Communication avec un système embarqué : l'exemple du port série

Communication avec un système embarqué : l'exemple du port série

La liaison série fait le lien entre le microcontrolleur (la cible) et le PC. Elle assure deux rôles :

- La liaison pour le développement : la commande stampbc prend a sa charge la transmission du code objet (les token) sur la cible. Sitôt téléchargé, le code est excuté sur la cible
- La liaison pour les communications. Lorsqu'un programme est chargé sur le basic stamp, plusieurs instructions permettent de gérer les entrées et sorties :
  - DEBUG permet a la cible d'écrire des octets sur le port série.
  - SERIN permet au micro-contrôleur de lire et de formater des octets sur le port série.
- Pour l'instant, nous utilisons le programme minicom pour envoyer des octets depuis le PC.

Nous allons maintenant automatiser cette dernière partie de l'application afin de pouvoir développer des programmes distants qui pilotent le duspositif embarqué via liaison série.

Communication avec un système embarqué : l'exemple du port série

• La communication série est l'action d'envoyer des données bits par bits, séquentiellement, sur un canal de communication ou un bus informatique.

• Elle est a mettre en opposition avec la communication parallèle ou tous les bits sont envoyés simultanément.

• La communication série est facile a synchroniser, fiable sur les longues distances, et utilisée dans la plupart des communications modernes :

- USB
- FireWire
- SCSI
- PCI
- SATA
- RS232

La Liaison serie RS232

• RS-232 est une norme standardisant un port de communication de type série. Disponible sur presque tous les PC jusqu'au milieu des années 2000, il est communément appelé le « port série

- •COM1, COM2, sous windows
- •/dev/ttyS0 /dev/ttyS1 sous linux
- •/dev/cua0 /dev/cua1 sous solaris

• il est de plus en plus remplacé par le port USB, mais il est toutjours possible de l'utiliser via un adaptateur "USB-Serie".

•Le port RS-232 est fréquemment utilisé dans l'industrie pour connecter différents appareils électroniques (automate, appareil de mesure, etc..).

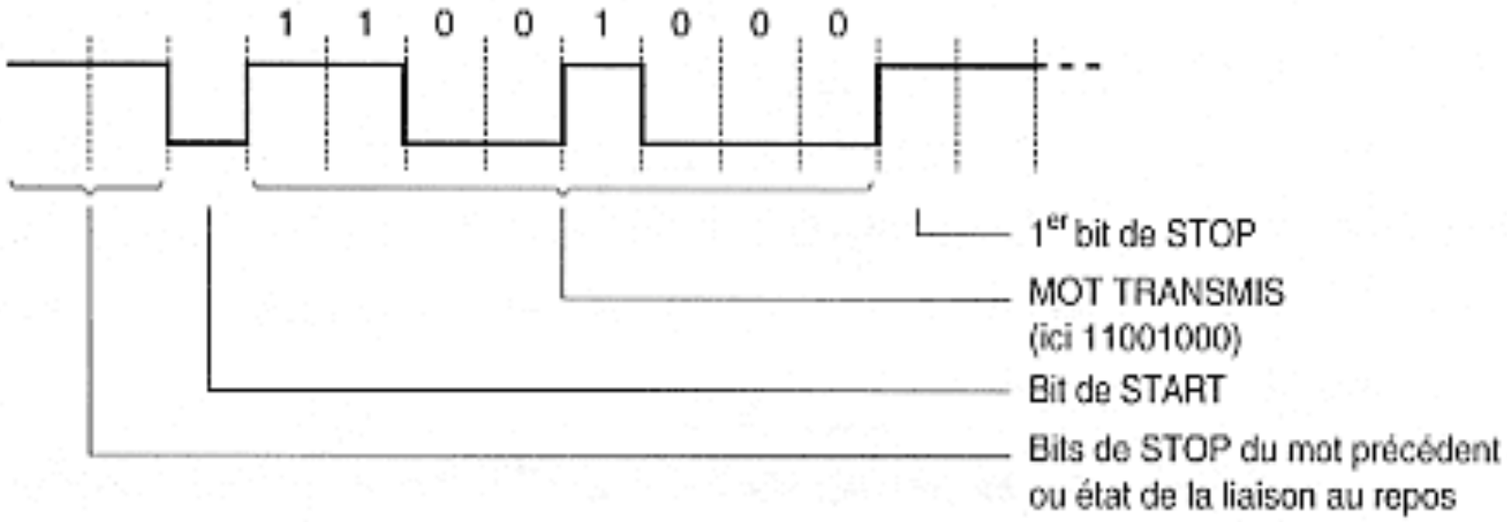
La Liaison serie RS232

- L'interface série est une interface asynchrone, ce qui signifie que le signal de cette interface n'est pas synchronisé avec celui d'un bus quelconque.
- les bits des données sont envoyés les uns après les autres.
- Un caractère = une valeur comprise entre 0 et 255 = 1 octect = 8 bits.
- Chaque caractère est délimité par
  - un signal de début : un bit à 0 STARTBIT
  - par signal de fin standard qui peut correspondre à un ou deux bits de fin, cela permet d'indiquer que le caractère a été envoyé. les STOPBITS.
- L'interface asynchrone est orienté caractère, c'est à dire que l'on doit utiliser les signaux de début et de fin pour indentifier un caractère. L'inconvénient de ce processus c'est qu'il augmente la duré des transferts de presque 25 %. En effet pour chaque ligne de 8 bits il faut au minimum 2 bits.

- Le terme "série" viens juste du fait que les bits sont envoyés les uns après les autres sur un seul fil pour l'émission et un autre fil pour la reception, comme pour le téléphone. Il existe de nombreuse carte d'extension permettant d'avoir plusieurs ports séries ou port parallèle.

La Liaison serie RS232

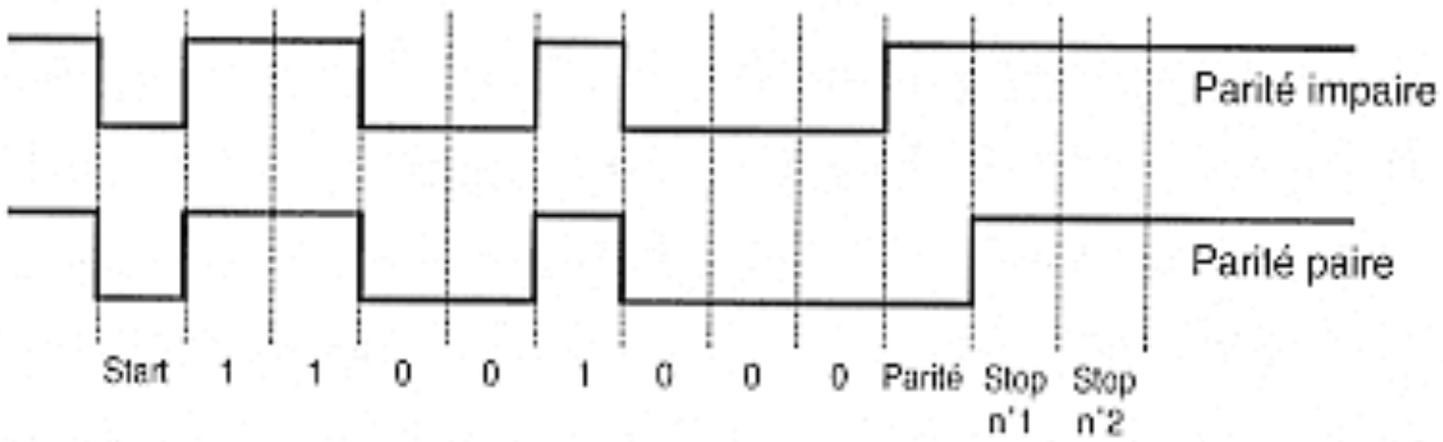
- Exemple : transmission de l'octet 11001000



- transmission d'un octet = 10 bits (8+2)
- vitesse de transmission exprimée en Bauds, i.e : Bits par secondes , il suffit donc de diviser par 10 la vitesse pour obtenir le débit en otects/s. 9600 Bauds = 960 octest/s < 1ko/s

La Liaison serie RS232

•Detection d'erreurs :



•On compte le nombre de bits a 1 de l'octet (on exclue le startbit et les stopbits).
•nombre pair de bits : bit de parité a 1
•nombre impair de bits : parité a 0
•permet lors de la réception de détecter si un bit a été altéré lors de la transmission
• Attention : si deux bits sont altérés pas de détection d'erreur

Programmation en C sous Linux

• Le port serie est géré sous unix comme un flux classique

•on utilise donc les instructions open(), close(), read() et write() tel qu'on le ferait pour effectuer la gestion de bas niveau d'un fichier.

•int open(char*, int flags) avec flags valant :

- O_RDWR "read ans write"
- O_NOCTTY "no control terminal" -> ne nous concerne pas
- O_NDELAY "no delay" -> ne nous concerne pas

•la valeur de retour est un entier "file descriptor" utilisé par le processus pour identifier les flux d'E/S

•ex :

```
int fd;
fd = open("/dev/ttyS0",O_RDWR) ;
```

•La fermeture se fait avec close()

```
close(fd);
```

Programmation en C sous Linux

- Ecriture : on utilise write() issu de la libc

```
#include <unistd.h>
...
ssize_t write(int fd, const void*buf,size_t count);
```

- write permet d'ecrire *count* octets sur le flux referencé par *fd*. Les octets à écrire sont dans *buf*
- write retourne le nombre d'octects écrits si succès
  - -1 si echec

•Enfin, pour se confirmer a une gestion en mode asynchrone, on utilise les routines *tcflush* et *tcdrain* sur le descripteur de fichier associé.

•tcflush(int fd, TCIOFLUSH) empêche la transmission des données "résiduelles" sur le port associé a fd. tcflush() est à appeler AVANT le write().

•tcdrain(int fd) attend que tout soit transmit sur fd. à apeler APRES le write().

Programmation en C sous Linux

- Ce qui nous donne :

```
#include <unistd.h>
#include <termios.h>
...


tcflush(....,TCIOFLUSH)
write(..., .... , .....)
tcdrain(....);
```

Programmation en C sous Linux

- Ecriture : on utilise read() issu de la libc

```
#include <unistd.h>
...
ssize_t read(int fd, const void*buf,size_t count);
```

- read permet de lire *count* octets sur le flux referencé par *fd*. Les octets lu sont placés dans *buf* (attention, buff doit alloué au préalable).

- read retourne le nombre d'octets lus si succès
  - -1 si echec

-

Programmation en C sous Linux