial tutorial can be found [here](http://www.arduino.cc/en/Tutorial/SoftwareSerial).

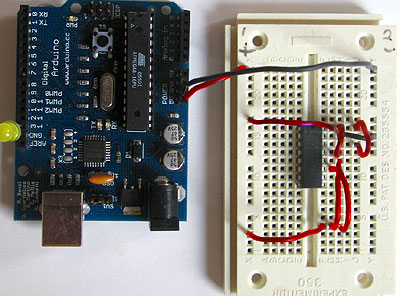
Materials needed:

* Computer with a terminal program installed (ie. HyperTerminal or RealTerm on the PC, Zterm on Mac)
* Serial-Breadboard cable
* MAX3323 chip (or similar)
* 4 1uf capacitors
* Solderless breadboard
* Hookup wire
* Arduino Microcontroller Module
* Light emitting Diode (LED) - optional, for debugging

## Prepare the breadboard

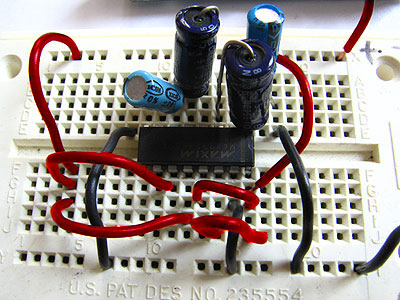


Insert the MAX3323 chip in the breadboard. Connect 5V power and ground from the breadboard to 5V power and ground from the microcontroller. Connect pin 15 on the MAX233 chip to ground and pins 16 and 14 - 11 to 5V. If you are using an LED connect it between pin 13 and ground.



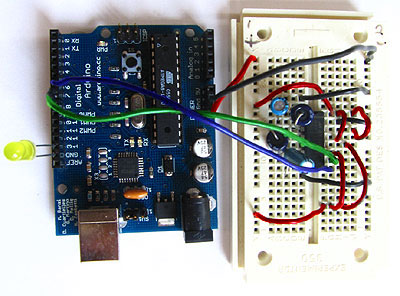
+5v wires are red, GND wires are black

Connect a 1uF capacitor across pins 1 and 3, another across pins 4 and 5, another between pin 2 and ground, and the last between pin 6 and ground. If you are using polarized capacitors make sure the negative pins connect to the negative sides (pins 3 and 5 and ground).



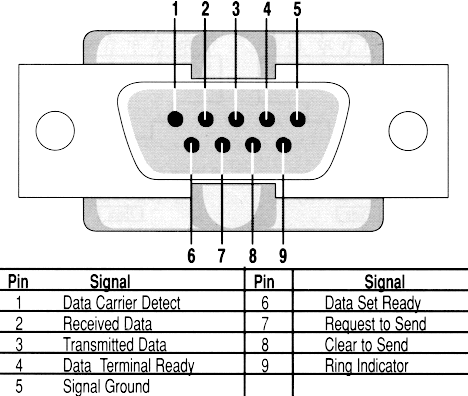
+5v wires are red, GND wires are black

Determine which Arduino pins you want to use for your transmit (TX) and recieve (RX) lines. In this tutorial we will be using Arduino pin 6 for receiving and pin 7 for transmitting. Connect your TX pin (7) to MAX3323 pin 10 (T1IN). Connect your RX pin (6) to MAX3323 pin 9 (R1OUT).



TX wire Green, RX wire Blue, +5v wires are red, GND wires are black

## Cables



(*DB9* Serial Connector Pin Diagram)

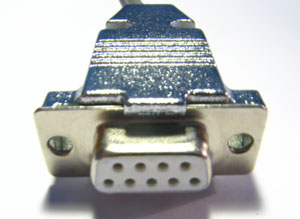
If you do not have one already, you need to make a cable to connect from the serial port (or USB-serial adapter) on your computer and the breadboard. To do this, pick up a female DB9 connector from radioshack. Pick three different colors of wire, one for TX, one for RX, and one for ground. Solder your TX wire to pin 2 of the DB9 connector, RX wire to pin 3 and Ground to pin 5.



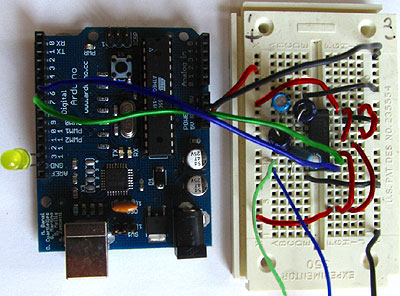
Connect pins 1 and 6 to pin 4 and pin 7 to pin 8. Heatshrink the wire connections to avoid accidental shorts.



Enclose the connector in a backshell to further protect the signal and enable easy unplugging from your serial port.



Connect the TX line from your computer to pin 8 (R1IN) on the MAX233 and the RX line to pin 7 (T1OUT). Connect the ground line from your computer to ground on the breadboard.



TX wires Green, RX wires Blue, +5v wires are red, GND wires are black

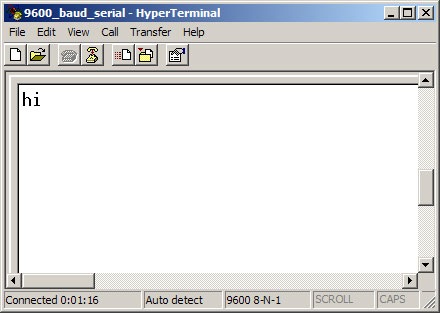
## Program the Arduino

Now we will write the code to enable serial data communication. This program will simply wait for a character to arrive in the serial recieving port and then spit it back out in uppercase out the transmit port. This is a good general purpose serial debugging program and you should be able to extrapolate from this example to cover all your basic serial needs. Upload the following code into the Arduino microcontroller module:

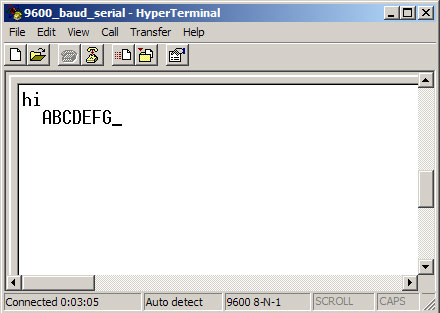
//Created August 23 2006  
//Heather Dewey-Hagborg  
//http://www.arduino.cc  
  
#include <ctype.h>  
  
#define bit9600Delay 100    
#define halfBit9600Delay 50  
#define bit4800Delay 188   
#define halfBit4800Delay 94   
  
byte rx = 6;  
byte tx = 7;  
byte SWval;  
  
void setup() {  
  pinMode(rx,INPUT);  
  pinMode(tx,OUTPUT);  
  digitalWrite(tx,HIGH);  
  delay(2);  
  digitalWrite(13,HIGH); //turn on debugging LED  
  SWprint('h');  //debugging hello  
  SWprint('i');  
  SWprint(10); //carriage return  
}  
  
void SWprint(int data)  
{  
  byte mask;  
  //startbit  
  digitalWrite(tx,LOW);  
  delayMicroseconds(bit9600Delay);  
  for (mask = 0x01; mask>0; mask <<= 1) {  
    if (data & mask){ // choose bit  
     digitalWrite(tx,HIGH); // send 1  
    }  
    else{  
     digitalWrite(tx,LOW); // send 0  
    }  
    delayMicroseconds(bit9600Delay);  
  }  
  //stop bit  
  digitalWrite(tx, HIGH);  
  delayMicroseconds(bit9600Delay);  
}  
  
int SWread()  
{  
  byte val = 0;  
  while (digitalRead(rx));  
  //wait for start bit  
  if (digitalRead(rx) == LOW) {  
    delayMicroseconds(halfBit9600Delay);  
    for (int offset = 0; offset < 8; offset++) {  
     delayMicroseconds(bit9600Delay);  
     val |= digitalRead(rx) << offset;  
    }  
    //wait for stop bit + extra  
    delayMicroseconds(bit9600Delay);   
    delayMicroseconds(bit9600Delay);  
    return val;  
  }  
}  
  
void loop()  
{  
    SWval = SWread();   
    SWprint(toupper(SWval));  
}

[[Get Code]](https://www.arduino.cc/en/Tutorial/ArduinoSoftwareRS232?action=sourceblock&num=1)

Open up your serial terminal program and set it to 9600 baud, 8 data bits, 1 stop bit, no parity, no hardware flow control. Press the reset button on the arduino board. The word "hi" should appear in the terminal window followed by an advancement to the next line. Here is a shot of what it should look like in Hyperterminal, the free pre-installed windows terminal application.



Now, try typing a lowercase character into the terminal window. You should see the letter you typed return to you in uppercase.



If this works, congratulations! Your serial connection is working as planned. You can now use your new serial/computer connection to print debugging statements from your code, and to send commands to your microcontroller.

code and tutorial by Heather Dewey-Hagborg, photos by Thomas Dexter

<https://www.jujens.eu/posts/2014/May/05/Communication-serie/>

# [La communication série avec arduino](https://www.jujens.eu/posts/2014/May/05/Communication-serie/)

Mon 05 May 2014 | tags: [Communication série](https://www.jujens.eu/tag/communication-serie.html)[Arduino](https://www.jujens.eu/tag/arduino.html)[Python](https://www.jujens.eu/tag/python.html)[Robotique](https://www.jujens.eu/tag/robotique.html)

La communication série est indispensable pour dialoguer avec votre Arduino puisque c'est le mode de transmission utilisé pour communiquer avec la carte. Dans ce tuto je vais vous expliquer comment ça fonctionne avec des exemples pratiques pour faire communiquer une carte arduino (le modèle n'importe pas) et votre PC grâce à une connexion USB. Nous utiliserons dans un premier temps l'IDE Arduino puis du code Python. Je suppose que vous avez quelques bases en programmation. Les exemples seront suffisamment simples pour que la connaissance préalable de python ne soit pas nécessaire.

Le tuto peut se réaliser indifféremment sous Windows, GNU/Linux, Mac ou autre. Néanmoins, les exemples fonctionnent tel quel uniquement sous linux. Sous d'autres systèmes, vous pouvez être amené à changer les chemins des ports. Le reste devrait rester inchangé.

Tous les codes sont disponibles en [téléchargement](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#telechargement) à la fin de ce document sous licence [Apache v2](http://www.apache.org/licenses/LICENSE-2.0.html).

**Sommaire**

* [Présentation](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#presentation)
* [Pré-requis](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#pre-requis)
  + [Logiciels et bibliothèques à installer](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#logiciels-et-bibliotheques-a-installer)
  + [Vocabulaire](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#vocabulaire)
* [Communiquer avec le moniteur série](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#communiquer-avec-le-moniteur-serie)
  + [Communication la plus simple possible](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#communication-la-plus-simple-possible)
  + [Echo](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#echo)
* [Lire des entiers](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#lire-des-entiers)
  + [Lecture basique d'un entier](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#lecture-basique-d-un-entier)
  + [Faire des maths avec deux entiers](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#faire-des-maths-avec-deux-entiers)
* [Communication entre un programme et la carte](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#communication-entre-un-programme-et-la-carte)
  + [Lecture](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#lecture)
  + [Écriture](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#ecriture)
* [Influence du nombre de bauds](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#influence-du-nombre-de-bauds)
* [Conclusion](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#conclusion)
  + [En résumé](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#en-resume)
* [Quelques problèmes fréquents](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#quelques-problemes-frequents)
  + [Problème de téléchargement](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#probleme-de-telechargement)
  + [Lecture de données de taille indéterminées](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#lecture-de-donnees-de-taille-indeterminees)
* [Téléchargements](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#telechargements)

## [Présentation](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id2)

En communication série, on découpe l'information à transmettre en petits blocs de taille fixe avant de la transmettre. La taille des blocs correspond au nombre des lignes disponibles pour la transmission des données.

Ce type de communication s'oppose à la communication parallèle. En communication parallèle, il y a une ligne par bits à transmettre. Tous les bits sont donc transmis en même temps. Pour une même fréquence de communication, la communication parallèle est donc plus rapide.

L'avantage de la communication série sur la communication parallèle est qu'elle nécessite moins de lignes, donc moins de broches, donc moins de composants. Son coût est donc plus faible.

Les protocoles de communication série les plus connus sont :

* Le protocole [USB](http://fr.wikipedia.org/wiki/Universal_Serial_Bus)
* Le protocole [I2C](http://fr.wikipedia.org/wiki/I2C)
* Le protocole [PCI Express](http://fr.wikipedia.org/wiki/PCI_Express)

## [Pré-requis](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id3)

### [Logiciels et bibliothèques à installer](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id4)

* Arduino IDE : l'IDE officiel arduino. Le moyen le plus simple d'écrire et de télécharger les programmes sur la carte. Il est disponible sur le site officiel d'Arduino : <http://arduino.cc/en/Main/Software#toc1>
* Python3 : indispensable pour exécuter les certains programmes d'exemple (uniquement pour la partie [communication entre un programme et la carte](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#communication-entre-un-programme-et-la-carte)). Il se trouve dans les dépôts de votre distribution linux ou par là : [Python](http://www.python.org/)
* Pyserial : c'est la bibliothèque python qui permet de faire de la communication série. Elle est normalement dans les dépôts de votre distribution linux. Elle peut aussi être installée via [pip](https://pypi.python.org/pypi/pyserial/2.7) : pip install pyserial.

### [Vocabulaire](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id5)

Commençons par définir quelques termes de vocabulaire avec lesquels vous pouvez ne pas être familier et que nous allons rencontrer tout au long de ce tuto :

* Ascii : c'est l'acronyme de American Standard Code for Information Exchange, soit code américain pour l'échange d'information. Ce code permet de représenter avec des nombres compris entre 0 et 127 les caractères utilisés dans l'alphabet américain soit :
  + Les lettres, minuscules et majuscules
  + Les symboles de ponctuation : . , ;
  + Des caractères spéciaux : $ # \* +
  + Pas les accents
* Baud (Bd) : Unité de mesure du nombre de symboles transmissibles par seconde. Il ne faut pas le confondre avec le nombre de bits par seconde (bps). Par exemple, considérons que l'on souhaite transmettre le symbole ascii a en 1 seconde exactement. D'après la [table ascii](http://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange#Table_des_128_caract.C3.A8res_ASCII) a s'écrit en binaire 01100001. Pour transmettre ce symbole en 1 seconde, il faut communiquer avec une vitesse de 1 Bd mais de 8 bps.
* Bytes (en python3) : Petite précision de vocabulaire spécifique à python 3. C'est un type de base du langage qui représente une chaîne d'octets. C'est ce type de donnée que nous manipulerons dans [communication entre un programme et la carte](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#communication-entre-un-programme-et-la-carte)

## [Communiquer avec le moniteur série](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id6)

Avant de rentrer vraiment dans la communication, je vous signale que deux diodes présentes sur la arduino, peuvent être utiles :

* TX : s'allume lors d'une transmission
* RX : s'allume lors d'une réception



### [Communication la plus simple possible](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id7)

Nous allons aborder notre premier exemple. Il consiste à envoyer des données sur le port série de la arduino de d'utiliser l'outil intégré à l'IDE pour voir les données envoyées. Lancez l'éditeur arduino et entrez le code ci-dessous :

void setup() {

Serial.begin(9600);

}

void loop() {

Serial.println("Coucou");

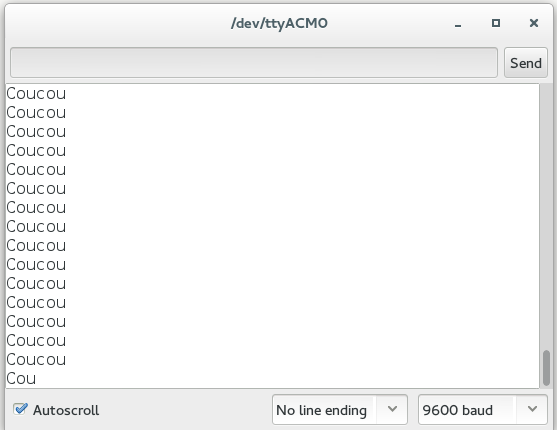
}

Détaillons le :

* Dans la fonction setup, on initialise simplement le port série en donnant sa vitesse avec Serial.begin(9600). En effet, pour que la carte et l'ordinateur communique correctement, ils doivent « parler » à la même vitesse. Une vitesse différente est une erreur fréquente au début. Pensez à la vérifier si vous rencontrez des problèmes. On choisit ici 9600 Bd ce qui est une valeur classique. Nous verrons plus loin l'[influence du nombre de bauds](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#influence-du-nombre-de-bauds).
* Dans la fonction loop, on se contente d'envoyer "Coucou" avec Serial.println("Coucou").

Vérifiez dans Outils > Carte que la bonne carte est sélectionnée et dans Outils > Port série que le bon port est choisi. Si tout est bon, utilisez le bouton Téléchargements pour envoyer le code sur la carte.

Ouvrez le moniteur série avec Outils > Moniteur série. Vérifiez en bas à droite que la valeur est bien à 9600 Bd. Si tout va bien, vous devriez voir le moniteur se remplir de Coucou.



Si ce n'est pas le cas, reprenez les étapes précédentes, vous avez raté quelque chose. Vous pouvez aussi regarder du côté des [quelques problèmes fréquents](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#quelques-problemes-frequents).

### [Echo](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id8)

Après cet exemple d'autant moins intéressant qu'il ne fonctionne que dans un sens, faisons un programme d'écho : la arduino va nous renvoyer exactement ce qu'on lui envoie. Si elle ne reçoit rien, elle envoie Rien. Voilà le code de la arduino :

void setup() {

Serial.begin(9600);

}

void loop() {

if ( Serial.available() ) {

int lu = Serial.read();

Serial.println(lu);

} else {

Serial.println("Rien");

}

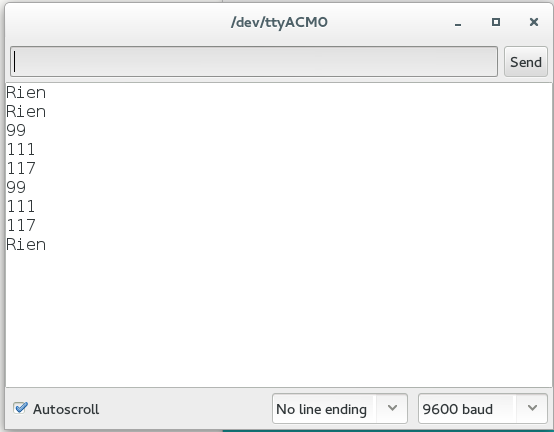
delay(2000);

}

Regardons le d'un peu plus près. Pas de changement du côté de setup. Dans loop :

* Nous commençons par vérifier si des données en attente de lecture sur le port avec : Serial.available(). Cette fonction renvoie true si telle est le cas.
* Si on a des données, on définit une variable de type int dans laquelle on place une donnée en attente. Puis on affiche cette donnée.
* Sinon, on affiche Rien.
* Enfin, on attend 2 secondes histoire d'avoir le temps de voir ce qui se passe.

Testez ce code. Pour envoyer une donnée à la carte, tapez du texte dans la barre du haut puis presser Entrée ou appuyez sur Envoyer. Vous devriez obtenir quelque chose comme ça (j'ai tapé coucou) :



Mais ce n'est pas ce que j'ai tapé ! En effet, Serial.read() lit un octet de données sur le port série. Ni plus, ni moins. Et c'est la [seule manière](http://arduino.cc/en/Serial/Read) de lire les données. Mais rassurez-vous, vous pouvez facilement donner un sens à ce qui s'affiche : les données sont encodées en ascii. Par conséquent, le numéro qui s'affiche est la version décimale du code ascii. Vous pouvez consulter la [table](http://fr.wikipedia.org/wiki/American_Standard_Code_for_Information_Interchange#Table_des_128_caract.C3.A8res_ASCII) pour convertir ce code dans le caractère standard.

## [Lire des entiers](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id9)

Dans cette partie, nous allons utiliser la table ascii pour convertir le nombre reçu en quelque chose d'exploitable. Nous ferons tout d'abord un programme qui lit et affiche correctement un nombre puis un second programme qui fera des opérations sur deux entiers que vous choisirez.

### [Lecture basique d'un entier](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id10)

Ce n'est pas très compliqué. Il suffit de prendre la table ascii et de constater que les chiffres sont codés en décimal entre 48 et 57 puis de décaler.

int byte\_read = 0; ///< The current byte

int recieved\_integer = 0; ///< The current recieved integer

void setup() {

Serial.begin(9600);

}

boolean is\_a\_number(int n)

{

return n >= 48 && n <= 57;

}

int ascii2int(int n, int byte\_read)

{

return n\*10 + (byte\_read - 48);

}

void loop() {

recieved\_integer = 0;

while ( Serial.available() ) {

byte\_read = Serial.read();

if ( is\_a\_number(byte\_read) ) {

recieved\_integer = ascii2int(recieved\_integer, byte\_read);

}

}

Serial.println( recieved\_integer );

delay(1000);

}

Détaillons ce programme :

* On définit les variables globales en début de programme. Ça ne sert à rien de les redéfinir à chaque tour de boucle.
* Ensuite on définit deux fonctions :
  + boolean is\_a\_number(int n) qui renvoie vrai si on lui donne un nombre ascii codé en décimal.
  + int ascii2int(int n, int byte\_read) qui renvoie le nombre sous la forme d'un entier. Elle prend en paramètre l'entier déjà calculé ainsi que l'octet à ajouter.
* Dans loop : on fait la conversion si un entier est disponible.

Je vous laisse constater que ça marche bien.

### [Faire des maths avec deux entiers](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id11)

Maintenant on va faire un peu mieux : lire deux entiers séparés par une virgule et les additionner. Le code est ci-dessous :

const int baudrate = 9600;

int byte\_read = 0; ///< The current byte read.

int coords[2]; ///< Contains x and y.

int index = 0; ///< 0: reading x, 1: reading y.

int separator = 44; ///< The separator between the integers (44: ,)

void setup()

{

Serial.begin(baudrate);

}

boolean is\_a\_number(int n)

{

return n >= 48 && n <= 57;

}

int ascii2int(int n, int byte\_read)

{

return n\*10 + (byte\_read - 48);

}

void loop()

{

coords[0] = 0;

coords[1] = 0;

index = 0;

while ( Serial.available() > 0 )

{

byte\_read = Serial.read();

if ( is\_a\_number(byte\_read) )

{

coords[index] = ascii2int( coords[index], byte\_read );

}

else if ( byte\_read == separator )

{

++index;

}

}

if ( index )

{

Serial.print("x + y = ");

Serial.println(coords[0] + coords[1]);

Serial.print("x \* y = ");

Serial.println(coords[0] \* coords[1]);

}

else

{

Serial.println();

}

delay(1000);

}

Décortiquons :

* Concernant les variables :
  + Nos deux nombres seront stockés dans un tableau d'entier.
  + index nous permet de savoir si on lit x ou y.
* Dans loop :
  + On prend soin de remettre à 0 les variables pour éviter les problèmes.
  + Puis suivant si l'octet lut représente un nombre ou le séparateur, on lit un nombre ou on incrémente l'index.
  + Concernant les calculs : on vérifie que index n'est pas à 0 et donc qu'on a bien lut deux nombres. Si tel est le cas, on fait l'addition et la multiplication de x et y et on affiche le résultat. La fonction print affiche ses paramètres sans revenir à la ligne comme println.

Vous pouvez vous amuser à écrire une calculatrice plus complète si vous voulez. Vous pouvez même envoyer l'opération à effectuer si vous voulez. Utilisez éventuellement l'instruction [switch](http://arduino.cc/en/Reference/SwitchCase).

## [Communication entre un programme et la carte](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id12)

Cette partie reprend en partie ce qui a déjà été évoqué [ici](https://www.jujens.eu/posts/2014/Jan/11/communication-serie-facile-python/) mais de façon plus détaillée et un peu moins complète.

### [Lecture](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id13)

Nous allons maintenant faire communiquer un programme de votre ordinateur (autre que l'IDE Arduino) et la carte. Le code sera écrit directement dans l'interpréteur, mais vous pouvez sans problème le mettre dans un fichier puis l'exécuter. Il sera écrit en python, mais les principes restent les mêmes dans les autres langages. Reprenons le code « echo » pour la carte :

void setup() {

Serial.begin(9600);

}

void loop() {

if ( Serial.available() ) {

int lu = Serial.read();

Serial.println(lu);

} else {

Serial.println("Rien");

}

delay(2000);

}

Lancez l'interpréteur python sur votre PC et importer la bibliothèque pyserial :

Python 3.3.2 (default, Mar 5 2014, 08:21:05)

[GCC 4.8.2 20131212 (Red Hat 4.8.2-7)] on linux

Type "help", "copyright", "credits" or "license" for more information.

>>> from serial import Serial

Créez la communication série avec (en remplaçant /dev/ttyACM0 par le nom de votre port) :

>>> serial\_port = Serial(port='/dev/ttyACM0', baudrate=9600)

Pour lire des données, rien de plus simple :

>>> serial\_port.read()

b'R'

>>> serial\_port.readline()

b'Rien\r\n'

Les résultats sont de type bytes. Si vous comptez faire des manipulations par la suite (hors transmission série), je vous conseille de passer dans le type string plus adapté (ou un autre type qui représente correctement vos données) :

>>> lu = serial\_port.readline()

>>> lu

b'Rien\r\n'

>>> type(lu)

<class 'bytes'>

>>> chaine = lu.decode('ascii')

>>> chaine

'Rien\r\n'

>>> type(chaine)

<class 'str'>

### [Écriture](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id14)

Pour écrire des données c'est aussi très simple :

>>> while True:

... nombre = input('Entrez un chiffre : ')

... serial\_port.write(nombre.encode('ascii'))

... serial\_port.readline()

...

Entrez un nombre : 78

1

b'55\r\n'

Le nombre 1 est la valeur de retour de serial\_port.write qui renvoie la longueur de la chaîne écrite.

## [Influence du nombre de bauds](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id15)

Comme vous avez dû le comprendre en lisant [vocabulaire](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#vocabulaire), plus le nombre de bauds est élevé, plus on peut transmettre de données dans un temps court. Nous allons illustrer ceci dans cette partie. Télécharger tout d'abord le fichier [donnee.txt](https://www.jujens.eu/static/communication-serie/donnees.txt) qui contient le texte à envoyer. Enregistrez de façon à pouvoir l'ouvrir facilement avec l'interpréteur python. Programmez votre arduino avec le code suivant :

void setup() {

Serial.begin(117500);

}

void loop() {

while ( Serial.available() ) {

Serial.read();

}

}

Puis dans l'interpréteur python, tapez :

>>> port = Serial(port='/dev/ttyACM0', baudrate=117500)

>>> f = open('donnees.txt', 'r')

>>> data = f.read()

>>> a.write(data.encode('ascii'))

L'interpréteur devrait répondre rapidement 8000. Remplacez maintenant 117500 par 4800 dans les deux codes. C'est **beaucoup** plus long, non ?

Le principal est d'utiliser la bonne vitesse de transmission pour son application. Avec une grande vitesse, vous transmettez plus d'informations mais cela peut poser des problèmes sur les grandes distances. À vous de tester et de choisir la bonne, sachant qu'en général sur des cas simples, 9600 Bd est suffisant.

## [Conclusion](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id16)

À partir de ces informations, vous devriez pouvoir faire ce que vous voulez tant que ce n'est pas trop compliqué. Je publierais sûrement d'ici quelques temps un tuto plus complet sur pyserial. En attendant, vous pouvez aller consulter la [documentation](http://pyserial.sourceforge.net/).

### [En résumé](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id17)

* Choisir la bonne vitesse de communication.
* Que les deux programmes qui communiquent le fasse à la même vitesse.
* Se souvenir que l'arduino lit les données en code ascii.
* Ne pas désespérer si ça ne fonctionne pas.

## [Quelques problèmes fréquents](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id18)

### [Problème de téléchargement](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id19)

Ils sont malheureusement courants et je n'ai pas de méthode miracle pour les régler. Vous pouvez :

* Vérifiez le port de communication
* Débranchez/Re-branchez la carte
* Relancez l'éditeur

### [Lecture de données de taille indéterminées](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id20)

Lors de tous nos tests de lecture, nous avons utilisé soit des int soit des tableaux de taille fixe de int. Vous ne pouvez pas lire des données de taille indéterminée avec la arduino. Par exemple, faire :

String data = Serial.read();

En effet, contrairement à votre ordinateur qui a beaucoup de mémoire, une arduino est beaucoup plus limitée : pas plus de quelques kilos octets. La lecture de données sous cette forme empêche le compilateur de connaître la taille nécessaire pour la donnée et va donc refuser de compiler le code.

Dans tous les cas, souvenez-vous que la mémoire est très limitée et que vous ne pourrez donc pas tout traiter.

## [Téléchargements](https://www.jujens.eu/posts/2014/May/05/Communication-serie/#id21)

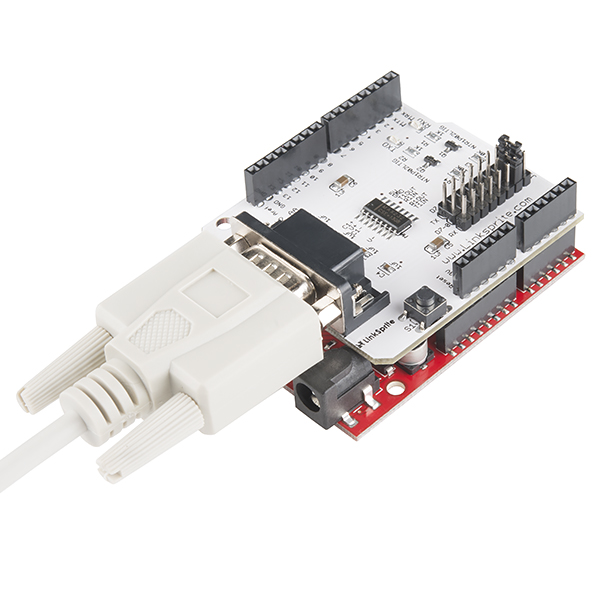
Tout est sous licence [Apache v2](http://www.apache.org/licenses/LICENSE-2.0.html).

* [serial\_port\_com.ino](https://www.jujens.eu/static/communication-serie/serial_port_com/serial_port_com.ino)
* [lire\_entier.ino](https://www.jujens.eu/static/communication-serie/lire_entier/lire_entier.ino)
* [echo.ino](https://www.jujens.eu/static/communication-serie/echo/echo.ino)
* [maths.ino](https://www.jujens.eu/static/communication-serie/maths/maths.ino)
* [baud.ino](https://www.jujens.eu/static/communication-serie/baud/baud.ino)

L'archive avec tous les codes : [codes.zip](https://www.jujens.eu/static/communication-serie/codes.zip)

# Shield Sparkfun RS232 v2 DEV-13029

<https://www.sparkfun.com/products/13029>



$11.95

**Description:** As you know, the Arduino micro controller only has a USB port and a TTL UART interface, so if you need an RS232 port directly connected to your Arduino, look no further than the RS232 Shield.

The RS232 serial port used to be the standard connection for most peripheral ports on PCs, but now the port is primarily used on industrial equipment and automation prototyping, with the RS232 Shield and its DB9 connector, you can now easily access those elements straight from your Arduino.

This rev of the RS232 Shield provides you with the option to choose between two pins from an Arduino (D0 to D7) as software serial ports to communicate with RS232 Shield. Also headers are included with this shield but will need to be soldered on by the end user.

**Documents:**

* [Schematics](https://cdn.sparkfun.com/datasheets/Dev/Arduino/Shields/SHD_RS232_V2_AA_SCH.pdf)
* [Datasheet](https://cdn.sparkfun.com/datasheets/Dev/Arduino/Shields/max232.pdf) (MAX232)
* [Product Wiki](http://linksprite.com/wiki/index.php5?title=RS232_Shield_V2_for_Arduino)

# RS-232 (by Arduino.cc)

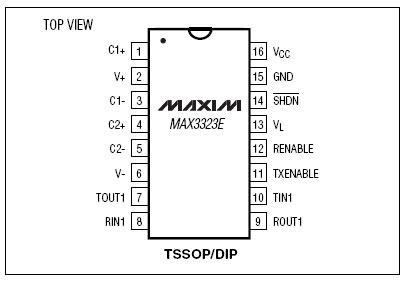
<https://www.arduino.cc/en/Tutorial/ArduinoSoftwareRS232>

In this tutorial you will learn how to communicate with a computer using a MAX3323 single channel RS-232 driver/receiver and a software serial connection on the Arduino. A general purpose software serial tutorial can be found [here](http://www.arduino.cc/en/Tutorial/SoftwareSerial).

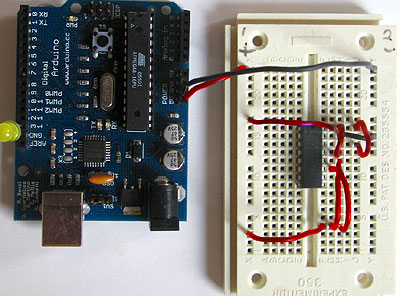
Materials needed:

* Computer with a terminal program installed (ie. HyperTerminal or RealTerm on the PC, Zterm on Mac)
* Serial-Breadboard cable
* MAX3323 chip (or similar)
* 4 1uf capacitors
* Solderless breadboard
* Hookup wire
* Arduino Microcontroller Module
* Light emitting Diode (LED) - optional, for debugging

## Prepare the breadboard

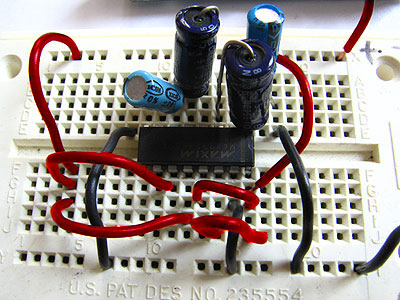


Insert the MAX3323 chip in the breadboard. Connect 5V power and ground from the breadboard to 5V power and ground from the microcontroller. Connect pin 15 on the MAX233 chip to ground and pins 16 and 14 - 11 to 5V. If you are using an LED connect it between pin 13 and ground.



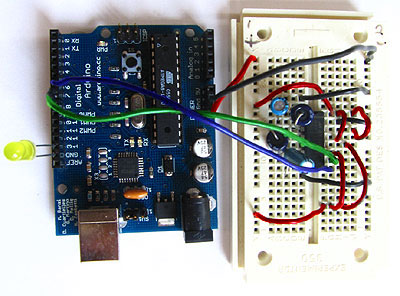
+5v wires are red, GND wires are black

Connect a 1uF capacitor across pins 1 and 3, another across pins 4 and 5, another between pin 2 and ground, and the last between pin 6 and ground. If you are using polarized capacitors make sure the negative pins connect to the negative sides (pins 3 and 5 and ground).



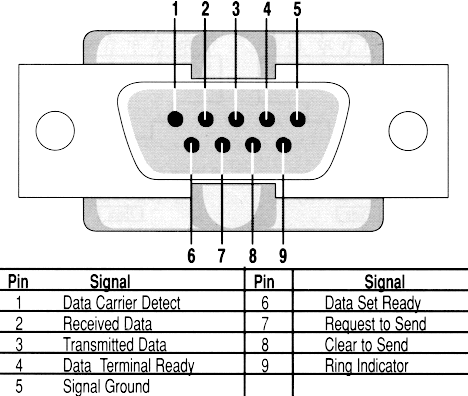
+5v wires are red, GND wires are black

Determine which Arduino pins you want to use for your transmit (TX) and recieve (RX) lines. In this tutorial we will be using Arduino pin 6 for receiving and pin 7 for transmitting. Connect your TX pin (7) to MAX3323 pin 10 (T1IN). Connect your RX pin (6) to MAX3323 pin 9 (R1OUT).



TX wire Green, RX wire Blue, +5v wires are red, GND wires are black

## Cables



(*DB9* Serial Connector Pin Diagram)

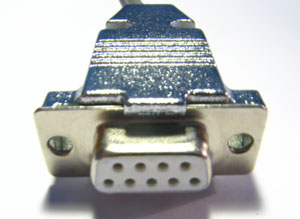
If you do not have one already, you need to make a cable to connect from the serial port (or USB-serial adapter) on your computer and the breadboard. To do this, pick up a female DB9 connector from radioshack. Pick three different colors of wire, one for TX, one for RX, and one for ground. Solder your TX wire to pin 2 of the DB9 connector, RX wire to pin 3 and Ground to pin 5.



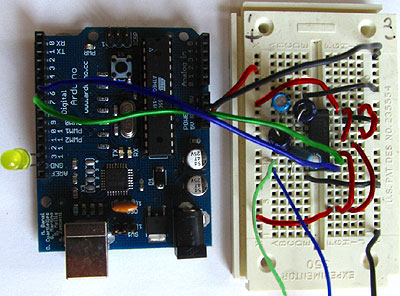
Connect pins 1 and 6 to pin 4 and pin 7 to pin 8. Heatshrink the wire connections to avoid accidental shorts.



Enclose the connector in a backshell to further protect the signal and enable easy unplugging from your serial port.



Connect the TX line from your computer to pin 8 (R1IN) on the MAX233 and the RX line to pin 7 (T1OUT). Connect the ground line from your computer to ground on the breadboard.



TX wires Green, RX wires Blue, +5v wires are red, GND wires are black

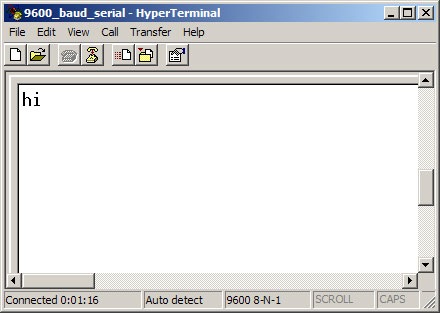
## Program the Arduino

Now we will write the code to enable serial data communication. This program will simply wait for a character to arrive in the serial recieving port and then spit it back out in uppercase out the transmit port. This is a good general purpose serial debugging program and you should be able to extrapolate from this example to cover all your basic serial needs. Upload the following code into the Arduino microcontroller module:

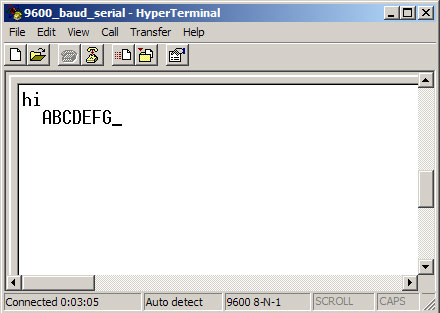
//Created August 23 2006  
//Heather Dewey-Hagborg  
//http://www.arduino.cc  
  
#include <ctype.h>  
  
#define bit9600Delay 100    
#define halfBit9600Delay 50  
#define bit4800Delay 188   
#define halfBit4800Delay 94   
  
byte rx = 6;  
byte tx = 7;  
byte SWval;  
  
void setup() {  
  pinMode(rx,INPUT);  
  pinMode(tx,OUTPUT);  
  digitalWrite(tx,HIGH);  
  delay(2);  
  digitalWrite(13,HIGH); //turn on debugging LED  
  SWprint('h');  //debugging hello  
  SWprint('i');  
  SWprint(10); //carriage return  
}  
  
void SWprint(int data)  
{  
  byte mask;  
  //startbit  
  digitalWrite(tx,LOW);  
  delayMicroseconds(bit9600Delay);  
  for (mask = 0x01; mask>0; mask <<= 1) {  
    if (data & mask){ // choose bit  
     digitalWrite(tx,HIGH); // send 1  
    }  
    else{  
     digitalWrite(tx,LOW); // send 0  
    }  
    delayMicroseconds(bit9600Delay);  
  }  
  //stop bit  
  digitalWrite(tx, HIGH);  
  delayMicroseconds(bit9600Delay);  
}  
  
int SWread()  
{  
  byte val = 0;  
  while (digitalRead(rx));  
  //wait for start bit  
  if (digitalRead(rx) == LOW) {  
    delayMicroseconds(halfBit9600Delay);  
    for (int offset = 0; offset < 8; offset++) {  
     delayMicroseconds(bit9600Delay);  
     val |= digitalRead(rx) << offset;  
    }  
    //wait for stop bit + extra  
    delayMicroseconds(bit9600Delay);   
    delayMicroseconds(bit9600Delay);  
    return val;  
  }  
}  
  
void loop()  
{  
    SWval = SWread();   
    SWprint(toupper(SWval));  
}

[[Get Code]](https://www.arduino.cc/en/Tutorial/ArduinoSoftwareRS232?action=sourceblock&num=1)

Open up your serial terminal program and set it to 9600 baud, 8 data bits, 1 stop bit, no parity, no hardware flow control. Press the reset button on the arduino board. The word "hi" should appear in the terminal window followed by an advancement to the next line. Here is a shot of what it should look like in Hyperterminal, the free pre-installed windows terminal application.



Now, try typing a lowercase character into the terminal window. You should see the letter you typed return to you in uppercase.



If this works, congratulations! Your serial connection is working as planned. You can now use your new serial/computer connection to print debugging statements from your code, and to send commands to your microcontroller.

code and tutorial by Heather Dewey-Hagborg, photos by Thomas Dexter

# Shield Aliexpress MoreSuns 15€

<https://www.aliexpress.com/store/1548016>

<https://fr.aliexpress.com/item/Fast-Free-Ship-RS232-Shield-V2-for-pcDuino-Arduino-RS232-serial-interface-communication-module/32664256235.html?spm=2114.06010108.3.1.hV3d1z&ws_ab_test=searchweb0_0,searchweb201602_3_10065_10068_10501_10503_10000032_119_10000025_10000029_430_10000028_10060_10062_10056_10055_10000062_10054_10059_10099_10000022_10000012_10103_10000015_10102_10096_10000018_10000019_10000056_10000059_10052_10053_10107_10050_10106_10051_10000053_10000007_10000050_10084_10083_10118_10000047_10080_10082_10081_10110_10111_10112_10113_10114_10115_10033_10000041_10000044_10078_10079_10000038_429_10073_10000035_10121-10503_10501,searchweb201603_10,afswitch_1,single_sort_1_default&btsid=edc0cfd6-673f-46c8-a22c-4d3647bed128>



