Les communications série

Plan

- Principe de la communication série
- Le type de bus séries
- Le bus rs232
- Le cas de l'ATmega
- L'API Arduino

Principe de la communication série

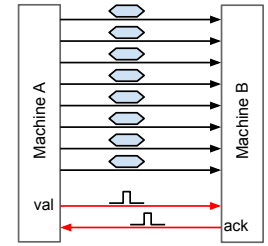
Pour "transmettre" des données entre deux ordinateurs (ou périphérique) par des fils, on dispose de 2 modes:

- mode parallèle
  - tous les bits de données d'un même caractère sont envoyés en même temps
- mode série
  - les bits de données sont envoyés l'un après l'autre

Actuellement :
le mode série est le plus utilisé pour la communication avec les E/S, le mode parallèle est confiné sur le circuit imprimé.

Différence : Série / Parallèle

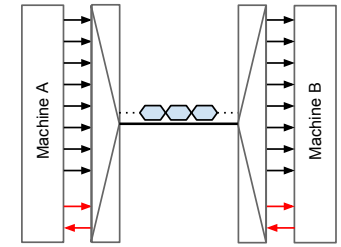
Transmission parallèle



Les bits sont envoyés en parallèle les caractères sont envoyés en série.

a priori plus simple, mais tous les signaux doivent arriver en même temps, c'est donc cher et difficile pour les grandes distances à haute fréquence.

Transmission série

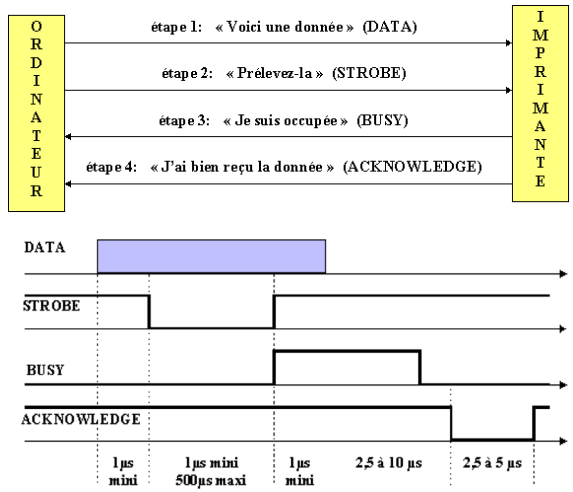


Les bits de chaque caractères sont envoyés en série.

nécessite un sérialiseur/désérialiseur, mais tous les bits arrivent dans l'ordre cela semble plus long, mais on peut augmenter la fréquence.

Port Parallèle EPP (chronogramme)

géré par le pic16f877 avec les ports D et E



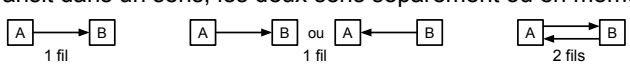
dessins: http://sitelec.org/cours/abati/centronic.htm

Les bus séries

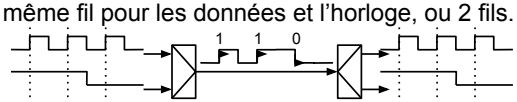
- Il existe une très grande variété car aucun bus n'est universel.
- Il faut choisir en fonction :
  - de la distance entre les équipements (cm, m, km)
  - du nombre d'équipements à relier
  - du débit de données (contrôle ou data)
  - de la consommation autorisée (pile/secteur)
  - de la fiabilité nécessaire (bruit)
  - de la maintenabilité (hotplug)
  - des contraintes temporelles (QoS)
  - du catalogue disponible
  - du coût (en générale, la bonne solution est trop chère)
  - etc...
- Chaque bus existe en plusieurs versions, en général compatibles entre elles mais avec un rapport débit / distance différent.
- Le microcontrôleur propose RS232, I2C et SPI natifs.
- La carte ATmega propose RS232(via USB), RS232, I2C, SPI

Différences technologiques

- Half duplex ou Full duplex
  - transit dans un sens, les deux sens séparément ou en même temps.

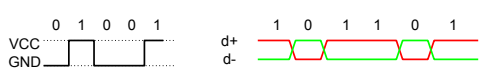


- Horloge et Data mélangés ou séparés
  - 1 même fil pour les données et l'horloge, ou 2 fils.

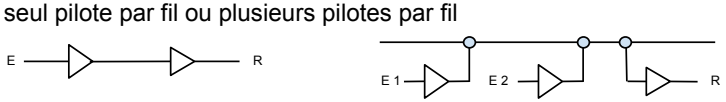


(ici codage Manchester)

- Signal différentiel ou simple
  - une donnée utilise 2 fils de valeurs opposées ou 1 seul valant 0 ou 1.



- Signal point-à-point ou bussé
  - 1 seul pilote par fil ou plusieurs pilotes par fil



Différences technologiques

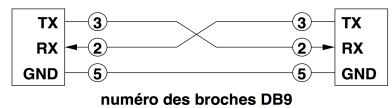
- RS232
  - full duplex,
  - pas de signal d'horloge
  - 2 data (3 fils mininum : RX, TX, GND),
  - signal non différentiel
  - point à point
  - de 75 bits/s à 115 kb/s
- SPI
  - full duplex,
  - horloge et data séparés (4 fils mininum : SCLK, MISO, MOSI, SS)
  - signal non différentiel
  - point à point
  - adhoc jusqu'à 100Mb/s
- I2C ls / hs
  - half duplex
  - horloge et data séparé (3 fils : SDA, SCL, GND),
  - signal non différentiel,
  - bussé
  - 100 kb/s à 3.4 Mb/s
- USB 1 / 2 / 3
  - half duplex
  - horloge et data mélangé (4 fils : VBUS, D+, D-, GND),
  - signal différentiel
  - point-à-point
  - de 1.5 Mb/s à 5 Gb/s

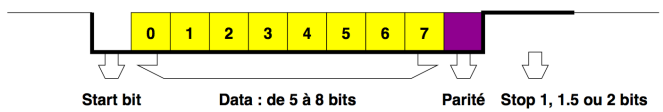
La vitesse est gagnée au prix de la complexité des protocoles et du matériel

RS232

- Protocole faible débit, simple et très diffusé, datant des années 60
- Pas d'horloge: l'émetteur et le récepteur s'entendent avant.
- Protocole *handshake* optionnel : CTS, RTS, ...
- Liaison point-à-point, pas de notion d'adresse.
- Trame de données de 5 à 8 bits avec parité.
- La parité est optionnelle:
  parité paire: le nombre de 1 de la donnée et du bit de parité doit être pair
  parité impaire: c'est le contraire

- RS232 prévoit plusieurs types de cablages:
  le cablage null-modem définit la communication entre 2 terminals
- au minimum 3 fils : TX, RX et GND
- on peut avoir besoin d'un convertisseur de niveaux électriques :
  - 0 logique : +8 à +12V
  - 1 logique : -8 à -12V



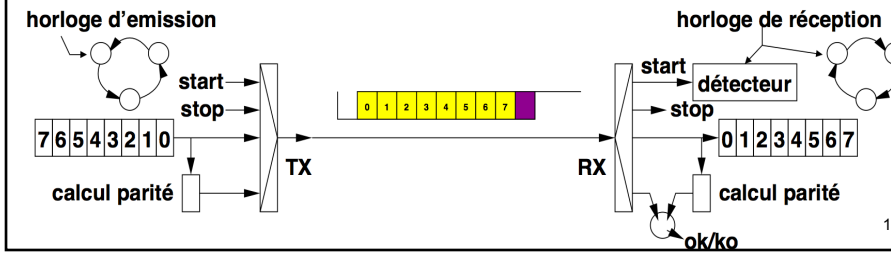


RS232 Schéma de principe

Émetteur — La trame est produite par un automate qui vide un registre à décalage

Récepteur — La trame est lue par un automate qui remplit un registre à décalage

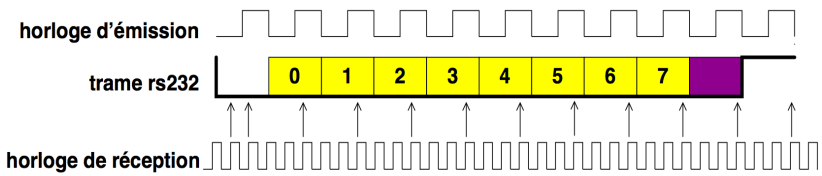
Parité — Un bit supplémentaire qui signe la donnée
parité paire : le nombre de bit à 1 de la donnée est rendu pair grâce au bit de parité.
parité impaire : c'est le contraire



RS232 Synchro émetteur/récepteur

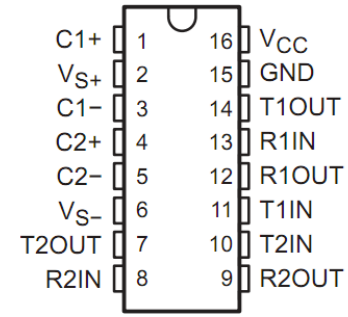
- L'émetteur transmet à une fréquence standardisée (1200, 2400, 4800,...)
- Le **récepteur connait cette fréquence** et **sur-échantillonne** pour répérer **start**
  - Si la fréquence d'échantillonnage est 3 fois la fréquence d'émission
  - Lorsqu'on lit 0, on est au dans le premier tiers du **start**
  - L'échantillon suivant est dans le tiers du milieu
  - Les bits de la trame sont alors lu toutes les 3 périodes

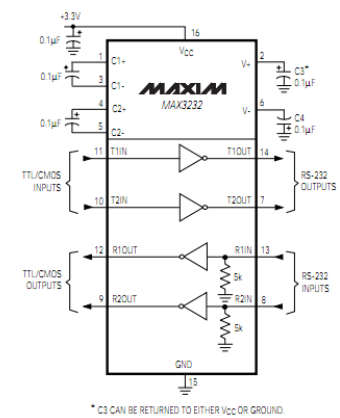
- Le récepteur a une petite marge d'erreur possible sur la fréquence.



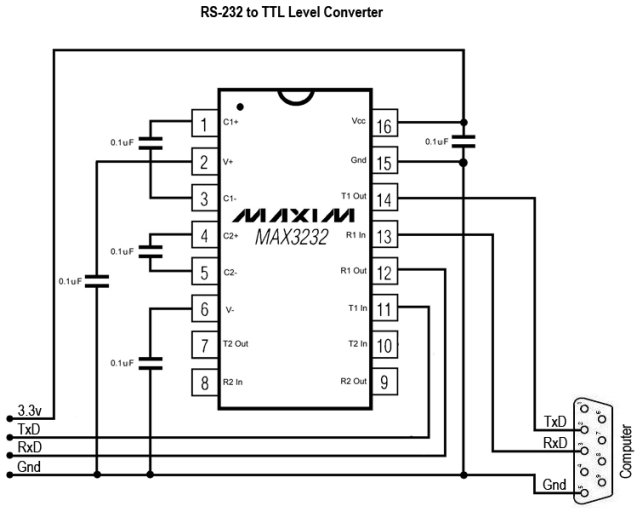
RS232 adaptation de niveau 1/2

Pour le microcontroleur 1 : 5V et 0 : 0V
Sur les lignes 1 : -12V et 0 : +12V



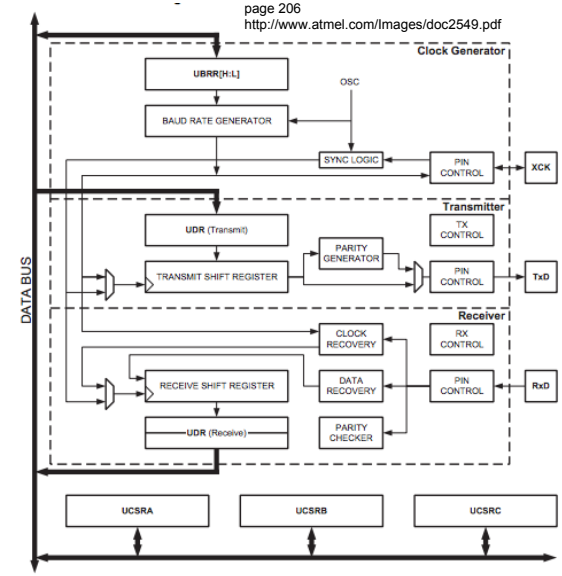


RS232 adaptation de niveau 1/2

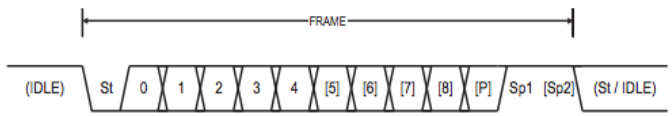


ATmega USART : Block diagram

page 206
http://www.atmel.com/Images/doc2549.pdf



ATmega USART : Frame TX



St	Start bit, always low.
(n)	Data bits (0 to 8).
P	Parity bit. Can be odd or even.
Sp	Stop bit, always high.
IDLE	No transfers on the communication line (RxDn or TxDn). An IDLE line must be high.

page 210
http://www.atmel.com/Images/doc2549.pdf

ATmega USART : Registers

Bit	7	6	5	4	3	2	1	0	
	RXB[7:0]								UDRn (Read)
	TXB[7:0]								UDRn (Write)
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
	RXCIEn	TXCIEn	UDRIEn	RXENn	TXENn	UCSZn2	RXB8n	TXB8n	UCSRnB
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Bit	7	6	5	4	3	2	1	0	
	UMSELn1	UMSELn0	UPMn1	UPMn0	USBSn	UCSZn1	UCSZn0	UCPOLn	UCSRnC
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	1	1	0	

page 222
http://www.atmel.com/Images/doc2549.pdf

ATmega USART : Registers

Bit	7	6	5	4	3	2	1	0	
	RXCn	TXCn	UDREn	FEn	DORn	UPEn	U2Xn	MPCMn	UCSRnA
Read/Write	R	R/W	R	R	R	R	R/W	R/W	
Initial Value	0	0	1	0	0	0	0	0	

- **Bit 7 – RXCn: USART Receive Complete**

This flag bit is set when there are unread data in the receive buffer and cleared when the receive buffer is empty (that is, does not contain any unread data). If the Receiver is disabled, the receive buffer will be flushed and consequently the RXCn bit will become zero. The RXCn Flag can be used to generate a Receive Complete interrupt (see description of the RXCIEn bit).

- **Bit 6 – TXCn: USART Transmit Complete**

This flag bit is set when the entire frame in the Transmit Shift Register has been shifted out and there are no new data currently present in the transmit buffer (UDRn). The TXCn Flag bit is automatically cleared when a transmit complete interrupt is executed, or it can be cleared by writing a one to its bit location. The TXCn Flag can generate a Transmit Complete interrupt (see description of the TXCIEn bit).

- **Bit 5 – UDREn: USART Data Register Empty**

The UDREn Flag indicates if the transmit buffer (UDRn) is ready to receive new data. If UDREn is one, the buffer is empty, and therefore ready to be written. The UDREn Flag can generate a Data Register Empty interrupt (see description of the UDRIEn bit).

UDREn is set after a reset to indicate that the Transmitter is ready.

- **Bit 4 – FEn: Frame Error**

This bit is set if the next character in the receive buffer had a Frame Error when received, that is, when the first stop bit of the next character in the receive buffer is zero. This bit is valid until the receive buffer (UDRn) is read. The FEn bit is zero when the stop bit of received data is one. Always set this bit to zero when writing to UCSRnA.

page 223
http://www.atmel.com/Images/doc2549.pdf

RS232 Arduino

Arduino Mega 4 ports série : serial, serial1, serial2, serial3

- serial.begin(speed[,config]) → *explication après*
- serial.end() → ferme la connexion
- serial.available() → rend le nombre de char en attente (jusqu'à 64)
- serial.read() → retire et rend le caractère en attente ou -1
- serial.peek() → rend le caractère en attente ou -1
- serial.flush() → vide le buffer en sorties
- serial.print(var[,base|frac]) → imprime la variable (int, char, string)
  base : BIN, OCT, DEC, HEX
  frac : nombre de chiffres après la virgule
- serial.println() → comme print + '\n' (ascii 10)
- serial.write(val) → val entier ou string rend le nombre de char écrits
- serial.write(buf, len) → buf pointeur, len size
- serialEvent() → appelée quand il y a une donnée présente

RS232 Arduino : begin

speed: in bits per second (baud) - long
300, 600, 1200, 2400, 4800, 9600, 14400, 19200, 28800, 38400, 57600, or 115200.

config: sets data, parity, and stop bits. Valid values are :

SERIAL_5N1	SERIAL_5E2
SERIAL_6N1	SERIAL_6E2
SERIAL_7N1	SERIAL_7E2
SERIAL_8N1 (the default)	SERIAL_8E2
SERIAL_5N2	SERIAL_5O1
SERIAL_6N2	SERIAL_6O1
SERIAL_7N2	SERIAL_7O1
SERIAL_8N2	SERIAL_8O1
SERIAL_5E1	SERIAL_5O2
SERIAL_6E1	SERIAL_6O2
SERIAL_7E1	SERIAL_7O2
SERIAL_8E1	SERIAL_8O2