Ce document se retrouve à la fois dans le repo public d'enseignement de dccote et sur le site Web de DCC Lab. Le format Markdown est celui supporté par Typora.io. Une version PDF est disponible.

DAQ: Entrées-sorties numériques avec le UM232R

Le monde numérique

En classe comme dans la vie, la question la plus simple se répond par oui ou par non. La première tâche la plus simple que l'on peut faire avec un appareil d'acquisition ou de contrôle sera donc de lui faire dire Oui ou Non. On peut s'imaginer plusieurs interprétations de réponses binaire exclusives: vrai ou faux, 1 ou 0, "Je suis prêt" ou "Je ne suis pas prêt".

Les pionniers de l'informatique ont dû concevoir un système qui permettrait de représenter les valeurs sans ambiguïtés pour les stocker et les manipuler avec un circuit logique. Pour bien des raisons qui viennent à la fois de l'ingéniosité (il fallait y penser) et de l'opportunité technologique (le transistor semionducteur et son potentiel de fabrication et miniaturisation), il est devenu apparent qu'un système basé sur une représentation binaire des nombres et de leur transformation par des circuits logiques serait idéal. Les premiers circuits électroniques de type TTL (ou *Transistor-Transistor-Logic*) dans les années 60 sont devenus les premiers circuits logiques, c'est-à-dire des circuits traitant des données d'entrée en représentation binaire pour produire de l'information à la sortie, aussi sous forme binaire. L'ensemble des circuits logiques permet d'implémenter toutes les opérations nécessaires à un calcul. Nous reviendrons plus tard au très passionnant côté *hardware* de la naissance de l'informatique, mais pour l'instant il suffira de savoir que la représentation réelle des nombres et des instructions dans les chips est sous forme binaire: une tension est à 5V est VRAI (ou 1) et une tension à 0V est FAUX (ou 0)[1].

Bits et octets

Un **bit** a 2 valeurs possibles, 0 ou 1. C'est la plus petite quantité d'information que nous puissions avoir sur quelque chose [2]. Une collection ordonnée de 8 bits est un octet. Chaque bit pouvant avoir deux valeurs, un octet peut donc avoir $2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 \times 2 = 2^8 = 256$ valeurs différentes. Par commodité, nous représentons cela sous forme de nombre en base 2 (binaire). La séquence suivante de 8 bits 0 0 1 0 0 1 0 1 peut être prise pour avoir une valeur de:

$$0 \times 2^7 + 0 \times 2^6 + 1 \times 2^5 + 0 \times 2^4 + 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 37 \text{ dec.} \quad (1)$$

Comprenons bien que l'interprétation d'un octet en nombre entier allant de 0 à 255 n'est rien de plus qu'une *interprétation*: on pourrait aussi interpréter ce nom autrement. Par exemple, on pourrait déterminer que si le dernier bit est 1.

Dans de nombreux textes, une valeur binaire est notée %00100101 (Python: `0b00100101`) pour la différencier de la valeur décimale 100101, qui a pour valeur «cent mille cent un». Cependant, la notation binaire n'est pas pratique: elle prend beaucoup de place lorsqu'elle est écrite et il est difficile d'évaluer rapidement une valeur pour le non-expert. Par conséquent, une notation raccourci est l'hexadécimale ou la notation en base 16. Les "chiffres" hexadécimaux sont {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}. Le nombre ci-dessus peut donc être ré-écrit sous la forme %10100101 = $10 \times 16^1 + 5 \times 16^0$ ou 0xA5 en hexadécimal, le préfixe 0x (souvent seulement x) désignant hexadécimal. Évidemment, puisque la base $16 = 2^4$on voit que la notation hexadécimale permet de regrouper les bits par groupe de 4: les moins significatifs (à droite) sont représentés par le chiffre 5 = %0101 et les plus significatifs (à gauche) sont représentés par la lettre 0xA = %1010.

L'importance de la notation binaire vient du fait que tout langage de programmation que l'on utilisera pour communiquer avec notre appareil devra invariablement passer par des représentations en binaires et des octets, puisque c'est la représentation interne naturelle de l'ordinateur.

Exercices

1. Écrivez la valeur suivante en binaire, décimal et hexadécimal: a. 12 b. 0x45 c. %1001101 d. 230 e. 0xF3
2. Effectuez les opérations suivantes: a. % 00101101 +% 00100001 b. 0xF6 - 0x41 c. 0x29 - 0x56
3. Multipliez %00111001 par 2, 4 et 8 et écrivez les réponses en binaire.
4. Divisez %00111001 par 2 et écrivez la réponse en binaire.
5. Multipliez 0xF3 par 16.

Premier branchement

Le plan ici est d'obtenir un système pour expérimenter. Il y aura bien sûr des moments où l'on pourra se questionner sur le fonctionnement exact d'un ou l'autre aspect. Cependant, pour l'instant, il s'agit simplement de mettre en place un "banc d'expérimentation" pour aider à la compréhension, pour démystifier certains aspects et pour permettre l'exploration. Ainsi, on utilise le module de communication **UM232R** de FTDI (specifications) de type UART. Ce module est très intéressant pour plusieurs raisons:

1. Il se branche simplement dans le port USB de n'importe quel ordinateur
2. Il ne demande aucun branchement supplémentaire pour être au moins fonctionnel
3. Il permet de construire un systeme pour communiquer avec d'autres appareils. En fait, il permet de mettre à jour les vieux systèmes RS232 pour USB.
4. Les *drivers* (pilotes en français) de FTDI existent pour toutes les plateformes
5. Les puces USB de FTDI sont utilisées dans un très grand nombre d'appareils
6. Les librairies de FTDI sont disponibles, simples et sans bugs, et sont supportées par Python.

Pour commencer, il faut se procurer le fameux module UM232R. La façon la plus simple, venez me voir à mon bureau POP-2141 et demandez-le moi, j'en ai quelques uns. Sinon, on peut les commander pour 28$ chez Digi-Key , numéro de pièce Digi-Key 768-1019-ND, ou FTDI UM232R. Idéalement, un petit kit de breadboard comme le 438-1047-ND serait acheté en même temps, mais n'importe quel breadboard fait l'affaire. On remarque:

1. Il y a 24 lignes (ou *pins* en anglais), on commence a compter en haut à gauche en tournant dans le sens trigonométrique.
2. Plusieurs lignes sont identifiées RST, GND, VCC, VIO mais d'autres sont identifiées avec des termes génériques DB0, DB1, etc...
3. Le manuel indique qu'il y a une ligne nommée TXD (*transmission data*) et une nommée RXD (*receiving data*)
4. Il y a une ligne RESET. Le manuel indique clairement que mettre cette ligne à 0V forcera un "reset" de la puce.
5. Il y a deux lignes configurées par défaut `CB0` et `CB1` pour servir d'indicateurs de transmission et de réception avec une DEL.
6. Un groupe de lignes est identifié comme faisant du *hardware handshake*: DTR, RTS, DSR, DTS
7. Bien que nous soyions en 2018, il reste des lignes identifiées comme: Ring Indicator (RI, ligne 6) Data Carrier Detect Control Input (DCDC, ligne de tonalité d'un modem)

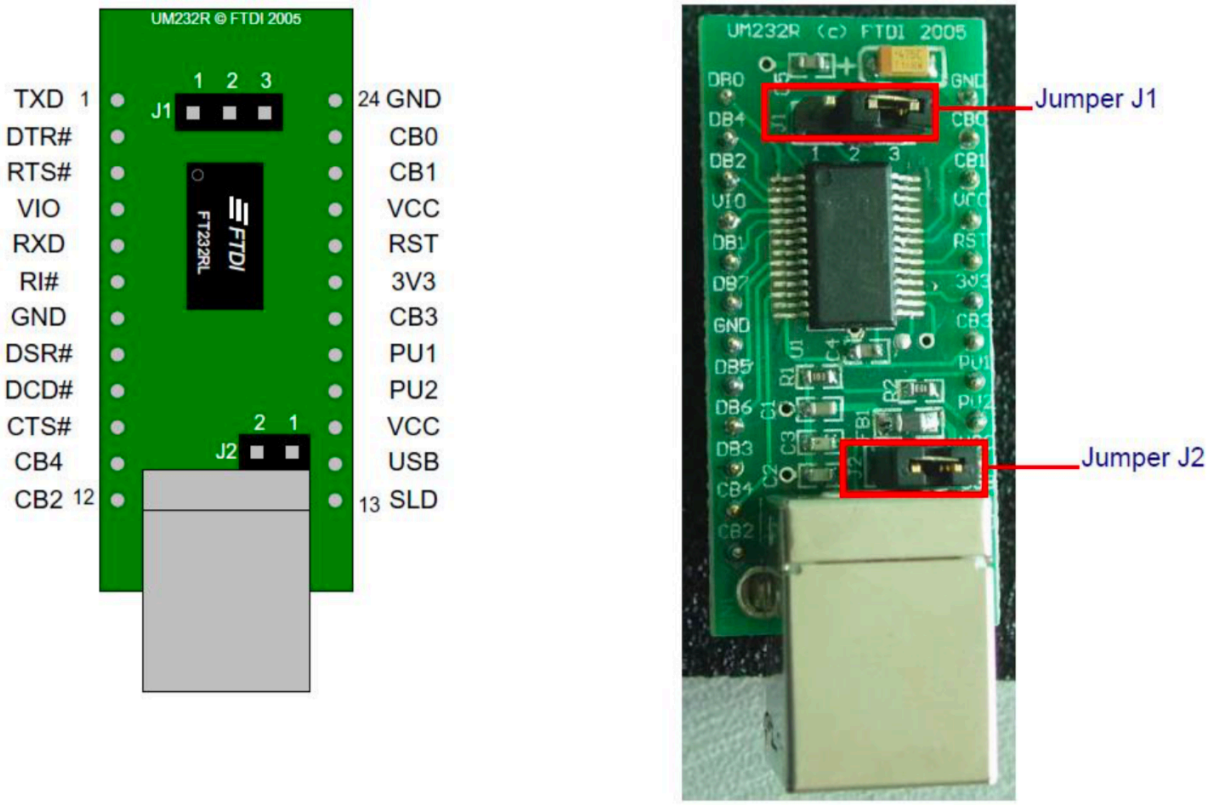


Figure 4.1 Module Pin Out and Jumper Locations

Pour l'instant, mes instructions seront pour macOS/Linux car c'est ce que j'utilise. D'autres instructions devraient suivre pour les autres plateformes, même si la tâche me donne plutôt le goût d'aller chez le dentiste me faire enlever la totalité des dents d'en bas. Sans anesthésie.

Expérience 1: brancher

Si vous branchez un câble USB dans le module, la magie du standard USB[3] devrait faire son oeuvre si le driver FTDI est installé (souvent standard, sinon obtenez et installez-le). Votre "puce" apparaîtra comme un port série ou *Virtual Comm Port* (VCP). Les systèmes qui utilisent le standard POSIX (macOS[4] et Linux) le montreront dans `/dev/` :

```
Nestor:anaconda2 dccote$ ls -l /dev/*usbserial*
crw-rw-rw-  1 root  wheel   18,  51 14 Oct 22:42 /dev/cu.usbserial-FTCBGW24
crw-rw-rw-  1 root  wheel   18,  50 14 Oct 22:42 /dev/tty.usbserial-FTCBGW24
Nestor:anaconda2 dccote$
```

Le port série en POSIX apparait en deux formes pour des raisons historiques: `cu` (callout) et `tty` (teletype). Nous prendrons `cu*` pour la communication avec les appareils. Le nom du port série (ici `cu.usbserial-FTCBGW24` est programmé par FTDI directement dans la puce à la fabrication). La valeur `FTCBGW24` est un numéro de série unique à chaque puce, donc la vôtre sera différente de la mienne.

Expérience 2: débrancher

Si on débranche le câble USB, le port de communication associé au module disparaitra du système car le module n'est plus disponible. On peut faire la même chose à l'aide d'un fil en connectant la ligne `RST` au `GND` pour forcer un reset du module. Lorsqu'on le fait, on voit:

```
Nestor:anaconda2 dccote$ ls -l /dev/*usbserial*
ls: /dev/*FT*: No such file or directory
Nestor:anaconda2 dccote$
```

Avant de continuer, on enlève la connexion entre `RST` et `GND` .

Expérience 3: parler

Le langage de référence dans ces tutoriels est Python, non pas pour sa puissance, sa convivialité ou l'élégance de sa syntaxe mais bien pour son universalité. La façon la plus rapide d'être opérationnel est de télécharger Anaconda2. Vous aurez un environnement et la majorité des modules importants pour travailler, incluant un petit module appelé `libftdi`. Surprenamment, les routines de port serie `PySerial` n'y sont pas, on les installe avec: `easy_install PySerial`. Par la suite, les programmes Python qui suivent peuvent être exécuter.

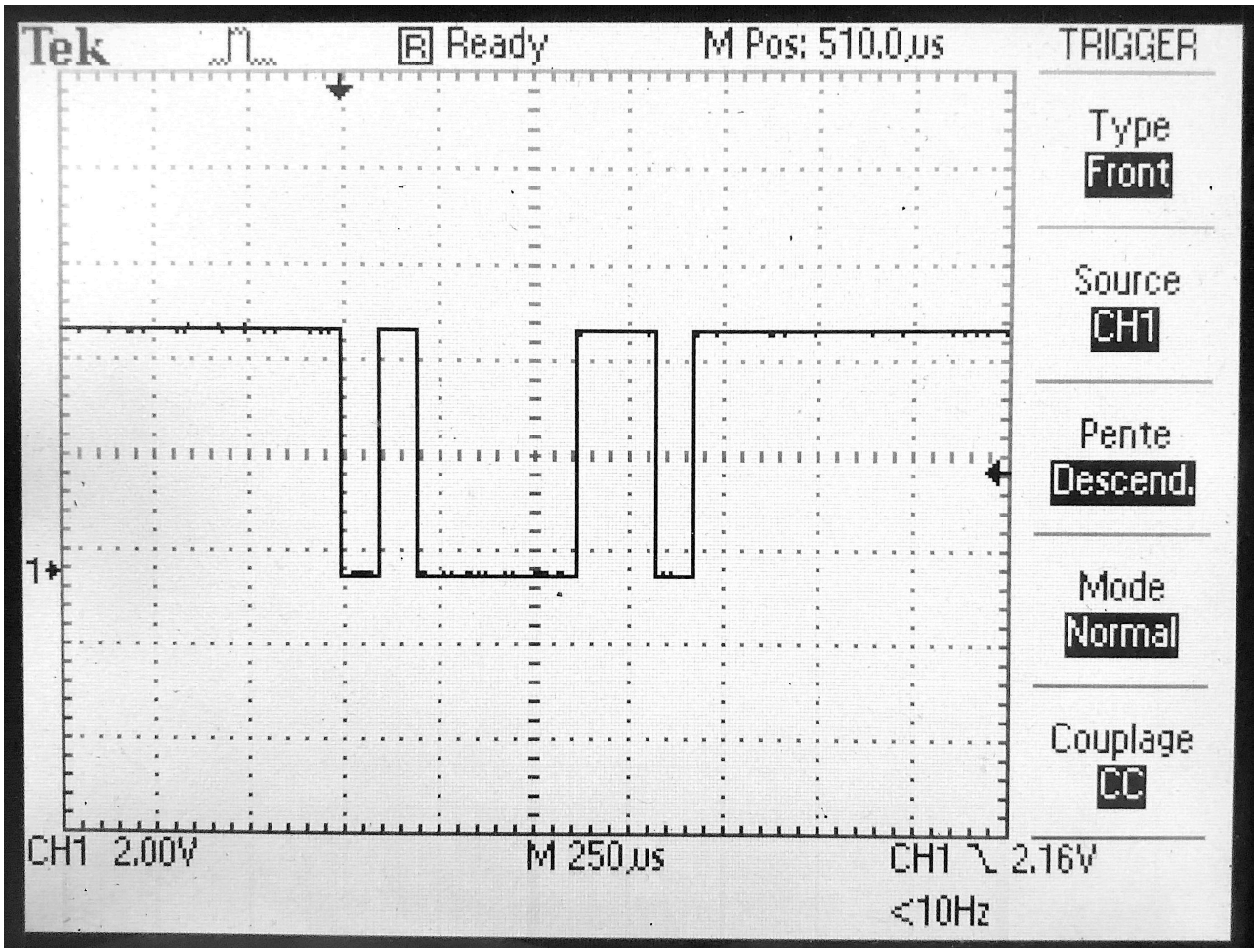
Le UM232R est une puce pour la communication série de type UART (*Universal Asynchronous Receiver-Transmitter*), c'est à dire qu'elle sert à transférer l'information vers un récepteur un bit à la fois. Pour l'instant, ignorons les détails et concentrons-nous sur la ligne de transmission TXD (ou DB0). Dans Python, on peut ouvrir la communication avec cette puce UM232R par "le port série". En effet, tout ce que l'on écrit sera sur la ligne de transmission TXD et tout ce qui est sur la ligne RXD nous pourrons lire par le port série.

```
import serial
import time

text = 'a'
path = '/dev/cu.usbserial-FTCBGW24'
try:
    port = serial.Serial(path, 9600)
    bytesWritten = port.write(text)
    if bytesWritten == len(text):
        print('Wrote to port: %s' % port.name)
    else:
        print('Error when writing to port: %s' % port.name)

    time.sleep(.1)
    port.close()
except IOError:
    print('Unable to open the port with path: %s' % path)
```

```
    except:
        print('Unknown error')
```

Pour commencer, idéalement, on utilise un oscilloscope. Ne vous en faites pas, l'expérience suivante ne nécessite pas d'oscilloscope.

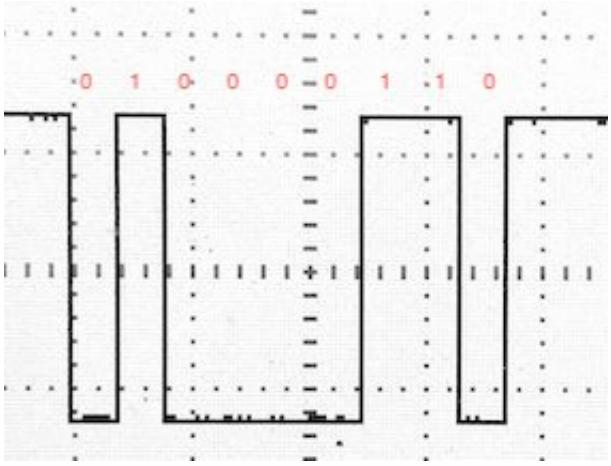
En exécutant le code précédent (`python parler.py`) avec une sonde d'oscilloscope sur la ligne TXD/DB0 (la premiere ligne en haut à gauche), on verra la ligne osciller entre 0V et 5V, comme sur l'image suivante. La ligne est d'abord à 5V, ensuite descend à 0 pour environ 100 µs, remonte à 5V, redescend pendant 400 µs, remonte pour 200 µs et finalement descend 100 µs et remonte pour rester à 5 V.



Nous verrons les details de cette communication UART plus tard, mais on remarque:

1. environ 100 µs est en fait précisément 104 µs, c'est-à-dire 1/9600 de secondes,
2. la lettre 'a' est codée comme le code ASCII 97. En binaire, 97 s'écrit %10000110,
3. à partir de la première descente, il y a 9 périodes de 104 µs pour un total de 936 µs sur l'écran d'oscilloscope. Si on remplace les période de 104 µs à 5V par 1 et celle de 104 µs à 0V par zéro, on obtient 0 suivi de 10000110.

On comprendra donc que la communication envoie la lettre 'a' (code 97) à 9600 bits per seconde. Dans le cas de la puce ici présente qui implémente le protocole UART, la ligne commence toujours à 5V pour commencer, descend à 0 le temps d'un "coup d'horloge", et ensuite écrit la valeur en binaire:



Rien n'est connecté, donc notre module "parle dans le vide". On vient de dire au module d'envoyer un "a" sur "ses lignes de sortie" mais cette information n'a pas été utilisée ou capté par personne, sauf notre oscilloscope.

Expérience 4: parler et illuminer

Ce n'est pas tout le monde qui a un oscilloscope. Pour visualiser un peu ce qui se passe, on peut utiliser une diode électroluminescente (DEL) et la connecter directement sur la ligne de transmission avec une résistance de 220 Ohms au ground. Pour bien voir ce qui se passe, on écrit plusieurs 0 en ligne, à la vitesse la plus lente permise (300 bits par seconde). Ainsi, la ligne TXD sera à 5V pour commencer, et descendra à 0 pour 30 ms, remontera brièvement à 5V pour descendre tout de suite, et ce 15 fois en ligne. La DEL sera allumée, ensuite essentiellement éteinte pendant 0.5 seconde (avec un minuscule petit flash de 3 ms à chaque 30 ms) et ensuite rallumée. Voir le vidéo.

```
import serial
import time

data = [0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0]
path = '/dev/cu.usbserial-FTCBGW24'
try:
    port = serial.Serial(path, 300)
    bytesWritten = port.write(data)
    if bytesWritten == len(data):
        print('Wrote to port: %s' % port.name)
    else:
        print('Error when writing to port: %s' % port.name)

    time.sleep(1)
```

```
    port.close()
except IOError:
    print('Unable to open the port with path: %s' % path)
except:
    print('Unknown error')
```

Expérience 5: écouter dans le vide

Si on essaie de lire le module, on n'obtiendra rien: la ligne `read()` ne complètera jamais (il n'y a pas de *timeout*, il est infini par défaut):

```
import serial

path = '/dev/cu.usbserial-FTCBGW24'
try:
    port = serial.Serial(path)
    text = port.read() ## Bloquera ici. Faites Ctrl-C pour quitter.
    port.close()
except IOError:
    print('Unable to open the port with path: %s' % path)
except:
    print('Unknown error')
```

En effet, le module n'est aucunement connecté à quoi que ce soit pour lire des données. Il peut transmettre, mais n'a rien à lire.

Expérience 6: écouter l'écho

On peut par contre prendre le module et le connecter pour qu'il s'écoute lui-même. En effet, on connecte la ligne de sortie (TXD) à la ligne d'entrée (RXD). Ainsi, en excéutant le code suivant, on pourra écrire "hello" et lire "hello" par la suite.

```
import serial

text = 'hello'
```

```
path = '/dev/cu.usbserial-FTCBGW24'
try:
    port = serial.Serial(path)
    bytesWritten = port.write(text)

    if bytesWritten == len(text):
        print('Wrote to port: %s' % port.name)
    else:
        print('Error when writing to port: %s' % port.name)

    echo = port.read(bytesWritten)

    if bytesWritten == len(echo):
        print('Read from port: %s' % port.name)
    else:
        print('Error when reading to port: %s' % port.name)

    port.close()
except IOError:
    print('Unable to open the port with path: %s' % path)
except:
    print('Unknown error')
```

On appelle ce mode le mode ECHO, puisque tout ce qui est écrit sur le port est lu sur le même port. En exécutant le code précédent, on obtient "hello" en lecture après avoir écrit "hello" sur le port série. Le mode ECHO permet de tester notre compréhension de plusieurs façon car nous savons toujours ce qui devrait être lu sur le port: il s'agira de ce que l'on vient d'écrire.

---

1. En fait, le standard TTL demande une tension supérieure à 2V pour être considéré comme un logique VRAI et moins de 0.8V pour un logique FAUX. ↩

2. Bien qu'il soit omniprésent en informatique, le terme «bit» a été inventé par Claude Shannon, père de la théorie de l'information.↩

3. Le standard USB est un standard complexe qui n'est pas nécessaire de comprendre pour l'instant. Cependant, un bon ingénieur devra comprendre la reconnaissance des appareils, l'association des drivers, les classes, les vendor ID, product ID, la sérialisation des ports, les endpoints, etc... Bien sur, nous verrons tout cela plus loin lorsque ce sera nécessaire.↩

4. Apple fournit depuis macOS 10 un driver FTDI avec le système, mais celui-ci n'est pas directement de FTDI mais bien d'Apple. On peut tout de même installer le driver de FTDI. Cependant, depuis macOS 10, qui resserre la sécurité, le driver doit obtenir la permission de l'Administrateur pour pouvoir s'executer.↩