de zéros, le bit de stop serait aussi à zéro et cette condition provoquerait une erreur. C'est simple mais efficace. Du moins tant qu'un des appareils n'utilise pas une version du protocole avec 2 bits de stop, ou même un et demi. Tant que le récepteur accepte autant ou moins de bits de stop que l'émetteur, cela ne fait que ralentir le débit. Mais je ne crois pas avoir jamais vu plus d'un bit de stop dans un système réel.

Chaque trame peut contenir un bit optionnel de parité (avant le(s) bit(s) de stop), de polarité positive ou négative. Cela peut être intéressant sur des liaisons de faible qualité (comme par téléphone ou par radio) mais on ne l'utilise pas souvent pour un petit montage sur un coin de bureau. On préfère alors établir des protocoles de plus haut niveau pour assurer l'intégrité des données, avec des sommes de contrôle ou des CRC. Le plus souvent, c'est même ignoré par simplicité (Fig. 1).

Ainsi, une liaison typique aura le format suivant :

- 1 bit START
- 8 bits de données
- 1 bit de STOP.

Le débit effectif de transmission sera alors F/10, il suffit d'enlever le dernier zéro du nombre de bauds pour obtenir le nombre d'octets par seconde. Par exemple : 115200 bauds devrait donner 11520 octets par seconde. En théorie.

## 1.3 Manipulations sous Linux

Toutes les manipulations de cet article ont été effectuées sous Fedora Linux :

```
[yg@Pavilion]$ uname -a
Linux Pavilion.ygdes 3.12.8-300.fc20.i686
#1 SMP Thu Jan 16 01:28:49 UTC 2014 i686
i686 i386 GNU/Linux
```

Lorsque vous branchez un appareil sur un port USB, comme nos dongles série, le noyau devrait le détecter rapidement et vous pouvez vous en assurer en consultant la liste des messages du kernel :

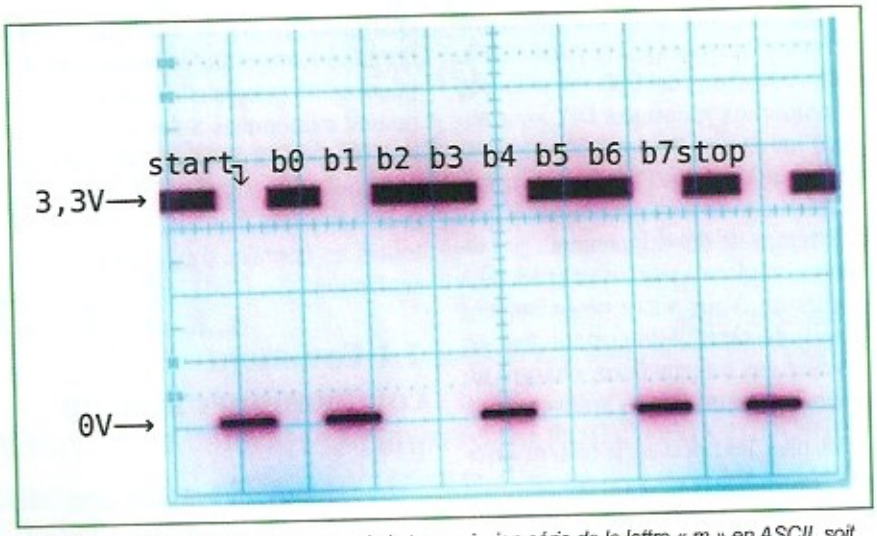


*Figure 1 : Affichage à l'oscilloscope de la transmission série de la lettre « m » en ASCII, soit l'octet 109 ou 01101101 en binaire. Le signal numérique commence par le bit de start à zéro, suivi par les bits de données (le bit de poids faible en premier) et un éventuel bit de parité. La fin est indiquée par le bit de STOP à 1, et le mot suivant peut suivre immédiatement.*

```
[yg@Pavilion] dmesg
[  367.223939] usb 5-2: new full-speed USB device number 3 using xhci_hcd
[  367.290752] usb 5-2: New USB device found, idVendor=0403, idProduct=6001
[  367.290768] usb 5-2: New USB device strings: Mfr=1, Product=2, SerialNumber=3
[  367.290778] usb 5-2: Product: FT232R USB UART
[  367.290786] usb 5-2: Manufacturer: FTDI
[  367.290794] usb 5-2: SerialNumber: A603VBKV
[  367.304780] ftdi_sio 5-2:1.0: FTDI USB Serial Device converter detected
[  367.304875] usb 5-2: Detected FT232RL
[  367.304882] usb 5-2: Number of endpoints 2
[  367.304889] usb 5-2: Endpoint 1 MaxPacketSize 64
[  367.304896] usb 5-2: Endpoint 2 MaxPacketSize 64
[  367.304902] usb 5-2: Setting MaxPacketSize 64
[  367.310093] usb 5-2: FTDI USB Serial Device converter now attached to ttyUSB0
```

La dernière ligne indique le nom qui a été assigné à ce dongle USB. C'est un périphérique qui peut normalement se faire passer pour un télétype (**tty**) et donc on le retrouve parmi les autres télétypes dans le répertoire **/dev** :

```
[yg@Pavilion]$ l /dev/ttyU*
crw-rw----. 1 root dialout 188, 0 2 avril 11:03 /dev/ttyUSB0
```

Nous allons garder son chemin pour plus tard, en le mettant dans une variable du terminal.

```
[yg@Pavilion]$ export TTY=/dev/ttyUSB0
```

Ainsi, il n'est plus nécesaire de taper tout le chemin à chaque commande, on écrit juste **$TTY** et Bash fait la substitution.

On voit que ce périphérique appartient à l'utilisateur **root**, mais aussi au groupe **dialout**. Pour la suite des manipulations, nous serons en utilisateur normal (ici **yg**) et il faut s'assurer que cet utilisateur est aussi présent dans le groupe **dialout**. On peut le vérifier en consultant le fichier **/etc/group** :

```
[yg@acer DCIM]$ grep yg /etc/group
...
dialout:x:18:root,yg
...
```

S'il n'est pas présent, ajoutez-le, le reste pourra fonctionner.