ÉLECTRONIQUE | EMBARQUÉ | RADIO | IOT

HACKABLE

L'EMBARQUÉ À SA SOURCE

N° 54
MAI / JUIN 2024

FRANCE MÉTRO. : 14,90 €
BELUX : 15,90 € - CH : 23,90 CHF ESP/IT/PORT-CONT : 14,90 €
DOM/S : 14,90 € - TUN : 35,60 TND - MAR : 165 MAD - CAN : 24,99 $CAD

L 19338 - 54 - F: 14,90 € - RD

CPPAP : K92470

RISC-V / FREERTOS

Prise en main du Milk-V Duo, un petit SBC RISC-V double cœur sous Buildroot à moins de 8 € p.64

RP2040 / I2C

Ajoutez une interface i2c externe à n'importe quelle machine grâce au Raspberry Pi Pico p.14

IA / M5STACK

Découvrons le monde de l'Edge AI appliqué à l'IoT avec MicroPython et le MCU Kendryte K210 p.04

Domotique / Radio / LoRa

Comment déployer vos capteurs sans Wi-Fi, sans Bluetooth, sans Zigbee et sans 4G ?

COMMUNICATIONS GRATUITES

HORS RÉSEAU ET LONGUE DISTANCE AVEC MESHTASTIC !

p.50

- Matériel et firmware
- Configuration des nœuds
- Intégration avec Home Assistant

SMARTCARD / MCU

Faites communiquer votre ESP32 et votre Raspberry Pi Pico avec des smartcards p.78

ESP32 / WS2812

Créez un afficheur 8 fois 7 segments RGB indiquant les heures de lever/coucher du soleil p.30

RETRO / SHARP / GO

Le Sharp PC-G850V : un ordinateur de poche des années 2000 embarquant un compilateur C p.112

DU 9
AU 11
MAI
2024

ATELIERS
LUDIQUES
POUR TOUS

FESTIVAL
WE R TECH'

COUPE DE FRANCE DE ROBOTIQUE
SENIOR ET JUNIOR

CONCOURS INTERNATIONAL EUROBOT

PARC EXPO
LA ROCHE-SUR-YON

WWW.COUPEDEROBOTIQUE.FR

#CDFR24

Coupe de France ROBOTIQUE
Coupe de France ROBOTIQUE JUNIOR
eurobot

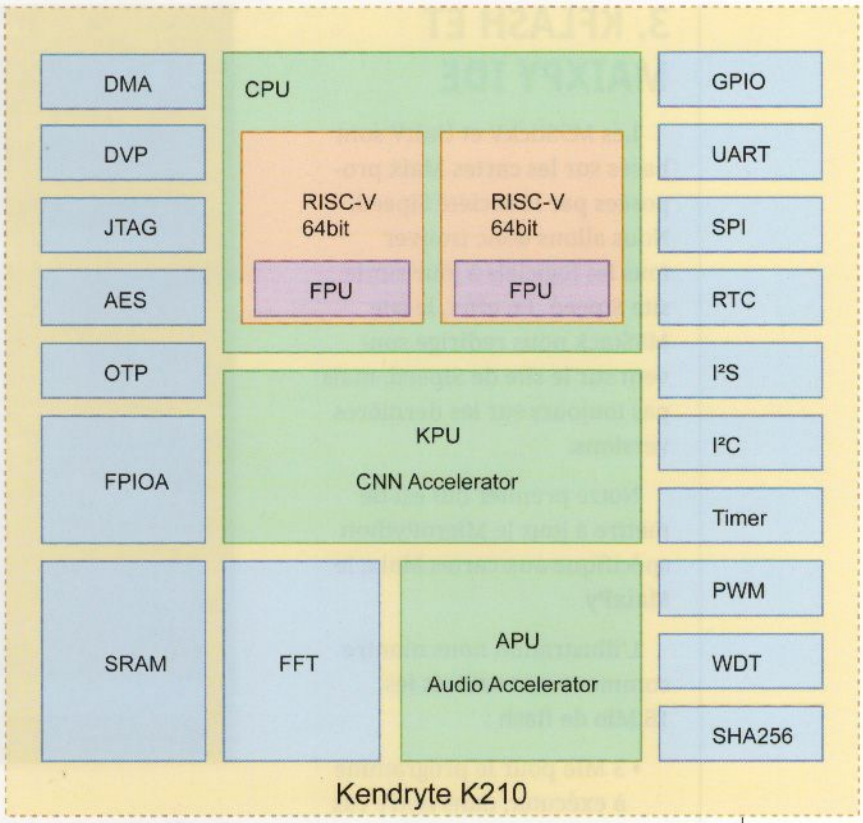
ORYON
Planète Sciences Une aventure pour les jeunes
La Roche-sur-Yon Agglomération Le cœur Vendée
RÉGION PAYS DE LA LOIRE
EXOTEC
LINUX MAGAZINE / FRANCE
HACKABLI MAGAZINE

IAOT AVEC M5STACK

Sébastien Colas

Dans cet article, nous allons parler IAoT, en français l'intelligence artificielle de l'objet. En effet, de nos jours il est de plus en plus fréquent d'embarquer de l'intelligence artificielle au plus prêt des objets connectés comme une caméra, par exemple, ce concept est appelé l'EdgeIA. L'EdgeIA appliquée à l'IoT devient donc IAoT. M5Stack propose des périphériques IAoT à des prix très abordables. Dans cet article, nous allons donc prendre en main le M5StickV ainsi que le UnitV.

Le M5StickV ainsi que le UnitV sont tous les deux basés sur une puce Kendryte K210. Le matériel est fourni avec un interpréteur MicroPython appelé MaixPy, en version 0.5.0. Nous allons donc commencer par parler du K210, puis nous verrons comment construire notre propre *firmware* avec une version à jour de MaixPy. Pour finir, nous effectuerons quelques tests. Tout au long des différentes étapes, nous en profiterons pour prendre en main les différents outils disponibles pour manipuler notre matériel.



1. KENDRYTE K210

Le Kendryte K210 est composé de :

- un CPU RISC-V 64 bits à 2 cœurs ;
- un KPU (*Knowledge Processor Unit*), une unité de calcul de 1 TOPS (*Tera Operations Per Second*) qui intègre un réseau de neurones haute performance, optimisé pour réaliser des traitements en temps réel. Le KPU nécessite qu'on lui fournisse la configuration du réseau de neurones dans un format appelé kmodel.

Le Kendryte K210 est donc idéal pour la détection d'objets dans une image en temps réel.

2. M5STICKV VS UNITV

La grosse différence entre les 2 modèles est que le M5StickV peut être utilisé de façon autonome grâce à sa batterie 200 mA h ainsi qu'à son écran LCD 135*240, le reste des spécifications étant les mêmes. Voici les plus importantes :

- puce Kendryte K210 ;
- SRAM 8 Mio ;
- flash 16 Mio ;
- connecteurs USB-C et Grove ;
- 2 boutons ;
- 1 LED RGB ;
- une caméra OV7740 ;
- un lecteur de carte Micro SD.

3. KFLASH ET MAIXPY IDE

Les M5StickV et UnitV sont basés sur les cartes Maix proposées par la société Sipeed. Nous allons donc trouver tous les logiciels à jour sur le site Sipeed. En effet, le site M5Stack nous redirige souvent sur le site de Sipeed, mais pas toujours sur les dernières versions.

Notre premier but est de mettre à jour le MicroPython spécifique aux cartes Maix, le MaixPy.

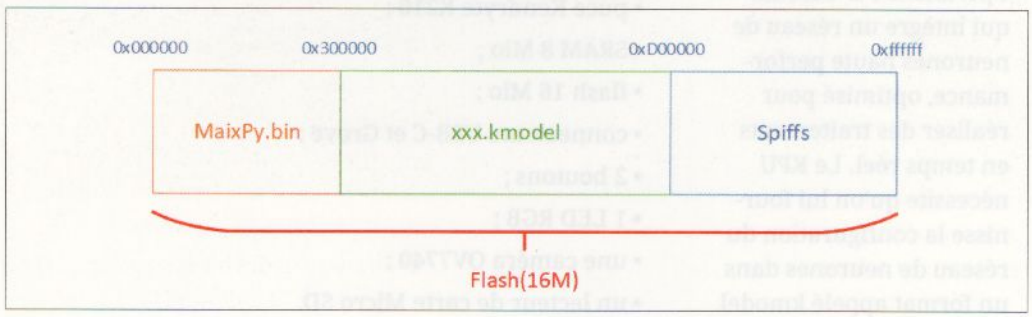
L'illustration nous montre comment sont utilisés les 16 Mio de flash :

- 3 Mio pour le programme à exécuter, dans notre cas le MaixPy (interpréteur micropython) ;
- 10 Mio pour stocker la configuration de notre réseau neuronal appelé kmodel ;
- 3 Mio de système de fichiers en SPIFFS pour stocker le programme Python à exécuter.

L'utilitaire `kflash` (dernière version 1.8.1) **[1]** permet donc de mettre à jour tout ou partie des 16 Mio de flash. Il vous faudra taper la commande `ldd --version` pour connaître la version de GlibC sur votre système GNU/Linux.

Sur la capture d'écran, on peut voir les paramètres les plus importants :

- ***Board*** `M5StickV` ;
- ***Burn To*** `Flash` ;
- ***Port*** `/dev/ttyUSB0 (M5Stack Intf)`.



Sur le site officiel M5Stack, il est possible de télécharger un *firmware* **[2]** au format `.kfpkg`. Ce format n'est rien d'autre qu'une compression ZIP. Le but étant d'installer notre propre *firmware*, nous ne nous attarderons pas sur celui fourni par M5Stack, vous pouvez néanmoins vous en servir pour tester l'utilitaire `kflash`.

```
 __  __       _____ _             _    __      __
|  \/  |     /\    |_   _| \ \ / / |  __ \  \ \   / /
| \  / |    /  \     | |    \ V /  | |__) |  \ \_/ /
| |\/| |   / /\ \    | |     > <   |  ___/    \   /
| |  | |  / ____ \  _| |_   / . \  | |         | |
|_|  |_| /_/    \_\|_____| /_/ \_\ |_|         |_|

M5StickV by M5Stack : https://m5stack.com/
M5StickV Wiki       : https://docs.m5stack.com
Co-op by Sipeed     : https://www.sipeed.com

Traceback (most recent call last):
  File "_boot.py", line 62, in <module>
  File "_boot.py", line 62, in <module>
  File "<string>", line 9, in <module>
  File "pmu.py", line 53, in __init__
NotFoundError:
MicroPython v0.5.0-98-g7ec09ea22-dirty on 2020-07-21; Sipeed_M1 with kendryte-k210
Type "help()" for more information.
>>>
```

Il nous faut ensuite installer l'outil nous permettant le développement Python. Il s'agit de MaixPy IDE version 0.2.5 **[3]**. Lors de la connexion de MaixPy IDE à notre `M5StickV`, on découvre que la version de MaixPy n'est pas à jour : version 0.5.0. Il est aussi possible de connecter le `M5StickV` en mode terminal à l'aide de la commande `screen` :

```
screen /dev/ttyUSB0 115200
```

4. COMPILATION ET INSTALLATION DE MAIXPY

Comme nous l'explique la documentation **[4]**, il y a un certain nombre d'étapes pour pouvoir compiler un nouveau *firmware* pour notre `M5StickV`.

Tout d'abord, récupérons les sources du projet, ses dépendances :

```
mkdir ~/project
cd ~/project
https://github.com/sipeed/maixpy-v1
cd maixpy-v1
git submodule update --recursive --init
sudo apt install python3 python3-pip build-essential cmake
sudo pip3 install -r requirements.txt
```

Nous avons les sources de la dernière version du *firmware*. La prochaine étape consiste à télécharger et à compiler sur notre machine locale le compilateur dédié à notre périphérique, il s'agit du compilateur `kendryte` : Kendryte RISC-V GNU Compiler Toolchain **[5]**.

```
cd ~/project
git clone --recursive https://github.com/kendryte/kendryte-gnu-toolchain
sudo apt-get install autoconf automake autotools-dev curl libmpc-dev
libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool
patchutils bc zlib1g-dev libexpat-dev
cd kendryte-gnu-toolchain
./configure --prefix=/opt/kendryte-toolchain --with-cmodel=medany --with-
arch=rv64imafc -with-abi=lp64f
make -j8
```

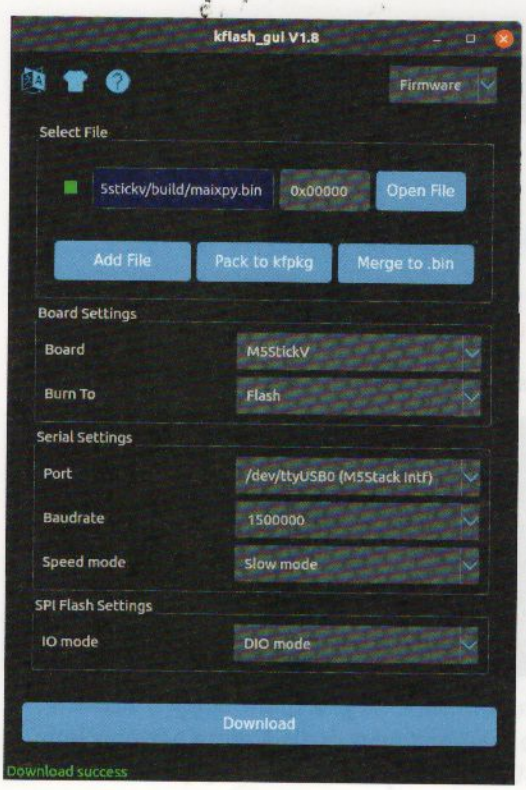
Le compilateur `kendryte` sera installé dans le répertoire `/opt/kendryte-toolchain`. Nous pouvons maintenant configurer le *firmware* avant de le compiler :

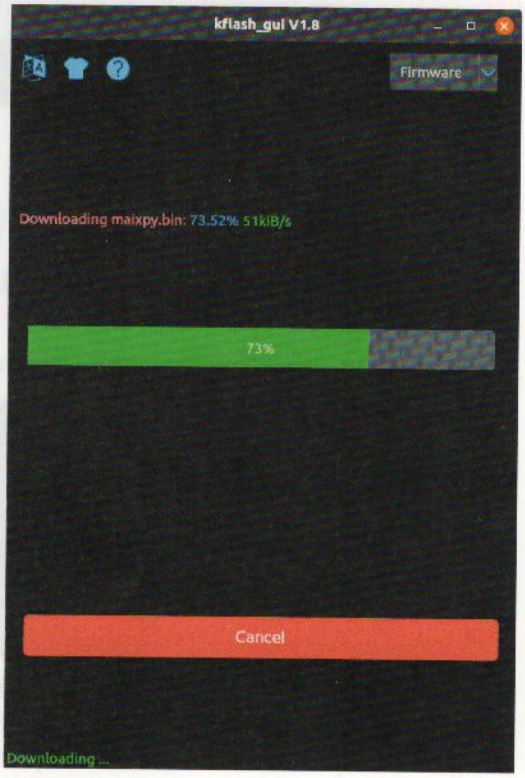
```
cd ~/project/maixpy-v1/projects/maixpy_m5stickv
PATH=$PATH:/opt/kendryte-toolchain/bin
python3 project.py menuconfig
```

La configuration par défaut est suffisante, il est temps de lancer la compilation :

```
python3 project.py build
```

Le nouveau *firmware* est disponible ici : `build/maixpy.bin`.





Plaçons-le dans un répertoire de travail :

```
mkdir ~/project/firmware
mv ~/project/maixpy-v1/projects/maixpy_m5stickv/build/maixpy.bin ~/project/
firmware/maixpy-0.6.2.bin
```

Utilisons `kflash` pour mettre à jour le *firmware*.

Pour tester notre nouveau *firmware*, il nous faut un réseau de neurones préentraîné au format kmodel ainsi qu'un système de fichiers au format SPIFFS contenant le programme python à exécuter. C'est ce que nous allons installer dans les sections suivantes.

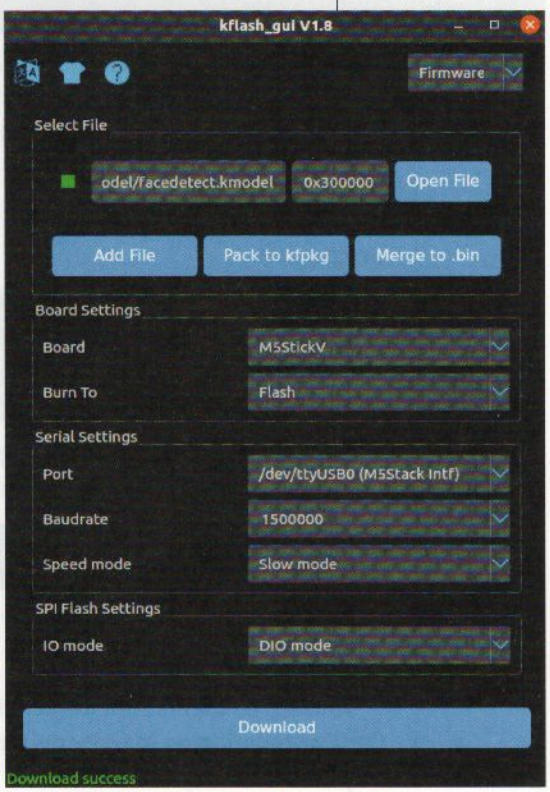
5. INSTALLATION D'UN KMODEL

Téléchargeons un modèle de détection des visages préentraîné :

```
mkdir ~/project/kmodel
cd ~/project/kmodel
wget https://api.dl.sipeed.com/fileList/MAIX/MaixPy/model/face_model_
at_0x300000.kfpkg
unzip face_model_at_0x300000.kfpkg
mv ~/project/kmodel/facedetect.kmodel ~/project/firmware/
```

Le fichier téléchargé est un `.kfpkg`, il s'agit en fait d'une compression ZIP ainsi qu'un fichier `flash-list.json` contenant l'adresse où déployer le modèle. Nous créerons notre propre `.kfpkg` par la suite. Pour le moment, contentons-nous de récupérer le kmodel s'y trouvant.

Utilisons `kflash` pour installer notre fichier kmodel à l'adresse `0x300000` de la mémoire flash.



6. PRISE EN MAIN DU SYSTÈME DE FICHIERS SPIFFS

Notre MaixPy et notre kmodel étant prêts, créons le programme python qui sera déployé :

```
mkdir ~/project/fs
```

Créons le fichier `~/project/fs/boot.py` :

```
import sensor, image, time, lcd
import KPU as kpu
```

```
lcd.init(freq=15000000,type=3)
lcd.rotation(2)
sensor.reset()
sensor.set_pixformat(sensor.RGB565)
sensor.set_framesize(sensor.QVGA)
sensor.run(1)

task = kpu.load(0x300000)
anchor = (1.889, 2.5245, 2.9465, 3.94056, 3.99987, 5.3658, 5.155437, 6.92275,
6.718375, 9.01025)
a = kpu.init_yolo2(task, 0.5, 0.3, 5, anchor)

while(True):
    img = sensor.snapshot()
    code = kpu.run_yolo2(task, img)
    if code:
        for i in code:
            print(i)
            a = img.draw_rectangle(i.rect())
    lcd.display(img)

a = kpu.deinit(task)
```

Rien de particulier à signaler dans le programme, si ce n'est la variable `anchor` contenant une liste de valeurs. Cette liste de valeurs est attachée au kmodel que nous utilisons. En effet, si nous créons notre propre kmodel, un programme Python nous sera fourni, contenant ces valeurs.

Créons maintenant un système de fichiers au format SPIFF spécifique pour déployer notre programme.

```
cd ~/project
python3 maixpy-v1/tools/spiffs/gen_spiffs_image.py maixpy-v1/projects/
maixpy_m5stickv/config_defaults.mk
```

Le résultat se trouve dans le répertoire `fs_image`. Copions le système de fichiers dans le répertoire accueillant notre nouveau *firmware* :

```
cp fs_image/maixpy_spiffs.img firmware/
```

Utilisons `kflash` pour installer notre système de fichiers SPIFF à l'adresse `0xD00000` de la mémoire flash.

Tous les éléments sont désormais présents, notre périphérique doit fonctionner. Pour simplifier l'installation de MaixPy du kmodel ainsi que du système de fichiers SPIFF, nous allons créer un *firmware* complet.

7. CRÉATION D'UN FIRMWARE MAISON

Pour créer un *firmware*, il nous faut créer un fichier décrivant où enregistrer les éléments dans la mémoire flash. Créons donc le fichier `~/project/firmware/flash-list.json` :

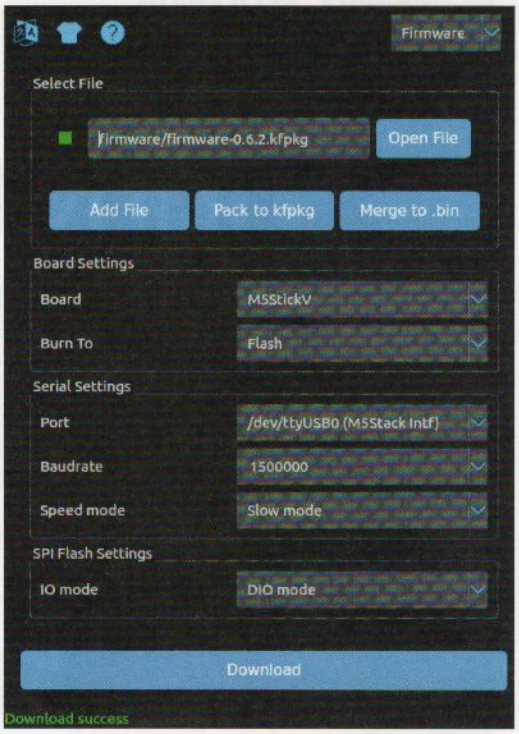
```
{
 "version": "0.6.2",
 "files": [
   {
     "address": 0x00000000,
     "bin": "maixpy-0.6.2.bin",
     "sha256Prefix": true
   },
   {
     "address": 0x00D00000,
     "bin": "maixpy_spiffs.img",
     "sha256Prefix": false
   },
   {
     "address": 0x00300000,
     "bin": "facedetect.kmodel",
     "sha256Prefix": false
   }
 ]
}
```

Notre répertoire `~/project/firmware` contient désormais tout ce qui est nécessaire pour construire notre *firmware* :

```
firmware/
├── facedetect.kmodel
├── flash-list.json
├── maixpy-0.6.2.bin
└── maixpy_spiffs.img
```

Il est temps de créer notre *firmware*.

```
cd ~/project/firmware
zip firmware-0.6.2.kfpkg *
```



8. TESTS ET CONCLUSION

Déployons notre *firmware* à l'aide de `kflash`.

Il est temps de tester le bon fonctionnement de notre *firmware*. Notre premier test utilisera MaixPy IDE :

```
./maixpy-ide/bin/maixpyide
```

Une fois le logiciel lancé, nous pouvons exécuter le programme de notre choix. Ouvrons donc notre programme de test, puis cliquons sur les boutons ***Connect*** puis ***Start*** (les boutons sont en bas à gauche de l'interface).

Dans le deuxième test, utilisons à nouveau `screen` pour lancer le programme enregistré sur notre périphérique. Si tout fonctionne, nous devrions voir les coordonnées des visages détectés.

```
 __  __              _____  __   __  _____   __     __
|  \/  |     /\     |_   _| \ \ / / |  __ \  \ \   / /
| \  / |    /  \      | |    \ V /  | |__) |  \ \_/ /
| |\/| |   / /\ \     | |     > <   |  ___/    \   /
| |  | |  / ____ \   _| |_   / . \  | |         | |
|_|  |_| /_/    \_\ |_____| /_/ \_\ |_|         |_|

M5StickV by M5Stack : https://m5stack.com/
M5StickV Wiki       : https://docs.m5stack.com
Co-op by Sipeed     : https://www.sipeed.com

init i2c:2 freq:100000
[MAIXPY]: find ov7740
[MAIXPY]: find ov sensor
{"x":190, "y":81, "w":107, "h":144, "value":0.500000, "classid":0, "index":0, "objnum":1}
{"x":182, "y":67, "w":107, "h":144, "value":0.611305, "classid":0, "index":0, "objnum":1}
{"x":176, "y":81, "w":107, "h":115, "value":0.884690, "classid":0, "index":0, "objnum":1}
{"x":166, "y":69, "w":107, "h":144, "value":0.859508, "classid":0, "index":0, "objnum":1}
{"x":165, "y":78, "w":108, "h":144, "value":0.829885, "classid":0, "index":0, "objnum":1}
```

Notre caméra intelligente est donc prête à être utilisée :

- en mode autonome pour le M5StickV ;
- en mode connecté via USB pour le UnitV.

Pour aller plus loin :

- modifier le programme Python pour utiliser efficacement le périphérique, carte SD, LED...
- construire son propre kmodel ou en utiliser un préentraîné. SC

RÉFÉRENCES

[1] KFlash :
https://github.com/sipeed/kflash_gui/releases

[2] Firmware M5Stack :
https://m5stack.oss-cn-shenzhen.aliyuncs.com/resource/docs/M5StickV_Firmware_v5.1.2.kfpkg

[3] MaixPy IDE :
https://api.dl.sipeed.com/shareURL/MAIX/MaixPy/ide/v0.2.5

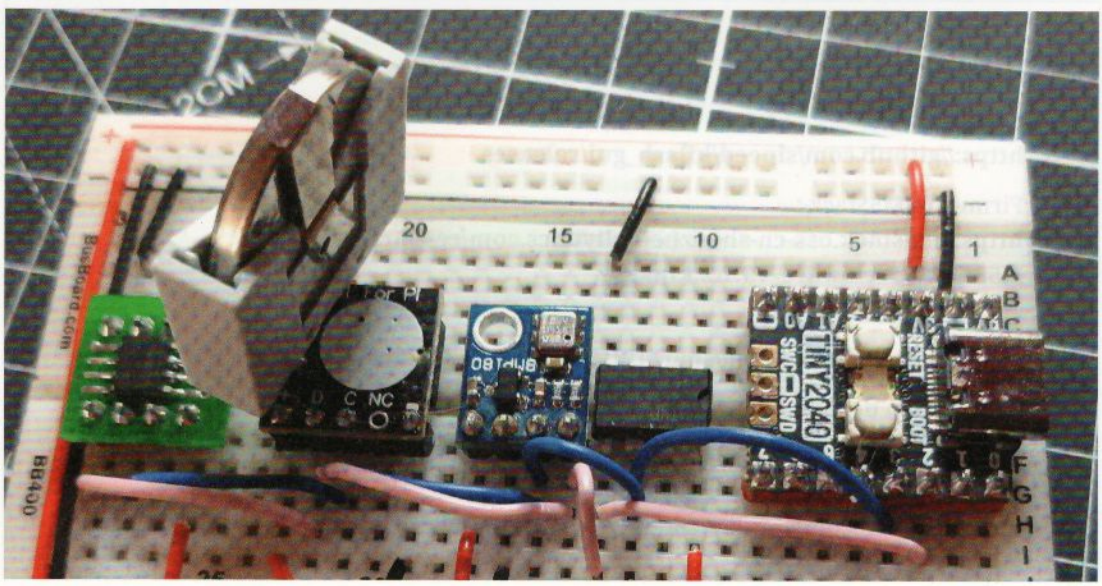
[4] Compilation de MaixPy :
https://wiki.sipeed.com/soft/maixpy/en/course/advance/compile.html

[5] Compilateur Kendryte :
https://github.com/kendryte/kendryte-gnu-toolchain

I2C-TINY-USB : UN BUS I2C FACILEMENT ACCESSIBLE POUR VOTRE PC

Denis Bodor

Lorsqu'on cherche à développer un support pour un composant interfacé en i2c qui ne dispose pas encore de bibliothèque, que ce soit pour Arduino ou toute autre plateforme pour microcontrôleur (RP2040, ESP32, etc.), la phase de mise au point et de test n'est pas toujours très aisée. Il faut travailler sur un code de firmware, programmer le MCU, tester et itérer, ce qui fait perdre un temps conséquent. Il est bien plus facile de développer sur PC ou Pi pour ensuite porter le code sur MCU. Mais encore faut-il disposer d'un bus i2c facilement accessible…



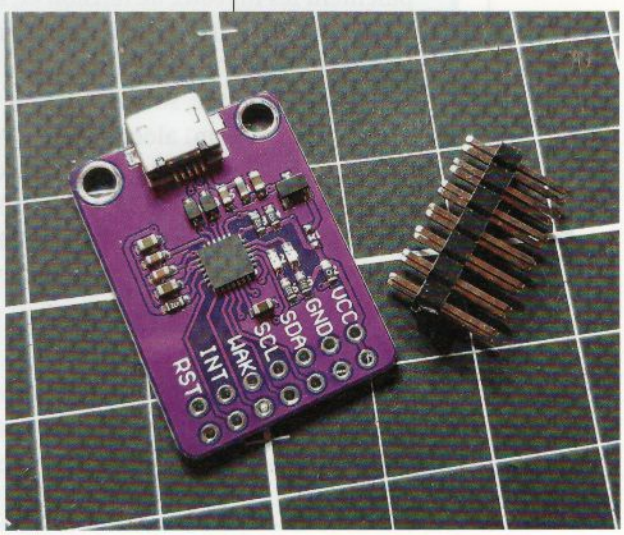
– i2c-tiny-usb : un bus i2c facilement accessible pour votre PC –

J'ai évoqué dans l'introduction le fait d'utiliser une carte Raspberry Pi, mais le vrai confort de développement concerne, bien entendu, un environnement sur PC, avec tous les outils à disposition, et ce, sans avoir à passer par une connexion SSH. Le problème cependant est que, contrairement à une Pi ou n'importe quel SBC, un PC moderne ne dispose pas de bus i2c vraiment accessible. Bien sûr, les fonctionnalités sont matériellement là puisque c'est ainsi que les capteurs de température, par exemple, sont utilisés et consultés par le système. Vous pouvez d'ailleurs en avoir un aperçu très facilement avec une machine sous GNU/Linux en utilisant une simple commande :

```
$ sudo i2cdetect -l
i2c-0   smbus   SMBus PIIX4 adapter port 0 at 0b00  SMBus adapter
i2c-1   smbus   SMBus PIIX4 adapter port 2 at 0b00  SMBus adapter
i2c-2   smbus   SMBus PIIX4 adapter port 1 at 0b20  SMBus adapter
i2c-3   i2c     AMDGPU DM i2c hw bus 0              I2C adapter
i2c-4   i2c     AMDGPU DM i2c hw bus 1              I2C adapter
i2c-5   i2c     AMDGPU DM i2c hw bus 2              I2C adapter
i2c-6   i2c     AMDGPU DM i2c hw bus 3              I2C adapter
i2c-7   i2c     AMDGPU DM aux hw bus 2              I2C adapter
i2c-8   i2c     AMDGPU DM aux hw bus 3              I2C adapter
```

Cette machine, un PC Minisforum HX90 à base d'AMD Ryzen 9 5900HX, a pas moins de 8 bus disponibles, mais aucun d'entre eux n'est effectivement accessible facilement pour y connecter des capteurs ou composants à tester. Ici, il s'agit d'un mini PC de type « NUC » et même s'il était possible d'y « hacker » un accès, via le port HDMI ou DisplayLink, ou encore en se connectant directement sur la carte mère, on peut considérer que l'opération est relativement risquée, en particulier en raison de la densité et la compacité de ce genre de machine. L'âge des cartes mères avec des composants SOIC sur lesquels se greffer sans trop de problèmes semble définitivement révolu. Et utiliser une ancienne machine comme plate-forme de développement n'est pas vraiment une option viable, autant utiliser un SBC, cela ne changerait pas grand-chose au niveau du confort de développement.

Ce module à base de CP2112 est un pont USB/SMBus permettant de disposer non seulement d'un bus i2c externe à la machine hôte (PC ou SBC), mais propose également un certain nombre de GPIO pilotables depuis l'espace utilisateur de l'OS.



Une solution bien plus intéressante serait de disposer d'un périphérique externe, donc en USB, fournissant un bus i2c et s'intégrant parfaitement au système comme n'importe quel autre bus du même type. Nous parlons ici de quelque chose de directement pris en charge par le noyau Linux

i2c, SMBus et TWI

Ces trois termes désignent des technologies compatibles sinon identiques. i2c pour *Inter-Integrated Circuit*, parfois également écrit IIC, a été initialement développé par Philips, maintenant NXP, pour des applications domotiques et l'interfaçage de capteurs au début des années 80.

« i2c » étant au départ une marque déposée, tout comme le logo associé, d'autres constructeurs comme Atmel ont implémenté cette même technologie en la désignant sous l'acronyme TWI pour *Two-Wire Interface* (et parfois TWSI, *Two-Wire Serial Interface*). Les différences effectives avec i2c sont si minimes que les deux termes sont presque totalement permutables.

De la même manière, SMBus, pour *System Management Bus*, est un protocole compatible et totalement interopérable avec i2c. C'est un bus de communication créé par Intel et Duracell dès 1994 et littéralement construit sur i2c. Cette désignation, plutôt que simplement « i2c », est généralement utilisée dans le monde informatique puisque c'est par exemple le bus utilisé pour qu'un système récupère les données de configuration d'un module DRAM. Les différences entre i2c et SMBus concernent principalement les niveaux de tension et, selon la version des spécifications, la fréquence d'horloge du bus. Les tensions des niveaux logiques en i2c, par exemple, sont relatives à VDD (la tension d'alimentation) alors qu'avec SMBus, elles sont fixes (< 0,8 V et > 2,1 V). Une autre différence concerne la fréquence avec 10 à 100 kHz pour SMBus (< 3.0), contre 0 à 3,4 MHz pour i2c (selon le mode utilisé). Techniquement, un bus i2c cadencé à moins de 10 kHz n'est donc pas compatible SMBus.

Dans les grandes lignes cependant, et sauf cas très particulier, i2c, SMBus et TWI désignent donc dans les faits peu ou prou la même chose.

et non d'un outil reposant, par exemple, sur la LibUSB fournissant une sorte d'émulation i2c en espace utilisateur. Le but est de pouvoir, si nécessaire, prendre un code développé de cette manière et le migrer, sans la moindre modification, sur un SBC comme une Pi ou autre, possédant ses propres bus i2c « internes ».

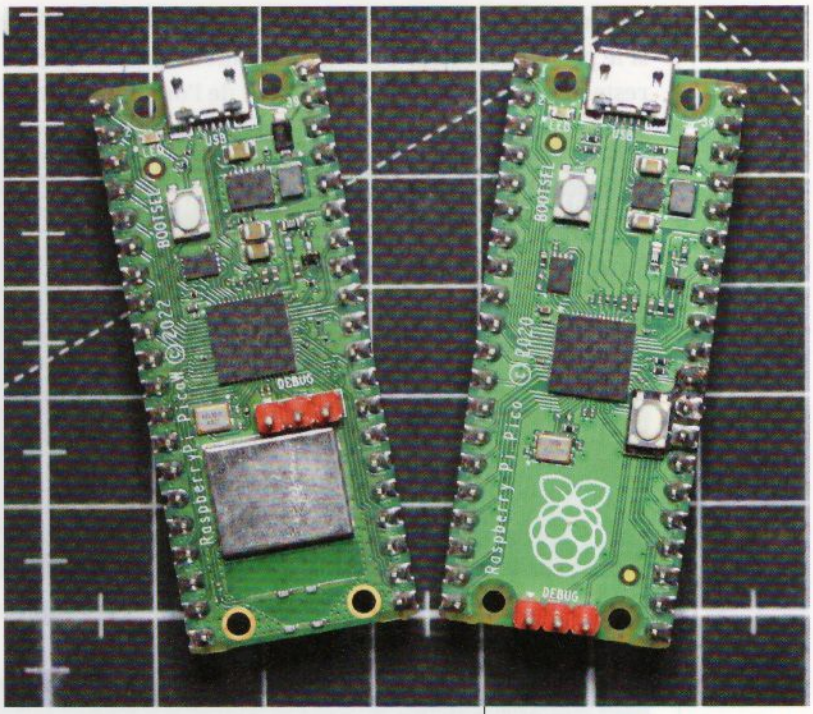
Fort heureusement, ceci existe et il y a même deux solutions envisageables...

1. OPTION 1 : CP2112

La puce CP2112 **[1]** de Silicon Laboratories est un pont USB vers SMBus. Celle-ci ne nécessite aucun *firmware* et se présente comme un périphérique USB HID pour un système USB hôte. Il s'agit d'une solution intégrée pouvant être supportée au niveau noyau comme en espace utilisateur pour tout système capable de prendre en charge les périphériques de la classe HID (*Human Interface Device*). Pour rappel, cette classe est initialement destinée au support de périphériques d'entrée comme les claviers, souris ou *gamepads*, mais permet également d'interfacer de nombreux autres types d'accessoires et gadgets.

L'intérêt premier de l'USB HID dans ce cas est que la totalité des systèmes d'exploitation modernes prend déjà en charge ce type de standard et donc que le CP2112

pourra être utilisable sans avoir à développer de pilote spécifique. Dans notre situation cependant, l'utilisation d'un pilote est très précisément ce que nous souhaitons, puisque le bus i2c doit être intégré au système au même titre que ceux déjà présents dans la machine. Fort heureusement, Linux comme FreeBSD possèdent un tel pilote. À noter, de plus, que le CP2112 n'est pas qu'une simple interface SMBus, mais propose également un jeu de GPIO qui seront utilisables de la même manière avec un système compatible.



L'implémentation RP2040 de i2c-tiny-usb fonctionnera tout aussi bien sur une carte Pico que sur une Pico-W, mais les fonctionnalités matérielles disponibles étant déjà largement sous-utilisées par le firmware, on préférera immobiliser la carte de moindre importance (et coût).

Trouver ce composant déjà prêt à servir sous forme de module, sans avoir à créer un circuit soi-même, n'est pas très difficile puisqu'il est disponible un peu partout, que ce soit Amazon (~16 €) ou AliExpress (~6 €). Celui-ci se présente alors sous la forme d'un petit circuit de 30 mm par 21 mm équipé du CP2112, d'un régulateur de tension, deux LED et quelques composants passifs. Un connecteur USB micro B permet la liaison avec un PC et 14 broches fournissent tension d'alimentation, masse, SDA, SCL, 3 GPIO (5 à 7) et un certain nombre de signaux de contrôle (*reset*, INT, *suspend*, etc.).

La mise en œuvre est relativement aisée avec la plupart des distributions GNU/Linux puisqu'il suffit de brancher le module :

```
cp2112 0003:10C4:EA90.0007: hidraw4:
  USB HID v1.01 Device [Silicon Laboratories CP2112
  HID USB-to-SMBus Bridge] on
  usb-0000:04:00.4-2.1.3.3/input0
cp2112 0003:10C4:EA90.0007: Part Number:
  0x0C Device Version: 0x03
gpio gpiochip2: (cp2112_gpio): not an
  immutable chip, please consider fixing it!
```

La commande `i2cdetect -l` vous montre alors un bus i2c supplémentaire appelé `"CP2112 SMBus Bridge on hidraw*"` que vous pourrez utiliser comme n'importe lequel de ceux déjà présents. Il vous suffira alors d'utiliser SDA/SCL avec deux résistances de rappel à la tension d'alimentation d'environ 10 kΩ (recommandé), ainsi que Vcc et la masse pour vous interfacer facilement

avec un ou plusieurs composants i2c. L'outil `i2cdetect` permet de scanner un bus i2c, même si le protocole n'offre à l'origine aucun mécanisme dédié à cet effet, mais pour une raison qui reste mystérieuse, le comportement de l'outil n'est pas celui attendu :

```
$ i2cdetect -y 9
Warning: Can't use SMBus Quick Write command,
         will skip some addresses
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:
10:
20:
30: -- -- -- -- -- -- -- --
40:
50: 50 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60:
70:
```

En effet, l'adaptateur ne supporte pas certaines commandes SMBus comme le montre la sortie de l'option `-F` :

```
$ i2cdetect -F 9
Functionalities implemented by /dev/i2c-9:
I2C                              yes
SMBus Quick Command              no  <------
SMBus Send Byte                  yes
SMBus Receive Byte               yes
SMBus Write Byte                 yes
SMBus Read Byte                  yes
SMBus Write Word                 yes
SMBus Read Word                  yes
SMBus Process Call               yes
SMBus Block Write                yes
SMBus Block Read                 yes
SMBus Block Process Call         yes
SMBus PEC                        no
I2C Block Write                  yes
I2C Block Read                   yes
```

Ce qui est relativement surprenant, étant donné que le même module, utilisé sous FreeBSD, sera non seulement reconnu sans problème, mais affichera effectivement l'ensemble des composants présents sur le bus lors d'un scan :

```
$ i2c -f /dev/iic8 -s
23 3c 50 60 68 76
```

Même une implémentation équivalente d'un scan `i2cdetect` fonctionnera sans problème :

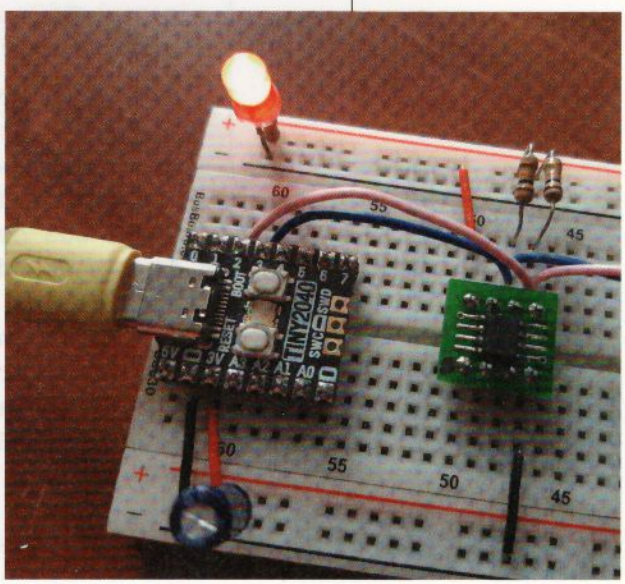
```
$ ./i2cscan /dev/iic8
Checking device: /dev/iic8
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- 23 -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- 3C -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: 50 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: 60 -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
70: -- -- -- -- -- -- 76 -- -- -- -- -- -- -- -- --
```

Il doit s'agir d'un problème d'implémentation du pilote, mais j'avoue ne pas avoir creusé la question, ayant opté pour la seconde solution assez rapidement. J'ai également constaté des différences de comportement lors de l'accès aux composants et en particulier avec un écran OLED 128×64 contrôlé par un SSD1306 dont le comportement était instable sous Linux **[2]**, alors qu'il fonctionnait sans problème sous FreeBSD ou en utilisant l'autre option que nous allons découvrir à présent.

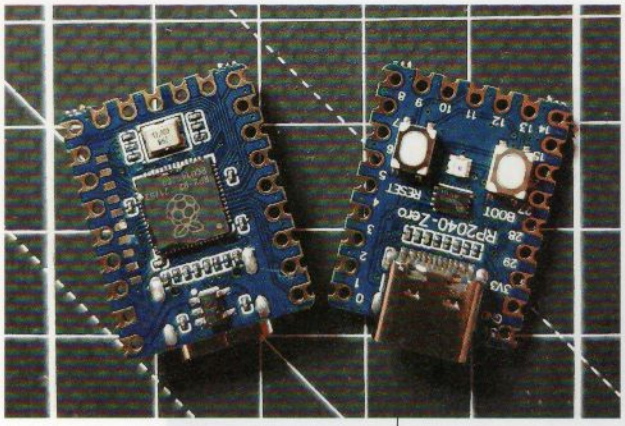
2. OPTION 2 : I2C-TINY-USB

Le CP2112 est un composant très intéressant et qui pourra venir compléter la boîte à outils électronique, mais c'est à la fois une solution très spécialisée et quelque chose qui n'est pas très souple en termes d'évolutions et de développements futurs. Une autre approche consiste à plutôt opter pour quelque chose de générique et réalisable avec « les moyens du bord ».

Il y a plusieurs années, Till Harbaum a rencontré une problématique similaire à la nôtre et a décidé de la solutionner avec un circuit de sa conception : l'i2c-tiny-usb **[3]**. Basé sur le microcontrôleur Atmel AVR ATtiny45, son circuit implémente une interface USB entièrement logicielle et un protocole relativement simple permettant d'obtenir un pont USB/i2c exactement comme le fait le CP2112. Il n'est plus question ici de reposer sur USB HID et un support dans le noyau est alors absolument nécessaire. Fort heureusement, du fait de sa popularité, i2c-tiny-usb est depuis devenu une sorte de standard et le pilote est intégré au



Cette minuscule carte est la TINY2040 de Pimoroni. Celle-ci, bien que comportant moins de broches qu'une Pico standard se prêtera parfaitement à la réalisation de ce montage.



Vous trouverez sur AliExpress des clones de TINY2040, appelés RP2040-zero, pour 2,5 € / pièce (+ 1,9 € de port), tout aussi compacts et avec un connecteur USB-C, mais cinq GPIO accessibles supplémentaires et une LED RVB WS2812 intégrée.

noyau Linux depuis la version 2.6.22 (2007 !). Il est donc plus que probable que cette fonctionnalité soit d'ores et déjà présente dans la distribution que vous utilisez, que ce soit sur PC ou SBC (elle est même activée dans HAOS, la distribution Home Assistant, j'ai vérifié).

i2c-tiny-usb a évolué avec le temps et des versions ATmega8, ATmega88p, ATmega168p et ATmega328p sont disponibles pour une plateforme appelée tinyUSBboard **[4]**, même si le code n'a pas réellement changé depuis plus de 7 ans. C'est une option intéressante, mais le point le plus important est le fait que le protocole lui-même s'est retrouvé « normalisé » au fil du temps. Ainsi, si vous n'avez pas vraiment envie de confectionner un circuit ou d'adopter une solution basée sur un MCU 8 bits en voie de disparition, c'est vers un autre dépôt GitHub qu'il faudra tourner votre attention : rp2040-i2c-interface **[5]** de Nicolai Electronics. Là, vous trouverez une réimplémentation plus moderne du code de Till Harbaum reposant sur l'utilisation d'une carte Raspberry Pi Pico.

Pour transformer une Pico en pont USB/i2c, rien de plus simple, il vous suffit de télécharger les sources depuis GitHub avec `git clone https://github.com/Nicolai-Electronics/rp2040-i2c-interface.git`, vous placer dans le répertoire `rp2040-i2c-interface`, créer un sous-répertoire pour la compilation et vous y rendre, pour enfin invoquer CMake (`cmake ../`) puis GNU Make (`make`) pour produire `i2c_adapter.bin` pour une utilisation avec `picotool` et `i2c_adapter.uf2` pour le flashage via l'émulation de stockage en mode BOOTSEL.

Dès le *firmware* chargé en flash et la Pico redémarrée, le noyau Linux détecte la présence du nouveau périphérique USB comme attendu :

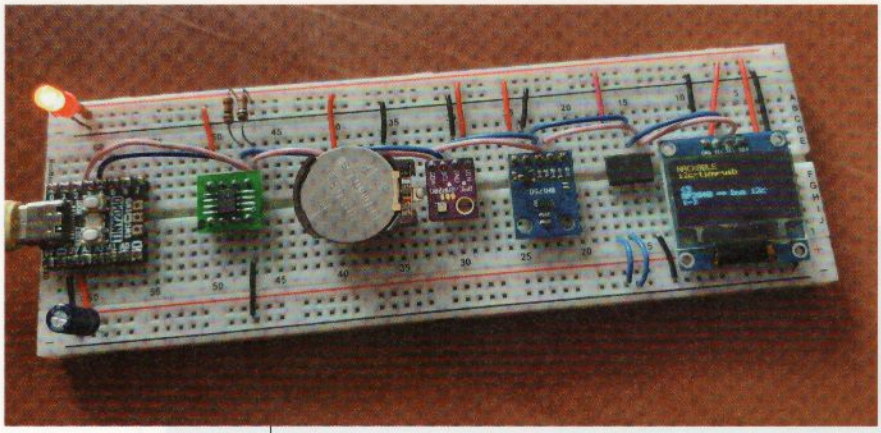
```
usb: new full-speed USB device number 15 using xhci_hcd
usb: unable to get BOS descriptor or descriptor too short
usb: New USB device found, idVendor=1c40, idProduct=0534,
     bcdDevice= 1.00
usb: New USB device strings: Mfr=1, Product=2, SerialNumber=4
usb: Product: I2C adapter
usb: Manufacturer: Nicolai Electronics
usb: SerialNumber: E660583883856E34
i2c-tiny-usb 3-2.1.3.4:1.0: version 1.00 found
                            at bus 003 address 015
i2c i2c-3: connected i2c-tiny-usb device
```

Vous disposez maintenant d'un adaptateur USB/i2c pris en charge par le système comme vous l'indique `i2cdetect -l` (`i2c-tiny-usb at bus 003 device 015`). Les broches utilisées par défaut par le *firmware* sont GP2 (broche 4) pour SDA, et GP3 (broche 5) pour SCL. Ajoutez simplement les deux résistances de rappel et vos composants i2c sur ce bus, avec l'alimentation prise directement depuis la Pico (broches 36 pour 3V3 et 38/33/28/23/, etc., pour la masse) et le tour est joué :

```
$ i2cdetect -F 3
Functionalities implemented by /dev/i2c-3:
I2C                              yes
SMBus Quick Command              yes
SMBus Send Byte                  yes
SMBus Receive Byte               yes
SMBus Write Byte                 yes
SMBus Read Byte                  yes
SMBus Write Word                 yes
SMBus Read Word                  yes
SMBus Process Call               yes
SMBus Block Write                yes
SMBus Block Read                 no
SMBus Block Process Call         no
SMBus PEC                        no
I2C Block Write                  yes
I2C Block Read                   yes

$ i2cdetect -y 3
     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f
00:                         -- -- -- -- -- -- -- --
10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
20: -- -- -- 23 -- -- -- -- -- -- -- -- -- -- -- --
30: -- -- -- -- -- -- -- -- -- -- -- -- 3c -- -- --
40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
50: 50 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --
60: 60 -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --
70: -- -- -- -- -- -- 76 --
```

Le code du *firmware* est relativement simple puisqu'il se contente de réceptionner des commandes USB de contrôle à destination de l'interface *vendor*, d'en extraire les données, et de les relayer sur le périphérique i2c matériel du RP2040. C'est également un excellent point de départ pour expérimenter autour de l'USB *device* avec le RP2040, étant donné que le protocole utilisé est simple et que la bibliothèque TinyUSB (également présente dans l'environnement ESP-IDF) est relativement facile à prendre en main.

Mais ce qui nous intéresse ici, ce n'est pas tant le *firmware* que le résultat qu'il permet d'obtenir. Nous disposons maintenant d'une interface i2c externe nous donnant l'opportunité de développer très facilement du code pour prendre en charge différents composants sur ce bus. À noter que le support pour FreeBSD existe **[6]**, mais devra être ajouté manuellement en compilant un module noyau. Une contribution officielle (via un PR GitHub) a été proposée **[7]** et trouvera peut-être place dans une future version stable du système.



Difficile de résister lorsqu'il s'agit de mettre le matériel (et le pilote) à l'épreuve. Nous avons là un bus bien étoffé avec, de gauche à droite, un HSM Atmel ECC608A, une RTC DS3231, un capteur pression/temp/hygro BMP280, un capteur d'intensité lumineuse BH1750, une EEPROM 24c256 et un écran OLED SSD1306. Et tout cela fonctionne sans le moindre problème avec le firmware i2c-tiny-usb pour RP2040, sous Linux comme sous FreeBSD.

3. UTILISATION DE L'INTERFACE I2C

En guise d'exemple et pour pousser le raisonnement jusqu'au bout, nous allons à présent nous pencher sur la manière d'accéder à ce nouveau bus. Notez bien que ceci n'est aucunement spécifique à i2c-tiny-usb ou même au CP2112 et que vous pourrez faire très exactement la même chose avec un bus i2c intégré à un SBC. De plus, nous utilisons ici du C puisque l'objectif est d'expérimenter dans le but de produire du code qui, au final, est destiné à être porté sur un microcontrôleur, et parce que c'est mon langage préféré, n'en déplaise aux amateurs de crustacés ou de reptiles. Ces derniers trouveront d'ailleurs un script `example/ssd1306.py` dans le dépôt de rp2040-i2c-interface, ce qui devrait pleinement les satisfaire...

Pour faire connaissance avec l'utilisation du bus i2c depuis un système GNU/Linux, nous allons rester dans le classique avec un simple module RTC équipé d'une puce DS3231. Ceci nous permettra de vérifier le bon fonctionnement de l'ensemble en obtenant des données connues et vérifiables. La RTC aura, bien entendu, été initialisée avec la date et heure courante par ailleurs (Pi ou montage Arduino) et notre objectif sera donc de simplement extraire ces informations.

Un bus i2c sous Linux est accessible via un pseudofichier `i2c-n` placé dans `/dev` où `n` est un chiffre dépendant de l'ordre dans lequel les différents bus présents auront été détectés au démarrage ou par la suite. Dans notre cas, il est fort probable que ce chiffre soit tout bonnement le plus important de la liste, mais l'utilisation de `i2cdetect -l` vous donnera l'information explicitement.

L'accès au périphérique (le bus) se fera simplement en accédant à l'entrée `/dev` correspondante et en utilisant les opérations classiques de lecture et d'écriture sur les fichiers, ainsi qu'un certain nombre d'IOCTL à notre disposition. Notez qu'il est possible de n'utiliser que les IOCTL pour communiquer avec un composant sur le bus, mais que la méthode la plus courante est celle que nous verrons ici.

La première chose à faire pour accéder au bus est d'ouvrir le fichier :

```
if ((fd = open(optdevice, O_RDWR)) < 0) {
    free(optdevice);
    err(EXIT_FAILURE, "Error opening device");
}
```

`optdevice` provient de la gestion des options avec `getopt()` et est une simple chaîne de caractères. Si vous rencontrez des problèmes à ce niveau, et en particulier en rapport avec les permissions, vous pourrez invoquer votre programme en `root` via `sudo` ou `doas`, ajuster la configuration *udev* en ajoutant une règle, ou plus raisonnablement avec Debian GNU/Linux et consorts, placer l'utilisateur courant dans le groupe `i2c`. Une fois le fichier ouvert en lecture/écriture, nous pouvons spécifier avec quel composant nous comptons dialoguer, via son adresse sur le bus. Dans le cas du DS3231, c'est `0x68` :

```
#define DS3231_ADDR  0x68
[...]
int i2caddr = DS3231_ADDR;
[...]
    if (ioctl(fd, I2C_SLAVE, i2caddr) < 0) {
        close(fd);
        err(EXIT_FAILURE, "Error changing slave address");
    }
```

Ceci fait, toutes les opérations suivantes de lecture et d'écriture concerneront ce composant. Nous utiliserons deux *buffers* pour les échanges, un pour l'écriture (`buf[10]`) et l'autre pour la réception (`rbuf[10]`). Avec le DS3231, les informations qui nous intéressent sont stockées dans des registres de 8 bits numérotés de `0x00` à `0x12`. On trouve là aussi bien la date et l'heure courante que la configuration des alarmes, celle générale du composant lui-même ou encore une mesure de température.

Pour accéder à un ou plusieurs registres, nous devons écrire la valeur du registre de départ de la future opération de lecture, puis lire un ou plusieurs octets correspondant au registre initial et les suivants. Nous allons donc commencer par récupérer la date se trouvant stockée dans les registres `0x03` à `0x06` (jour de la semaine, jour, mois, année) :

```
    buf[0] = 3;
    if (write(fd, buf, 1) != 1) {
        close(fd);
        err(EXIT_FAILURE, "Error writing to DS3231");
    }
    usleep(1500);
    if (read(fd, rbuf, 4) != 4) {
        close(fd);
        err(EXIT_FAILURE, "Error talking to the device");
    }
    printf("Date: %s %02u/%02u/%04u\n",
       semaine[rbuf[0]],
       (rbuf[1] & 0x0f) + (rbuf[1] >> 4) * 10,
       (rbuf[2] & 0x0f) + ((rbuf[2] >> 4) & 0x01) * 10,
       ((rbuf[3] & 0x0f) + ((rbuf[3] >> 4) & 0x0f) * 10) + 2000
       );
```

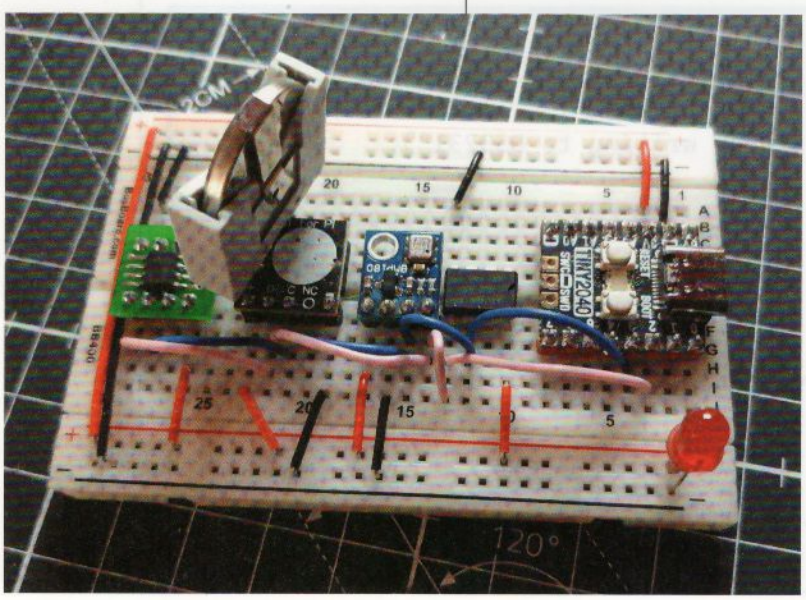
Comme vous pouvez le voir, nous écrivons `0x03` et lisons 4 octets, puis décodons (format BCD) le contenu de `rbuf` pour formater les données correctement, en suivant les indications dans la *datasheet* **[8]**. Nous faisons de même avec l'heure :

```
buf[0] = 0;
if (write(fd, buf, 1) != 1) {
    close(fd);
    err(EXIT_FAILURE, "Error writing to DS3231");
}
usleep(1500);
if (read(fd, rbuf, 3) != 3) {
    close(fd);
    err(EXIT_FAILURE, "Error talking to the device");
}
printf("Heure: %02u:%02u:%02u\n",
    (rbuf[2] & 0x0f) + (rbuf[2] >> 4) * 10,
    (rbuf[1] & 0x0f) + (rbuf[1] >> 4) * 10,
    (rbuf[0] & 0x0f) + (rbuf[0] >> 4) * 10
    );
```

Bien entendu, cette approche en deux étapes est purement démonstrative. Nous aurions tout aussi bien pu lire 7 octets de `0x00` à `0x06`, mais je voulais présenter autre chose qu'une simple lecture séquentielle.

Chaque composant interfacé en i2c possède sa propre structure interne et une façon bien précise d'obtenir des informations ou de déclencher des actions. Ici, nous avons procédé à une lecture en spécifiant un registre de départ et toute lecture consécutive incrémentait un pointeur d'adresse automatiquement, bouclant à `0x12` pour revenir à `0x00`. Tous les composants i2c ne fonctionnent pas nécessairement de cette manière, loin de là, et chacun d'eux est différent. Tout est généralement relativement bien décrit dans leur documentation, si celle-ci est effectivement disponible (certaines ne sont accessibles que sous NDA, malheureusement). Une autre source d'information peut être le code du noyau lui-même ou encore différentes implémentations pour

Le principal problème des modules RTC très compacts, comme ceux généralement utilisés pour les Raspberry Pi, est la pile soudée au circuit. Lorsque celle-ci est épuisée, le module devient inutilisable et on n'a d'autre choix que d'en acheter un autre ou, comme ici, d'opter pour un hack barbare utilisant un support de pile bouton.



l'embarqué disponibles sur GitHub ou GitLab (y compris lorsqu'une *datasheet* n'est pas publiquement disponible, **tousse** ArduinoECCX08 **tousse**).

4. VARIATION AVEC FREEBSD

GNU/Linux est le système par excellence lorsqu'on développe pour l'embarqué ou un microcontrôleur. Énormément de périphériques sont pris en charge, les environnements de développement sont généralement compatibles (voire dédiés) et les outils disponibles, quelles que soient ses préférences, sont légion. Mais on voit également que FreeBSD commence petit à petit à se faire une place auprès d'utilisateurs souhaitant davantage de maîtrise sur leur système (dont moi).

FreeBSD, comme Linux, met à disposition des points d'accès aux bus i2c en espace utilisateur via des entrées `/dev/iic*` et propose un outil similaire à `i2cdetect`, appelé sobrement `i2c`. Et, bien sûr, tout comme avec Linux, nous pouvons accéder aux composants sur un bus en manipulant ces pseudofichiers (autorisés en écriture pour les membres du groupe `wheel`). Nous pourrions également utiliser les appels système `read`/`write`, mais je vous propose de voir une autre approche, entièrement basée sur les IOCTL. Toute la partie ouverture du fichier est strictement identique au code pour Linux (merci POSIX), mais la manière de préparer les données est différente dans ce cas.

Nous allons utiliser deux structures, la première pour stocker les messages du bus i2c et la seconde pour gérer la transaction en lecture/écriture. Nous avons toujours nos deux *buffers*, mais la manière de les utiliser change :

```
struct iic_msg msg[2];
struct iic_rdwr_data rdwr;

// message 1
msg[0].slave = i2caddr << 1;
msg[0].flags = IIC_M_WR;
msg[0].len = 1;
msg[0].buf = buf;

// message 2
msg[1].slave = i2caddr << 1;
msg[1].flags = IIC_M_RD;
msg[1].len = 1;
msg[1].buf = rbuf;

// 2 messages
rdwr.nmsgs = 2;
rdwr.msgs = msg;
```

Le premier message correspond à l'écriture sur le bus (drapeau `IIC_M_WR`), permettant de spécifier le registre de départ, et le second est l'opération de lecture (`IIC_M_RD`). La taille des données par défaut est ici spécifiée à 1 octet et les deux *buffers* sont respectivement référencés en initialisant le membre `buf` avec le pointeur correspondant. Ces deux messages sont ensuite utilisés pour configurer la transaction.

Bien sûr, comme nous procédons de la même manière qu'avec le code précédent, nous ajustons le contenu de `buf[]` ainsi que la taille des données attendues en lecture (`msg[1].len`) juste avant d'utiliser l'IOCTL `I2CRDWR` :

```
buf[0] = 3;
msg[1].len = 4;
if (ioctl(fd, I2CRDWR, &rdwr) < 0) {
    close(fd);
    err(EXIT_FAILURE, "Error talking to the device");
}
printf("Date: %s %02u/%02u/%04u\n",
       semaine[rbuf[0]],
       (rbuf[1] & 0x0f) + (rbuf[1] >> 4) * 10,
       (rbuf[2] & 0x0f) + ((rbuf[2] >> 4) & 0x01) * 10,
       ((rbuf[3] & 0x0f) + ((rbuf[3] >> 4) & 0x0f) * 10) + 2000
       );
```

Et nous faisons exactement de même pour récupérer l'heure courante :

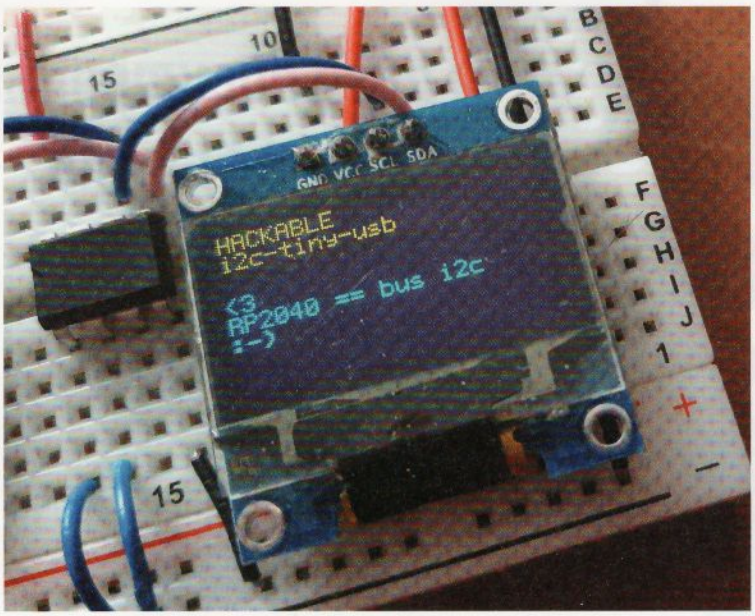
```
buf[0] = 0;
msg[1].len = 3;
if (ioctl(fd, I2CRDWR, &rdwr) < 0) {
    close(fd);
    err(EXIT_FAILURE, "Error talking to the device");
}
printf("Heure: %02u:%02u:%02u\n",
       (rbuf[2] & 0x0f) + (rbuf[2] >> 4) * 10,
       (rbuf[1] & 0x0f) + (rbuf[1] >> 4) * 10,
       (rbuf[0] & 0x0f) + (rbuf[0] >> 4) * 10
       );
```

Notez que vous pouvez également procéder de la sorte sous Linux puisque la structure `iic_msg` est compatible avec `i2c_msg` sur ce système. La piste NetBSD n'a pas été explorée (pour l'instant), pas plus que celle d'OpenBSD qui n'offre pas d'accès *userland* pour les bus i2c.

CONCLUSION

La réimplémentation d'i2c-tiny-usb sur une base RP2040 est, je trouve, une fantastique découverte, qui va grandement améliorer mon confort de développement pour la prise en charge de futurs composants i2c ne disposant pas pour l'heure de bibliothèque dédiée dans un environnement comme ESP-IDF ou Pico-SDK. Mais en réalité, le fait de disposer d'un bus i2c facilement accessible, pour le modique coût de devoir temporairement immobiliser une carte Pico, offre de nombreux autres avantages.

Si nous reprenons notre bête exemple avec une RTC DS3231, par exemple, développer rapidement un outil permettant non pas de lire l'heure, mais d'initialiser le composant, sur la base de la date et heure courante du système, peut grandement nous faciliter la vie. Plus besoin de passer par une Pi ou de devoir gérer une initialisation manuellement via la saisie de valeurs, ni même de commencer à jouer avec NTP/SNTP (cf. l'article sur le lever/coucher de soleil dans



Cet écran OLED de 128 par 64 pixels est piloté par une puce SSD1306 interfacée en i2c. L'utilisation d'un pont USB/i2c est absolument idéale pour la mise au point de programmes en faisant usage. Il suffira ensuite de porter le code sur microcontrôleur et le tout fonctionnera à l'identique.

le présent numéro). Il suffit de connecter la Pico, d'ajouter la RTC au bus, d'exécuter l'outil et le tour est joué. D'ailleurs, j'ai trouvé l'idée tellement plaisante que j'ai finalement créé un tel petit outil **[9]** en finissant le présent article.

Ceci est également potentiellement applicable aux EEPROM sur i2c (type Atmel 24c256, par exemple), pouvant être ainsi initialisées avec des données, sur PC, avant de compléter un montage à base de MCU. C'est la porte ouverte à énormément de projets. Merci Till Harbaum et merci Nicolai Electronics ! **DB**

RÉFÉRENCES

[1] **https://www.silabs.com/documents/public/data-sheets/cp2112-datasheet.pdf**

[2] **https://github.com/armlabs/ssd1306_linux**

[3] **https://github.com/harbaum/I2C-Tiny-USB**

[4] **http://matrixstorm.com/avr/tinyusbboard/**

[5] **https://github.com/Nicolai-Electronics/rp2040-i2c-interface**

[6] **https://gitlab.com/0xDRRB/kernfbsd_i2ctinyusb**

[7] **https://github.com/freebsd/freebsd-src/pull/1123**

[8] **https://www.analog.com/media/en/technical-documentation/data-sheets/DS3231.pdf**

[9] **https://gitlab.com/0xDRRB/dsrtcinit**

LEVER ET COUCHER DE SOLEIL SUR ESP32

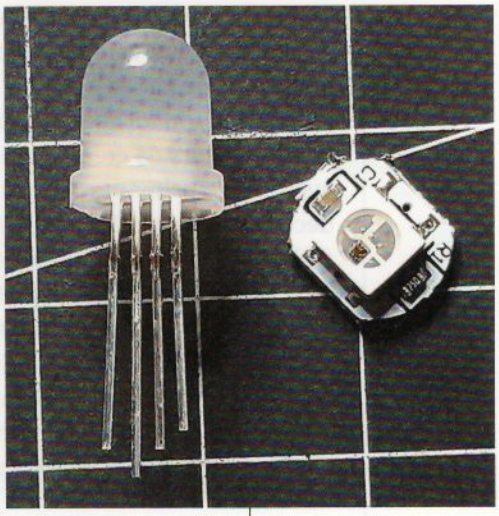
Denis Bodor

Ce projet peut être utile pour deux catégories de personnes, ceux qui font du vélo tôt le matin et veulent profiter du spectacle qu'offre notre étoile locale au matin, et ceux qui sont des créatures de la nuit susceptibles d'être détruites ou pétrifiées à l'aube. Je suis dans l'une de ces deux catégories et je ne vous dirai pas laquelle. Quoi qu'il en soit, pour nombre de bonnes ou mauvaises raisons, il peut être intéressant de savoir, d'un coup d'œil, quand le soleil va se lever et se coucher, et c'est précisément l'objet du présent projet.

– Lever et coucher de soleil sur ESP32 –

Cette réalisation soulève un certain nombre de problématiques. Nous avons bien sûr l'algorithme de base, permettant le calcul des positions respectives des éléments et donc des instants précis des événements qui nous intéressent, mais ça c'est quelque chose qui est largement demandé, répondu et traité de par le Web, mais nous avons surtout des choses plus pratiques en rapport avec les fonctionnalités de la plateforme choisie. Ici, j'ai opté pour de l'ESP32 et plus précisément un ESP32-C3 à cœur RISC-V. Les coûts de ce type de plateformes, aujourd'hui, sont tellement ridicules (parfois moins de 2 euros pour un clone de NodeMCU) qu'il n'est même pas nécessaire de se poser la question de savoir si un Arduino, un ESP8266 ou encore une RPi Pico devraient être envisagés.

Le principal avantage de l'ESP32 se divise en deux éléments. D'une part, le support du Wi-Fi est directement intégré (comme pour un ESP8266), de l'autre, nous pouvons également nous reposer sur un environnement de développement mature et qui ne cesse de s'étoffer et d'évoluer. Et je parle ici d'un **vrai** environnement, donnant accès direct au *framework* du constructeur, sans avoir à passer par une couche intermédiaire comme celle imposée par les bibliothèques (et l'IDE simpliste) Arduino. Que vous soyez *old school*, comme moi, avec votre ligne de commande et votre éditeur Vi (NeoVim, pour être précis) ou préférez quelque chose de plus intégré et graphique comme VSCode avec PlatformIO, l'environnement ESP-IDF offre, je trouve, un bon équilibre.



Les LED adressables se déclinent en bien des versions et formats, mais toutes ne sont pas directement supportées par toutes les plateformes. À gauche, une soi-disant APA106 8 mm et à droite un module utilisant une WS2812b.

Parmi les fonctionnalités qui nous intéressent ici, nous avons premièrement la connectivité à Internet. Plutôt que de devoir reposer sur un composant supplémentaire comme un module RTC, nous allons récupérer l'heure exacte via le Net en utilisant le protocole SNTP, puis garder l'heure synchronisée automatiquement pour afficher des données quotidiennement. Étant donné les valeurs manipulées, la précision n'est pas critique et nous arrondissons de toute manière à la minute. Il est d'ailleurs assez intéressant de chercher les heures de lever et de coucher de soleil pour le jour sur le Web et de se rendre compte que des variations de l'ordre d'une, deux, voire trois minutes ne sont pas rares, d'un site à l'autre. Nous, nous voulons simplement afficher une indication sommaire étant donné que, que vous soyez un vampire ou un cycliste, bien d'autres paramètres entrent en jeu pour capter le bon moment (ou y échapper).

À propos d'affichage justement, nous avons une pléthore de solutions, qu'il s'agisse d'un affichage sur le bureau ou de quelque chose de plus massif : LCD, *e-paper*, LED, écran OLED, etc. Mon objectif est d'avoir un dispositif mural et donc un affichage de grande taille, de préférence agréable à regarder, tout en restant modulable et évolutif. Les afficheurs 7 segments à LED offrent un rendu plaisant, mais

dès lors qu'on cherche quelque chose de 6 à 8 centimètres de haut, ou plus, cela se complique un peu. Si vous ajoutez à cela le fait de pouvoir piloter la couleur de chaque segment (ou chiffre), on touche à l'impossible. La solution la plus évidente est alors de créer soi-même un tel afficheur, que ce soit avec une imprimante 3D ou, comme ici, avec une graveuse/découpeuse laser. Ceci permet de créer une structure de base où l'éclairage des segments est le fait de LED adressables (type WS2812b, APA106, SK6812, HD108, etc.). Cette approche « maison » permet non seulement d'avoir une totale liberté de conception, mais nous évite également de devoir passer par un circuit intégré dédié comme un max7219 ou un registre à décalage, l'ESP32 prenant très bien cela en charge. Et c'est d'ailleurs le premier point à couvrir.

1. PILOTER DES LED ADRESSABLES AVEC UN ESP32

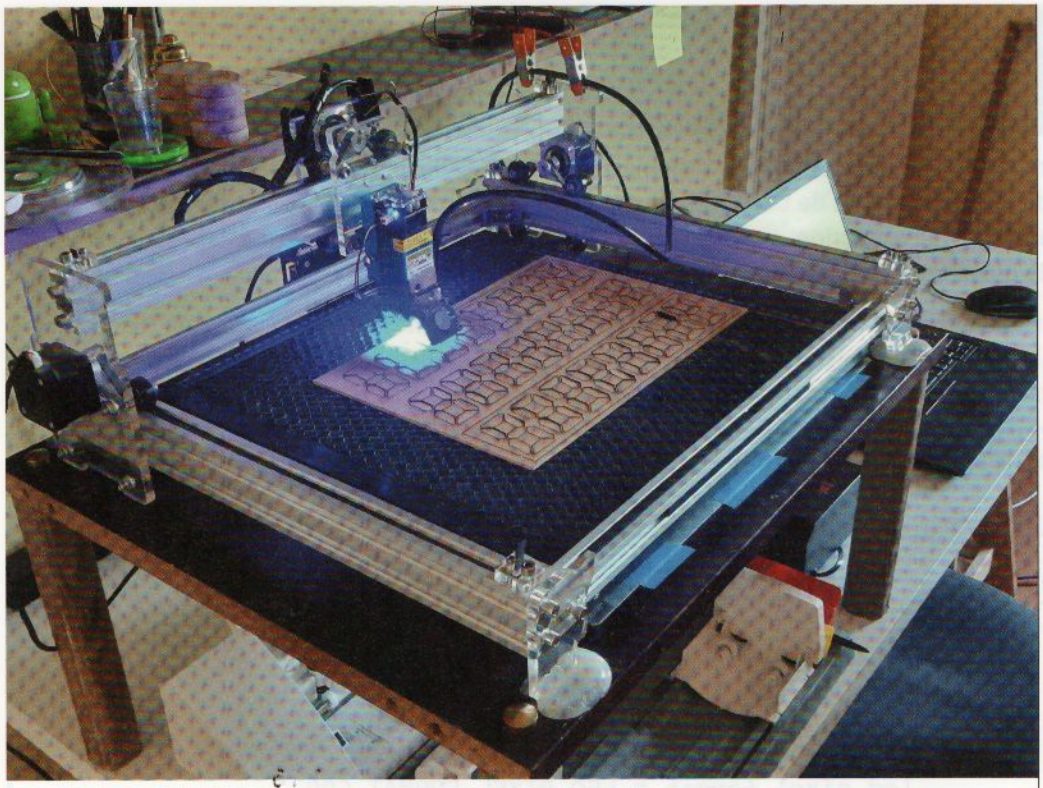
Dans l'écosystème Arduino, lorsqu'on se penche sur des WS2812b par exemple, deux noms viennent immédiatement à l'esprit : Adafruit NeoPixel **[1]** et FastLed **[2]**. Adafruit NeoPixel est une bibliothèque purement Arduino servant principalement de support pour les produits Adafruit utilisant le nom commercial, inventé de toutes pièces, que ce revendeur attribue arbitrairement à toutes sortes de LED adressables dans son catalogue. Fastled est plus « ouvert », plus complet, plus propre et surtout plus performant, mais aucun support ESP-IDF n'est officiellement développé. Il existe quelques tentatives de portages sur GitHub, mais la plupart n'ont pas vu un *commit* depuis au moins quatre bonnes années.

En soi, ceci n'est pas réellement un problème, car l'une comme l'autre solution existant pour Arduino va bien plus loin que le simple pilotage de ces composants. Ceci est particulièrement vrai pour FastLed qui s'apparente davantage à une bibliothèque graphique haute performance qu'à un simple support pour les WS2812b et consorts. Nous, pour ce projet et pour la plupart des choses qu'on peut vouloir faire avec des LED adressables, n'avons besoin que du strict minimum : attribuer une couleur à une LED ponctuellement, sur la base des trois composantes primaires que sont le rouge, le vert et le bleu. Point d'animation, d'ajustement ou de correction chromatique, de *framerate* infernal ou de manipulation de matrices. Juste attribuer des couleurs à quelque 56 LED, une fois par heure, au maximum.

Pour cela, l'ESP-IDF fournit déjà le nécessaire sous la forme d'un périphérique très intéressant : l'encodeur RMT (*Remote Control Transceiver*). Comme son nom l'indique, l'encodeur est initialement prévu pour piloter des émetteurs à infrarouge et transmettre un signal pouvant être, par exemple, reçu et décodé par le très connu TSOP1738. À noter que le périphérique RMT peut également fonctionner en réception et démoduler un tel signal.

Quel rapport avec une LED adressable ? Un signal IR envoyé par une télécommande est un signal modulé sur la base d'une fréquence de 30 kHz à 56 kHz (pour assurer une immunité aux signaux parasites comme l'éclairage ou les reflets), mais ceci n'a pas grande importance ici. C'est une succession rapide d'émissions du signal avec une absence de ce dernier qui encode l'information. Retirez la modulation (ce que fait le TSOP1738) et vous vous retrouvez avec un encodage tout ou rien.

Une WS2812b, pour sa part, utilise un signal carré basé sur une période à l'état



La gravure laser, ou plus exactement ici, la découpe laser, est une option et une alternative intéressante à l'impression 3D pour réaliser des boîtiers, des panneaux, etc.

haut et une période à l'état bas pour encoder un 1 ou un 0. Un rapide coup d'œil à la *datasheet* de ce composant nous montre que 0,4 µs à l'état haut (T0H) et 0,85 µs à l'état bas (T0L) est un 0, et que 0,8 µs à l'état haut (T1H) et 0,45 µs (T1L) à l'état bas est un 1. Tout état bas qui dépasse 50 µs est un *reset*.

En désactivant la modulation, l'encodeur RMT peut donc être utilisé pour transmettre non pas un signal à une LED infrarouge, mais à une LED adressable comme la WS2812b, et donc, puisque ce genre de composant est « chaînable », à toute une collection de WS2812b.

L'environnement ESP-IDF fournit plusieurs exemples utilisant le périphérique RMT dans le répertoire `examples/peripherals/rmt/`. On trouve là, entre autres, un émetteur-récepteur IR, une interface 1-Wire, un pilotage de moteur pas-à-pas et, bien entendu, un exemple d'utilisation avec des LED WS2812 (sans « b »). J'avoue que mon premier réflexe a été de me baser sur cet exemple pour mon projet, jusqu'à avoir un résultat pleinement fonctionnel, avant de finalement changer d'avis et de basculer sur quelque chose de plus simple, dans tous les sens du terme.

Espressif propose, en effet, un système de gestion de composants logiciels dans l'environnement, permettant à la fois de moduler son code, mais aussi de reposer sur des développements tiers. Nous en avons d'ailleurs parlé dans le numéro 53 **[3]**. Le site *ESP Registry* **[4]** référence les plus populaires, stables et actifs, tout en permettant une installation automatique via une simple commande. C'est là qu'on trouve le composant `led_strip` **[5]** qu'on intégrera sans problème avec un simple :

```
$ idf.py add-dependency "espressif/led_strip^2.5.3"
Executing action : add-dependency
Created "/[...]/esp32sunrise/main/idf_component.yml"
```

À noter qu'une autre implémentation existe dans la *ESP-IDF Components library* **[6]**, supportant davantage de types de LED et assez similaire dans sa structure et son utilisation. Le choix de l'une ou l'autre solution, sachant que le `led_strip` ici installé provient d'Espressif, est une simple question de préférence.

Certes, dans les deux cas, nous sommes loin de quelque chose d'aussi étoffé que Fastled, mais pour la plupart des usages, c'est largement suffisant. Initialiser le périphérique RMT via cette bibliothèque Espressif est relativement aisé, tout comme l'accès aux valeurs RVB des LED :

```
#include <led_strip.h>

led_strip_handle_t led_strip;

[...]

    // init leds strip
    led_strip_config_t strip_config = {
        .strip_gpio_num = LED_STRIP_GPIO_NUM,
        .max_leds = LED_NUMBERS,
        .led_pixel_format = LED_PIXEL_FORMAT_GRB,
        .led_model = LED_MODEL_WS2812,
        .flags.invert_out = false,
    };

    led_strip_rmt_config_t rmt_config = {
        .clk_src = RMT_CLK_SRC_DEFAULT,
        .resolution_hz = 10 * 1000 * 1000, // 10MHz
        .flags.with_dma = false,
    };
    ESP_ERROR_CHECK(
      led_strip_new_rmt_device(
          &strip_config,
          &rmt_config,
          &led_strip));

[...]

  led_strip_set_pixel(led_strip, 42, 128, 64, 32);
  led_strip_refresh(led_strip);
```

`LED_STRIP_GPIO_NUM` et `LED_NUMBERS` sont définis par ailleurs (`main.h`) et proviennent de macros prises en charge par le système de configuration d'ESP-IDF (`main/Kconfig.projbuild`). Comme pour d'autres périphériques (par exemple, le générateur de fréquences utilisé pour l'accès

aux *smartcards* que je décris dans l'article sur ce sujet dans ce même numéro), la configuration se décompose en deux parties avec d'un côté la configuration du canal utilisé pour la sortie, et de l'autre le périphérique RMT lui-même. Ceci fait, choisir une couleur pour une LED se résume à un simple appel à `led_strip_set_pixel()` qui impacte le « framebuffer » et `led_strip_refresh()` pour « pousser » le contenu sur la série de LED.

2. AFFICHEUR 7 SEGMENTS À BASE DE WS2812B

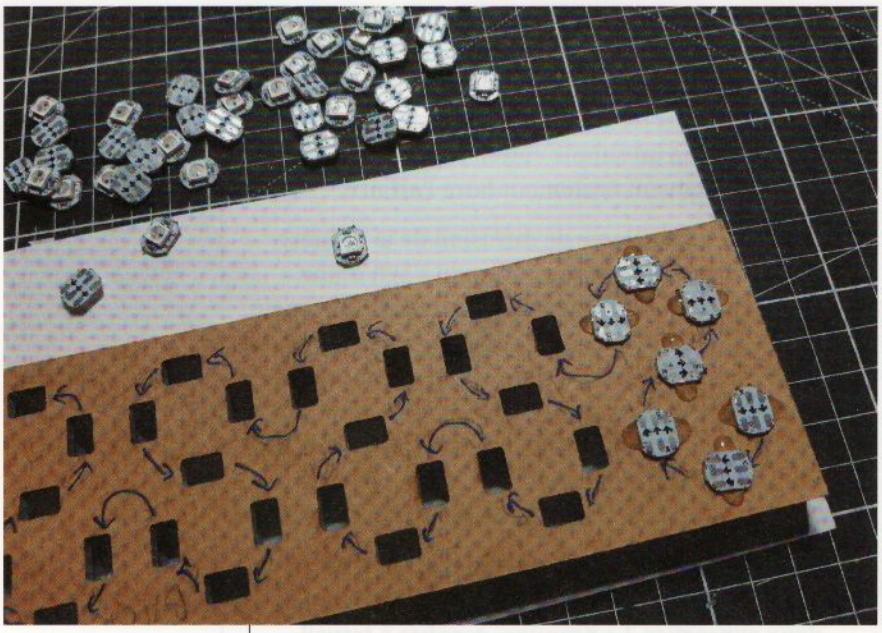
Le principe de l'afficheur en lui-même est assez simple. Il s'agit d'avoir une structure présentant les différents segments agencés de manière à obtenir 8 chiffres sur lesquels sont présentées heures et minutes du lever, et idem pour le coucher. J'ai utilisé une graveuse/découpeuse laser EleksMaker A3 Pro, modifiée depuis achat par un remplacement du bloc laser (15 W), un plan en nid d'abeille et un kit *air assist*, pour obtenir cette structure en découpant trois plaques de MDF 5 mm qui ont ensuite été collées ensemble. Ceci avant de compléter d'une quatrième plaque, gravée avec les motifs des segments (pour l'alignement) et découpée pour accueillir de minuscules modules WS2812b.

J'ai opté pour le MDF et la découpe laser, mais il va de soi qu'une imprimante 3D ferait également le travail sans le moindre problème. Et si on est vraiment doué de ses mains, une scie à chantourner ou même un simple cutter et beaucoup de patience peuvent également faire l'affaire.

Régler le problème de diffusion de la lumière est étonnamment simple puisqu'il s'agit d'une simple feuille de papier collée en façade. Ceci peut sembler « bricolo » mais en réalité, après avoir testé d'autres solutions (film vinyle blanc et polycarbonate translucide), le très commun papier 80g donne les meilleurs résultats (c'est salissant, mais un film plastique autocollant, comme celui pour protéger les livres, règle facilement le problème).

Au-delà de l'aspect purement matériel, l'agencement des LED est un point intéressant. J'ai testé deux approches. La première consiste à travailler ligne par ligne en « Z » avec les huit segments supérieurs des chiffres chaînés de gauche à droite, les 16 segments de la ligne du dessous de droite à gauche, les 8 segments du milieu de gauche à droite et ainsi de suite. Ceci se prête bien à l'utilisation de LED 8 mm comme les APA106 (non supportées par le composant `led_strip` Espressif), où l'on peut utiliser les pattes des LED pour relier DATA IN et DATA OUT.

Pour obtenir du volume à partir de simples plaques de MDF (médium), la technique est toute simple : il suffit de multiplier les couches et de tout coller avec de la colle à bois.



L'agencement des LED est une affaire de choix et de préférences, puisque tout est géré de manière transparente par le code. Le motif en « S » est cependant une excellente approche si on compte jouer sur des dégradés de couleurs « transchiffres ».

Pour les mini-modules WS2812b utilisés au final, l'idée est de travailler chiffre par chiffre, en commençant au segment supérieur gauche et en chaînant en « S » pour finir avec le segment inférieur gauche. Ceci permet de repartir sur le segment inférieur droit du chiffre suivant, en « S » inversé, pour se connecter au troisième chiffre, et ainsi de suite.

La manière de chaîner les LED importe peu en réalité, car c'est quelque chose que nous allons gérer de manière purement logicielle. Ce qui est important, en revanche, c'est d'assurer une bonne distribution du courant en connectant masse et tension d'alimentation en plusieurs points de la chaîne, et en ajoutant un condensateur électrolytique d'assez grande capacité en parallèle à GND/VCC (les modules disposent déjà d'un condensateur céramique et d'une résistance de protection).

Pour créer chaque chiffre de 7 segments, nous allons établir une correspondance entre l'indice de la LED utilisée avec le segment concerné, sous la forme d'une table (*lookup table*). Nous commençons avec un chiffre structuré ainsi :

```
    a
   ---
f|   |b
   --- g
e|   |c
   ---
    d
```

Dans la table que nous allons créer, l'ordre des segments est : *a*, *f*, *b*, *g*, *e*, *c* et *d*. Nous aurons donc 8 fois 7 valeurs d'indice correspondant à la position des LED dans la chaîne. Pour l'agencement en « S », ceci nous donne :

```
uint8_t digit[8][7] = {
    { 54, 55, 53, 52, 51, 49, 50 }, // 0
    { 43, 44, 42, 45, 48, 46, 47 }, // 1
    { 40, 41, 39, 38, 37, 35, 36 }, // 2
    { 29, 30, 28, 31, 34, 32, 33 }, // 3
    { 26, 27, 25, 24, 23, 21, 22 }, // 4
    { 15, 16, 14, 17, 20, 18, 19 }, // 5
    { 12, 13, 11, 10,  9,  7,  8 }, // 6
    {  1,  2,  0,  3,  6,  4,  5 }  // 7
};
```

Ainsi, pour afficher un « 8 » sur le quatrième chiffre, il nous suffit de choisir une valeur de rouge, vert et bleu pour toutes les LED de la ligne 3 : 29, 30, 28, 31, 34, 32 et 33. Nous pouvons alors dans la foulée créer une police de caractères pour les chiffres de « 0 » à « 9 » et les lettres de « A » à « F » (l'hexadécimal n'est pas utile ici, mais ça ne coûte rien de le gérer en plus). Nous avons 7 segments par chiffre, donc plutôt que de gâcher de la mémoire pour rien, chaque segment peut être un simple bit (toujours en gardant le même ordre de segments « afbgecd ») :

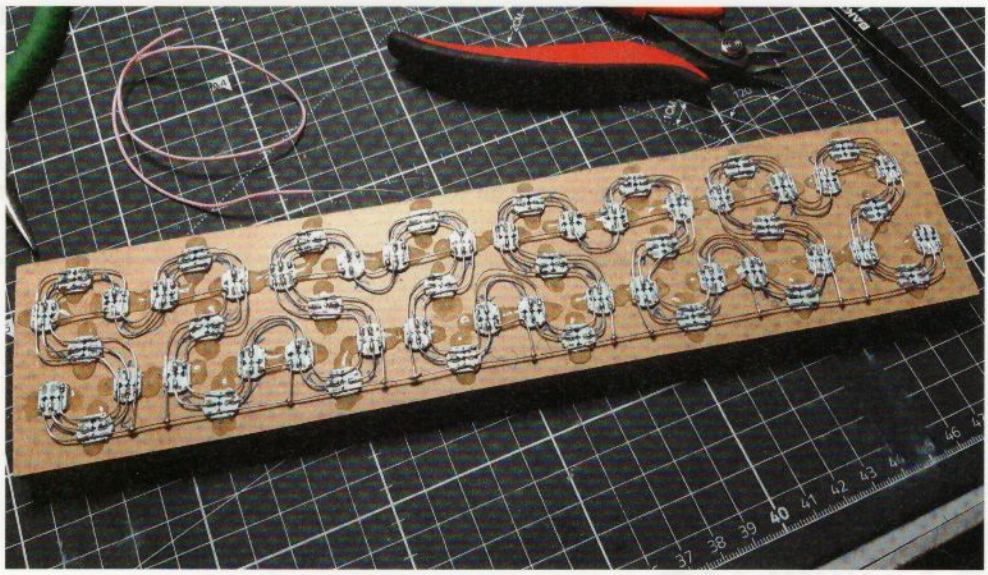
```
0 = 1110111 = 0x77
1 = 0010010 = 0x12
2 = 1011101 = 0x5d
3 = 1011011 = 0x5b
4 = 0111010 = 0x3a
5 = 1101011 = 0x6b
6 = 1101111 = 0x6f
7 = 1010010 = 0x52
8 = 1111111 = 0x7f
9 = 1111011 = 0x7b
A = 1111110 = 0x7e
B = 0101111 = 0x2f
C = 1100101 = 0x65
D = 0011111 = 0x1f
E = 1101101 = 0x6d
F = 1101100 = 0x6c
```

Ce qui se transforme en une nouvelle table pour le dessin de chaque caractère, où la position de l'octet correspond à la valeur à afficher :

```
uint8_t digifont[16] = {
//     0     1     2     3     4     5     6     7
    0x77, 0x12, 0x5d, 0x5b, 0x3a, 0x6b, 0x6f, 0x52,
//     8     9     A     B     C     D     E     F
    0x7f, 0x7b, 0x7e, 0x2f, 0x65, 0x1f, 0x6d, 0x6c
};
```

Poussons le vice un poil plus loin. Nous avons ici 56 LED RVB et rien ne nous oblige à choisir la couleur de chaque segment individuellement. Nous pouvons travailler avec des couches ou calques, exactement comme avec un logiciel de retouche comme The Gimp. Si nous partons du principe qu'afficher un chiffre revient à copier la valeur de couleur du fond vers le premier plan, nous pouvons avoir une carte de 56 segments de couleurs prédéterminées comme source et la forme des chiffres comme destination. Vous pouvez voir cela comme un masque de calque dans The Gimp.

Ceci nous permet, par exemple, de créer un dégradé sur l'ensemble des segments dans un sens ou dans l'autre, sans nous soucier de la forme des symboles qui seront affichés. De la même manière, nous pouvons également utiliser plusieurs « fonds » avec des nuances différentes et choisir lequel est utilisé, sans avoir à nous préoccuper des couleurs de chaque segment en affichant les symboles.



La soudure des LED, précédemment sécurisées avec un peu de résine UV, est une étape pénible (pour le dos) mais relativement simple. On prendra toutefois soin de relier VCC et masse en plusieurs points du circuit pour éviter les tensions parasites.

Pour nous simplifier la vie, nous créons un nouveau type de données basé sur la structure suivante :

```
typedef struct RGBcolor_t {
        uint8_t r;
        uint8_t g;
        uint8_t b;
} RGBcolor;
```

Créer un fond, que nous appelons désormais notre `colormap`, revient à déclarer un tableau de 56 `RGBcolor` :

```
RGBcolor colormap[LED_NUMBERS] = { 0 };
```

Ensuite, pour remplir cette `colormap`, nous pouvons créer tout un tas de fonctions utilisant le tableau `digit[][]` pour déterminer la position des LED pour chaque segment. Nous pouvons également faire fi de cette information, et grâce à l'agencement en « S », remplir la `colormap` avec un dégradé sur 56 positions. Les options sont nombreuses et les implémentations dépendent de vos besoins. À titre d'exemple, voici une fonction déterminant la couleur pour un chiffre :

```
void setdigitcolor(RGBcolor *map, int pos,
  uint8_t r, uint8_t g, uint8_t b)
{
    for (int i = 0; i < 7; i++) {
        map[digit[pos][i]].r = r;
        map[digit[pos][i]].g = g;
        map[digit[pos][i]].b = b;
    }
}
```

Et voici celle permettant d'utiliser un dégradé dans le modèle de couleur TSV (Teinte / Saturation / Valeur) :

```
void gengradhsv(RGBcolor *map, uint32_t startH, uint32_t stopH,
    uint32_t s, uint32_t v, bool allseg)
{
    int i, j;
    int jump = LED_NUMBERS / DIGIT_NUMBERS;

    if (s > 100) s = 100;
    if (v > 100) v = 100;
    if (allseg) jump = 1;

    for (i = 0; i < LED_NUMBERS; i += jump) {
        if (stopH >= startH) { // clockwise
            for (j = 0; j < jump; j++) {
                hsv2rgb(startH + ((stopH - startH) * i) / LED_NUMBERS,
                  s, v, &(map[i + j].r), &(map[i + j].g), &(map[i + j].b));
            }
        } else { // counterclockwise
            for (j = 0; j < jump; j++) {
                hsv2rgb(startH - ((startH - stopH) * i) / LED_NUMBERS,
                  s, v, &(map[i + j].r), &(map[i + j].g), &(map[i + j].b));
            }
        }
    }
}
```

Celle-ci est un peu plus complexe, car elle se veut relativement polyvalente. En premier lieu, elle utilise une fonction `hsv2rgb()` qu'il n'est pas intéressant de lister ici (c'est la même qu'on trouve partout sur le Net) et permet, à partir d'un jeu d'arguments de teinte, de saturation et de valeur, d'obtenir les valeurs de rouge, de vert et de bleu correspondantes (entre 0 et 255). L'astuce ici consiste à obtenir ce qu'on veut à partir des deux arguments de teinte que sont `startH` et `stopH` (avec « H » pour *Hue*, la teinte). En espace TSV, la teinte est représentée par une valeur en degrés entre 0 et 360. Comme nous utilisons une comparaison entre `startH` et `stopH` pour déterminer si nous devons incrémenter ou décrémenter pour aller de `startH` à `stopH`, on pourrait penser que certains dégradés ne sont pas possibles.

Si nous prenons l'exemple où les arguments sont respectivement 90 et 270, nous allons incrémenter la teinte vers 270 et, ce faisant, passer du vert (90), au vert-bleu (135), cyan (180), bleu (225) et violet (270). Si nous inversons les valeurs, nous irons du violet au vert, en passant par le cyan, mais nous n'irons pas du violet (270) en passant par le magenta (315), le rouge (0), le jaune (45) pour finir au vert (90). Comment faire ? Faut-il ajouter un argument ? Non. Il suffit de spécifier des valeurs correspondant à plus d'un tour dans le cercle et, ici, spécifier 270 et 450 (ou 450 et 270 pour le sens inverse). Problème réglé.

Une fois les diverses fonctions de remplissage de la `colormap` créées, il ne nous reste qu'à combiner tout cela pour afficher les chiffres :

```
void setdigit(RGBcolor *map, uint8_t pos, uint8_t val)
{
    // si pas [0..9a..f] alors noir
    if (val > 15) {
        for (int i = 0; i < 7; i++) {
            led_strip_set_pixel(led_strip,
               digit[pos][i], 0, 0, 0);
        }
    } else {
        for (int i = 0; i < 7; i++) {
            if (digifont[val] & (0x40 >> i)) {
                led_strip_set_pixel(
                   led_strip,
                   digit[pos][i],
                   map[digit[pos][i]].r,
                   map[digit[pos][i]].g,
                   map[digit[pos][i]].b);
            } else {
                led_strip_set_pixel(led_strip,
                   digit[pos][i], 0, 0, 0);
            }
        }
    }
}
```

C'est un simple jeu de tableau, d'index et de *bitmap* pour la forme des symboles, tout en utilisant `led_strip_set_pixel()` (de `led_strip`) pour définir le contenu du *buffer* d'affichage (auquel nous n'avons pas accès). Pour chaque segment de chaque chiffre, nous testons si le bit correspondant est à `1` dans le symbole. Si c'est le cas, nous copions les valeurs de rouge, vert et bleu de l'emplacement correspondant dans la `colormap` vers le *buffer* à transmettre à la chaîne de LED. Sinon, tout est à 0 et ça va être tout noir (oui, « ta gueule », forcément [7]).

Sur cette fonction, on peut ensuite en construire d'autres, comme celle permettant d'afficher un nombre en décimal :

```
void setnumber(RGBcolor *map, uint32_t num)
{
    if (num > (pow(10, DIGIT_NUMBERS) - 1))
        return;

    unsigned long div =
      (unsigned long)pow(10, (DIGIT_NUMBERS - 1));
    for (int i = 0; i < DIGIT_NUMBERS; i++) {
        setdigit(map, i, num / div);
        num = num - ((num / div) * div);
        div = div / 10;
    }
}
```

Pas d'inquiétude ici à propos du `pow()`, le compilateur voyant qu'on travaille avec des constantes optimisera tout cela de lui-même. Et c'est tant mieux, car faire cela uniquement en langage macro du préprocesseur nécessite de la récursion et de l'aspirine.

Mais ce qui nous intéressera le plus ici, c'est de pouvoir afficher un nombre en spécifiant une position et une taille en nombre de chiffres (pour préfixer d'un « 0 » si nécessaire) :

```
void setpnumber(RGBcolor *map,
  uint8_t pos, size_t len, uint32_t num)
{
    if (num >= pow(10, len))
        return;

    int div = pow(10, len - 1);
    for (int i = 0; i < len; i++) {
        setdigit(map, i + pos, num / div);
        num = num - ((num / div) * div);
        div = div / 10;
    }
}
```

Là, malheureusement `pow()` ne sera pas optimisé, c'est un problème classique lorsqu'on travaille dans une base qui n'est pas « native » à l'informatique (qui sait, peut-être compterait-on en hexadécimal si nous avions huit doigts à chaque main (les pieuvres comptent-elles en octal ?)). Oui, nous pourrions jouer avec `snprintf()`, mais je pense que le résultat serait pire, à moins d'avoir besoin d'un afficheur 16 segments (ce qui est une évolution possible de cette partie du projet).

3. WI-FI, SNTP, RTC ET GESTION DU TEMPS

La connectivité Wi-Fi des ESP32 n'est pas la seule raison pour laquelle je n'ai pas opté pour une plateforme plus basique comme une carte Arduino (AVR). Le *framework* utilisé par ce microcontrôleur, tout comme celui pour l'ESP8266, intègre une gestion de RTC. Je ne parle pas ici de quelque chose comme la présence d'un DS1307/DS3231 au cœur du composant, capable d'utiliser un oscillateur externe et/ou une pile bouton, mais de la présence d'une RTC capable de maintenir une information temporelle durant le fonctionnement d'un programme, sans avoir recours a des fonctions comme `millis()` d'Arduino, qui n'ont rien à voir avec la gestion d'une horloge temps réel au sens « humain » du terme.

Via les bibliothèques standard de l'ESP32 et le *framework* ESP-IDF, nous avons accès à des fonctions que vous reconnaîtrez sans doute, puisqu'il s'agit tout simplement de fonctions POSIX qu'on retrouve sous GNU/Linux et les *BSD. Et quoi qu'en disent certains qui semblent prompts à répondre sans avoir la moindre idée de quoi ils parlent **[8]**, oui, ceci est également accessible sans le moindre problème avec un ESP8266 dans l'environnement Arduino. Ne croyez donc pas tout ce que vous lisez sur le Net, les ignorants y sont généralement bien trop loquaces, surtout dans les forums Arduino...

Quoi qu'il en soit et même si le système maintiendra une date/heure à jour lorsqu'il est en cours de fonctionnement, nous n'avons pas de source fiable pour l'initialisation et/ou une éventuelle rupture d'alimentation. Nous pourrions utiliser une RTC externe comme nous l'avons déjà fait par le passé, mais la connectivité Wi-Fi offre une bien meilleure solution :

SNTP (*Simple Network Time Protocol*). Ce protocole est une déclinaison simplifiée de NTP, mais offre quasiment les mêmes fonctionnalités : obtenir un référentiel temporel exact et également maintenir cette information à jour de façon récurrente en tâche de fond pour pallier une éventuelle dérive de notre horloge interne.

Je n'aborderai pas ici l'aspect « connexion au Wi-Fi », car il s'agit, ni plus ni moins, que d'une reprise du code livré en exemple avec ESP-IDF (`examples/wifi/getting_started/station/`) et placé dans un fichier `wifistuff.c`, pour pouvoir simplement appeler `wifi_init_sta()` depuis `app_main()`. Notez que, par défaut, cette implémentation utilise la mémoire NVS pour stocker les paramètres de connexion et que celle-ci devra donc être initialisée au démarrage avec :

```
ret = nvs_flash_init();
if (ret == ESP_ERR_NVS_NO_FREE_PAGES ||
    ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
    ESP_ERROR_CHECK(nvs_flash_erase());
    ret = nvs_flash_init();
}
ESP_ERROR_CHECK(ret);
```

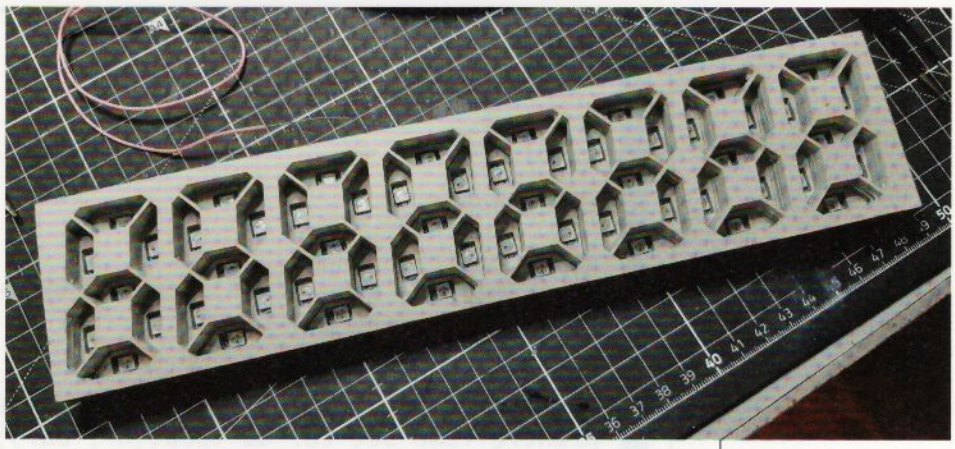
Une fois la connexion Wi-Fi établie, obtenir l'heure et se synchroniser est relativement simple :

```
esp_sntp_config_t config =
ESP_NETIF_SNTP_DEFAULT_CONFIG(CONFIG_ESPSUNRISE_SNTP_SERVER);
config.sync_cb = time_sync_notification_cb;

esp_netif_sntp_init(&config);

printf("Syncing time from %s...\n", CONFIG_ESPSUNRISE_SNTP_SERVER);
if (esp_netif_sntp_sync_wait(pdMS_TO_TICKS(10000)) != ESP_OK) {
    printf("Failed to update system time within 10s timeout");
    abort();
}
```

Plusieurs points sont importants ici. `CONFIG_ESPSUNRISE_SNTP_SERVER` est issu directement de la configuration du projet et est initialisé par défaut avec la chaîne `"pool.ntp.org"`, un des nombreux serveurs NTP/SNTP publiquement utilisables. ESP-IDF fournit une macro `ESP_NETIF_SNTP_DEFAULT_CONFIG` permettant d'initialiser la structure `esp_sntp_config_t` avec des paramètres courants, mais nous ajustons cependant le membre `sync_cb` permettant de spécifier une fonction *callback* appelée à chaque synchronisation (par défaut, toutes les heures). Cette fonction est la suivante :

```
volatile int update;

void time_sync_notification_cb(struct timeval *tv)
{
    update = 1;
}
```

update sera utilisé dans la boucle principale pour conditionner la mise à jour de l'afficheur, ce qui nous évite de mettre en place une quelconque temporisation. L'initialisation du support SNTP est déclenchée par un simple appel à esp_netif_sntp_init(), mais nous attendons une première réponse du serveur avec esp_netif_sntp_sync_wait() et, le cas échéant, puisque l'heure exacte est impérative à notre projet, redémarrerons brutalement en cas de problème.



Une fois le collage des quatre couches réalisé, on peut éventuellement passer par une étape de peinture pour améliorer la diffusion. Dans les faits cependant, je ne suis pas certain que l'impact soit réellement important ou significatif.

Suite à ces manipulations, le référentiel temporel interne est prêt à être utilisé, mais comme pour un système POSIX ou autre, il nous faut prendre en compte notre fuseau horaire pour obtenir une heure locale et non une valeur UTC/GMT. La configuration du fuseau (ou TZ pour *Time Zone*) provient de la configuration via la macro CONFIG_ESPSUNRISE_LOCAL_TIMEZONE et sera spécifiée sous la forme d'une variable d'environnement via :

```
setenv("TZ", CONFIG_ESPSUNRISE_LOCAL_TIMEZONE, 1);
tzset();
```

Cette variable d'environnement TZ fait bien plus que de simplement déterminer le fuseau et utilise le format provenant de la GNU Libc **[9]**. Certains pays, dont la France, ont eu l'excellente idée de perturber le rythme circadien de leurs citoyens à la fin des années 70 en instaurant deux changements d'heures dans l'année. Peut-être vous êtes-vous déjà demandé comment votre PC faisait pour changer d'heure automatiquement (même sans Net), et dans le cas d'un système basé sur la GNU Libc (ou compatible), ceci est directement intégré à l'information concernant le fuseau horaire, et donc la variable TZ. Il ne faut donc pas simplement spécifier CET (*Central European Time*) et/ou CEST (*Central European Summer Time*) pour *Europe/Paris* mais : CET-1CEST,M3.5.0,M10.5.0/3. Cette information peut être obtenue d'un système GNU/Linux avec strings /usr/share/zoneinfo/Europe/Paris ou tout simplement en cherchant sur le Net **[10]**. Elle encode directement le « changement d'heure » ou plus exactement le mécanisme de DST (pour *Daylight Saving Time*) et, vous l'aurez compris, nous évitera d'avoir à régler/changer le fuseau deux fois l'an (le faire sur le four est déjà assez pénible).

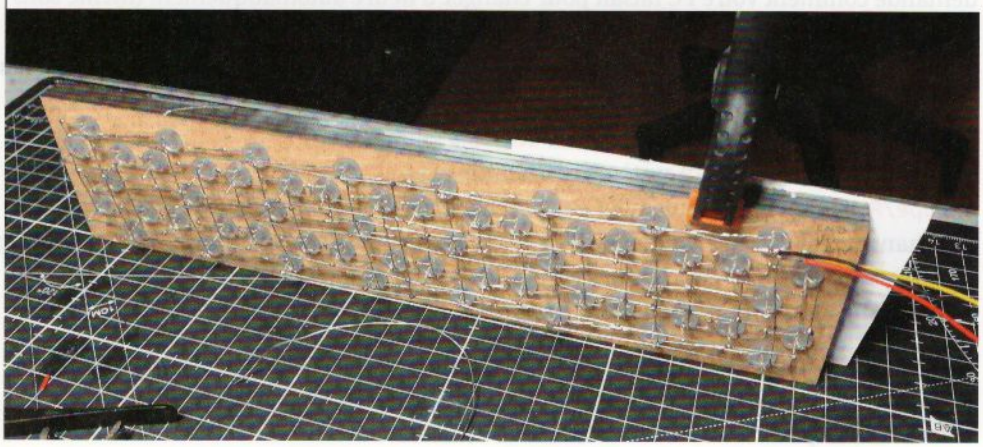
Une fois TZ défini et appliqué, toutes les fonctions standard prendront en compte ce paramètre et nous pouvons utiliser gmtime() pour obtenir l'heure GMT ou localtime() pour l'heure locale ainsi :

```
char buf[80];
time_t now;
struct tm *ntpts;

[...]

time(&now);

ntpts = gmtime(&now);
strftime(buf, sizeof(buf), "%b %d %Y %H:%M:%S", ntpts);
printf("Time set to: %s UTC\n", buf);

ntpts = localtime(&now);
strftime(buf, sizeof(buf), "%b %d %Y %H:%M:%S", ntpts);
printf("Time set to: %s local\n", buf);
```

Nous n'avons plus à intervenir sur les réglages, nous pouvons à tout moment utiliser `time()` pour obtenir l'heure courante et l'utiliser pour nos calculs, et c'est précisément ce que nous allons faire.

4. BASE DU CODE ET CALCULS ASTRONOMIQUES

Encore une fois, je ne rentrerai pas dans le détail ici, car je vous avoue que mes connaissances en astronomie et/ou en mécanique solaire sont très superficielles et, de plus, il semble qu'énormément de codes utilisent exactement le même algorithme décliné en une myriade d'implémentations en différents langages (exemple : **[11]**, **[12]** ou **[13]**). L'aspect important ici est davantage l'adaptation du code à nos besoins, en particulier concernant la gestion du temps. Les implémentations en C qu'on trouve sur le Web utilisent généralement une tripotée d'arguments spécifiant, bien sûr, les coordonnées géographiques sous la forme d'une latitude et d'une longitude, mais également la date souvent précisée via trois entiers (jour, mois, années), ainsi que le décalage horaire dû au fuseau utilisé.



La version APA106 de la réalisation repose sur un agencement en lignes plutôt qu'en « S ». Ici, on peut utiliser directement les pattes du composant pour les liaisons DIN/DOUT et le format de LED (8 mm) est plus facile à intégrer dans le support. Ces composants ne sont cependant pas correctement gérés par les ESP32 (si ce sont effectivement des APA106) et il faut se rabattre sur un ESP8266 avec Arduino et FastLed.

Ici, nous n'avons pas besoin de ces éléments « discrets » et ajustons cela pour n'avoir à fournir que :

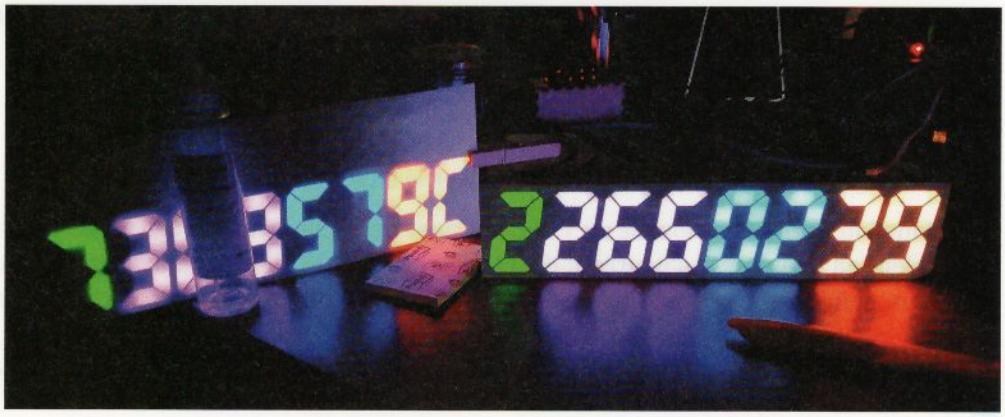
- la latitude et la longitude sous la forme de deux `float` ;
- la date/heure via un pointeur sur un `time_t` ;
- et un argument spécifiant l'événement concerné (lever ou coucher) avec un `int`.

La fonction de départ retourne un `float` en guise de résultat et nous devons transformer cette valeur en un nombre d'heures et de minutes avec `modff()`, qui retourne la partie décimale d'un nombre en virgule flottante (il existe aussi `modf()` pour les `double` et `modfl` pour les `long double`), ainsi :

```
// lever
float riselocalT;
float risehours;
float riseminutes;

// coucher
float setlocalT;
float sethours;
float setminutes;

// durée
float daytimelocalT;
float dayhours;
float dayminutes;

riselocalT = calculateSunrise(latitude, longitude, &now, SUNRISE);
riseminutes = modff(riselocalT, &risehours) * 60;
setlocalT = calculateSunrise(latitude, longitude, &now, SUNSET);
setminutes = modff(setlocalT, &sethours) * 60;
daytimelocalT = setlocalT - riselocalT;
dayminutes = modff(daytimelocalT, &dayhours) * 60;

setpnumber(colormap, 0, 2, (int)risehours);
setpnumber(colormap, 2, 2, (int)riseminutes);
setpnumber(colormap, 4, 2, (int)sethours);
setpnumber(colormap, 6, 2, (int)setminutes);
led_strip_refresh(led_strip);
```

La durée du jour n'est pas envoyée sur l'afficheur à LED, mais c'est quelque chose que je garde en réserve pour une future version, avec éventuellement l'ajout de l'heure courante et une sorte de barre de progression (plus il y a de LED, mieux c'est). Vous l'aurez compris, c'est `calculateSunrise()` qui fait tout le travail mathématique nécessaire et c'est cette fonction qui a été adaptée pour ce projet.

L'implémentation des fonctions POSIX (ou plus exactement GNU) avec l'ESP-IDF n'est pas complète (`timelocal()` et `timegm()` sont absents), pas plus que les extensions concernant la structure `tm` qui, sous les systèmes de la famille *BSD par exemple, intègrent les membres



Le rendu final est très satisfaisant, même si l'appareil photo ne restitue pas fidèlement le résultat. À l'œil, les segments des chiffres apparaissent uniformes et sans ombre interne (comme le « 2 » vert), que ce soit avec les WS2812b ou les APA106.

`tm_zone` (`char *`) et `tm_gmtoff` (`long`) correspondant respectivement à un pointeur vers une chaîne décrivant le fuseau utilisé et le décalage entre l'heure UTC et l'heure locale. Exactement ce qu'il nous faudrait puisque la fonction originale utilise cette valeur en l'additionnant au résultat du calcul, juste avant de retourner le résultat.

Ce n'est pas grave, nous pouvons ajouter quelques lignes de code pour régler le problème et nous passer de devoir utiliser le décalage en argument de la fonction.

Pour cela, il nous suffit de jongler avec `localtime()`, `gmtime()` et `mktime()`, mais nous devons contourner un petit problème. En effet, la façon la plus aisée d'obtenir une différence en seconde entre l'heure locale et l'heure UTC consisterait à obtenir le nombre de secondes depuis le 1er janvier 1970 (heure Unix) avec `mktime()` et `timegm()` (inverse de `gmtime()`) et de simplement faire une soustraction. Malheureusement, comme le précise un commentaire de « igrr » dans un ticket (#10876) ouvert sur le GitHub de l'ESP-IDF, « *timegm n'est pas encore supporté par la bibliothèque C Newlib utilisée par l'IDF* ». Nous n'avons donc d'autre choix que d'implémenter une fonction pour faire le calcul en nous basant sur les informations tirées de `struct tm` créées pour l'occasion :

```
long tzoffset(time_t t) {
    struct tm local = *localtime(&t);
    struct tm utc = *gmtime(&t);
    long diff = ((local.tm_hour - utc.tm_hour) * 60
        + (local.tm_min - utc.tm_min)) * 60L
        + (local.tm_sec - utc.tm_sec);
    int deltamday = local.tm_mday - utc.tm_mday;

    // attention au changement de mois
    if ((deltamday == 1) || (deltamday < -1)) {
        diff += 24L * 60 * 60;
    } else if ((deltamday == -1) || (deltamday > 1)) {
        diff -= 24L * 60 * 60;
    }
    return diff;
}
```

Nous pouvons alors utiliser cette fonction pour obtenir et utiliser le décalage dans `calculateSunrise()` :

```
    struct tm tmutc;

    if (!gmtime_r(tt, &tmutc))
        return -1;

    int delta = tzoffset(*tt) / (60 * 60);

    tmutc.tm_sec = 0;

    //2. convert the longitude to hour value and
    //   calculate an approximate time
    float lngHour = lng / 15.0;

    float t;
    if (event == SUNRISE) {
        t = tmutc.tm_yday + (6.0 - lngHour) / 24.0;
    } else {
        t = tmutc.tm_yday + (18.0 - lngHour) / 24.0;
    }
[...]
    //9. adjust back to UTC
    float UT = fmod(24 + fmod(T - lngHour, 24.0), 24.0);

    //10. convert UT value to local
    //    time zone of latitude/longitude
    return UT + delta;
```

Et enfin, nous avons un dernier problème à régler, qui est celui du positionnement. Mettre « en dur » cette information dans le code est une mauvaise idée, d'autant que les autres éléments comme le nombre de LED, le serveur NTP à utiliser ou les éléments d'authentification pour le Wi-Fi sont déjà gérés dans la configuration (`main/Kconfig.projbuild`). La difficulté cependant concerne le format de données et les types pouvant être utilisés dans le fichier : entier, chaîne ou booléen, il n'y a nulle trace d'une valeur en virgule flottante.

Deux options s'offrent alors à nous :

- utiliser des chaînes et devoir les convertir en `float` pour initialiser `latitude` et `longitude` ;
- ou multiplier la valeur initiale jusqu'à obtenir un entier et, dans le code, la diviser d'autant lors de l'initialisation.

J'ai choisi la seconde option, car soyons honnête, il n'y a rien de pire que de devoir jouer avec `strtof()` et ses cousins, d'autant qu'une macro peut parfaitement définir un `float` ou un `double`, c'est une limitation de *Kconfig*, pas du préprocesseur. Ceci nous donne donc :

```
config ESPSUNRISE_LATITUDE
  int  "Latitude (in 1/1000000 of degree)"
  default 48107362
  help
    Latitude of your geographical position

config ESPSUNRISE_LONGITUDE
  int  "Longitude (in 1/1000000 of degree)"
  default 7508348
  help
    Longitude of your geographical position
```

dans le `main/Kconfig.projbuild`, et :

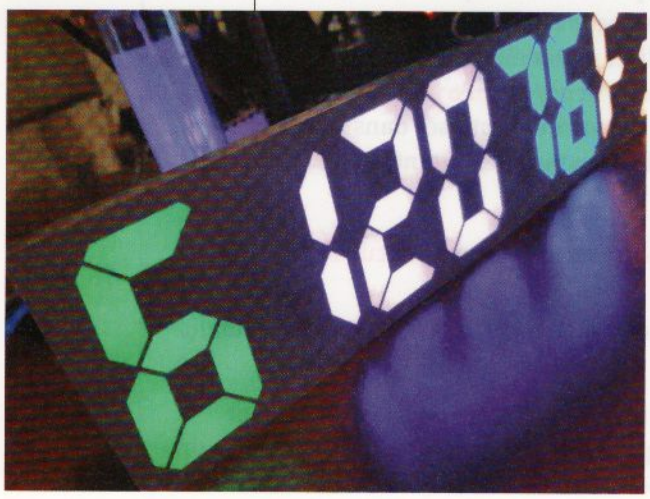
```
float latitude = CONFIG_ESPSUNRISE_LATITUDE / 1000000.0;
float longitude = CONFIG_ESPSUNRISE_LONGITUDE / 1000000.0;
```

dans le code.

CONCLUSION

À ce stade, le projet est fonctionnel, mais n'est pas encore, à mon sens, visuellement satisfaisant. Souhaitant conserver le plus longtemps possible un aspect générique à la réalisation, celle-ci ressemble davantage à un simple afficheur 8 chiffres 7 segments qu'à une double horloge. Mais le concept étant à présent validé, ce n'est l'affaire que de quelques nouvelles découpes laser après une légère phase de redesign dans InkScape. Réagencer les chiffres sera la priorité et ceci se verra, peut-être, complété de l'ajout de 4 LED supplémentaires en guise de séparateurs heures/minutes (comme une horloge digitale).

Après de nombreux essais, le diffuseur finalement utilisé en façade est du papier blanc 80g tout ce qu'il y a de plus standard. Un tantinet fragile et salissant, mais rien qui ne saurait être corrigé par l'application d'un simple film transparent autocollant.



J'ai également fini par implémenter une version Arduino, car le pilotage des APA106 (si tant est que les LED 8 mm que je possède sont effectivement des APA106) semble difficilement possible avec le périphérique RMT. Un délai supplémentaire de 12µs appelé WT et correspondant au temps de traitement des données, qui doit être inséré entre les trames de 24 bits pour chaque LED, semble perturber le fonctionnement (j'ai depuis commandé des YF923-F8 sur AliExpress en espérant une compatibilité avec les WS2812). La solution a donc été de me rabattre sur l'ESP8266 parfaitement capable d'utiliser FastLed avec le *framework* Arduino, et donc de réimplémenter tout cela sans grande difficulté. FastLed semble considérer les APA106

comme des SK6822, alors que les deux *datasheets* affichent des valeurs pour T0H/T0L/T1H/T1L très différentes... mais ça fonctionne et c'est tout ce qui importe (pour l'instant). Les deux versions se trouvent dans leurs dépôts respectifs sur mon GitLab (ESP-IDF **[14]** et Arduino **[15]**).

Enfin, les perspectives d'avenir pour ce projet sont nombreuses :

- utiliser un module GPS série pour acquérir la position et heure, et donc se passer de Wi-Fi ;
- changer d'échelle et doubler ou tripler le nombre de LED par segments, ce qui permettrait des effets intéressant comme des dégradés dans les segments ;
- compléter avec 4 chiffres supplémentaires pour l'heure courante ;
- ajouter un décompte en heure/minutes pour la durée restante de jour ;
- voir encore plus grand et passer de 7 segments à 16 pour ajouter la date et une abréviation du jour de la semaine ;
- réutiliser le concept d'afficheur, mais pour réaliser une horloge mondiale occupant tout un mur ;
- travailler sur une version extérieure capable de résister aux intempéries ;
- etc.

« *So many snacks, so little time.* » -- Venom, 2018. **DB**

RÉFÉRENCES

[1] **https://github.com/adafruit/Adafruit_NeoPixel**

[2] **https://fastled.io/**

[3] **https://connect.ed-diamond.com/hackable/hk-053/esp32-creer-ses-composants-reutilisables-avec-esp-idf**

[4] **https://components.espressif.com/**

[5] **https://components.espressif.com/components/espressif/led_strip**

[6] **https://github.com/UncleRus/esp-idf-lib**

[7] **https://www.youtube.com/watch?v=wnSelKDe4sE**

[8] **https://forum.arduino.cc/t/setting-the-internal-clock-on-the-esp8266/1147142**

[9] **https://www.gnu.org/software/libc/manual/html_node/TZ-Variable.html**

[10] **https://rainmaker.espressif.com/docs/time-service.html**

[11] **https://edwilliams.org/sunrise_sunset_example.htm**

[12] **https://itecnote.com/tecnote/sunrise-sunset-times-in-c/**

[13] **https://stackoverflow.com/questions/7064531/sunrise-sunset-times-in-c**

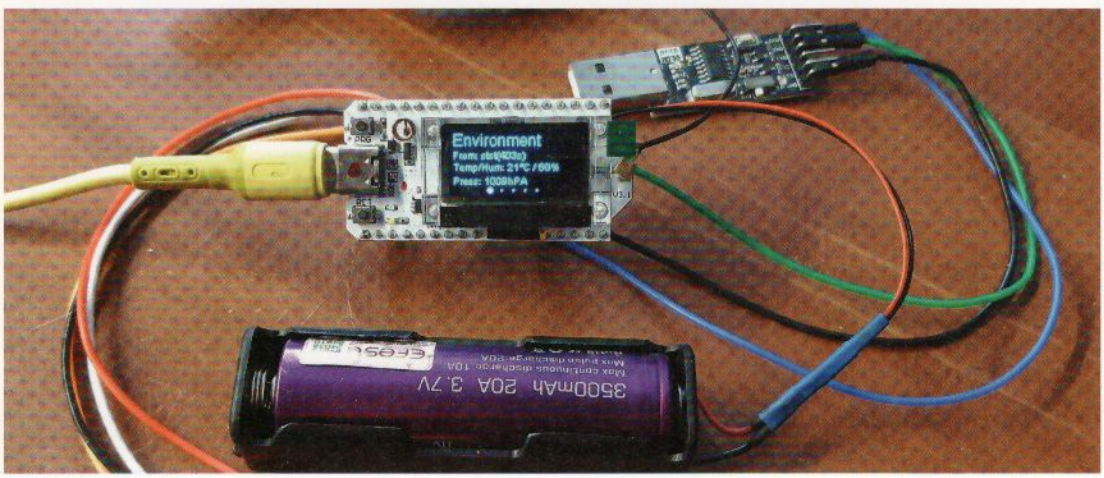
[14] **https://gitlab.com/0xDRRB/esp32sunrise**

[15] **https://gitlab.com/0xDRRB/ardu7segSunrise**

MESHTASTIC : COMMUNICATION LONGUE DISTANCE, HORS RÉSEAUX ET GRATUITE

Denis Bodor

Une installation domotique ou un déploiement de capteurs repose généralement sur différents types de communication radio ou filaires : Wi-Fi, Zigbee, Z-Wave, Bluetooth, i2c, EIA-485, Ethernet, etc. Mais lorsque les distances se comptent en centaines de mètres voire en kilomètres, le problème devient plus épineux. L'option 4G est toujours possible, toutefois comme avec d'autres solutions d'IoT, ceci suppose de non seulement ajouter une charge financière au projet, mais implique également de reposer sur un réseau contrôlé par un opérateur susceptible de changer les règles comme il l'entend. Meshtastic est une réponse à cette contrainte : une réponse hors réseau (*off grid*), gratuite, participative et très économique...



– Meshtastic : communication longue distance, hors réseaux et gratuite –

Prenons un exemple simple. Vous êtes à la campagne et votre habitation est domotisée. Vous êtes très heureux ainsi, supervisant à loisir le chauffage, la consommation électrique, les luminaires, etc. Seulement voilà, vous avez un potager, un terrain de loisir ou un petit chalet qui se trouve hors de portée du Wi-Fi ou de tout autre protocole radio « local ». Comment ajouter des capteurs en ce lieu sans avoir à payer un abonnement (typiquement 3G, 4G ou GPRS) et sans avoir à intégrer un réseau hors de votre maîtrise, le tout parfaitement légalement ? Cette problématique fonctionne de la même manière avec plusieurs situations, que ce soit du côté particulier ou professionnel (on pense naturellement aux installations agricoles, par exemple).

Premier démarrage d'un Heltec V3 après flashage : pas de connectivité LoRa sans définir la région. On ne rigole pas avec l'ARCEP, et c'est très bien comme ça !

1. MESHTASTIC ET MATÉRIEL

Meshtastic propose une solution à ce problème, est *open source* et géré par sa communauté d'utilisateurs. Ce nom désigne, en même temps, une initiative, un projet, un *firmware* et un ensemble d'outils. Ce dont il ne s'agit pas, en revanche, c'est de quelque chose de centralisé, comme le réseau TTN (*The Things Network*) que nous avons évoqué dans le numéro 19, il y a fort longtemps **[1]**. Meshtastic repose sur la technologie LoRa, un protocole de communication radio longue distance (record de 254 km pour Meshtastic, voir **[2]**) utilisant des fréquences de la bande ISM (Industriel, Scientifique et Médical) incluant 433 MHz et 868 MHz chez nous. Il est donc possible d'utiliser des équipements LoRa sans autorisation préalable, sans demande de licence et sans coût supplémentaire autre que le matériel lui-même. Bien entendu, la puissance est régulée ainsi que l'utilisation (rapport cyclique de 10 % par heure), mais ceci est parfaitement pris en charge par le code du projet.

Comme son nom l'indique, Meshtastic repose sur la notion de maillage (*mesh*) où chaque installation, ou nœud, établit une communication avec les nœuds à portée et peut relayer les messages (un peu comme Tor). L'ensemble des communications est chiffré de façon à garantir la sécurité des données lorsque des messages transitent de cette manière d'un nœud à un

L'écran du nœud vous affiche le code vous permettant de procéder à l'association Bluetooth avec votre smartphone.

autre (message direct), tout en permettant également des diffusions globales (*broadcast*). C'est le concept de base et l'aspect participatif du projet décentralisé, mais en disposant de ses propres nœuds, cela fonctionne également. Il n'est donc pas nécessaire, pour profiter de Meshtastic, d'avoir des nœuds préexistants dans sa zone. LoRa est un standard spécialement destiné à ce genre d'utilisation et la portée des liaisons peut s'étendre sur plusieurs kilomètres en terrain dégagé (notion de « ligne de vue » ou « *line of sight* » en anglais). En environnement urbain, les choses sont drastiquement différentes et il est plus raisonnable de parler de centaines de mètres dans de bonnes conditions.

Ce qui nous amène justement au choix du matériel permettant de faire fonctionner le *firmware* Meshtastic. Le site officiel **[3]** liste un certain nombre de plateformes, pour la plupart basées, sans surprise, sur des microcontrôleurs ESP32, dont un grand nombre de chez LILYGO, le tout trouvable facilement sur les sites habituels (comprendre « Amazon » si vous êtes pressé ou « AliExpress » si vous êtes patient et économe). Notez qu'il existe également une solution reposant sur Raspberry Pi Pico avec un module LoRa additionnel ainsi qu'une approche consistant à utiliser un SBC GNU/Linux (typiquement Raspberry Pi) faisant fonctionner un service/démon `meshtasticd`.

Le point important, sinon critique, dans le choix du matériel concerne la partie LoRa et plus précisément la fréquence que le circuit va utiliser. En Europe, nous avons le choix entre deux bandes de fréquences pour LoRa :

- celle des 433 MHz limitant la puissance apparente rayonnée ou PAR (ERP en anglais) à +10 dBm (10 mW) ;
- et celle des 868 MHz avec la sous-bande P (869,4 MHz - 869,65 MHz) permettant une PAR jusqu'à +27 dBm (500 mW) avec un rapport cyclique de 10 % par heure.

On voit clairement que, pour une utilisation très intermittente maximisant la portée, comme c'est le cas pour des sondes et capteurs, mieux vaut privilégier le matériel supportant la bande des 868 MHz (863 - 928 MHz) plutôt que celle des 433 MHz déjà bien occupée par de nombreux périphériques type sonnettes et télécommandes. Attention, la bande 902 - 928 MHz n'est pas utilisable en Europe, car hors ISM, ceci ne concerne que les USA et le Canada.

À cela s'ajoute le fait que tous les contrôleurs LoRa ne se valent pas et n'utilisent pas un *transceiver* forcément capable d'émettre à la puissance maximale autorisée. Nous avons là deux puces, la SX1276 et la SX1262. Cette dernière équipe le matériel récent et permet une émission avec une PAR de +22 dBm, par opposition au SX1276

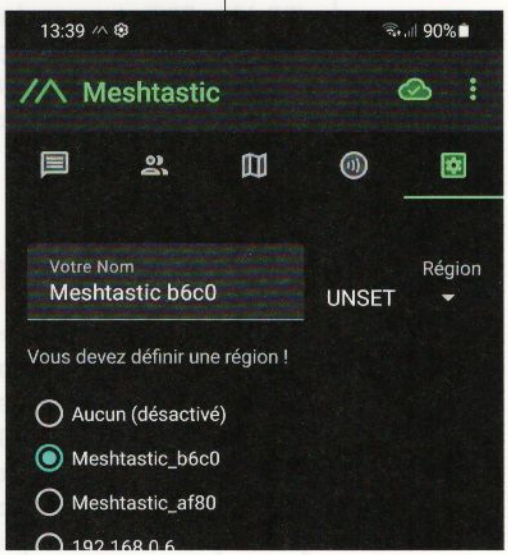
limité à +20 dBm et équipant les cartes et modules les moins coûteux. 2 dBm de différence semblent peu, mais on parle ici de +50 % de puissance au bénéfice du SX1262, avec une sensibilité plus importante en réception et une consommation moindre.

Ici, j'ai jeté mon dévolu sur une paire de cartes Heltec Wi-Fi LoRa 32 V3 (attention, la V2 utilise un SX1276), acquises sur Amazon (pressé donc) pour environ 50 euros (les deux). Il s'agit de cartes basées sur de l'ESP32-S3, intégrant un contrôleur Semtech SX1262 ainsi qu'un écran OLED monochrome (bleu) de 0,96 pouce (128×64 pixels), un contrôleur de charge pour accu LiPo (non inclus) et une interface CP2102 pour la liaison USB/série. Le tout est livré avec un connecteur à souder pour l'accu et une antenne LoRa avec un connecteur IPEX1.0.

2. ARCHITECTURE GÉNÉRALE

Comme précisé précédemment, la structure générale de Meshtastic consiste en un maillage et même si ici nous n'avons que deux nœuds, ceci ne change rien à l'affaire. Dans la configuration par défaut, vous devez voir tous les nœuds en votre possession comme des passerelles vers le réseau Meshtastic et chacun d'eux communique à la fois avec le réseau via LoRa et potentiellement avec vous par un autre biais. Il peut s'agir de Bluetooth, d'une liaison série, de Wi-Fi, d'USB ou encore, tout simplement, d'une IHM (écran + périphérique d'entrée). Ceci est totalement dépendant du matériel utilisé et dans le cas des Heltec V3, nous avons de base le choix entre l'interface série accessible via CP2102 et le Bluetooth activé par défaut. Ceci permet une connexion et une configuration soit via un PC/Mac (outil CLI Python ou *Web client*), soit un smartphone et l'application Meshtastic dédiée (Android ou iOS).

Un peu à la manière d'ESPHome, le *firmware* Meshtastic est avant toute chose le socle logiciel permettant de faire fonctionner le réseau et d'établir la liaison avec votre ou vos équipements (PC, Pi, montage Arduino, etc.). De la même manière qu'un périphérique ESPHome s'intègre dans une configuration Home Assistant, un nœud s'intègre au réseau Meshtastic, mais peut également prendre en charge des capteurs directement et, en cas de connexion Wi-Fi locale, établir le contact avec un *broker* MQTT pour diffuser l'information. Un certain nombre de « modules » (logiciels) peuvent être activés dans la configuration pour adapter le comportement du *firmware* à vos besoins. De manière générale, cependant, il sera plus courant et plus souple de tout simplement utiliser un montage quelconque à base de microcontrôleur et de garder le nœud dans son simple rôle de passerelle.



L'application Android est plus « modeste » que celle pour iOS, mais fait parfaitement le travail. Ici, la première connexion via Bluetooth pour le choix de la région.

En parlant de rôle justement, plusieurs profils sont prévus dans la configuration que nous allons voir sous peu. Celui par défaut est *CLIENT*, définissant que le nœud en question participe au réseau en relayant les messages et supporte l'utilisation de clients (Bluetooth, Wi-Fi, série, etc.). D'autres rôles peuvent être choisis, comme *REPEATER* ou *ROUTER*, principalement destinés aux nœuds placés stratégiquement pour une émission/réception LoRa optimale. Ici, la partie « cliente » sera donc désactivée et les nœuds en question ne feront que participer à la topologie du réseau.

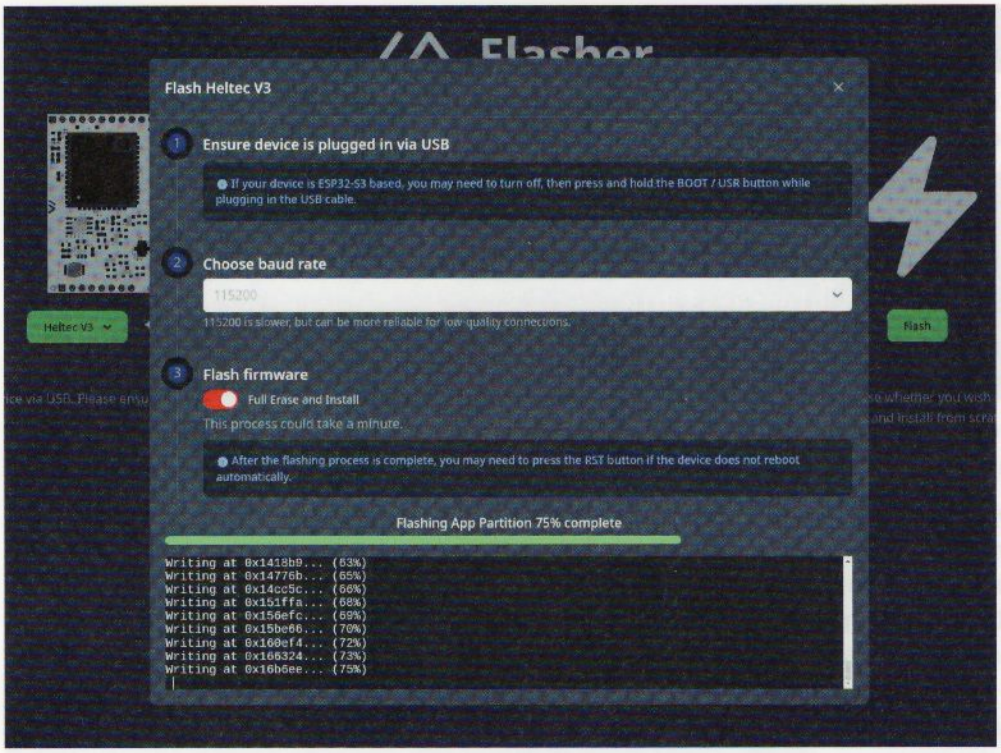
Ce dernier point nous amène également à parler du GPS et des données associées. Le Heltec V3 n'intègre pas de récepteur GPS contrairement à d'autres périphériques, mais il est toutefois possible d'associer une information de position fixe « en dur » dans la configuration. Le positionnement géographique du nœud peut être intéressant pour cartographier le réseau et positionner les nœuds voisins afin de déterminer la portée de la liaison. Il peut également être utile de mettre en œuvre un récepteur GPS dans le cas d'un nœud mobile, sur accu ou dans un véhicule, par exemple. Il s'agit là, à mon sens, d'installations ayant des besoins spécifiques, et mieux vaut donc faire l'économie de quelques euros qui seront bien plus utiles ailleurs (autres nœuds, antenne plus performante, accu LiPo, etc.).

Étant donné la très faible densité de nœuds sur le territoire français (pour l'instant), la structure de l'installation consistera typiquement en un nœud relié à un ordinateur (PC, Mac, SBC) et un ou plusieurs autres nœuds faisant office de capteurs et dialoguant localement avec des montages divers. Meshtastic constituera ainsi le « média » sur lequel les informations circuleront d'un bout à l'autre de la liaison radio.

3. FIRMWARE ET MISE À JOUR

Une fois le matériel réceptionné, il est très peu probable que le *firmware* installé soit dans une version récente. La première opération consistera donc à mettre à jour le logiciel et pour cela, la façon la plus simple de procéder consiste à pointer votre navigateur sur l'URL **https://flasher.meshtastic.org**. Vous trouverez là un *flasher* permettant de reprogrammer la mémoire de vos nœuds, quel que soit leur modèle, à condition d'utiliser un navigateur compatible avec l'API WebSerial. Ceci ne fonctionnera pas avec Firefox qui ne supporte pas cette fonctionnalité (et ne la supportera sans doute jamais) et vous devrez vous tourner vers Chromium, Chrome, Opera ou Edge.

Pour flasher votre Heltec V3, maintenez le bouton « PRG » enfoncé en connectant le périphérique en USB-C (« PRG » puis « RST » ne semble pas fonctionner, contrairement à d'autres cartes ESP32). Ceci le placera en mode *bootloader* et vous pourrez alors choisir « Heltec V3 » dans la liste, sélectionner une version récente (mais stable) du *firmware* (2.3.2.63df972 Beta à ce jour) et cliquer sur ***Flash***. Un résumé des informations de la version s'affichera et vous pourrez défiler et cliquer sur ***Continue***. Dans le formulaire qui apparaît alors, assurez-vous de cliquer ***Full Erase and Install***, puis ***Erase Flash and Install***. Le navigateur vous demandera de choisir le port série à utiliser pour terminer la procédure (attention aux permissions). Ceci prend un certain temps et vous pourrez suivre le déroulé des opérations directement à l'écran.



Le « flasheur » web est sans doute l'option la plus rapide pour installer ou mettre à jour le firmware. Mais ceci ne fonctionnera pas avec Firefox qui ne supporte pas l'API WebSerial.

Voilà pour l'approche rapide. Si vous ne comptez pas utiliser un autre navigateur que votre Firefox (ce que je comprends tout à fait), une alternative est à votre disposition via la ligne de commandes. Précisons de suite que, généralement, j'aime reconstruire les *firmwares* moi-même pour ce type d'opération. Malheureusement ici, les développeurs, sans doute en raison du caractère très multiplateforme du projet, ont opté pour le duo PlatformIO + Visual Studio Code, ce qui ne va pas du tout dans le sens de mes préférences. J'aurais, en effet, préféré simplement utiliser l'ESP-IDF pour construire et flasher ce qui est, après tout, un simple ESP32. Les sources du *firmware* **[4]** comprennent un `Dockerfile`, ce qui laisse supposer un fonctionnement possible similaire à ESPHome, mais aucune indication n'est fournie.

On ne peut donc que se rabattre sur la solution la plus proche consistant à flasher le *firmware* binaire manuellement. Dans le cas de plateforme ESP32 comme le « Heltec V3 », nous pouvons utiliser l'outil `esptool.py` livré avec l'environnement ESP-IDF (ou installable séparément via `pip3 install --upgrade esptool`). Vous trouverez sur le GitHub officiel une section « Releases » où, pour la dernière version **Beta** (la plus stable) vous pourrez obtenir une archive ZIP (`firmware-2.3.2.63df972.zip` actuellement) regroupant tous les *firmwares* binaires pour tout le matériel supporté. Téléchargez et décompressez cette archive et parmi tous les fichiers, vous trouverez `firmware-heltec-v3-2.3.2.63df972.bin`. Nous n'allons pas utiliser le script fourni (`device-install.sh`), mais plutôt flasher manuellement avec `esptool.py`. Nous commençons par placer le périphérique en mode *bootloader* en le branchant tout en pressant le bouton « PRG », puis nous effaçons la flash :

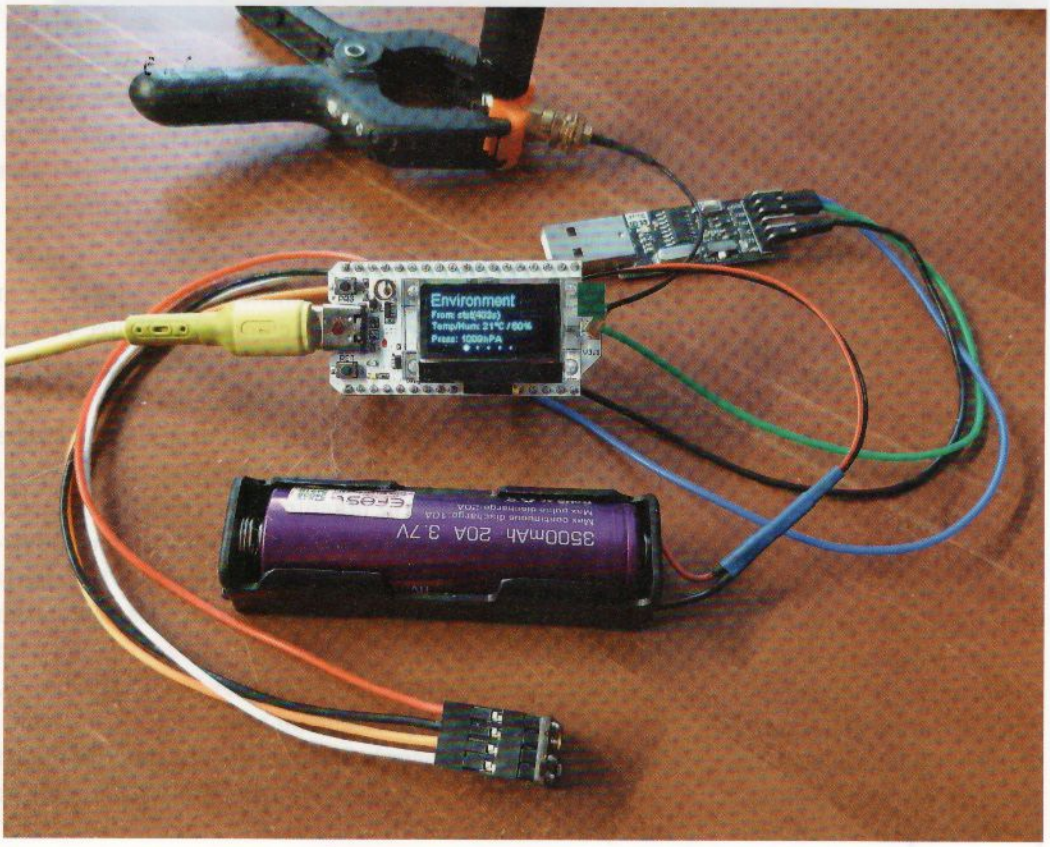
```
$ esptool.py --after no_reset erase_flash
```

Et enchaînons immédiatement sur le flashage du *firmware*, de l'image OTA BLE et du système de fichiers embarqué :

```
$ esptool.py --after no_reset write_flash \
  0x00 firmware-heltec-v3-2.3.2.63df972.bin
$ esptool.py --after no_reset write_flash \
  0x260000 bleota-s3.bin
$ esptool.py --after no_reset write_flash \
  0x300000 littlefs-2.3.2.63df972.bin
```

Comme nous avons utilisé `--after no_reset`, notre ESP32 reste en mode *bootloader*. Nous devons alors le déconnecter/reconnecter pour démarrer sur ce nouveau *firmware*.

Quelle que soit la méthode que vous avez utilisée, et si vous avez également opté pour ce matériel Heltec, l'écran OLED devrait vous afficher un message précisant que le périphérique n'est pas configuré et que vous devez choisir une région avec l'un des outils de configuration disponibles. En effet, il n'est pas question pour le *firmware* de commencer à utiliser LoRa sans savoir à quelle régulation locale se plier.



Voici un nœud Meshtastic dûment équipé : antenne digne de ce nom, accu LiPo format 16550, capteur BME280 en i2c et seconde liaison série pour l'échange de messages en texte brut.

4. CONFIGURATION

De base, avec les Heltec V3 et leur ESP32, le Wi-Fi n'est pas actif et vous avez alors deux options de connexion : le port série (via le CP2102 intégré) ou le Bluetooth avec votre smartphone et l'application Meshtastic (disponible dans les *stores* Google et Apple). Pour la liaison série, là encore nous avons deux options : le client web (local ou via **https://client.meshtastic.org/**) ou le client Python en ligne de commandes.

Commençons par le plus simple en procédant à une configuration minimale permettant, au moins, de communiquer via LoRa. Commencez par installer l'application Meshtastic sur votre smartphone (ici, Android) et assurez-vous de bien avoir le Bluetooth activé. Lancez ensuite l'application et, dans la configuration (le dernier écran) utiliser le bouton +. Votre appareil va alors détecter un périphérique avec un nom débutant par « Meshtastic » suivi d'une valeur hexadécimale correspondant aux deux derniers octets de son adresse MAC. Dès ce moment, l'écran OLED du Heltec V3 vous affichera un message vous présentant un code à 6 chiffres permettant de vous appairer en Bluetooth. Ceci fait, la communication est établie et, sur le même écran, vous verrez en haut à droite la région définie à « UNSET ». Tapotez et choisissez « EU_868 » (Europe 868 MHz), votre nœud Meshtastic va alors redémarrer pour utiliser cette nouvelle configuration et immédiatement rechercher d'autres nœuds via LoRa. Votre smartphone est maintenant un « terminal » connecté au nœud et, si vous avez des voisins Meshtastic, ils devraient apparaître dans le second écran de l'application.

Dans la continuité de cette configuration, nous allons maintenant nous occuper du second nœud, mais cette fois via l'outil en ligne de commande sur Raspberry Pi, par exemple (ou PC GNU/Linux). Pour cela, commencez par créer un environnement virtuel Python avec la commande `python3 -m venv /chemin/meshtastic`. Ceci vous permettra de télécharger et installer l'outil et ses dépendances (modules) sans interférer avec le reste du système. Pour activer cet environnement, utilisez `source /chemin/meshtastic/bin/activate`, puis installez avec `pip3 install --upgrade pytap2 meshtastic`. Ceci fait, tant que vous serez dans l'environnement (que vous pouvez quitter avec `deactivate`), vous aurez accès à la commande `meshtastic`.

Votre nœud connecté en USB, vous devez avoir un nouveau port série (`/dev/ttyUSB*`) et c'est via cette interface que vous pourrez utiliser le client Python. Pour tester le bon fonctionnement, vous pouvez utiliser l'option `--info` accompagnée de `--port /dev/ttyUSBx` pour spécifier le port. Vous devrez alors voir apparaître un gros paquet d'informations concernant le périphérique, sous forme de données JSON. L'option `-h` vous indiquera, classiquement, toutes les commandes utilisables. Un autre test qui peut s'avérer très utile par la suite, surtout si vous êtes déjà habitué à la syntaxe de configuration d'ESPHome et/ou Home Assistant, est l'export de la configuration actuelle en YAML avec `meshtastic --port /dev/ttyUSB1 --export-config`, qu'on redirigera vers un fichier avec quelque chose comme `> maconfig.yaml`. Ce fichier pourra ensuite être modifié, puis réimporté avec `--configure`.

Mais pour l'heure, nous souhaitons juste activer LoRa et donc spécifier la région. Chose que nous faisons avec :

```
$ meshtastic --port /dev/ttyUSB1 \
--set lora.region EU_868
```

Notez qu'à chaque utilisation de `--set`, la configuration est immédiatement modifiée puis appliquée, ce qui provoque un redémarrage du périphérique. Ceci peut être embêtant lorsqu'il s'agit de modifier plusieurs paramètres liés entre eux. Un bon exemple est la configuration du Wi-Fi, et nous pouvons faire cela, immédiatement, en cumulant plusieurs options en une ligne :

```
$ meshtastic --port /dev/ttyUSB1 \
--set network.wifi_enabled true \
--set network.wifi_ssid "monSSID" \
--set network.wifi_psk "mot2passe"
```

Avec les plateformes ESP32, le fait d'activer le Wi-Fi désactive le Bluetooth automatiquement, mais vous pouvez également décider de le faire explicitement avec :

```
$ meshtastic --port /dev/ttyUSB1 \
--set bluetooth.enabled false
```

Une fois le périphérique redémarré une dernière fois, celui-ci sera connecté à notre réseau et accessible via **http://meshtastic.local/** (mDNS) ou directement avec l'IP obtenue par DHCP. En pointant votre navigateur à cet endroit, vous verrez une page web fournie par l'ESP32, en tout point similaire à celle de **https://client.meshtastic.org/**. Un nœud configuré de cette manière est également accessible localement (LAN) via l'application mobile. Vous pouvez donc désactiver le Bluetooth sur votre smartphone (ce qui est toujours une bonne chose). Notez qu'il est également possible d'utiliser le client Python via LAN en utilisant `--host` et l'IP du nœud en lieu et place de `--port`.

L'idée est ici d'avoir l'un des nœuds connecté en Wi-Fi et le ou les autres étant déconnectés pour être éventuellement reconfigurés via le port USB ou en Bluetooth. Notez qu'il est également possible de configurer ces nœuds pour les rendre administrables à distance, via le réseau *mesh*, de façon sécurisée pour éviter des accès physiques potentiellement pénibles.

L'antenne livrée avec les Heltec V3 (en bas à gauche) n'est pas la solution la plus efficace pour maximiser la portée. On préférera investir un peu (AliExpress, ~6€) et opter pour quelque chose de plus performant comme l'antenne 3,5 dBi Ziisor (en haut) recommandée par un développeur Meshtastic sur le Discord du projet [9].

Une fois nos deux nœuds ainsi configurés, vous devez voir, sur l'un comme sur l'autre, le second nœud présent dans le « voisinage », que ce soit dans l'application smartphone, dans l'interface web (série ou HTTP) à la section « *Nodes* » ou via le

client Python, avec l'option `--nodes`. Chaque nœud est désigné par un nom, un alias de 4 caractères et son identifiant composé des 4 derniers octets de son adresse MAC. Le nom et l'alias sont deux éléments que vous pouvez changer dans la configuration.

5. CANAUX

Pour configurer plus avant nos nœuds, nous devons nous pencher sur la notion de « canal ». Les canaux ne sont pas des « bandes de fréquences », mais un mécanisme permettant la ségrégation des messages. Par défaut, un seul canal de communication est actif, c'est le canal 0 également désigné comme *primary* et celui-ci est chiffré avec AES en utilisant une clé partagée (ou PSK pour *Pre-Shared Key*) connue, c'est `"AQ=="` (ou `0x01` en hexadécimal). Le canal primaire est celui utilisé pour diffuser les informations comme la télémétrie et la position entre les nœuds, c'est le canal par défaut. Il ne peut exister qu'un seul canal primaire dans la configuration.

Dans un réseau plus vaste que deux nœuds isolés (pour le moment), les échanges sur le canal primaire sont donc publics et visibles par tous (puisque tout le monde à la même PSK). Ce n'est pas nécessairement ce que vous pouvez souhaiter si vous devez faire circuler des informations concernant vos capteurs. Pour cela, vous avez à disposition 7 autres canaux configurables, utilisant des PSK de votre choix à renseigner en base64 dans l'interface web ou en base64/hexadécimal via l'outil Python. Pour configurer un canal, je pense que la manière la plus simple est de commencer par un nœud et via l'outil en ligne de commande ainsi :

```
$ meshtastic --port /dev/ttyUSB1 \
--ch-set name "MonCanal" --ch-set psk random \
--ch-index 1
```

Nous activons ainsi le canal 1 en lui donnant un nom et réglons un mot de passe aléatoire. Nous pouvons ensuite afficher les informations du périphérique pour retrouver l'information qui nous intéresse :

```
$ meshtastic --port /dev/ttyUSB1 --info
[...]
Channels:
  Index 0: PRIMARY psk=default {
    "psk": "AQ==", "moduleSettings":
       { "positionPrecision": 32 },
       "channelNum": 0, "name": "",
       "id": 0, "uplinkEnabled": false,
       "downlinkEnabled": false }
  Index 1: SECONDARY psk=secret {
    "psk": "ZggZyjGrS1Z0lxgjsXlflfxgTtJmLcudFszS92/Q048=",
    "name": "MonCanal", "channelNum": 0,
    "id": 0, "uplinkEnabled": false,
    "downlinkEnabled": false }
```

Les Heltec V3 arrivent dans une jolie boîte contenant la carte, deux fois 18 broches à souder, une antenne très modeste et un connecteur JST pour un accu LiPo (non visible ici).

Nous avons, à présent, le nom et le PSK en base64 que nous pouvons répercuter sur les autres nœuds via l'interface web ou l'application smartphone d'un simple copier/coller. Une fois cet ajout fait sur l'ensemble des nœuds, nous avons alors trois moyens de communication à notre disposition :

- via le canal primaire : pour tout le monde sur tout le réseau *mesh* ;
- via des messages directs d'un nœud à un autre en spécifiant l'ID de la cible : ***Messages***/***Nodes*** dans l'interface web, tapez sur un nœud et ***Message direct*** dans l'application mobile ou `--sendtext "mon message"` accompagné de `--dest` suivi de l'ID précédé d'un `!`. Attention, il faut échapper ce caractère qui a une utilisation spécifique dans le shell, c'est donc `\!` qu'il faut utiliser ;
- via un canal privé protégé par une clé AES où les messages restent donc privés entre les nœuds qui possèdent cette clé, notre « sous-réseau *mesh* », en somme.

Notez que si vous configurez plusieurs canaux, ceux-ci doivent être consécutifs. Vous ne pouvez pas avoir un « trou » dans la liste des canaux.

6. MODULES

Le *firmware* Meshtastic ne fait pas que de s'occuper de la communication sur le réseau maillé et de fournir l'interface avec le monde extérieur. Il intègre également, selon la plateforme, plusieurs modules logiciels permettant d'activer des fonctionnalités sans avoir à modifier ou reconstruire le *firmware*. En réalité, il ne s'agit pas du tout d'une sorte de SDK prévu pour être adapté en fonction des capteurs à mettre en œuvre. Un nœud doit surtout être vu comme une interface à mettre en œuvre avec un microcontrôleur complémentaire, chargé de gérer les capteurs. La question est donc, comment faire cela et, en même temps, ajouter des fonctionnalités au *firmware*, et la réponse tient en un mot : module.

Si vous faites un tour dans la section éponyme, dans l'interface web par exemple, vous trouverez une série de formulaires vous permettant d'activer et configurer ces modules. Nous n'allons pas tout traiter ici, mais uniquement nous concentrer sur trois d'entre eux.

Premièrement, nous avons ***Serial*** permettant d'établir une nouvelle liaison série, plus simple à utiliser que celle disponible via

l'interface USB (CP2102 avec les Heltec V3) permettant certes de récupérer les messages reçus et d'en envoyer, mais utilisant un protocole qui permet aussi la configuration, impliquant de passer, par exemple, par un client Python. Ici, avec cette nouvelle interface, nous définissons deux GPIO pour RX/TX (47 et 48, par exemple) et ce sont uniquement les messages qui transiteront. Ceci nous permettra alors de connecter une carte ESP8266, RP2040 ou Arduino (attention c'est 3,3 V) pour simplement envoyer et recevoir des messages sans autre forme de procès, en texte brut.

Nous avons ensuite le module ***Telemetry*** qui, comme son nom l'indique, permet de mesurer et diffuser des informations sur « l'état de santé » d'un nœud. En activant ce module, de base, ce sont les mesures de charge d'accu, de tension, d'utilisation des canaux et du « temps d'antenne » qui sont concernées, mais il est possible d'aller plus loin. Dans le cas des Heltec V3, par exemple, les GPIO 41 et 42 (respectivement SDA et SCL) donnent accès par défaut à un bus i2c scanné automatiquement au démarrage. Si un capteur pris en charge par le module (voir **[5]**) est détecté, il sera automatiquement utilisé. C'est le cas, par exemple, du BME280 (température, pression et hygrométrie) qu'il suffira de brancher à ces GPIO (plus la masse et l'alimentation) pour que ces mesures s'ajoutent et apparaissent directement dans la liste des nœuds voisins sur l'application smartphone, par exemple.

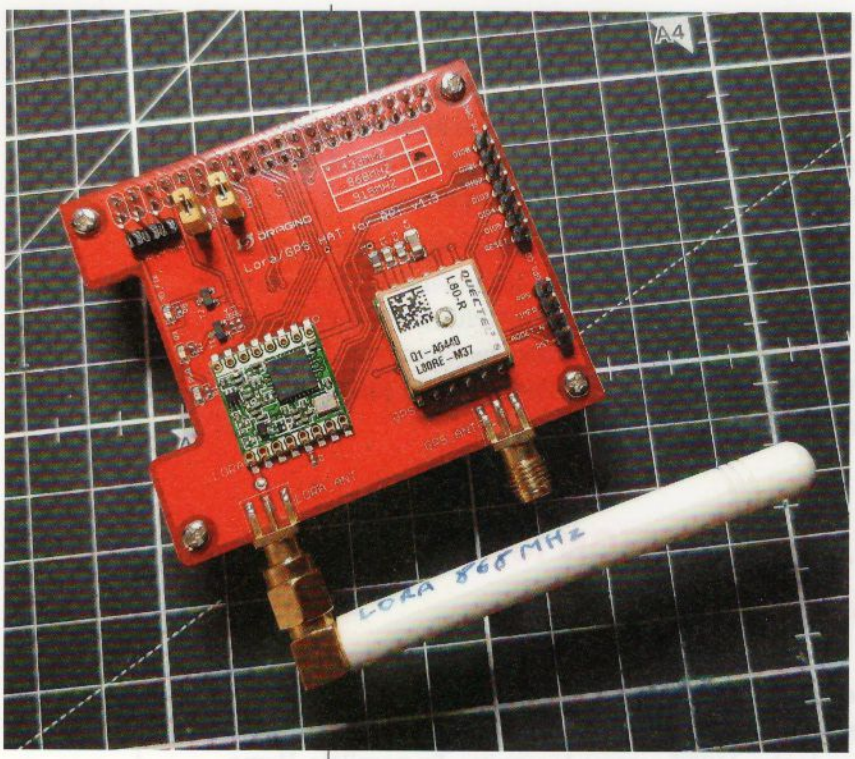
D'autres modules de ce genre sont également disponibles comme la notification externe, la lumière ambiante, la détection de changement d'état d'une GPIO (capteur PIR, par exemple) ou encore MQTT (avec une connexion Wi-Fi) qui offre une possibilité d'intégration à Home Assistant, et est le troisième module qui nous intéresse...

7. INTÉGRATION RAPIDE À HOME ASSISTANT

En ayant un de nos nœuds connecté en Wi-Fi au réseau local, le module MQTT activable dans le *firmware* Meshtastic peut remonter les informations qu'il reçoit directement à un *broker* MQTT comme Mosquitto. En parallèle, Home Assistant (ou HA dans la suite du texte) peut également utiliser ce *broker* comme source d'information (et plus encore) pour collecter des mesures et intégrer les valeurs dans ses statistiques. En d'autres termes, il est possible de déployer ses capteurs avec Meshtastic et en faire des entités dans HA.

Pour ce faire, nous devons tout d'abord disposer d'un *broker* MQTT sur notre réseau. Inutile de chercher plus loin que HA lui-même, proposant directement Mosquitto sous la forme d'un module complémentaire (ou *add-on*) installable via le menu ***Paramètres*** en un simple clic. Le *broker* local détecté apparaîtra alors automatiquement dans les intégrations et pourra être configuré. Sans plus de configuration, le *broker* utilisera les informations des utilisateurs locaux (votre compte pour l'interface web HA) pour authentifier les connexions. Vous pourrez donc rapidement vérifier le bon fonctionnement de l'installation de base en allant dans ***Paramètres***, ***Appareils et services***, ***MQTT***, ***CONFIGURER*** et ***Écouter un sujet***. Là, mettez un # comme sujet (pour « tout » écouter), activez ***Mettre en forme le contenu JSON*** et cliquez ***COMMENCER A ÉCOUTER***. Sur une autre machine, comme une Pi, utilisez la commande

```
mosquitto_pub -u utilisateur -P mot2passe -h machineHA.local -t homeassistant/switch/1/on -m "ON"
```



Le HAT Dragino pour Raspberry Pi que nous avions utilisé pour un précédent article sur TTN (The Things Network) est équipé d'une puce Semtech SX1276/SX1278 qui n'est, pour l'heure, pas compatible avec le service meshtasticd basé sur portduino.

(paquet `mosquitto-clients`) et le message doit alors apparaître côté HA, démontrant que la configuration et l'authentification fonctionnent.

Tournez-vous ensuite vers votre nœud disposant d'une connexion Wi-Fi, puis activez et configurez le module MQTT. Renseignez l'adresse IP de votre hôte HA (la résolution mDNS ne semble pas fonctionner, il faut une IP), le nom d'utilisateur et son mot de passe, et activez l'option ***JSON Enabled***. Enregistrez les changements pour appliquer le tout (*reboot*).

À ce stade, nous devons avoir une connexion possible entre notre *mesh* et le *broker* MQTT, et de ce fait, vers l'intégration HA. Mais nous n'avons dit nulle part ce qui devait être transféré du réseau *mesh* au *broker*. Pour une utilisation privée, c'est via notre canal précédemment configuré que ceci va se passer. Retournez donc dans la configuration de ce canal et activez l'option ***Uplink Enabled*** pour que tous les messages en question soient publiés en MQTT.

La configuration appliquée, utilisez ce canal privé pour envoyer un message et vous verrez alors apparaître dans HA quelque chose comme :

```
{
    "channel": 1,
    "from": 4201363136,
    "hops_away": 0,
    "id": 195453466,
    "payload": {
        "text": "Coucou message"
    },
    "rssi": -13,
    "sender": "!fa6baf80",
    "snr": 6,
    "timestamp": 1712915779,
    "to": 4294967295,
    "type": "text"
}
```

Notre message sur ce canal a été capté par un nœud avec une liaison au LAN et celui-ci a été relayé au *broker* depuis lequel l'intégration HA l'a récupéré et affiché. Nous avons donc un moyen de remonter les informations de n'importe quel nœud directement dans HA ! Notez que dans cette sortie exemple, le message est envoyé (*sender*) par le nœud `!fa6baf80`, celui avec Wi-Fi et MQTT activé, mais qu'en réalité il provient de l'autre nœud, comme précisé par la propriété `from` qui vaut `4201363136`, et donc `0xfa6bb6c0`, l'ID du nœud qui n'est **pas** connecté en Wi-Fi. Cette gymnastique décimale/hexadécimale est pénible et les termes *sender* et *from* sont clairement source de confusion, mais c'est bel et bien `from` qu'il faut considérer.

Pour l'intégration à proprement parler, ceci dépendra de ce que vous comptez faire avec vos nœuds, quels capteurs seront utilisés et comment. Ceci sort du cadre du présent article qui se concentre sur le réseau *mesh* LoRa et non sur l'intégration MQTT dans HA. Mais la documentation de Meshtastic donne un bon point de départ **[6]** et il en va de même pour celle de HA **[7]**.

CONCLUSION

Pour l'heure, ne vous étonnez pas de ne pas voir vos voisins Meshtastic. Il n'existe, à ce jour, pas énormément de nœuds en France, selon l'une des cartes accessibles en ligne **[8]**. Celle-ci n'en montre seulement qu'un peu plus d'une centaine pour la France contre plus de 1500 au Royaume-Uni. Personnellement, je suis relativement isolé étant donné que le nœud le plus proche se trouve à quelque 50 km de chez moi à la frontière suisse (mais qui sait, avec une bonne antenne...). Espérons que le présent article suscite de la motivation pour participer au réseau, car en ce qui me concerne, je compte effectivement apporter ma pierre à l'édifice en procédant à une installation permanente sur le toit de mon habitat.

Ceci dit, même en dehors de l'aspect participatif, Meshtastic constitue une solution vraiment intéressante pour déployer des capteurs en dehors de la portée d'autres solutions, et ce pour un coût tout à fait raisonnable, voir nul pour ce qui est du fonctionnement même de l'installation.

Mais, oui, j'aimerais vraiment, mais **vraiment**, beaucoup voir apparaître des nœuds autour de moi. Mes Heltec V3 se sentent tellement seuls.... **DB**

RÉFÉRENCES

[1] https://connect.ed-diamond.com/Hackable/hk-019

[2] https://meshtastic.org/docs/overview/range-tests/

[3] https://meshtastic.org/docs

[4] https://github.com/meshtastic/firmware

[5] https://meshtastic.org/docs/configuration/module/telemetry/

[6] https://meshtastic.org/docs/software/integrations/mqtt/home-assistant/

[7] https://www.home-assistant.io/integrations/mqtt/

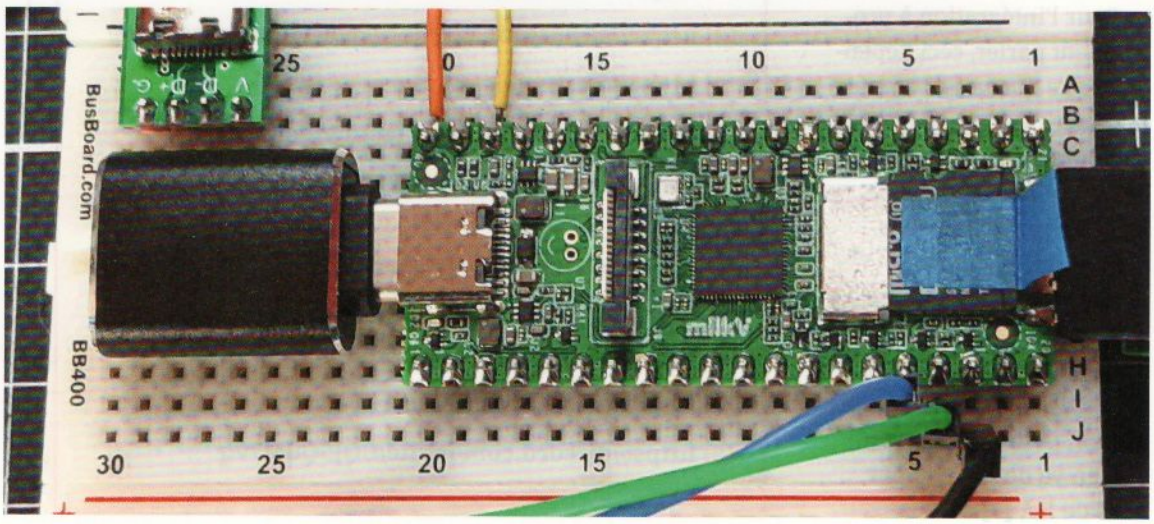
[8] https://meshtastic.liamcottle.net/

[9] https://discord.com/invite/UQJ5QuM7vq

MILK-V DUO : UN MINUSCULE SBC RISC-V À 8 €

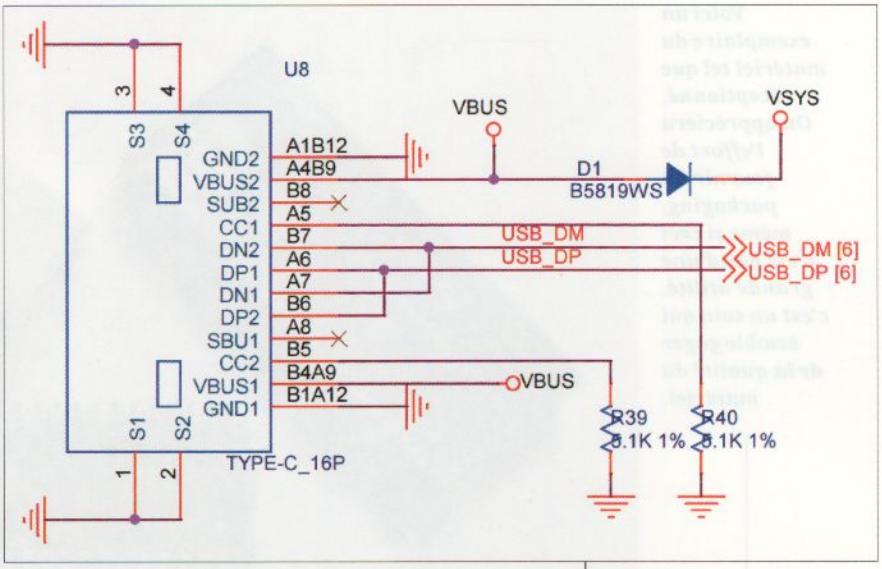
Denis Bodor

RISC-V (prononcé « RISC five ») est une ISA (ou architecture de jeu d'instructions) avec des spécifications ouvertes, pouvant être librement utilisée (contrairement à ARM). Cette architecture s'est enfin frayé un chemin dans le monde des microcontrôleurs où elle est maintenant clairement omniprésente (cf. ESP32-C3 et consorts, par exemple), mais peine encore à se populariser vraiment du côté des SoC. Nous sommes encore loin d'une invasion de SBC octa-core à prix défiant toute concurrence, mais cela ne saurait tarder (sauf si le sieur Xi Jinping décide d'envahir un pays démocratique voisin). En attendant, on peut déjà se faire la main avec quelque chose de plus modeste, comme le Milk-V Duo pour une petite poignée (ou pincée) d'euros…



Entre 8 € et 10 €, voilà ce que vous coûtera une carte Milk-V équipée d'un SoC (ou « microcontrôleur » selon la doc officielle) CVITEK CV1800B incluant deux processeurs RISC-V C906, à 1 GHz et 700 MHz, et 64 Mio de DRAM. Une version 256 Mio est également disponible, mais à un prix presque trois fois supérieur, car intégrant également un ARM Cortex A53. La carte possède 40 broches (20 de chaque côté, dont 26 GPIO) avec un format très réduit (21 mm par 51 mm, littéralement du DIP-40) et compatible platine à essais. Une interface Ethernet (PHY 100 Mbit/s) est intégrée au SoC, mais le transformateur/connecteur RJ45 est vendu séparément pour ~4 €.

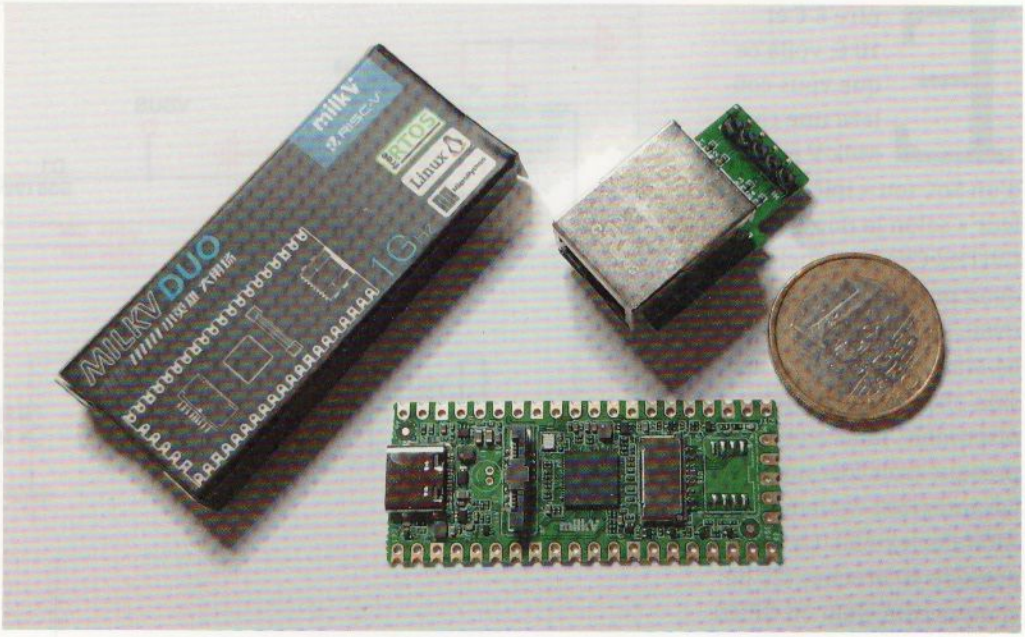
Le stockage est assuré par un support microSD (présent) ou un emplacement à souder pour une flash SD NAND (mémoire intégrant un contrôleur SD) et le reste de la connectivité se limite à un connecteur FPC (16 broches MIPI CSI 2-lane) pour une caméra et un port USB-C OTG (USB 2.0) servant en premier lieu à l'alimentation, mais parfaitement capable de fonctionner en hôte si on alimente la carte via les broches VBUS (40) et GND (38 par exemple). À noter que l'alimentation externe se fait normalement via VSYS (39), mais que le port USB-C ne fournit pas d'alimentation dans ce cas.



Le circuit d'alimentation via USB-C montre clairement la différence entre VBUS et VSYS, disponible par ailleurs via les broches de la carte (respectivement 40 et 39).

Nous n'avons donc ni sortie vidéo ni Wi-Fi intégré, mais pour ce prix, il serait déplacé de faire la fine bouche. À noter qu'une carte hôte existe (~20 €), au format rappelant celui d'une Arduino, permettant de disposer d'un RJ45 pour l'Ethernet en plus de 4 ports USB A (hub intégré). Ceci est une alternative au module Ethernet et permettra de facilement ajouter des périphériques USB (interface Wi-Fi, Bluetooth, support de stockage, etc.). Personnellement, je n'ai pas opté pour cette solution, préférant acquérir plusieurs Milk-V Duo pour un coût équivalent.

Enfin, le SoC intègre également un TPU (*Tensor Processing Unit*) pour les applications de *machine learning* (non traité ici), ainsi qu'un MCU 8 bits 8051 avec 8 Kio de SRAM (non traité non plus) alimenté par le circuit RTC et donc susceptible de réveiller le système en sommeil. Les documentations (schémas, *Product Brief* et *datasheet* de ~700 pages) sont disponibles en PDF directement dans le dépôt GitHub officiel **[1]**, pour cette carte et deux autres déclinaisons (Duo 256M et Duo S).



Voici un exemplaire du matériel tel que réceptionné. On appréciera l'effort de fournir un packaging, même si ceci n'est pas d'une grande utilité, c'est un soin qui semble gager de la qualité du matériel.

1. CONSTRUIRE UN SYSTÈME

Milk-V Technology met à disposition un SDK **[2]** pour sa plateforme avec une introduction pour la construction relativement détaillée et claire. Mais suivre simplement les instructions et les reproduire n'est pas vraiment le style de la maison et nous allons donc opter pour une approche différente.

Le SDK « officiel » est basé sur Buildroot sous la forme d'un clone maintenu en parallèle des sources officielles. Un développeur qui semble faire partie de Milk-V Technology a cependant proposé une contribution au projet pour ajouter le support de cette plateforme. Nous allons donc utiliser un Buildroot « standard » et lui appliquer les modifications proposées par le contributeur pour obtenir ce qui sera, à terme, la *release* officielle Buildroot intégrant ce support. Ceci nous permettra non seulement de faire un petit rappel de la procédure de construction d'un Buildroot pour une plateforme supportée, mais dans le même temps, voir comment intégrer une contribution non encore acceptée officiellement, à partir des messages échangés sur les listes de diffusion.

Pour nous faciliter la vie, nous n'allons pas utiliser les archives Pipermail officielles **[3]**, mais un miroir regroupant les messages de plusieurs listes de diffusion : **https://lore.kernel.org/**. Là, vous trouverez celle de Buildroot, parfaitement tenues à jour et nous recherchons alors les messages du contributeur, Hanyuan Zhao.

Les premiers messages datent de décembre 2023 avec une première proposition de contribution découlant sur un certain nombre de commentaires, questions et conseils pour adapter les patchs à la philosophie du projet. Suite aux échanges avec Giulio Benetti, Hanyuan Zhao reprend son code et produit plusieurs versions jusqu'à la V4 qui nous intéresse ici et qui a déclenché la réponse que tout contributeur souhaite recevoir : « *Everything looks good to me now* ». Trois messages nous seront utiles :

- « *Subject: [Buildroot] [PATCH v4 1/3] package/milkv-duo-libraries: new package* »
- « *Subject: [Buildroot] [PATCH v4 2/3] package/milkv-duo-smallcore-freertos: new package* »
- « *Subject: [Buildroot] [PATCH v4 3/3] configs/milkv_duo: new defconfig* »

Nous pourrions nous amuser à copier/coller les patchs depuis le navigateur, au risque de faire des erreurs, mais nous allons tout simplement utiliser le lien « raw » figurant à droite du « Message-ID » (d'où l'utilisation du miroir qui propose cela) pour obtenir un fichier mbox pour chaque message et les enregistrer respectivement dans :

- `package_milkv-duo-libraries.mbox`
- `package_milkv-duo-smallcore-freertos.mbox`
- et `configs_milkv_duo.mbox`

Nous avons des mbox, la belle affaire ! Nous n'allons tout de même pas les utiliser tels quels, si ? Hé bien si, justement. Nous commençons par obtenir une copie du dépôt officiel de Buildroot depuis GitHub, directement dans le même répertoire où sont les fichiers mbox, avec :

```
$ git clone https://gitlab.com/buildroot.org/buildroot.git
$ cd buildroot
```

Puis on applique, tout simplement, les patchs avec la commande `git am` :

```
$ git am ../configs_milkv_duo.mbox
Application de  configs/milkv_duo: new defconfig
$ git am ../package_milkv-duo-libraries.mbox
Application de  package/milkv-duo-libraries: new package
$ git am ../package_milkv-duo-smallcore-freertos.mbox
Application de  package/milkv-duo-smallcore-freertos: new package
```

Non seulement Git va se charger tout seul de prendre chaque message de la mbox (ici, un seul par fichier) pour en extraire le patch, mais il va automatiquement créer des *commits* correspondants en utilisant les informations de l'expéditeur et le contenu du ou des messages. C'est magique ! À noter qu'il peut être intéressant (si on y pense avant, pas comme moi) de créer une nouvelle branche Git avant d'appliquer ces modifications avec `git am`. Ceci gardera les modifications « parasites » clairement séparées des sources *upstream*.

Mais nous n'en avons pas fini. En effet, dans l'un des messages de Hanyuan Zhao, on peut lire :

```
"This patch depends on:
https://patchwork.ozlabs.org/project/buildroot/list/?series=393667
to work-around a gcc bug on htop package."
```

En suivant le lien, on découvre deux messages/patchs de Giulio Benetti corrigeant un problème avec la commande `htop` et l'optimisation effectuée par le compilateur GCC. L'interface web proposant, en haut à droite, le téléchargement des deux messages sous la forme de mbox, nous procédons donc de même que précédemment (fichiers `toolchain_gcc_109809.mbox` et `package_htop.mbox`).

Après avoir utilisé `git am` à nouveau, nous pouvons voir dans le log Git :

```
$ git log --oneline --abbrev-commit --all --graph
* f04dfcab21 (HEAD -> master) package/htop: fix build failure due to gcc bug 109809
* 978b75464d toolchain: introduce BR2_TOOLCHAIN_HAS_GCC_BUG_109809
* e10aea75e1 package/milkv-duo-smallcore-freertos: new package
* 2550c7584e package/milkv-duo-libraries: new package
* cdbdc5d654 configs/milkv_duo: new defconfig
* 0ee43b0157 (origin/master, origin/HEAD) package/libnss: bump version to 3.99
* 54dbd8e2c5 package/poco: needs C++17
* 9347905b95 package/squid: fix build with host gcc 10
[...]
```

Notre *HEAD* est maintenant 5 *commits* en avance sur `origin/master` et sont adorablement listées les trois modifications du contributeur et la correction du *bug* `htop`. Tout est prêt pour démarrer la construction de notre système Buildroot et nous suivons donc la procédure standard.

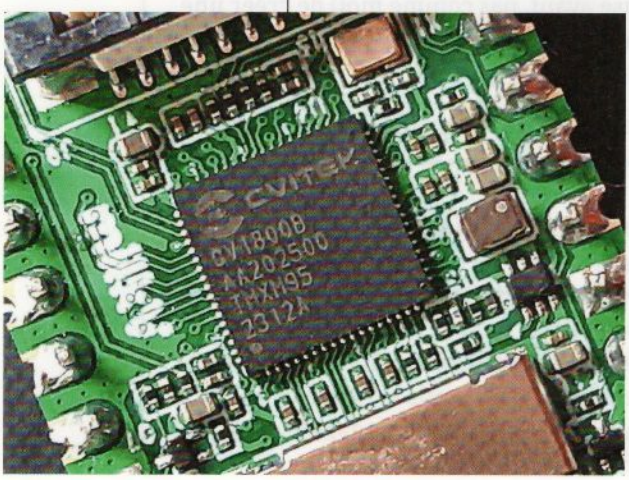
Nous commençons par initialiser le tout avec comme base la configuration par défaut de la plateforme (obtenue grâce aux patchs) :

```
$ make milkv_duo_musl_riscv64_defconfig
```

Ceci fait, nous avons l'opportunité de revisiter cette configuration via l'interface de menu de Buildroot accessible via :

```
$ make menuconfig
```

Le CVITEK CV1800B est un SoC double cœur RISC-V C906 cadencé à 1 GHz et 700 MHz. Il intègre de base 64 Mio de DRAM, 32 Kio de cache, un MCU 8051, un TPU et un contrôleur Ethernet.



Buildroot est un projet fantastique qui prend en charge directement tous les éléments qui composent la génération et la construction d'un système embarqué, de la chaîne de compilation aux applications utilisateurs en passant par le noyau Linux, le (ou les) *bootloader(s)*, les services, les outils système, etc. Et tout ceci est configurable via cette interface. Une fois vos choix faits, vous pouvez tout simplement quitter l'interface et la configuration sera enregistrée dans le fichier `.config`. Mais pour l'heure, nous voulons voir à quoi ressemble le système et pour cela nous devons le compiler/construire à l'aide d'un simple `make`. Qu'on aura toutefois la présence d'esprit de compléter d'une option `-j` pour paralléliser les compilations, en ajoutant le nombre de *jobs* à utiliser, correspondant généralement au nombre de cœurs/*threads* de votre CPU (sauf si vos voulez garder un peu de ressources en réserve, ***tousse*** Firefox ***tousse***).

Cette première construction prendra plus de temps que les suivantes du fait du téléchargement des éléments et de la création du compilateur croisé. Mais après quelques minutes, la procédure se conclura par :

```
[...]
>>>   Executing post-image script board/milkv/duo/post-image.sh
[Duo Post-Image fiptool.py] Integrating FreeRTOS
[Duo Post-Image fiptool.py] > fip.bin generated!
FIT description: Various kernels, ramdisks and FDT blobs
Created:         Mon Mar 18 16:37:56 2024
Image 0 (kernel-1)
  Description:  cvitek kernel
  Created:      Mon Mar 18 16:37:56 2024
  Type:         Kernel Image
  Compression:  lzma compressed
  Data Size:    2845767 Bytes = 2779.07 KiB = 2.71 MiB
  Architecture: RISC-V
  OS:           Linux
  Load Address: 0x80200000
  Entry Point:  0x80200000
  Hash algo:    crc32
  Hash value:   73030155
Image 1 (fdt-cv1800b_milkv_duo_sd)
  Description:  cvitek device tree - cv1800b_milkv_duo_sd
  Created:      Mon Mar 18 16:37:56 2024
  Type:         Flat Device Tree
  Compression:  uncompressed
  Data Size:    23473 Bytes = 22.92 KiB = 0.02 MiB
  Architecture: RISC-V
  Hash algo:    sha256
  Hash value:   9fc46f08a9f79de6b7235d4297a22c7a
                5e9ae41067153cf0d8bbf8feafd9965f
Configuration 0 (config-cv1800b_milkv_duo_sd)
  Description:  boot cvitek system with board cv1800b_milkv_duo_sd
  Kernel:       kernel-1
  FDT:          fdt-cv1800b_milkv_duo_sd
[Duo Post-Image] > boot.sd generated!
```

Vous trouverez dans `output/images/` le résultat de tous ces efforts (du CPU) et en particulier le fichier `sdcard.img` de quelque 70 Mio. C'est l'image de la carte SD que vous pourrez transférer avec `dd` :

```
$ sudo dd if=output/images/sdcard.img of=/dev/sdc
139265+0 enregistrements lus
139265+0 enregistrements écrits
71303680 octets (71 MB, 68 MiB) copiés,
         20,9799 s, 3,4 MB/s
```

Il suffira alors de glisser la SD/MMC dans l'emplacement prévu à cet effet sur la carte et, après avoir connecté un adaptateur USB/série aux broches 18 (GND), 17 (RX) et 16 (TXT) et lancé votre outil de communication préféré (Minicom ou GNU Screen, par exemple) configuré en 115200 8N1, vous pourrez alimenter la Milk-V Duo via son connecteur USB-C. Devraient alors défiler les messages du FSP (*First Stage Bootloader*), de U-Boot (SPL pour *Secondary Program Loader*) et enfin du noyau Linux. Et tout cela devrait se conclure par :

```
Welcome to Buildroot
buildroot login:
```

L'utilisateur est `root` et aucun mot de passe n'est nécessaire.

2. TOUR DU PROPRIÉTAIRE

Parmi les choses qu'on remarque au démarrage ou peu après, ce sont les étranges messages qui apparaissent dans le terminal, mêlés à ceux du noyau et des services :

```
RT: [6.430265]Hello Milk-V Duo from the small core! [2/3]
```

Le « *small core* » est le second cœur intégré au SoC CV1800B et qui, présentement, est autonome et fait fonctionner une application FreeRTOS **en parallèle** de Linux fonctionnant sur le premier cœur. Ce code a été compilé avec l'ensemble du reste des éléments et le binaire résultant est placé dans `output/images/cvirtos.bin`. Il sera intégré par le système de construction dans l'image FIP (*Firmware Image Package*) intégrant également le *bootloader* OpenSBI (qui lui-même charge U-Boot). On retrouve d'ailleurs une trace de ce chargement dès le début des messages de démarrage :

```
T: [1.418995]CVIRTOS Build Date:Mar 18 2024  (Time :16:31:11)
RT: [1.424569]Post system init done
RT: [1.427648]create cvi task
RT: [1.430271]Hello Milk-V Duo from the small core! [1/3]
```

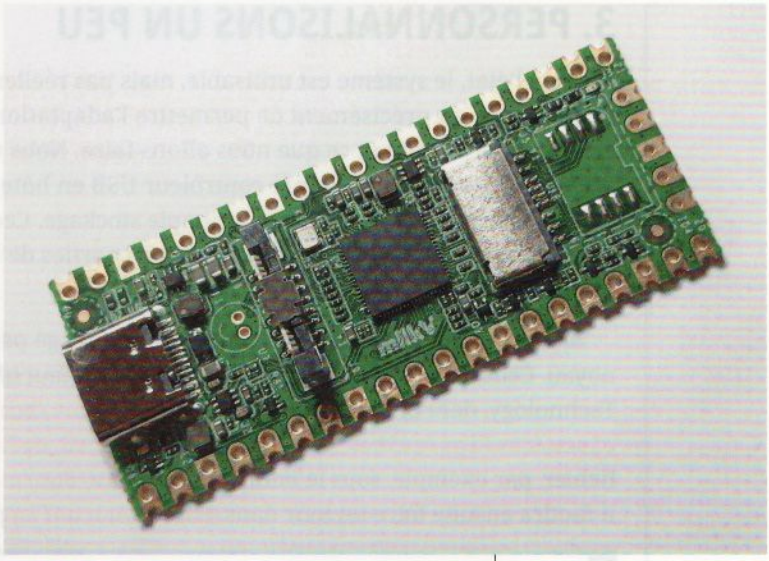
FIP est une structure de donnée liée au mécanisme *Trusted Firmware-A* (ou TF-A) permettant de sécuriser le *boot* d'un système embarqué. L'intérêt est d'avoir, en une seule archive binaire, un ensemble d'éléments à charger en mémoire depuis un support de stockage. Mais l'important ici n'est pas tant le mécanisme de chargement, que le code qui est utilisé. Suite à la compilation, vous trouverez les sources de ce simple « Hello World » dans une archive placée dans `dl/milkv-duo-smallcore-freertos` ou, plus simplement, dans le sous-répertoire `milkv-duo-smallcore-freertos` de `output/build/` (les sources sont également accessibles via un dépôt GitHub dédié **[4]**). Et, si vous le souhaitez, vous pouvez parfaitement désactiver sa construction (***Target packages***, ***Hardware handling***, et ***milkv-duo-smallcore-freertos*** (`BR2_PACKAGE_MILKV_DUO_SMALLCORE_FREERTOS`)).

J'avoue que cette surprise est fort intéressante, sachant qu'au départ mon objectif était surtout de disposer d'une plateforme RISC-V à un prix défiant toute concurrence. Ceci mérite d'être creusé exhaustivement et le sera peut-être dans un prochain article (c'est un gros morceau, tout comme la partie 8051).

Après cette surprise vient le moment de faire le tour de ce qu'a ajouté Milk-V Technology dans la configuration. Nous avons par exemple une LED bleue clignotant régulièrement, mais aucune entrée `/sys/class/leds`. Ceci est pris en charge d'une autre manière et on se tourne tout naturellement vers `/etc/init.d` (Dieu merci, pas de *systemd* ici) et `S99user`, en particulier.

Ce script lance `blink.sh` depuis `/opt/milkv-duo`, contenant d'autres choses intéressantes, qui est constitué d'une simple boucle utilisant `echo` (`DUO_WRITE` défini dans `/etc/milkv-duo.conf`) pour écrire dans `/sys/class/gpio/gpio440` à répétition (pas ce qu'il y a de plus élégant, je vous l'accorde).

Le contenu de `/etc/milkv-duo.conf` nous apprend également que la configuration du port USB est aussi gérée de cette manière. Nous y trouvons une variable `DUO_USB_FUNC` initialisée avec `RNDIS`. Celle-ci est utilisée par un autre script d'init, `/etc/init.d/S10hwinit`, pour appliquer la configuration choisie (`RNDIS`, `HOST` ou `MASS-STORAGE`) via les scripts `usb-rndis.sh`, `usb-host.sh` et `usb-mass-storage.sh` placés dans `/opt/milkv-duo`. Et en effet, en alimentant la carte via un hub connecté à un PC, nous avons effectivement une interface réseau qui apparaît : `usb0` renommé en une salade de caractères composée de « enx » et de la MAC de l'interface (merci, Red Hat, pour cette « fantastique » idée de *predictable NIC names* introduite avec... systemd v197).



La carte Milk-V duo se présente avec un format identique à celui d'une carte Raspberry Pi Pico et, comme la Pico, les libellés des broches sont sérigraphiés de l'autre côté du circuit. Notez le petit smiley, c'est en réalité le connecteur pour un micro.

Si, comme moi, vous avez opté pour l'option Ethernet à 4 €, ceci n'est pas d'une grande utilité et nous pouvons tout simplement modifier `/etc/milkv-duo.conf` pour ajuster la ligne en `DUO_USB_FUNC=HOST`. Il faudra, bien sûr, alimenter la carte via les broches VBUS/GND, mais nous disposerons alors d'une interface USB hôte au prochain démarrage, ou tout simplement en exécutant `/opt/milkv-duo/usb-host.sh` (un simple `echo host > /proc/cviusb/otg_role`). Notez que tous les pilotes USB ne sont pas compilés dans le noyau et que les modules ne sont pas activés. Le support du stockage USB (`CONFIG_USB_STORAGE` dans la configuration du *kernel*), en revanche, est supporté de base, mais c'est quelque chose qu'on peut souhaiter changer.

3. PERSONNALISONS UN PEU

Dans l'état, le système est utilisable, mais pas réellement agréable à gérer. La raison d'être de Buildroot est précisément de permettre l'adaptation de la configuration à ses besoins et c'est donc précisément ce que nous allons faire. Nous voulons, dans le désordre, disposer d'un utilisateur standard, utiliser le contrôleur USB en hôte par défaut et ajouter du support pour d'autres périphériques USB que le simple stockage. Ceci est totalement arbitraire, bien sûr, et clairement destiné à couvrir les différentes parties de Buildroot, plutôt que de constituer un exemple pratique réel.

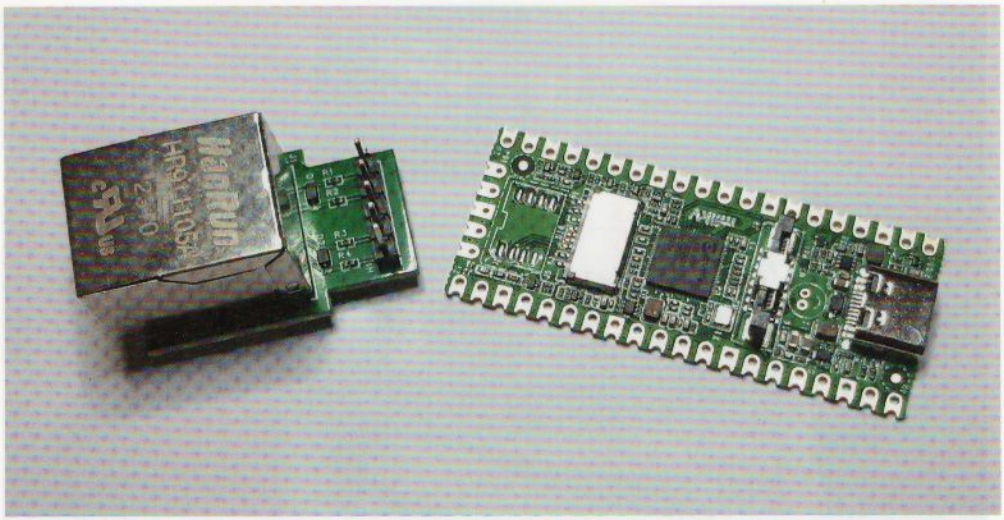
Ajouter le support du matériel non pris en charge par défaut passe par la configuration du noyau. Celle actuellement utilisée est celle par défaut (*defconfig*) du noyau en version Milk-V Technology, dont on trouve une copie dans `output/build/linux-duo-linux-5.10.4/arch/riscv/configs/cvitek_cv1800b_milkv_duo_sd_defconfig`. On commencera donc par copier ce fichier, par exemple, sous le nom `board/milkv/duo/cvitek_cv1800b_milkv_duo_sd.config`. Il faudra ensuite faire un tour dans `make menuconfig` pour préciser l'utilisation de ce fichier :

```
Kernel
  Kernel configuration
    Using a custom (def)config file
  Configuration file path
    cvitek_cv1800b_milkv_duo_sd.config
```

Buildroot sait donc à présent quelle configuration il doit appliquer aux sources du noyau et nous pouvons l'adapter à nos besoins. Pour cela, nous utilisons `make linux-menuconfig` pour entrer dans l'interface de configuration du noyau. Là, nous chargeons notre configuration existante via l'option ***Load*** et nous faisons le tour des options pour ajuster la configuration selon nos envies. Ici, ce sera par exemple l'ajout du support USB/série :

```
Device Drivers
  USB support
    USB Modem (CDC ACM)
    USB Serial Converter support
      USB Generic Serial Driver
      USB Serial Simple Driver
      USB Winchiphead CH341 Single Port Serial Driver
      USB CP210x family of UART Bridge Controllers
      USB FTDI Single Port Serial Driver
      USB Prolific 2303 Single Port Serial Driver
```

Enfin, nous enregistrons la configuration (***Save***) dans le même fichier que celui que nous avons précédemment chargé. La partie *kernel* est maintenant gérée, nous pouvons passer à l'ajustement de la configuration par défaut du système. Les fichiers que nous avons listés précédemment, qu'il s'agisse des scripts ou des fichiers de configuration, proviennent du contributeur et plus exactement d'une archive téléchargée par un paquet ajouté : `dl/milkv-duo-libraries/milkv-duo-libraries*.tar.gz`. On retrouve ces éléments, désarchivés par le système de construction, dans `output/build/milkv-duo-libraries*/overlay`.



Le connecteur/transformateur RJ45 pour l'interface Ethernet est vendu séparément et ne peut, dans l'état, pas être connecté à la carte, car les via sont d'un diamètre trop petit. Je suppose donc que ce « module » est relativement générique et absolument pas dédié à cette plateforme.

Notre approche sera relativement brouillonne, mais a le mérite d'être rapide. Nous allons, tout simplement, écraser ce contenu par nos propres fichiers. Ce n'est pas très fin, mais comme l'archive en question contient bien plus de choses que ces scripts et configurations, nous nous évitons de devoir adapter encore plus de choses encore. Pour écraser ces fichiers, nous reposerons sur le mécanisme d'*overlay* de Buildroot, en précisant dans la configuration que nous souhaitons « ajouter une couche » en fin de construction. Pour cela, nous commençons par copier l'arborescence complète du répertoire, spécifié précédemment, dans `board/milkv/duo/myoverlay`.

Nous pouvons alors modifier les fichiers selon nos besoins et en particulier configurer le contrôleur USB en hôte plutôt qu'en périphérique RNDIS, en changeant une simple ligne de `milkv-duo.conf` :

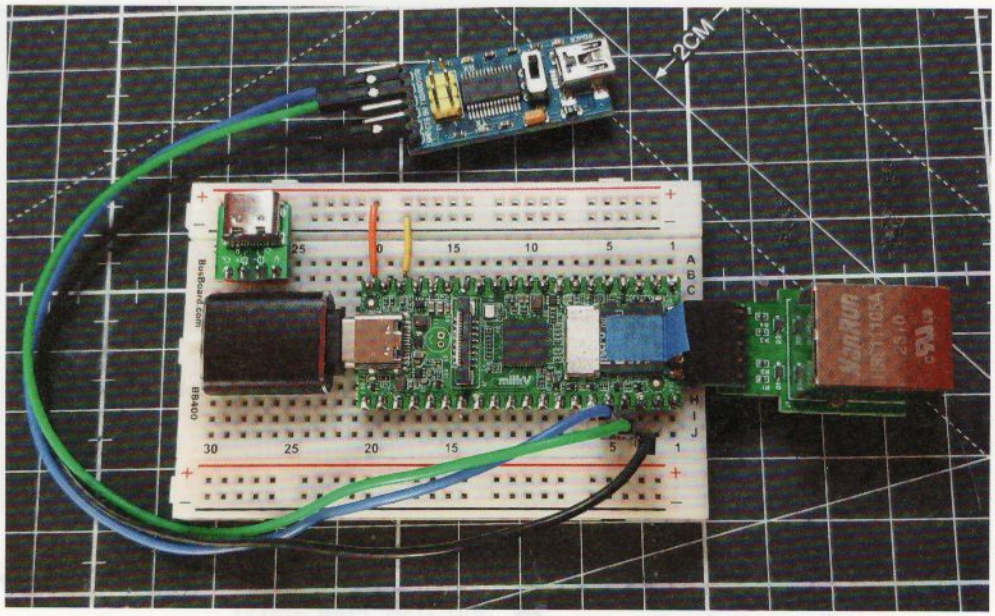
```
DUO_USB_FUNC=HOST
```

Notez que changer `ENABLE_BLINK` à `0` n'est pas d'une grande utilité. La LED bleue cesse effectivement de clignoter, mais `S99user` est tout de même exécuté et appelle toujours `blink.sh`, dont la boucle `while` ne cesse de sourcer `milkv-duo.conf` pour finalement ne rien faire. Mon dieu, que c'est laid ! Quoi qu'il en soit, nous refaisons un tour de `make menuconfig` pour préciser l'ajout de l'*overlay* :

```
System configuration
  Root filesystem overlay directories
    board/milkv/duo/myoverlay
```

Suite à la construction du système, nous pourrons nous assurer de la présence de ces fichiers en inspectant le contenu de `output/images/rootfs.tar` ou, plus simplement, en jetant un œil à `output/target/etc` (qui sert de base pour la création de `rootfs.tar`).

Il ne nous reste plus qu'à nous occuper de l'ajout d'un utilisateur standard, et ceci est relativement simple avec Buildroot. Il nous suffit de créer un fichier qui servira de base pour la création de l'utilisateur, que nous placerons dans `board/milkv/duo/mkusers.table` et contenant la ligne : `denis -1 denis -1 =coucou /home/denis /bin/sh wheel,plugdev,dialout,sudo Utilisateur de base`. Ceci aura pour effet de créer un



Voici l'une des cartes en situation de test avec le convertisseur USB/série pour la console, une alimentation par les broches (plutôt que par USB-C), le module RJ45 assemblé et un adaptateur USB-C mâle vers USB-A femelle permettant la connexion de périphériques (mode USB hôte).

utilisateur `denis`, avec un ID attribué automatiquement, dans un groupe `denis` avec ID automatique, un mot de passe `coucou` (il sera *hashé* SHA-256), ayant pour répertoire personnel `/home/denis`, utilisant le shell `/bin/sh` et faisant partie des groupes supplémentaires `wheel`, `plugdev`, `dialout` et `sudo`.

Une fois n'est pas coutume, pour intégrer ce fichier dans la configuration, cela se passe via `make menuconfig` :

```
System configuration
  Path to the users tables
    board/milkv/duo/mkusers.table
```

Et on en profitera pour activer `sudo` :

```
Target packages
  Shell and utilities
    sudo
```

On pourra éventuellement vouloir également modifier la configuration de BusyBox, fournissant un jeu de commandes standard avec une empreinte « mémoire » minimale. Comme pour le noyau, on repêchera la configuration actuellement utilisée (`package/busybox/busybox.config`) pour la copier en lieu sûr (`board/milkv/duo/busybox.config`, et on pourra l'ajuster avec `make busybox-menuconfig`. Il suffira ensuite d'intégrer cette configuration comme les précédentes :

```
Target packages
  BusyBox configuration file to use?
    board/milkv/duo/busybox.config
```

Et enfin, vous pourriez procéder de même pour U-Boot (`output/build/uboot-v2021.10_64mb/configs/cvitek_cv1800b_milkv_duo_sd_defconfig` et `make uboot-menuconfig`) :

```
Bootloaders
  U-Boot configuration
    Using a custom board (def)config file
  Configuration file path
    board/milkv/duo/uboot_cvitek_cv1800b_milkv_duo_sd.config
```

À l'issue de ces modifications, on fera un brin de ménage pour être sûr (normalement pas utile, sauf si on se mélange les pinceaux comme moi) avec un `make linux-dirclean` et on relancera une construction avec `make` (sans oublier le `-j`). L'image obtenue sera alors transférée, comme précédemment, sur la SD et *bootée* sur la carte.

Si rien n'a été oublié, le système se comportera alors par défaut comme nous l'attendons. Vous noterez cependant lors de l'utilisation de `sudo` qu'un avertissement est affiché :

```
random: sudo: uninitialized urandom read (40 bytes read)
```

Ceci disparaîtra naturellement au bout d'un certain temps et vous serez prévenu lorsque ce sera le cas par un autre message :

```
random: crng init done
```

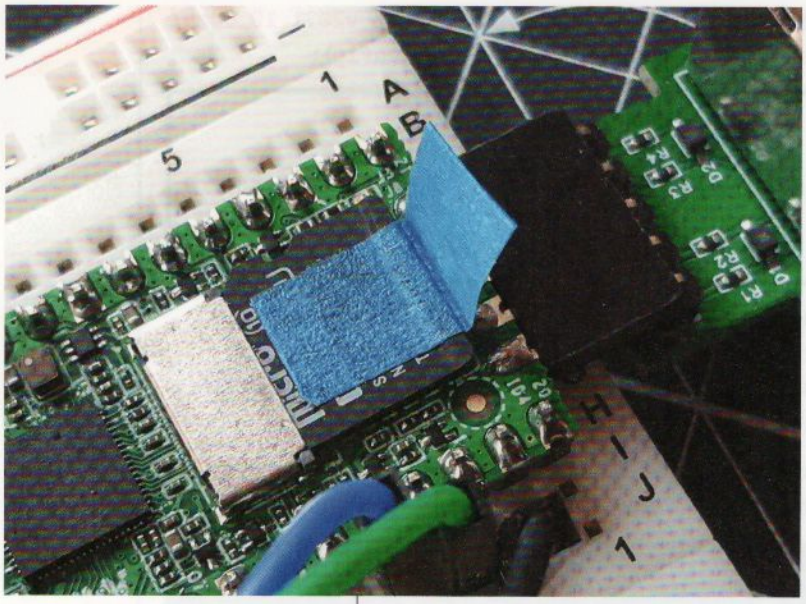
`sudo`, comme d'autres outils utilisant des fonctions cryptographiques, a besoin d'entropie et celle-ci est générée par le fonctionnement normal du système au fil du temps. La situation dans laquelle vous vous trouvez lors de ce premier démarrage est celle d'une « carence d'entropie » (*Boottime Entropy Starvation*). Vous pouvez accélérer le processus en chargeant le système, ou même en le *pingant* si vous avez l'interface Ethernet.

La connexion d'un adaptateur USB/série, quant à elle, fonctionnera sans le moindre problème :

```
usb 1-1: new full-speed USB device number 4 using dwc2
ch341 1-1:1.0: ch341-uart converter detected
usb 1-1: ch341-uart converter now attached to ttyUSB0
```

CONCLUSION À 8 €

Cette carte est très intéressante, mais tout n'est pas rose pour autant. Matériellement déjà, nous avons des comportements très suspects, comme le fait que selon l'adaptateur USB/série utilisé, la carte est plus ou moins toujours alimentée via un courant parasite présent sur RX/TX. La LED rouge témoignant de la mise sous tension est allumée dès que RX/TX/GND sont connectés et, avec au moins un adaptateur de ma collection, la carte a démarré, affichant des caractères aléatoires via la liaison série. Sans doute sans rapport, et peut-être en raison d'une erreur



Quelle que soit la manière de souder ou de connecter le module RJ45, les broches interfèrent avec l'insertion et le retrait du support de stockage SD/MMC. L'astuce consiste donc à se confectionner une petite « poignée » facilitant (un peu) les manipulations.

de manipulation de ma part, au moins une des mes cartes n'est plus capable de recevoir des caractères via la liaison série (tout en continuant à en afficher). Le système fonctionne parfaitement, mais les caractères n'arrivent plus.

Côté logiciel également, et bien que nous avons ici soigneusement évité d'utiliser le SDK du constructeur, il faut se rendre à l'évidence : c'est un *kernel* modifié et un *bootloader* modifié. Le support Milk-V n'est absolument pas présent dans les sources *vanilla* de Linux et de U-Boot. En parlant d'U-Boot justement, et même si ceci demande davantage d'investigation, il semblerait qu'une partie de la configuration soit « en dur » dans le code. Modifier certains paramètres comme le `bootdelay` fonctionnera sans problème, mais il ne semble pas être possible, dans l'état, de gérer un environnement stocké en fichier sur la partition FAT de la SD. La configuration n'a strictement aucun effet et pour cause, c'est le fichier `include/configs/cv180x-asic.h` qui intègre la quasi-totalité des éléments de l'environnement.

En soi, approcher le projet Buildroot pour contribuer au support de cette plateforme part d'une bonne intention (je suppose), mais il ne s'agit que de transposer des briques, pour certaines assez brouillonnes, provenant du SDK. Il aurait été plus judicieux, mais certes plus difficile, de proposer ce support plus *upstream*, directement dans Linux et U-Boot avant de le faire pour Buildroot.

Je suis peut-être un peu rude, mais cette plateforme, je pense, a énormément de potentiel et souffre finalement du même problème que les autres SBC peu chers que l'on trouve de-ci de-là : le support semble être excessivement ponctuel et dès qu'une nouvelle version du matériel sera mise sur le marché, tout s'arrêtera (voir disparaîtra). C'est un contexte qui fait que ce genre de plateforme ne sera sans doute jamais adaptée à une application industrielle ou même simplement commerciale, car sans assurance d'une certaine continuité dans le support (celle-là même pouvant être assurée en ayant un support dans des projets *upstream*), il n'est pas vraiment possible d'avoir un minimum de confiance dans l'avenir du produit. Si demain Milk-V Technology disparaît, ou décide de simplement fermer son compte GitHub

pour une raison ou une autre, l'intérêt de cette contribution à Buildroot sera réduit à néant, car la construction/compilation deviendra tout bonnement impossible.

Cependant, pour apprendre et explorer, à la fois le monde de Buildroot et celui des architectures RISC-V, l'investissement est loin d'être prohibitif. Huit malheureux euros pour une carte capable de faire fonctionner un système GNU/Linux 64 bits de la taille d'un composant DIP-40 (ou d'une Raspberry Pi Pico), qui en plus n'est pas de l'ARM ou du MIPS, est plus qu'on aurait pu en rêver il y a encore 4 ou 5 ans. Je dirai même que c'est une excellente plateforme pour un projet ponctuel ou une petite réalisation faite sur un coin de table, mais guère plus.

Espérons naïvement que le grand raz de marée RISC-V, tant attendu pour ajouter une concurrence bien méritée à la totale domination d'ARM, ne sera pas de cette nature, avec un écosystème fractionné en une multitude de plateformes qu'on ne peut supposer qu'éphémères. Ça, à mon sens, serait la pire chose qui puisse arriver à RISC-V, mais attendons un peu de voir comment cela va tourner avant d'ouvrir les paris, et croisons simplement les doigts... DB

RÉFÉRENCES

[1] https://github.com/milkv-duo/duo-files

[2] https://github.com/milkv-duo/duo-buildroot-sdk

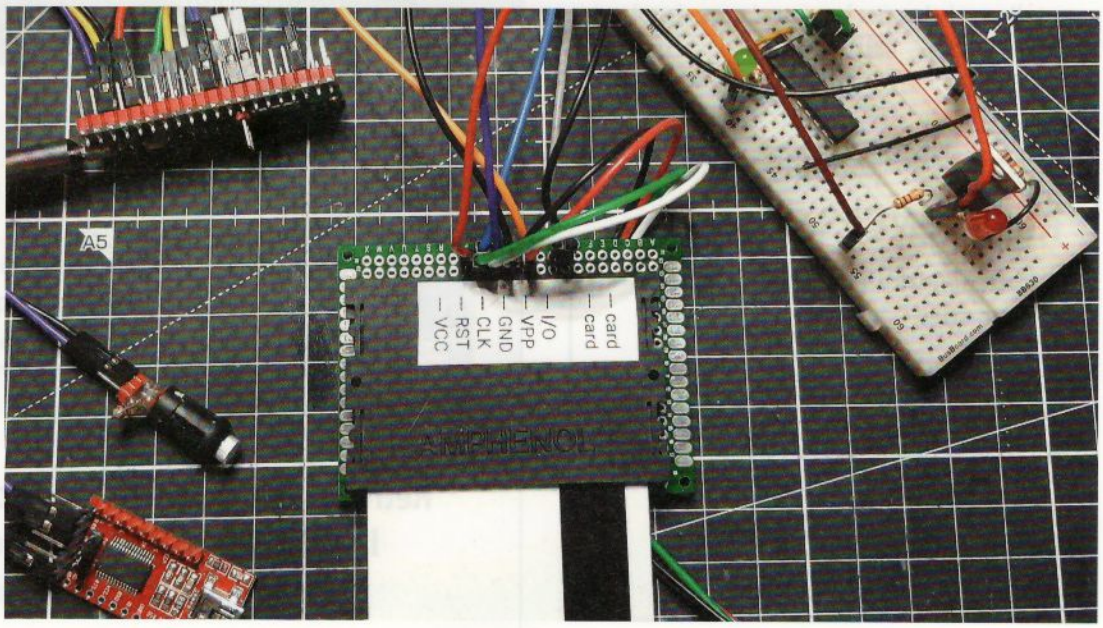
[3] https://lists.buildroot.org/pipermail/buildroot/

[4] https://github.com/milkv-duo/milkv-duo-smallcore-freertos

CARTE À PUCE ET MICROCONTRÔLEUR

Denis Bodor

Les cartes à puce ou smartcards reposant sur la norme ISO/IEC 7816 sont omniprésentes dans nos vies. Cartes bancaires, cartes Vitale, cartes SIM dans votre smartphone ou encore tantôt cartes permettant l'authentification ou la signature électronique sont autant d'exemples des services que ces petits bouts de plastique peuvent rendre. Il existe maintes solutions permettant d'interagir avec une smartcard sur PC, un Mac ou un SBC sous GNU/Linux, généralement via un lecteur USB CCID. Mais qu'en est-il lorsqu'on souhaite que notre montage à microcontrôleur utilise ce type de périphérique ? Il y a la voie simple avec l'utilisation d'une IC dédiée ou un support intégré au MCU, et la voie plus « créative » que nous allons explorer de ce pas…



Notre objectif ici n'est pas de créer un produit ou même de tenter de démontrer qu'il serait possible de le faire, mais de comprendre comment fonctionne la communication entre une carte à puce (ou plutôt une UICC pour *Universal Integrated Circuit Card*) et un lecteur (le terminal), et ce, au plus bas niveau. Ceci n'est pas possible avec un lecteur USB ou même un circuit intégré spécialisé comme le SEC1110 ou le SEC1210 de Microchip Technology qui est, de plus, relativement difficile à mettre en œuvre sans équipement adéquat (*package* QFN-24). Le fait d'arriver à échanger des données entre une smartcard et une carte à microcontrôleur comme une Raspberry Pi Pico, ou un module ESP32, permet de valider la compréhension de la technologie et fournit également les clés pour espionner (*sniffing*) les communications, lorsqu'elles ne sont pas chiffrées, avec ce type de dispositifs « dans la nature ».

Notez que je ne suis de loin pas le seul à avoir tenté l'expérience et qu'il existe plusieurs projets similaires, mais utilisant une approche différente. Citons par exemple l'extension NARD de *killergeek* **[1]** pour le Flipper Zero **[2]** reposant sur une puce SEC1210-I/PV-UR2, ou encore CardStalker **[3]** provenant de l'ANSSI, lié à Open-ISO7816-Stack **[4]** et implicitement au projet WooKey **[5]**, les trois étant très orientés vers les plateformes STM32 proposant des facilités d'interfaçage dans ce sens. Il en va de même avec l'Hydra-BUS **[6]** reposant sur un STM32F405. Personnellement, appréciant grandement les STM32, mais n'ayant pas spécialement d'affinité avec l'environnement de développement STM32Cube et la salade de code qu'il génère, j'ai opté pour un développement parallèle RP2040 et ESP32 puisqu'il n'y a, de plus, aucun intérêt à réimplémenter ce qui existe déjà. Ceci ajoute une petite difficulté, dans le sens où, contrairement aux STM32, les RP2040 et ESP32 n'ont aucun support matériel pour ce type de communication (cf. plus loin). Travailler en parallèle sur les deux plateformes nous permettra également de voir que de petites nuances dans les *frameworks* et bibliothèques peuvent parfois avoir un impact très important sur le développement (et sur sa capacité à se retenir de jurer comme un charretier, tout seul devant son écran).

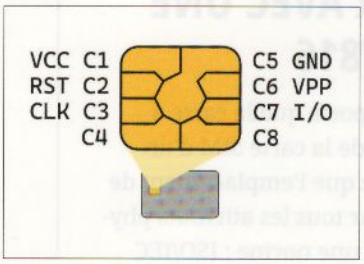
Un lecteur USB CCID comme celui-ci est pratique et peu cher. C'est une excellente solution, à condition d'utiliser un PC ou un SBC, mais n'a pas grand intérêt si l'on veut effectivement comprendre ce qui se passe au plus bas niveau.

1. COMMUNIQUER AVEC UNE SMARTCARD ISO-7816

Si vous prenez en main n'importe quelle carte à puce, qu'il s'agisse d'une CB ou de la carte SIM d'un smartphone, vous remarquerez que l'emplacement de la puce est toujours le même, car tous les attributs physiques du matériel répondent à une norme : ISO/IEC

7816. Celle-ci couvre tous les aspects, non seulement physiques, comme la taille de la carte et l'endroit où se trouve la puce, mais également le nombre et l'utilité des contacts, les tensions et protocoles utilisés, le format des données échangées, les mécanismes de sécurité, etc. Tout est dans la norme, mais bien entendu, celle-ci n'est pas accessible gratuitement en ligne, et à raison d'environ 150 € par partie (ISO/IEC 7816 en compte 15) via le *webstore* de l'ANSI, ceci devient rapidement prohibitif. Fort heureusement, et parfaitement légalement, les informations peuvent être obtenues via d'autres documentations, dont les notes d'application des constructeurs, les *datasheets* de certains composants liés à cette technologie ou encore les PDF de l'institut européen des normes de télécommunications (ETSI **[7]**).

En compilant toutes ces sources d'informations, plus d'autres au travers du Web, on finit par obtenir tout ce qui est nécessaire pour attaquer le sujet, en commençant par le niveau le plus bas, à savoir, le brochage des contacts :



En dehors de C4 et C8 qui sont utilisés pour des usages spécifiques dépendants des constructeurs (dont la connexion directe en USB pour certains modèles), nous avons :

- C5 GND : l'indispensable masse ;
- C1 VCC : l'alimentation de la puce avec une tension qui correspond à la classe de tension de la carte. Typiquement 1,8V, 3,3V ou 5V, correspondants respectivement aux classes C, B et A ;
- C6 VPP : c'est la tension de programmation qui n'est, actuellement, plus utilisée et laissée non connectée ;
- C3 CLK : le signal d'horloge permettant de cadencer le processeur dans la puce, avec une fréquence entre 1 MHz et 5 MHz et un rapport cyclique entre 40 % et 60 %. Typiquement, on utilisera un signal carré d'une fréquence de 4 MHz le plus stable possible ;
- C2 RST : le *reset* de la carte, actif à l'état bas ;
- C7 I/O : la ligne de données, bidirectionnelle, *half-duplex*. La communication est toujours initiée par le lecteur (terminal) qui, après l'envoi des données, passe immédiatement en lecture. Cette particularité est précisément celle directement prise en charge par certains STM32 et que nous devrons régler d'une autre manière avec le RP2040 et/ou un ESP32.

Pour connecter les contacts de la carte avec notre microcontrôleur, le plus simple est d'utiliser un matériel dédié comme un connecteur Amphenol FCI qui vous coûtera moins de 5 euros chez un revendeur comme Mouser (réf. 649-7434L0825S01LF, par exemple). Vous pouvez aussi tenter d'en récupérer un dans un équipement existant ou encore bricoler vous-même quelque chose de vaguement fiable. Très honnêtement, pour le coût, mieux vaut ne pas tenter le diable, opter pour un matériel dédié et éviter l'enfer des faux contacts, il y a suffisamment d'autres points qui peuvent poser problème...

1.1 Alimenter la carte

La première problématique qui se pose est celle de l'alimentation, car avec des tensions s'étalant de 1,8V à 5V, la potentialité d'une émission olfactive est non nulle. La documentation ETSI **[8]** précise clairement la procédure

à suivre qui se résume à : alimenter l'UICC (la carte) avec la tension la plus basse et si aucun ATR n'est reçu, passer à la classe et donc à la tension supérieure. Si l'ATR est corrompu, désactiver l'alimentation et procéder à un *reset* au moins trois fois de suite avant de passer à la tension supérieure. Enfin, si l'ATR stipule une classe précise, le terminal doit désactiver la carte (couper l'alimentation) et recommencer la procédure avec la tension correspondante.

Et là, vous vous demandez naturellement « c'est quoi, l'ATR ? ». Juste après un *reset*, une carte à puce conforme à la norme ISO/IEC 7816 émet une série de caractères constituant « la réponse au *reset* », ou « *Answer To Reset* » en anglais. C'est l'ATR. Celui-ci, d'une taille entre 2 et 33 caractères (ou octets) est la « salutation » et la carte d'identité de l'UICC. Il informe le terminal des caractéristiques à utiliser pour la communication, des spécifications de la carte (dont la tension), du protocole à utiliser (orienté « caractères » ou « blocs ») pour les échanges ou encore des informations spécifiques au constructeur (*historical bytes*). L'ATR est le seul message envoyé par la carte sans qu'un échange soit initié par le lecteur (sauf si on considère que le *reset* est une action initiée par le terminal).

Suivre la procédure donnée par le standard est idéal, mais pas forcément nécessaire dans le sens où vous pouvez parfaitement savoir à quelle carte à puce vous avez à faire. Dans le cadre de ces expérimentations, j'ai testé des NXP J2A081, NXP J3R150, NXP H3H145, ACS ACOSJ et même une carte SIM SFR ou encore une ancienne CB Crédit Mutuel sans le moindre problème en 3,3 volts. Je ne me serais certes pas risqué à faire de même en 5V, en particulier sans lire la documentation du constructeur lorsqu'elle est disponible, mais 3,3V est non seulement

Voici les smartcards utilisées pour les tests de ce petit (?) projet. Il s'agit de Java Cards de différentes générations et qualités. Deux d'entre elles utilisent le protocole T=0 et seule la J2A081 communique en T=1.

la tension utilisée par le RP2040 et l'ESP32, mais aussi celle la plus couramment utilisée.

Il est parfaitement envisageable de créer un circuit capable de suivre cette procédure, en utilisant plusieurs LDO (régulateur de tension) pour fournir les tensions adéquates. Les GPIO du microcontrôleur peuvent être utilisées pour piloter des MOSFET, mais il conviendra ensuite d'également adapter les niveaux de tension pour les signaux logiques, si nous travaillons en 1,8V ou en 5V. Cela devient vite compliqué...

À propos de GPIO justement, activer et désactiver une carte en fournissant VCC via une sortie du microcontrôleur peut ou ne peut pas fonctionner. En effet, le standard précise que le courant utilisé par l'UICC est au maximum de 50 mA (classe A et C) ou 60 mA (classe B). Dans le cas d'un ESP32, une broche GPIO configurée en sortie peut fournir un maximum de 40 mA, ce

Ce type de lecteur SIM se trouve sur AliExpress pour moins de 1,5 €/pièce (+port ~2 €). Ils ne sont pas compatibles CCID et utilisent un logiciel propriétaire pour Windows, mais à ce prix, c'est une bonne source d'approvisionnement pour un connecteur SIM (et potentiellement une excellente victime pour un peu de reverse engineering).

qui semble suffisant pour toutes les cartes testées. Il n'en va pas de même pour la Raspberry Pi Pico qui au-delà de 15 mA voit la tension s'effondrer. Il en résulte une communication impossible avec certaines cartes qui ne répondent tout simplement pas, et tantôt un comportement plus intéressant, avec des réponses vaguement aléatoires (ce *glitching* est furieusement intéressant et mériterait un article complémentaire). La solution est alors soit d'abandonner totalement l'idée d'alimenter la carte via une sortie du RP2040 et brancher VCC sur « 3V3 » (broche 36), soit utiliser un MOSFET pour piloter l'alimentation. Attention cependant, tous les MOSFET ne peuvent pas être contrôlés en 3,3V et selon le modèle, un transistor devra être ajouté (voir le très intéressant billet sur le blog ArduinoDIY **[9]**).

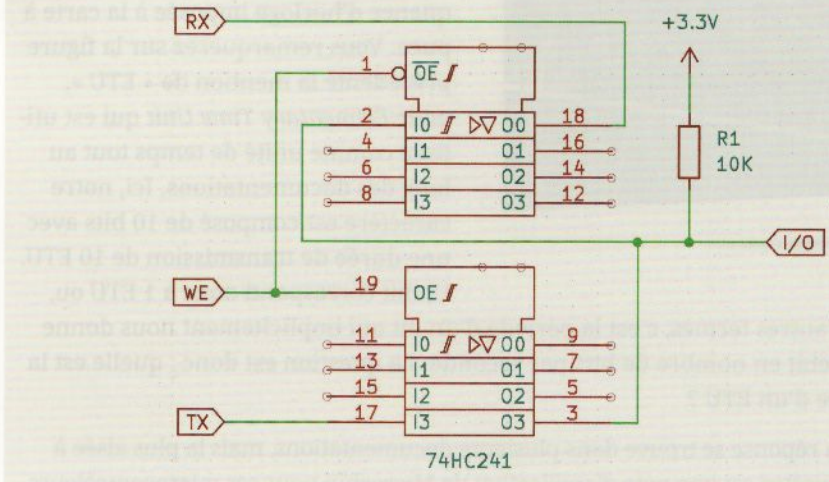
1.2 Le problème de la communication half-duplex

Vous l'aurez compris, le tout premier objectif à atteindre est d'arriver à obtenir l'ATR et donc de confirmer que la carte est correctement alimentée et cadencée. Après fourniture de l'alimentation, d'une manière ou d'une autre, un *reset* doit être effectué en mettant RST à la masse durant au minimum 400 cycles d'horloge. La carte va alors envoyer l'ATR via I/O et nous devons être à l'écoute. Mais suite à cela, ce sera au microcontrôleur d'envoyer des données et donc de contrôler l'état de I/O. Ceci n'est pas sans rappeler le bus I²C, à la différence que nous avons ici une communication asynchrone (pas de SCL) tout à fait similaire à une communication série, mais sur une seule ligne faisant office à la fois de TX et de RX.

Pour nous plier à ces caractéristiques peu courantes, nous avons plusieurs options. Utiliser un microcontrôleur prenant ceci en charge directement, comme un STM32, a déjà été écarté et devons donc procéder manuellement. Une approche possible consiste à n'utiliser qu'une seule broche GPIO et à changer la direction au besoin. Ceci suppose de ne plus reposer sur un périphérique UART intégré au microcontrôleur et de lui substituer une implémentation logicielle, comme le *SoftwareSerial* d'Arduino. Cela peut fonctionner et on trouve de-ci de-là des croquis Arduino relativement basiques reposant sur cette technique.

Je ne trouve pas cette solution spécialement élégante et potentiellement source de problèmes. Mieux vaut effectivement reposer sur le matériel à disposition et implémenter une solution à l'extérieur du microcontrôleur. En utilisant un octuple *buffer* à trois états comme le 74HC241 (« HC » est important ici pour un fonctionnement en 0/3,3V), nous pouvons connecter les lignes RX et TX du microcontrôleur respectivement à une entrée et une sortie du *buffer* et faire de même avec I/O sur deux autres entrées et sorties reliées entre elles. On pourra alors connecter les deux signaux d'activation (OE et /OE) du 74HC241 à une sortie du microcontrôleur et ainsi régler dans quelle

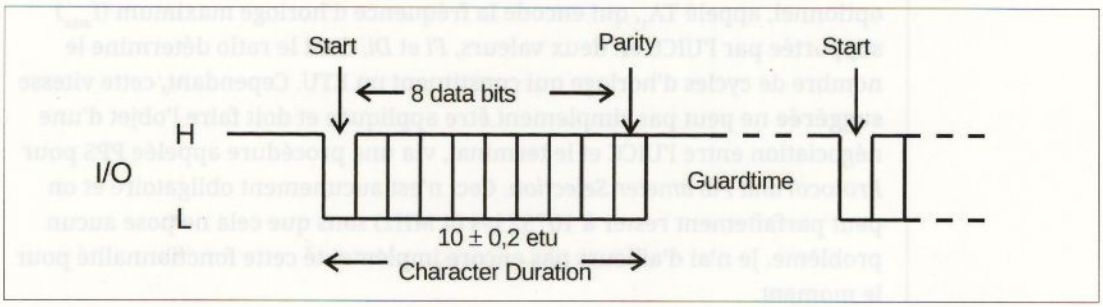
direction se passera la communication. L'intérêt du 74HC241 réside précisément dans les trois états des E/S utilisées : haut, bas et haute impédance (« en l'air »). Nous déconnectons effectivement et alternativement RX et TX du microcontrôleur de la ligne I/O de la carte en fonction de l'état de OE et /OE, que nous pilotons avec un signal appelé WE (ou WR) pour l'occasion (pour *Write Enable* ou *WRite*). Nous obtenons donc le circuit très simple suivant :

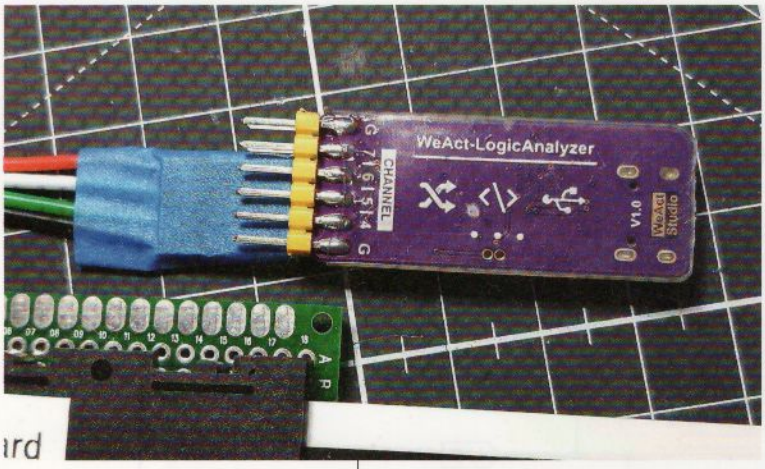


Dans notre code, tout ce que nous aurons à faire pour envoyer des données, c'est de passer WE à l'état haut, d'écrire sur le port série, puis d'immédiatement repasser WE à la masse pour recevoir la réponse de la carte. Ceci permettra non seulement les échanges, mais également la réception de l'ATR juste après le *reset*.

1.3 Horloge et vitesse de communication

Nous avons réglé le problème de la liaison physique avec l'UICC, mais ceci n'est qu'une partie du problème. La documentation montre que la communication asynchrone utilise un format de données sur 10 bits, avec un bit de *start*, huit bits de données et un bit de parité paire. Chaque octet (ou caractère) transmis de part et d'autre est suivi d'un temps supplémentaire où la ligne est à l'état haut (le *guardtime*). Ceci peut être réglé facilement en configurant deux bits de stop (ou 1,5, si c'est possible). Ce n'est pas exactement ce qu'attend la norme, mais ça fonctionne. Tout ceci est parfaitement résumé dans la documentation ETSI **[8]** via le schéma suivant :





Un analyseur logique USB, même d'entrée de gamme, est un outil de choix lorsqu'on fait ce genre d'expériences avec des smartcards. Il permet de mettre le doigt sur des points ambigus du standard et de voir effectivement ce qui se passe au niveau des signaux, en cas de problème.

Vient ensuite la vitesse de communication et là, les choses sont un peu plus délicates, car il ne s'agit non seulement pas de débits standard (4800, 9600, 19200, etc.), mais la vitesse est variable, car dépendante de la fréquence d'horloge imposée à la carte à puce. Vous remarquerez sur la figure précédente la mention de « ETU », pour *Elementary Time Unit* qui est utilisée comme unité de temps tout au long des documentations. Ici, notre caractère est composé de 10 bits avec une durée de transmission de 10 ETU. Un bit correspond donc à 1 ETU ou, en d'autres termes, c'est la période d'un bit qui implicitement nous donne un débit en nombre de bits par seconde. La question est donc : quelle est la durée d'un ETU ?

La réponse se trouve dans plusieurs documentations, mais la plus aisée à interpréter est une note d'application de Microchip pour ses microcontrôleurs PIC **[10]**. Au *reset* de l'UICC, la durée d'un ETU correspond à 372 fois la période d'un cycle d'horloge, appliquée à la carte. Comme la période d'un signal est l'inverse de sa fréquence, nous avons $ETU = 372*(1/f)$, où f est la fréquence. Avec 4 MHz, ceci nous donne donc 372/4000000 soit 0,000093 seconde. Comme un ETU est également la durée d'un bit, nous avons ainsi une communication à 1/0,000093, soit 10752 b/s. Pour nous faciliter la vie, nous pouvons définir quelques macros dans ce sens dans le code :

```
#define SC_CLK_FREQ          4000000
#define SC_ETU_PERIOD_US(f)  (unsigned int)(((1.0/f)*372)*1000000)
#define SC_BAUDRATE(f,e)     (unsigned int)(1/((1.0/f)*372))
```

Mais ce n'est pas tout. Bien que ceci soit suffisant en principe pour recevoir l'ATR et même communiquer par la suite avec la carte, nous pouvons éventuellement changer cette valeur. Inclus à l'ATR se trouve un octet optionnel, appelé TA_1, qui encode la fréquence d'horloge maximum (f_{max}) supportée par l'UICC en deux valeurs, *Fi* et *Di*, dont le ratio détermine le nombre de cycles d'horloge qui constituent un ETU. Cependant, cette vitesse **suggérée** ne peut pas simplement être appliquée et doit faire l'objet d'une négociation entre l'UICC et le terminal, via une procédure appelée PPS pour *Protocol and Parameter Selection*. Ceci n'est aucunement obligatoire et on peut parfaitement rester à 10752 b/s (4 MHz) sans que cela ne pose aucun problème. Je n'ai d'ailleurs pas encore implémenté cette fonctionnalité pour le moment.

L'ATR ne fait cependant pas que des suggestions, et dans sa version la plus minimaliste de deux octets, le premier, appelé TS, encode la *convention d'encodage*, non seulement de l'ATR lui-même, mais de toute communication qui s'en suivra. Deux conventions existent :

- Directe : un état haut sur la ligne I/O signifie un 1 logique et un état bas, un 0 logique. Le bit de poids le plus faible dans un octet est le premier.
- Indirecte : c'est l'inverse, haut = 0 et bas = 1. Le bit de poids le plus faible est le dernier d'un octet.

TS ne peut avoir que deux valeurs, 0x3b pour une convention directe et 0x3f pour indirecte, et vous comprendrez que la seconde option peut nous poser un gros problème, puisque nous ne sommes pas du tout dans le fonctionnement standard d'un UART. Fort heureusement, strictement aucune des cartes à puce que j'ai testées, que ce soit avec un montage comme celui présenté ici ou un lecteur sur PC, n'utilisait une convention indirecte. Ceci est confirmé par le fichier `smartcard_list.txt` livré avec `pcsc_scan` **[11]** qui répertorie les ATR de quelque 4300 cartes, et seules moins de 200 d'entre elles ont un TS à 0x3f. J'ai donc décidé, pour l'instant, de remettre ceci à plus tard (la TODO s'allonge).

Nous avons maintenant toutes les briques permettant d'avoir une première communication avec une carte à puce compatible ISO/IEC 7816, il est donc grand temps d'écrire un peu de code.

2. L'OBJECTIF DE DÉPART : L'ATR

En résumé, pour obtenir l'ATR, nous devons alimenter la carte à puce, fournir un signal carré de 4 MHz, configurer l'UART et lire les données qui arrivent juste après avoir brièvement mis à la masse le signal RST.

2.1 Configuration des GPIO

En termes d'entrées/sorties, nous avons besoin d'un certain nombre de signaux :

- Masse et VCC : Pico et ESP32 sont tout deux capables de fournir le courant nécessaire à l'UICC en 3,3V, ainsi que d'alimenter le 74HC241. Aucune alimentation supplémentaire n'est donc nécessaire.
- RX et TX de l'UART : ces deux broches sont totalement dépendantes de la plateforme utilisée. Dans le cas de l'ESP32, nous avons plus de liberté qu'avec le RP2040. Toutes les broches pouvant être utilisées en GPIO sont susceptibles d'être configurées avec l'un des 3 UART qu'intègre le microcontrôleur. L'UART0 étant souvent utilisé pour les messages console, on optera naturellement pour UART1. Dans le cas du RP2040, qui possède deux UART, seul un nombre limité de broches peuvent être utilisées. Comme, là aussi, nous réservons l'UART0 pour la console, seules les GPIO 4, 8, 20 ou 24 peuvent être utilisées pour TX et 5, 9, 21 ou 25 pour RX sur UART1 (voir *datasheet* **[12]** page 13 et 14).
- Activation émission sur le 74HC241 (WE/WR) : ceci est une simple sortie que nous pilotons très classiquement. Aucune précaution particulière n'est nécessaire à ce niveau.
- Signal d'horloge CLK : ceci est définitivement une sortie, mais très particulière, puisque l'objectif est de générer un signal carré d'une fréquence relativement élevée. ESP32 comme RP2040 proposent des fonctionnalités dans ce sens, avec une approche sensiblement différente.
- *Reset* de l'UICC (RST) : encore une simple sortie tout à fait standard.

- Contact présence carte : la seule entrée du projet, connectée à l'interrupteur se trouvant dans le connecteur de carte à puce, normalement fermé, et qui devient ouvert lors de l'insertion d'une carte. L'autre broche dédiée du connecteur sera reliée à la masse et l'entrée se verra donc configurée avec une résistance de *pull-up* interne au microcontrôleur.

Voyons maintenant ce que cela donne pour chacune des deux plateformes.

2.1.1 Version RP2040

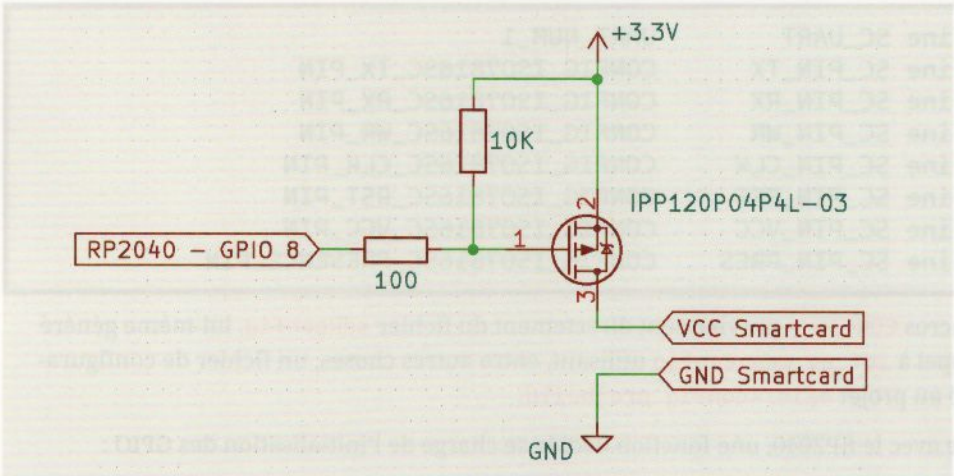
Pour simplifier la configuration et la portabilité, nous définissons quelques macros dans `main.h` :

```
#define SC_UART         uart1
#define SC_PIN_TX       4
#define SC_PIN_RX       5
#define SC_PIN_WR       6
#define SC_PIN_CLK      21
#define SC_PIN_RST      7
//#define SC_PIN_VCC    8
#define SC_PIN_PRES     9
```

Et, sur cette base, pouvons créer une fonction dédiée dans `main.c` :

```
void sc_gpio_init(void)
{
    gpio_init(SC_PIN_WR);
    gpio_set_dir(SC_PIN_WR, GPIO_OUT);
    gpio_put(SC_PIN_WR, 0);

    gpio_init(SC_PIN_RST);
    gpio_set_dir(SC_PIN_RST, GPIO_OUT);
    gpio_put(SC_PIN_RST, 1);

    /*
    gpio_init(SC_PIN_VCC);
    gpio_set_dir(SC_PIN_VCC, GPIO_OUT);
    gpio_put(SC_PIN_VCC, 0);
    */

    gpio_init(SC_PIN_PRES);
    gpio_set_dir(SC_PIN_PRES, GPIO_IN);
    gpio_pull_up(SC_PIN_PRES);
}
```

Notez la section en commentaire qui deviendra utile lorsque j'aurai reçu mes MOSFET Infineon IPP120P04P4L, utilisables directement (et sans transistor en complément) avec les niveaux de tension de la Pico. Pour l'heure, le VCC de la carte est connecté en permanence aux 3,3V régulés fournis sur la broche 36.



Les GPIO du RP2040 ne sont pas capables de fournir le courant nécessaire à l'alimentation d'une smartcard. Une solution possible est d'utiliser un MOSFET-P, comme l'Infineon IPP120P04P4L (plus de 3 €/pièce), avec un VGSth suffisamment bas pour être piloté directement en 3,3V. Ces MOSFET sont rares et plus chers, mais ils évitent d'utiliser un circuit plus complexe avec un transistor.

Pour utiliser les GPIO « standard », nous créons quelques fonctions placées dans `smartcard.c` :

```
void sc_enablewrite(void)
{
    gpio_put(SC_PIN_WR, 1);
}

void sc_disablewrite(void)
{
    gpio_put(SC_PIN_WR, 0);
}

void sc_reset(void)
{
    gpio_put(SC_PIN_RST, 0);
    sleep_ms(2);
    gpio_put(SC_PIN_RST, 1);
}

bool sc_ispresent(void)
{
    return gpio_get(SC_PIN_PRES);
}
```

Encore une fois, ceci est l'utilisation parfaitement courante des GPIO d'une Raspberry Pi Pico.

2.1.2 Version ESP32

L'implémentation ESP32 avec l'ESP-IDF est assez similaire à ce qu'on a avec le RP2040, mais l'environnement propose une fonctionnalité supplémentaire. Via le système de configuration (`idf.py menuconfig`), il est possible de rendre paramétrable l'assignation des GPIO à certaines fonctions et l'environnement se charge alors de créer des macros en conséquence. Notre `main.h` prend donc un aspect différent :

```
#define SC_UART         UART_NUM_1
#define SC_PIN_TX       CONFIG_ISO7816SC_TX_PIN
#define SC_PIN_RX       CONFIG_ISO7816SC_RX_PIN
#define SC_PIN_WR       CONFIG_ISO7816SC_WR_PIN
#define SC_PIN_CLK      CONFIG_ISO7816SC_CLK_PIN
#define SC_PIN_RST      CONFIG_ISO7816SC_RST_PIN
#define SC_PIN_VCC      CONFIG_ISO7816SC_VCC_PIN
#define SC_PIN_PRES     CONFIG_ISO7816SC_PRESENCE_PIN
```

Les macros `CONFIG_*` proviennent directement du fichier `sdkconfig`, lui-même généré par un appel à `idf.py menuconfig` utilisant, entre autres choses, un fichier de configuration dédié au projet `main/Kconfig.projbuild`.

Comme avec le RP2040, une fonction dédiée se charge de l'initialisation des GPIO :

```
static void sc_gpio_init(void)
{
    gpio_reset_pin(SC_PIN_WR);
    gpio_set_direction(SC_PIN_WR, GPIO_MODE_OUTPUT);
    gpio_set_level(SC_PIN_WR, 0);

    gpio_reset_pin(SC_PIN_RST);
    gpio_set_direction(SC_PIN_RST, GPIO_MODE_OUTPUT);
    gpio_set_level(SC_PIN_RST, 1);

    gpio_reset_pin(SC_PIN_VCC);
    gpio_set_direction(SC_PIN_VCC, GPIO_MODE_OUTPUT);
    gpio_set_level(SC_PIN_VCC, 0);

    gpio_reset_pin(SC_PIN_PRES);
    gpio_set_direction(SC_PIN_PRES, GPIO_MODE_INPUT);
    gpio_set_pull_mode(SC_PIN_PRES, GPIO_PULLUP_ONLY);
}
```

Et on retrouve un lot de fonctions dans `smartcard.c` pour simplifier les opérations :

```
void sc_enablewrite(void)
{
    gpio_set_level(SC_PIN_WR, 1);
}

void sc_disablewrite(void)
{
    gpio_set_level(SC_PIN_WR, 0);
}

void sc_reset(void)
{
```

```
    gpio_set_level(SC_PIN_RST, 0);
    vTaskDelay(2 / portTICK_PERIOD_MS);
    gpio_set_level(SC_PIN_RST, 1);
}

void sc_poweron(void)
{
    gpio_set_level(SC_PIN_VCC, 1);
}

void sc_poweroff(void)
{
    gpio_set_level(SC_PIN_VCC, 0);
}

bool sc_ispresent(void)
{
    return gpio_get_level(SC_PIN_PRES);
}
```

2.2 Génération du signal d'horloge

Jusqu'ici, et en dehors des noms des fonctions utilisées, RP2040 et ESP32 sont très similaires. Il n'en va pas de même pour la génération d'un signal directement sur une sortie du microcontrôleur. L'objectif ici est d'obtenir un signal propre (dans la mesure du possible), et ce, en reposant sur des mécanismes internes de la plateforme. Ceci signifie donc qu'il est absolument hors de question de faire cela de manière logicielle avec une boucle et des temporisations, ce qui serait non seulement peu avisé, mais surtout plus compliqué.

2.2.1 Version RP2040

Étonnamment, générer un signal d'horloge sur une sortie du RP2040 est excessivement aisé puisqu'un simple appel à une fonction suffit. Pour ce faire, on inclut simplement `hardware/clocks.h` et on utilise `clock_gpio_init()` avec, en argument, le port GPIO à utiliser, qui ne peut être **que** 21, 23, 24 ou 25, l'horloge source et un `float` servant de diviseur pour obtenir la fréquence souhaitée. Notez qu'il est également possible d'utiliser `clock_gpio_init_int_frac()` qui sépare le diviseur en deux entiers (`uint32_t` et `uint8_t`) avec la partie entière d'une part et la partie fractionnaire de l'autre.

L'élément important ici est l'horloge source qui peut être n'importe quel générateur d'horloge intégré au microcontrôleur, dont les plus utiles sont :

- `CLOCKS_FC0_SRC_VALUE_CLK_SYS` : l'horloge système ;
- `CLOCKS_FC0_SRC_VALUE_CLK_PERI` : horloge des périphériques, par défaut identique à celle système ;
- `CLOCKS_FC0_SRC_VALUE_CLK_USB)` : horloge de référence pour l'USB (48 MHz) ;
- `CLOCKS_FC0_SRC_VALUE_CLK_ADC)` : horloge du convertisseur analogique/numérique (48 MHz).

Pour notre projet, nous n'avons donc qu'à créer quelques fonctions très simples pour produire le signal souhaité (ici, à partir des 48 MHz pour l'ADC) ou stopper sa génération :

```
void clockonkhz(float freqkhz)
{
    uint32_t f_clk_adc = frequency_count_khz(CLOCKS_FC0_SRC_VALUE_CLK_ADC);
    clock_gpio_init(SC_PIN_CLK,
       CLOCKS_CLK_GPOUT2_CTRL_AUXSRC_VALUE_CLK_ADC,
       f_clk_adc/freqkhz);
}

void sc_clock_on(void)
{
    clockonkhz(SC_CLK_FREQ/1000.0);
}

void sc_clock_off(void)
{
    gpio_init(SC_PIN_CLK);
    gpio_set_input_enabled(SC_PIN_CLK, true);
}
```

Nous pourrions, dans le cas de l'horloge ADC, nous contenter d'un diviseur fixe puisque la fréquence source est forcément 48 MHz, mais autant faire générique et avoir `clockonkhz()` d'une part et `sc_clock_on()` de l'autre. Notez qu'arrêter l'émission du signal revient simplement à changer la fonction du port pour déconnecter le générateur de la sortie.

2.2.2 Version ESP32

Avec l'ESP32, les choses ne sont pas si simples, mais dans le même temps permettent également davantage de souplesse. Cette souplesse n'est cependant pas ici très intéressante, car la technique généralement utilisée pour générer un signal sur cette plateforme consiste à utiliser le périphérique LEDC, pour *LED Control*. Initialement présent pour permettre de générer un signal modulé en largeur d'impulsion, ou PWM, il est, comme son nom l'indique, prévu pour piloter l'intensité de LED. 4 MHz est une fréquence de base un peu surprenante pour ce type d'usage, mais LEDC est plus polyvalent qu'il n'y parait.

Le périphérique LEDC peut être configuré à partir de 4 *timers* différents, avec une résolution dépendant du modèle d'ESP32 (entre 1 et 20 bits, 1 et 14 pour l'ESP32-C3) et utiliser jusqu'à 8 canaux en sortie (mais 6 pour l'ESP32). Pour savoir quel microcontrôleur Espressif offre quelle fonctionnalité, regardez simplement le contenu du fichier `soc_caps.h` dans le sous-répertoire `components/soc/esp32*/include/soc/` correspondant au modèle qui vous intéresse.

Ici, nous n'avons besoin que d'un seul canal et il ne s'agit pas réellement de PWM puisque le signal a un rapport cyclique invariable de 50 %. Fort heureusement, le contrôleur LEDC de l'ESP32 est capable de fournir une fréquence importante, à condition que la résolution utilisée pour le rapport cyclique soit très faible. Ainsi, avec une résolution de 1 bit, nous pouvons

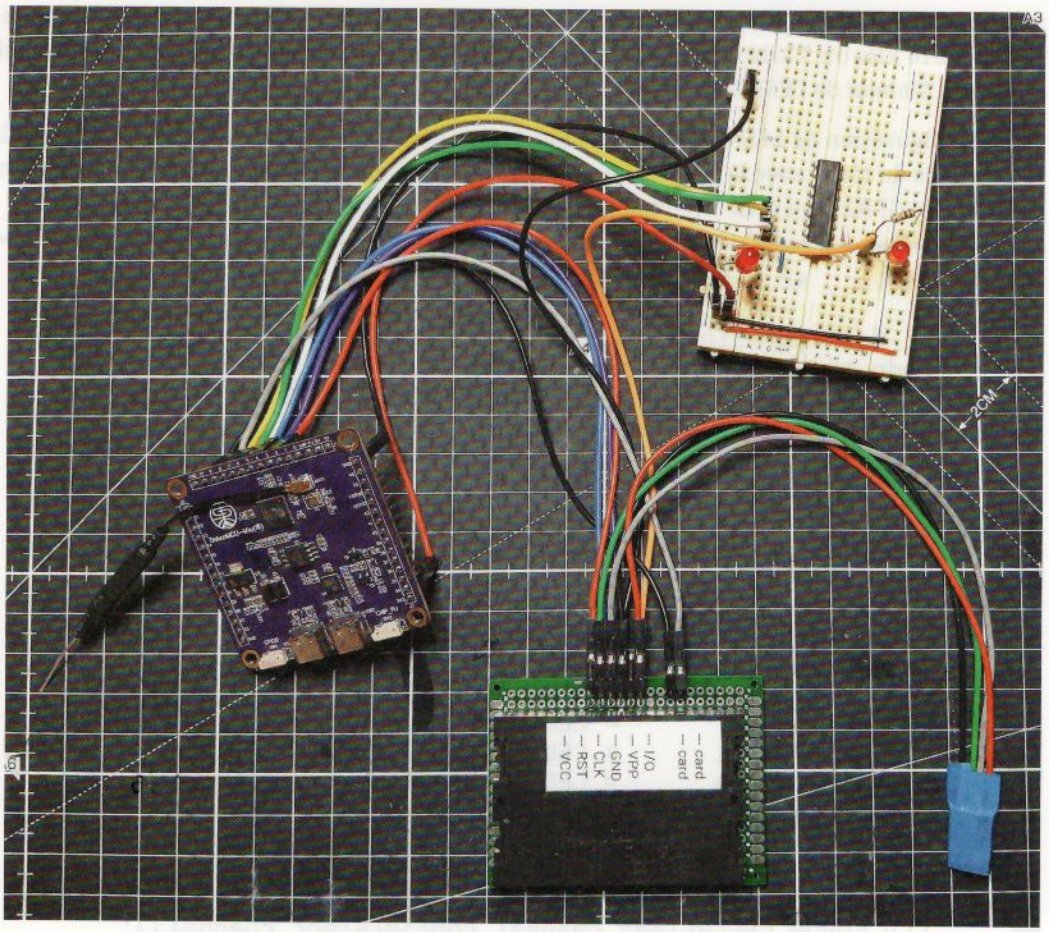
atteindre une fréquence de 40 MHz, alors qu'en utilisant 8 ou 13 bits nous ne pourrions arriver qu'à 310 kHz ou 9 kHz (si vous êtes curieux, la formule est donnée page 386 du *Technical Reference Manual* de l'ESP32 **[13]**).

La configuration de LEDC se passe en deux temps. Nous avons premièrement le réglage des paramètres du *timer* avec la résolution et la fréquence, et ensuite celui du canal, qui est associé à une sortie GPIO. Ceci peut être résumé dans une fonction dédiée :

```
static void sc_clock_init(void)
{
    // Configuration du timer LEDC PWM
    ledc_timer_config_t ledc_timer = {
        .speed_mode       = LEDC_LOW_SPEED_MODE,
        .duty_resolution  = LEDC_TIMER_1_BIT,
        .timer_num        = LEDC_TIMER_0,
        .freq_hz          = SC_CLK_FREQ,
        .clk_cfg          = LEDC_AUTO_CLK,
    };
    ESP_ERROR_CHECK(ledc_timer_config(&ledc_timer));

    // Configuration du canal LEDC PWM
    ledc_channel_config_t ledc_channel = {
        .speed_mode     = LEDC_LOW_SPEED_MODE,
        .channel        = LEDC_CHANNEL_0,
        .timer_sel      = LEDC_TIMER_0,
        .intr_type      = LEDC_INTR_DISABLE,
        .gpio_num       = SC_PIN_CLK,
        .duty           = 0, // rapport cyclique initial
        .hpoint         = 0,
    };
    ESP_ERROR_CHECK(ledc_channel_config(&ledc_channel));
}
```

Comme vous pouvez le constater, nous démarrons la génération avec un rapport cyclique à 0, ce qui signifie que la broche sera à la masse en permanence et qu'aucun signal d'horloge n'atteindra la carte à puce. Ceci est délibéré et nous permet d'ajouter deux fonctions permettant respectivement d'activer le signal et de l'arrêter :

```
void sc_clock_on(void)
{
    ESP_ERROR_CHECK(ledc_set_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0, 1));
    ESP_ERROR_CHECK(ledc_update_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0));
}

void sc_clock_off(void)
{
    ESP_ERROR_CHECK(ledc_set_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0, 0));
    ESP_ERROR_CHECK(ledc_update_duty(LEDC_LOW_SPEED_MODE, LEDC_CHANNEL_0));
}
```



Voici l'implémentation ESP32 de notre petite expérience. La plateforme utilisée repose sur un ESP32-S3 (DshanMCU-Mio), mais n'importe quel microcontrôleur de la famille ESP32 fera parfaitement l'affaire.

Nous utilisons un peu partout la macro `ESP_ERROR_CHECK` afin de ne pas avoir à gérer nous-mêmes la valeur retournée par les fonctions `ledc_*` et donc l'éventuelle présence d'une erreur. Ceci est l'équivalent d'un `assert()` tout en présentant l'avantage, si on utilise le moniteur ESP-IDF (`idf.py -p /dev/ttyXXX monitor`), d'apporter davantage d'informations concernant l'éventuelle erreur. Notez que, dans ce cas, la macro utilise `abort()`, qui affiche un *backtrace* et *reboote* le microcontrôleur.

2.3 Paramétrer l'UART

À ce stade, il ne nous reste plus qu'une chose à faire pour recevoir notre ATR, c'est d'ouvrir bien grand nos oreilles. En d'autres termes, il faut configurer l'UART pour être à l'écoute des caractères envoyés par l'UICC juste après son *reset*, et ceci signifie donc de configurer le périphérique avec le bon débit.

2.3.1 Version RP2040

La configuration du second UART de la carte Pico n'est pas très difficile puisque comme pour les autres réglages, ceci se limite à l'appel d'une simple fonction pour initialiser le périphérique et de quelques autres pour régler les broches utilisées et les paramètres de communication. Nous résumons, encore une fois, tout cela dans une fonction dédiée :

```
void sc_uart_init(unsigned int baudrate)
{
    uart_init(SC_UART, baudrate);
    gpio_set_function(SC_PIN_TX, GPIO_FUNC_UART);
    gpio_set_function(SC_PIN_RX, GPIO_FUNC_UART);
    uart_set_format(SC_UART, 8, 2, UART_PARITY_EVEN);
    gpio_pull_up(SC_PIN_RX);
}
```

Notez la présence de l'appel à `gpio_pull_up()` configurant une résistance de rappel à VCC sur la broche RX. Après maints essais avec un comportement très étrange, il s'est avéré que ceci n'est pas configuré par défaut et que, comme la broche 18 du 74HC241 passe en haute impédance lors de l'activation de l'envoi de données, des fluctuations et des tensions parasites se forment. Ainsi, sans cette résistance, le *buffer* de réception se remplit (généralement avec un ou plusieurs `0x00`) pendant l'envoi et ces octets s'ajoutent à ceux réceptionnés juste après. Le problème n'existe pas avec l'ESP32.

Avec cette fonction et la macro de calcul citée précédemment, configurer l'UART avec un débit dépendant de la fréquence appliquée à CLK se résume à :

```
sc_uart_init(SC_BAUDRATE(SC_CLK_FREQ,
  SC_ETU_PERIOD_US(SC_CLK_FREQ)
));
```

L'envoi et la réception de données sont également implémentés sous la forme de fonctions dédiées, plutôt que d'utiliser directement celles proposées par l'environnement/SDK. Ceci dans le cas du RP2040 était une obligation pour une raison que nous allons voir dans un instant, mais plus important encore, cela permet de créer un niveau d'abstraction vis-à-vis de la plateforme choisie. L'idée est de réunir l'ensemble des fonctions propres à l'environnement dans `smartcard.c`/`smartcard.h` et, par ailleurs, de n'avoir que des fonctions génériques, sans lien direct avec un microcontrôleur donné. À terme, car ce n'est pas terminé à ce jour, le but est de permettre l'utilisation du code avec n'importe quel type de microcontrôleur, en ayant besoin uniquement d'adapter `smartcard.c`/`smartcard.h` pour supporter un ATmega328, un MSP430 ou encore un STM32 à l'avenir.

Mais revenons à la raison initiale, qui est une limitation que je trouve très regrettable dans les fonctions UART fournies par le SDK Pico : il n'y a aucune gestion de *timeout* par défaut. À titre de comparaison, la lecture de l'UART côté ESP-IDF utilise une fonction avec ce prototype :

```
int uart_read_bytes(
  uart_port_t uart_num,
  void *buf,
  uint32_t length,
  TickType_t ticks_to_wait
)
```

On trouve en argument, respectivement, le port à utiliser, le *buffer* où stocker les données reçues, la taille maximum de ces données et un nombre de *ticks* système après lesquels stopper la lecture et retourner le nombre d'octets lus. Côté Pico en revanche, nous avons :

```
static void uart_read_blocking (
  uart_inst_t *uart,
  uint8_t *dst,
  size_t len
)
```

Ce qui dans le SDK (pico-sdk/src/rp2_common/hardware_uart/include/hardware/uart.h) est implémenté ainsi :

```
static inline void uart_write_blocking(uart_inst_t *uart,
const uint8_t *src, size_t len) {
    for (size_t i = 0; i < len; ++i) {
        while (!uart_is_writable(uart))
            tight_loop_contents();
        uart_get_hw(uart)->dr = *src++;
    }
}
```

C'est une simple boucle de lecture récupérant le contenu d'un registre jusqu'à atteindre la quantité spécifiée en argument de la fonction. Cela implique un potentiel blocage sans fin dès lors que l'on ne connaît pas précisément la quantité d'octets qu'on est censé recevoir, ce qui est précisément le cas des réponses de l'UICC.

D'autre part, nous avons également à notre disposition la fonction :

```
bool uart_is_readable_within_us (
  uart_inst_t *uart,
  uint32_t us
)
```

Celle-ci nous permet d'attendre un certain nombre de **microsecondes** (pas millisecondes) que le *buffer* de réception de l'UART soit non vide, et retournera immédiatement `true` si des données sont présentes, ou `false` si le *buffer* est vide après le délai imparti. Nous pouvons combiner ces deux fonctions/implémentations pour créer quelque chose d'équivalent à ce que l'on trouve sur ESP32 :

```
int sc_read_bytes(uart_inst_t *uart, uint8_t *dst,
size_t len, uint16_t timeoutms)
{
    int cpt = 0;

    // Attente du premier octet
    if (uart_is_readable_within_us(uart, timeoutms * 1000) == 0)
        return 0;

    while (uart_is_readable_within_us(uart, 5000) && cpt < len) {
```

```
        *dst++ = (uint8_t) uart_get_hw(uart)->dr;
        cpt++;
    }
    return cpt;
}
```

L'idée ici, c'est d'appliquer le *timeout* sur chaque octet attendu. Lorsque l'UICC répond, qu'il s'agisse de l'ATR suite au *reset* ou d'une réponse à une commande envoyée, il n'y a pas de pause dans l'émission. Il n'y a que trois options possibles : soit l'UICC ne répond pas, soit nous recevons une réponse d'une taille que nous n'attendons pas (généralement, un code erreur), soit la réponse est de la taille attendue. Le délai peut donc uniquement s'appliquer au premier octet et c'est précisément ce que nous faisons là, afin d'éviter d'attendre inutilement `timeoutms` millisecondes une fois le message reçu. Notez la post-incrémentation de `dst` lors de la copie du contenu du registre, cette syntaxe est directement reprise des sources du SDK.

Et puisque nous sommes dans le traitement des transactions via l'UART, nous en profitons également pour créer la fonction d'écriture :

```
void sc_write_bytes(uart_inst_t *uart,
const uint8_t *src, size_t len)
{
    uart_write_blocking(uart, (const uint8_t *)src, len);
    uart_tx_wait_blocking(uart);
}
```

Pas de gestion de *timeout* ici, mais une écriture bloquante avec l'ajout de `uart_tx_wait_blocking()`. Ceci est important dans le sens où nous, après envoi d'un message, nous repasserons immédiatement en lecture pour obtenir la réponse de l'UICC, mais l'exécution du code se déroule bien plus vite que l'envoi de données. Nous sommes donc obligés d'attendre que le *buffer* FIFO de l'UART soit effectivement vide pour poursuivre, sinon nous bloquons le 74HC241 en plein milieu de l'envoi.

2.3.2 Version ESP32

On commence maintenant à voir clairement qu'il existe d'importantes différences de « philosophie » entre les SDK RP2040 et Espressif et, une fois encore, la version ESP32 de la configuration de l'UART nécessitera davantage de code. Ici, ce n'est pas seulement le *framework* qui est en cause, mais également le fait que les *firmwares* pour ESP32 reposent sur le système temps-réel FreeRTOS, alors qu'avec le RP2040, nous sommes en *bare metal*.

Configurer et utiliser l'UART1 sur ESP32 passe donc par l'installation d'un pilote fonctionnant, dans les grandes lignes, comme une tâche indépendante du programme principal. Ceci signifie que pour obtenir ou envoyer des données, une communication interprocessus (ou intertâches) doit être mise en place. Avec FreeRTOS, ceci passe par l'utilisation de *queues* **[14]** qui peuvent être vues comme des *buffers* FIFO *thread safe*. Configurer l'UART implique donc de

régler les paramètres de communication (vitesse, format de données, parité, etc.), choisir les ports GPIO à utiliser (RX, TX et parfois RTS, CTS pour le contrôle de flux), mais aussi spécifier une *queue* pour établir la liaison entre notre code et le pilote.

Tout ceci est résumé, une fois n'est pas coutume, par une fonction d'initialisation dédiée :

```
static void sc_uart_init(unsigned int baudrate)
{
    // Queue FreeRTOS
    const int uart_buffer_size = (1024 * 2);
    QueueHandle_t uart_queue;

    uart_config_t uart_config = {
        .baud_rate = baudrate,
        .data_bits = UART_DATA_8_BITS,
        .parity = UART_PARITY_EVEN,
       //.stop_bits = UART_STOP_BITS_1_5,
        .stop_bits = UART_STOP_BITS_2,
        .flow_ctrl = UART_HW_FLOWCTRL_DISABLE,
        .source_clk = UART_SCLK_DEFAULT,
    };
    // Configuration des paramètres de l'UART
    ESP_ERROR_CHECK(uart_param_config(SC_UART, &uart_config));
    // Configuration des broches (TX, RX, RTS, CTS)
    ESP_ERROR_CHECK(uart_set_pin(
      SC_UART, SC_PIN_TX, SC_PIN_RX,
      UART_PIN_NO_CHANGE, UART_PIN_NO_CHANGE));
    // Installation du pilote et de la queue
    ESP_ERROR_CHECK(uart_driver_install(
      SC_UART, uart_buffer_size, uart_buffer_size,
      10, &uart_queue, 0));
}
```

Rien de bien particulier ici, on crée la *queue*, on remplit la structure permettant de configurer l'UART, on applique la configuration, on choisi, les broches utilisées, et enfin, on installe le pilote. Notez l'appel à `uart_set_pin()` qui utilise la macro `UART_PIN_NO_CHANGE` pour les signaux non utilisés (RTS et CTS) et qui permet de ne pas impacter de GPIO du tout.

L'utilisation de cette fonction dans `main()` est strictement identique à celle de l'implémentation RP2040 (c'est d'ailleurs le but) :

```
sc_uart_init(SC_BAUDRATE(SC_CLK_FREQ,
  SC_ETU_PERIOD_US(SC_CLK_FREQ)
));
```

De la même manière, et bien que ceci ne soit pas nécessaire dans l'absolu, nous créons également les fonctions de lecture et écriture sur l'UART avec le même nom et les mêmes arguments :

```
int sc_read_bytes(uart_port_t uartnum, uint8_t *data, size_t len, uint16_t timeoutms)
{
    int rlen;
    rlen = uart_read_bytes(uartnum, data, len, timeoutms / portTICK_PERIOD_MS);
    return rlen;
}

void sc_write_bytes(uart_port_t uartnum, const char *src, size_t len)
{
    uart_write_bytes(uartnum, src, len);
    ESP_ERROR_CHECK(uart_wait_tx_done(uartnum, 100));
}
```

2.4 Activation et réception de l'ATR

Atteindre l'objectif consistant à obtenir l'ATR est à portée de main, tout ce qu'il nous reste à faire est d'activer la carte et lire ce qui nous parvient. Le terme « activation » ne cache rien de bien mystérieux et se résume, selon la norme, en une succession d'étapes :

- mettre RST à la masse ;
- mettre VCC à la tension d'alimentation ;
- placer l'UART en réception ;
- fournir le signal d'horloge sur CLK ;
- laisser RST à la masse pendant au moins 400 cycles d'horloge ;
- repasser RST à l'état haut.

À 4 MHz, 400 cycles d'horloge représentent 0,1 milliseconde et nous arrondissons cela très généreusement à 10 ms. Encore une fois, nous plaçons tout cela dans une fonction, à commencer par la version RP2040 :

```
void sc_activate(void)
{
    gpio_put(SC_PIN_RST, 0);
    sleep_ms(1);
//  gpio_put(SC_PIN_VCC, 1);
//  sleep_ms(1);
    sc_clock_on();
    sleep_ms(10);

    // flush (ACOSJ)
    irq_set_enabled(UART1_IRQ, false);
    uart_set_fifo_enabled(SC_UART, false);
    sleep_ms(1);
    uart_set_fifo_enabled(SC_UART, true);
    irq_set_enabled(UART1_IRQ, true);

    sleep_ms(10);
    gpio_put(SC_PIN_RST, 1);
}
```

N'ayant toujours pas réceptionné mes MOSFET depuis le début de l'article, la partie concernant le contrôle de `SC_PIN_VCC` est pour le moment commentée, mais les lignes les plus intéressantes concernent le fait de vider le *buffer* FIFO de l'UART juste avant de relâcher le *reset*. Ceci ne s'avère pas nécessaire avec la plupart des cartes à puce qui sont totalement silencieuses lorsque RST est à la masse. Ce n'est malheureusement pas le cas pour les Java Card ACOSJ qui semblent avoir une fâcheuse tendance à faire apparaître tantôt des signaux sur I/O. Il n'existe pas, avec le SDK Pico, de fonction permettant de vider le *buffer* et la solution consiste à provoquer cette action de manière détournée en désactivant le FIFO et en le réactivant. Il faut, pour cela, préalablement désactiver les interruptions avec `irq_set_enabled()`.

Le code équivalent du côté ESP32 ressemble à ceci :

```
void sc_activate(void)
{
    gpio_set_level(SC_PIN_RST, 0);
    vTaskDelay(1 / portTICK_PERIOD_MS);
    gpio_set_level(SC_PIN_VCC, 1);
    vTaskDelay(1 / portTICK_PERIOD_MS);
    sc_clock_on();
    vTaskDelay(10 / portTICK_PERIOD_MS);
    uart_flush(SC_UART); // flush (ACOSJ)
    vTaskDelay(10 / portTICK_PERIOD_MS);
    gpio_set_level(SC_PIN_RST, 1);
}
```

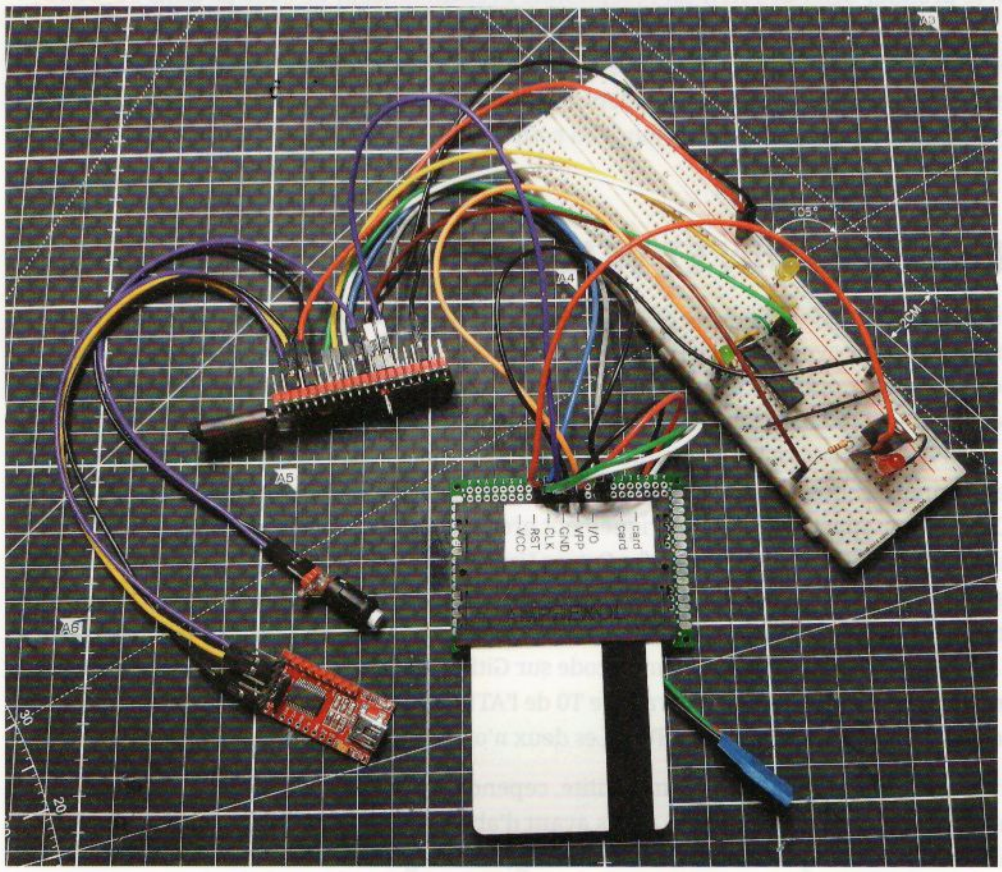
On retrouve la même logique, mais tout comme pour le *timeout* de lecture UART, l'ESP-IDF nous fournit une solution aisée et clé en main pour le *flush* du buffer FIFO de l'UART.

Le reste est commun aux deux plateformes et trouve place dans le `main()` (ou `app_main()` dans le cas de l'ESP32) :

```
stdio_init_all();
sc_gpio_init();

if (!sc_ispresent()) {
    printf("Error. No card present!\n");
    errorloop();
}

sc_uart_init(SC_BAUDRATE(SC_CLK_FREQ,
  SC_ETU_PERIOD_US(SC_CLK_FREQ)));

sc_activate();

rlen = sc_read_bytes(SC_UART, atr, 40, 200);

if (!rlen) {
    printf("no response\n");
```

```
    sc_poweroff();
    sc_clock_off();
    errorloop();
}

if (analyzeATR(atr, rlen, true) != 0) {
    sc_poweroff();
    sc_clock_off();
    errorloop();
}
```

Avant toute chose, je ferai l'impasse ici sur la fonction `analyzeATR()`, car cela dépasserait largement l'espace imparti à cet article et il n'y a pas vraiment de subtilité en lien avec le matériel dans le traitement des données. Si ce point précis attire votre attention plus que de nécessaire, Wikipédia propose une page **[15]** très détaillée (en anglais). Vous pourrez compléter cela avec les résultats affichés par le *parseur* ATR en ligne de Ludovic Rousseau **[16]** ou encore les sources en Perl de `ATR_analysis`, inclus dans les `pcsc-tools` (du même Ludovic Rousseau) et appelé par `pcsc_scan`. Enfin, bien entendu, il y a le résultat de mes propres expérimentations disponibles sur GitLab, en version Pico **[17]** et en version ESP32 **[18]**.



La version Raspberry Pi Pico du montage est presque en tout point similaire à celle utilisant un ESP32. La seule différence notable est l'incapacité du RP2040 à fournir suffisamment de courant sur ses GPIO pour alimenter la smartcard. Ceci explique la présence d'un MOSFET-P sur la platine à essai, qui bien sûr est arrivé, avec ses cousins, juste après que j'ai fini de rédiger l'article...

En attendant, une simple boucle pour afficher l'ATR en hexadécimal fera parfaitement l'affaire :

```
printf("ATR: ");
for (i = 0; i < atrlen; i++) {
     printf("%02x ", atr[i]);
}
printf(" (%d)\n", atrlen);
```

Une fois le code compilé et exécuté, et avec la présence d'une carte confirmée, vous devriez voir apparaître quelque chose comme ceci :

```
clock: 4000000Hz   baudrate: 10752b/s   ETU: 93µs
Init SC UART @10752bps
smartcard access...
ATR: 3b f8 13 00 00 81 31 fe 45 4a 43 4f 50 76 32 34 31 b7  (18)
```

En copiant/collant cet ATR sur le site de Ludovic Rousseau **[16]**, vous devriez voir le décodage complet de chaque champ/valeur. Ceci est l'ATR d'une Java Card J2A081 et des tests ont également été faits sans problèmes avec une J3R150, J3H145 et une ACOSJ. Les seuls problèmes rencontrés pour l'instant apparaissent sur la Pico avec une carte USIM Orange ainsi qu'une CB Visa. Seul le premier octet de l'ATR est réceptionné avec un *timeout* de 5000 µs dans `sc_read_bytes()`. Il a été nécessaire d'augmenter ce délai à 20000 µs (soit 20 ms) pour que la communication fonctionne avec la USIM et même 50 ms pour la CB. Le problème n'est pas apparu avec l'ESP32 et n'a pas encore été clairement identifié, mais seule une partie du protocole est actuellement gérée, c'est une expérience toujours en cours...

3. ALLER PLUS LOIN ET VRAIMENT COMMUNIQUER

Obtenir un ATR est un bon point de départ, mais l'objectif est de réellement échanger des messages avec l'UICC. Nous avons les fonctions de bas niveau pour envoyer et recevoir des octets en *half-duplex* et nous devons nous pencher maintenant sur la couche supérieure, concernant les protocoles de transmission. Et oui, c'est un pluriel puisqu'il en existe deux, appelés T=0 et T=1. Cette dénomination fait référence à l'une des informations véhiculées dans l'ATR qui précise, dans un caractère nommé TD, sous la forme des quatre bits de poids faible, le protocole de transmission à utiliser. Cette valeur appelée T peut théoriquement prendre une valeur entre 0 et 15, mais seuls 0 et 1 sont un sens (le reste est RFU, *Reserved for Future Use*). T peut donc être 0 ou 1, ce qui correspond aux désignations T=0 et T=1, voir tantôt T0 et T1, qu'on trouve dans les documentations. Encore une fois, pour une description complète des caractères de l'ATR, référez-vous à la page Wikipédia **[15]**, à l'outil de Ludovic Rousseau **[16]** et/ou à mon code sur GitLab (fonction `analyzeATR()`). Notez qu'il est important de ne pas confondre le caractère T0 de l'ATR (toujours présent en seconde position) et le protocole T=0, surtout écrit comme « T0 ». Les deux n'ont aucun rapport.

Les protocoles T=0 et T=1 ont la même utilité, cependant le premier est « orienté caractère » alors que le second est « orienté bloc ». Mais avant d'aborder cela en détail, il est important de définir clairement en quoi consistent les messages échangés entre le terminal et l'UICC.

Ces messages sont appelés APDU, pour *Application Protocol Data Unit*, et sont structurés d'une manière standard (ISO/IEC 7816 partie 4). Ils débutent toujours par un octet de classe (CLA), suivi d'un octet d'instruction (INS), deux octets de paramètre (P1 et P2) puis soit :

- 0, 1 ou 3 octets définissant la taille des données envoyées (L_c), suivi de ces dernières ;
- 0, 1, 2 ou 3 octets précisant la taille des données attendues dans la réponse (L_e).

La réponse à ce type d'APDU (parfois appelé C-APDU pour *Command APDU*) est également un APDU (R-APDU pour *Response APDU*), structuré avec les données de réponse (si présentes), suivies d'un code d'état sur deux octets (SW1 et SW2). Un grand nombre de ces codes sont décrits dans le standard et vous pourrez en consulter une liste sur différents sites **[19] [20] [21]**. Mais, dans les grandes lignes, ce qu'on espère généralement c'est `9000` (ou `61xx`).

Lorsqu'on parle de protocoles de transmission, on désigne la façon d'envoyer et recevoir ces APDU, et c'est là que T0 et T1 entrent en jeu.

3.1 Protocol T=1

Commençons par le protocole qui est à mon sens le plus simple. Il s'agit d'un protocole en mode bloc, c'est-à-dire que l'APDU, qu'il s'agisse d'une commande ou d'une réponse, est empaqueté (ou encapsulé) dans une trame, à la manière d'un paquet réseau. Dans cette trame, l'APDU en entier est une simple donnée et sont ajoutés 3 octets avant et un après. Une trame, ou TPDU (*Transport Protocol Data Unit*) ressemble à ceci :

```
|           prologue            |   information   | épilogue|
|   NAD   |   PCB   |   LEN     |       INF       |   EDC   |
| 1 octet | 1 octet | 1 octet   |   LEN octets    | 1 octet |
```

Avec :

- `NAD` : *Node ADdress byte*, se compose de 3 bits d'adresse de destination (DAD), 3 bits d'adresse source (SAD) et deux bits inutilisés. La seule valeur possible pour SAD et DAD est 0, toutes les autres valeurs sont RFU. NAD est donc toujours `0x00` ;
- `PCB` : *Protocol Control Byte*. Là, les choses sont un peu plus compliquées, cf. plus loin ;
- `LEN` : la taille des données ;
- `INF` : la donnée, à savoir l'APDU ;
- `EDC` : *Error Detection Code*, une somme de contrôle (LRC pour *Longitudinal Redundancy Check*) qui est un simple XOR (OU exclusif) de l'ensemble du TPDU, hormis l'EDC lui-même bien sûr.

Écartons de suite l'EDC, dont le calcul peut être résumé en une petite fonction :

```
uint8_t computelrc(uint8_t *data, size_t datalen)
{
    uint8_t lrc = 0;
    int i;
    if (!data || !datalen)
```

```
        return 0x00;
    for (i = 0; i < datalen; i++) {
        lrc = lrc ^ data[i];
    }
    return lrc;
}
```

Penchons-nous maintenant sur le PCB. Ce simple octet est différent en fonction de la nature de la transaction. Cela peut être un *I-block* s'il s'agit d'informations (d'échange de données, C-APDU et R-APDU), un *R-block* pour les accusés de réception ou un *S-block* pour les informations de contrôle et de supervision. Différencier les types de blocs n'est pas un problème :

- un *I-block* a le bit de poids le plus fort à `0` ;
- un *R-block* a les deux bits de poids les plus forts à `10` ;
- un *S-block* a les deux bits de poids les plus forts à `11`.

Nous ne nous intéresserons ici qu'aux *I-blocks* et donc aux échanges d'APDU. Dans ce cas, l'octet PCB est structuré comme suit :

- bits 0 à 4 : RFU ;
- bit 5 : c'est le bit de chaînage permettant d'envoyer les données en plusieurs fois, alias bit « M ». Celui-ci est à `1` si d'autres morceaux de données suivent le TPDU courant et `0` si ce TPDU est le dernier. Ceci n'est, pour l'heure, pas implémenté dans mon code, mais le sera peut-être quand vous lirez l'article ;
- bit 6 : le bit de « séquence » (« N(S) » dans la documentation) qui alterne simplement entre `0` et `1` au fil des envois de TPDU ;
- bit 7 : toujours à `0`.

Avec le protocole T1, pour envoyer notre APDU, nous devons donc dans une fonction `transceive_apdu_T1()` faire :

```
static uint8_t blkcount = 0;
uint8_t tpdu[255] = { 0 };
[...]
// Copier l'APDU à 3 octets du début du TPDU
memcpy(tpdu + 3, apdu, apdulen);
// Définir le NAD (toujours 0)
tpdu[0] = 0x00;
// Préciser la taille des données
tpdu[2] = apdulen;
// Basculer le bit de séquence
tpdu[1] = blkcount % 2 ? 0x40 : 0x00;
// Ajouter l'EDC en dernière position
tpdu[apdulen + 3] = computelrc(tpdu, apdulen+3);
```

On utilisera ensuite notre fonction d'écriture sur le port série avec :

```
sc_enablewrite();
sleep_ms(20);
sc_write_bytes(uart_num, (const uint8_t *)tpdu, apdulen + 4);
sc_disablewrite();

blkcount++;
```

Et on enchaînera immédiatement sur une lecture comme nous l'avons fait pour l'ATR. Une fois la réponse obtenue, nous procédons à quelques vérifications et si tout est correct, on stocke la réponse dans le tableau passé en argument de notre fonction (pointeur), ainsi que la taille des données reçues (R-APDU) :

```
  if (rlen < 4) {
    printf("Error. R-TPDU too short\n");
    return -1;
  }

  if (rlen == 4) {
    printf("Error. No R-APDU in R-TPDU\n");
    return -1;
  }

  if (rtpdu[rlen - 1] != computelrc(rtpdu, rlen - 1)) {
    printf("Error. Bad LRC in R-TPDU!\n");
    return -1;
  }

  memcpy(rapdu, rtpdu + 3, rlen - 4);
  *rapdulen = (size_t)rlen - 4;

  return 0;
}
```

C'est aussi simple que cela. Bien entendu, ceci est l'implémentation la plus naïve qui puisse exister et elle fait l'impasse sur bon nombre de cas particuliers. Le code actuel prend en réalité également en charge un *S-block* particulier qu'il s'avère nécessaire de gérer, au minimum, avec mon applet Java Card TOTP. Dans ce cas précis, lors de l'envoi d'un APDU demandant un certain nombre d'opérations de la part de l'UICC, celle-ci répond avec un *S-block* pour confirmer que le traitement est en cours et que la réponse arrivera bientôt (WTX pour *Waiting Time eXtension*). Ce à quoi il faut répondre par un autre *S-block*, confirmant la réception de cette information. Dans le cas contraire, le R-APDU n'arrive jamais parce que l'UICC pense qu'on est parti faire autre chose...

Si vous voulez en savoir plus sur T1 et les blocs, la documentation ETSI **[8]** détaille tout cela en pages 43 à 46. Notez que, pour une raison mystérieuse, ce document numérote les bits à partir de 1, ce qui est furieusement perturbant (des amateurs de Lua, peut-être).

3.2 Protocol T=0

Ce protocole est dit « orienté caractère », mais ceci n'a pas réellement de sens pour en décrire le fonctionnement. Certes, il n'y a pas de trame ou TPDU comme avec T1, mais il serait plus exact de décrire T0 comme un protocole orienté question/réponse *half-duplex*. Je m'explique...

Avec T0, vous envoyez :

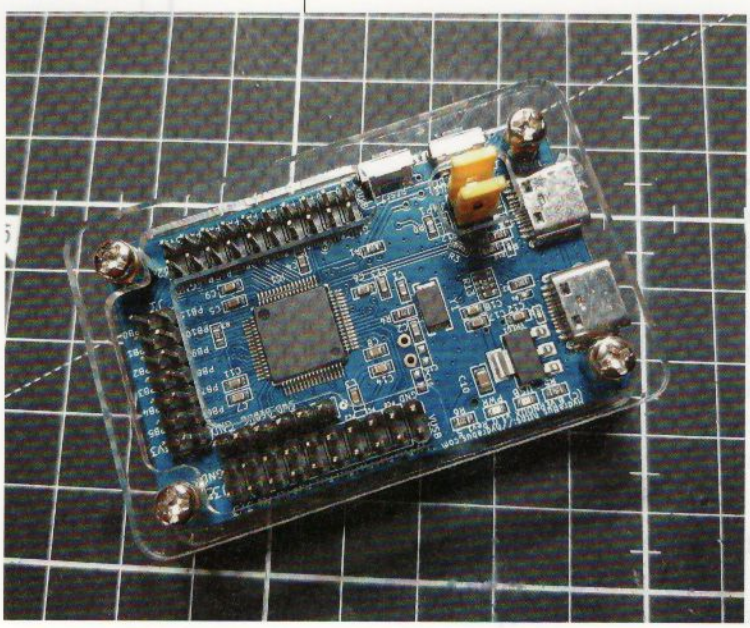
- soit CLA+INS+P1+P2+P3, puis les données après avoir reçu un octet de procédure en réponse, et aurez finalement SW1+SW2.

```
Terminal   <-->   UICC :
CLA+INS+P1+P1+P3 -->
<-- octet de procédure
Données -->
<-- SW1+SW2
```

- soit CLA+INS+P1+P2+P3 et vous recevez un octet de procédure en réponse, puis les données et SW1+SW2.

```
Terminal   <-->   UICC :
CLA+INS+P1+P2+P3 -->
<-- octet de procédure
<-- Données
<-- SW1+SW2
```

L'HydraBUS est un outil permettant de s'interfacer facilement avec une quantité impressionnante de bus et d'interfaces, y compris les smartcards ISO/IEC 7816. Le STM32F405 qui forme la base de ce périphérique supporte nativement ce type de communications séries asynchrones half-duplex. Ce qui n'est pas le cas du RP2040 ou des ESP32.



Deux points doivent être clarifiés. Le premier concerne P3 qui semble sortir de nulle part. C'est en réalité soit L_c, soit L_e, en fonction de l'APDU utilisé :

- CLA+INS+P1+P2 devient CLA+INS+P1+P2+00 (P3 = 00) ;
- CLA+INS+P1+P2+Le devient CLA+INS+P1+P2+Le (P3 = L_e) ;
- CLA+INS+P1+P2+L_c+DATA devient CLA+INS+P1+P2+L_c (P3 = L_c) ;
- CLA+INS+P1+P2+L_c+DATA+Le devient CLA+INS+P1+P2+L_c (P3 = L_c).

On constate que L_c comme L_e ne peut faire ici qu'un octet, mais ce n'est pas le plus important. En effet, un second point doit être expliqué : comment cela fonctionne-t-il si mon APDU transporte

des données et en attend également en retour, puisque dans les deux situations évoquées précédemment, nous ne recevons des données que si nous n'en envoyons pas ? La réponse est simple, ça se passe en deux fois :

```
Terminal   <-->   UICC :
CLA+INS+P1+P1+P3 -->
<-- octet de procédure
Données -->
<-- SW1+SW2 = 61xx

00+c0+00+00+xx -->
<-- octet de procédure
<-- Données
<-- SW1+SW2
```

En T0, nous avons besoin d'une transaction de plus par rapport à T1. L'UICC recevra les données, mais répondra avec un code d'état (normalement) composé de `61` et du nombre d'octets de la réponse. Ceci est généralement décrit par l'expression « *use GET RESPONSE* », la carte nous demande de demander la réponse plutôt que de la fournir directement. Nous devons alors utiliser un nouveau C-APDU avec CLA=`00`, INS=`c0`, P1=`00`, P2=`00` et L_e égal à la taille précisée dans le code d'état, c'est l'APDU *GET RESPONSE*. On se retrouve alors dans la seconde situation, avec un APDU sans données qui provoque une réponse qui en transporte.

Côté code, on laissera la gestion de l'état `61xx` et l'utilisation de l'ADPU *GET RESPONSE* dans `main()` et ne traiterons dans `transceive_apdu_T0()` que la partie de plus bas niveau. Là, nous conditionnons le type de message à écrire en fonction de la taille de l'APDU reçu en argument. Si celle-ci est supérieure à 5, l'APDU comprend des données à transmettre en deux temps, sinon non. Ainsi, si `apdulen > 5` nous avons :

```
    sc_enablewrite();
    sleep_ms(20);
    sc_write_bytes(uart_num,
      (const uint8_t *)apdu, headersize);
    sc_disablewrite();

    rlen = sc_read_bytes(uart_num, &pbyte, 1, timeout);

    // Vérification procedure byte
    if (rlen == 1 && pbyte == apdu[1]) {
      // OK, envoi données
      sc_enablewrite();
      sleep_ms(20);
      sc_write_bytes(uart_num,
        (const uint8_t *)(apdu + headersize),
        apdulen - headersize);
      sc_disablewrite();

      rlen = sc_read_bytes(uart_num,
        tmprapdu, *rapdulen, timeout);
```

```
    if (!rlen) {
      printf("Error. No response from card\n");
      return -1;
    }

    memcpy(rapdu, tmprapdu + i, rlen - i);
    *rapdulen = (size_t)rlen - i;
    return 0;
  } else {
    printf("Error! %d\n", rlen);
    return -1;
  }
```

Rien ici de bien différent de ce que nous avons déjà vu pour T1, c'est simplement le processus qui change avec une seconde écriture sur le port série pour les données. Si nous n'avons pas de données à transmettre, mais qu'il s'agit d'un APDU qu'on pourrait qualifier de commande ou de requête, les choses sont bien plus simples pour l'envoi :

```
  sc_enablewrite();
  sleep_ms(20);
  sc_write_bytes(uart_num,
    (const uint8_t *)apdu, headersize);
  sc_disablewrite();
```

Là, il nous suffira ensuite de lire le port pour obtenir une réponse et la retourner à la fonction appelante (`main()`) :

```
  rlen = sc_read_bytes(uart_num,
    tmprapdu, *rapdulen + 1, timeout);

  if (!rlen) {
    printf("Error. No response from card\n");
    return -1;
  }

  if (rlen < 3) {  // procedure byte + SW1 + SW2
    printf("Error. Response too short\n");
  }

  // On ne passe pas le procedure byte
  memcpy(rapdu, tmprapdu + 1, rlen - 1);
  *rapdulen = (size_t)rlen - 1;

  return 0;
}
```

Les deux fonctions `transceive_apdu_T0()` et `transceive_apdu_T1()` sont implémentées dans leurs codes sources respectifs `t0.c` et `t1.c`. Pour en faciliter l'utilisation, et étant donné que les prototypes sont identiques, nous utilisons un pointeur de fonction :

```
int (* transceive_apdu)(uart_inst_t *,
  uint8_t *, size_t, uint8_t *,
  size_t *, uint16_t, bool
);
```

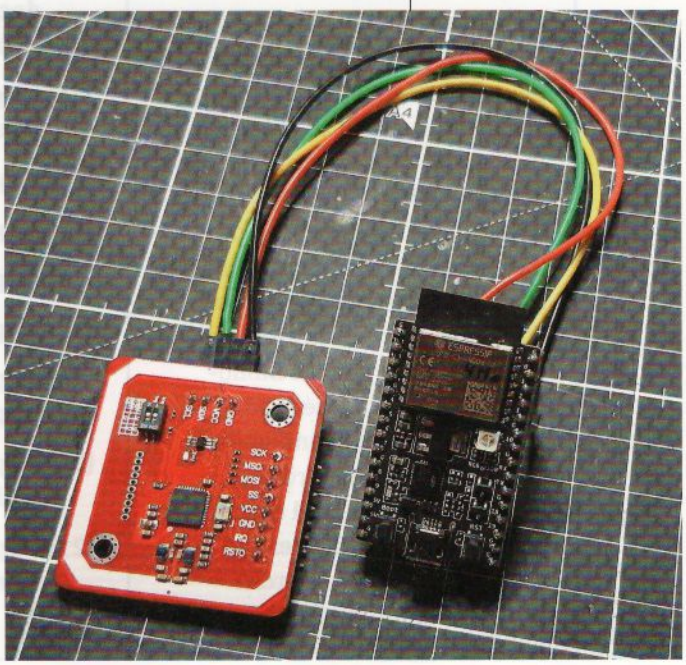
Ainsi, dans `main()`, nous pourrons utiliser directement `transceive_apdu()` sans nous soucier du protocole utilisé par l'UICC. Nous choisissons simplement la bonne fonction selon le *flags* `flag_t1_card` initialisé par l'analyseur d'ATR (`analyzeATR()`) :

```
if (flag_t1_card) {
  transceive_apdu = &transceive_apdu_T1;
} else {
  transceive_apdu = &transceive_apdu_T0;
}
```

Bien entendu, le code final est bien plus étoffé et implémente quelques points que nous n'avons pas abordés ici (comme l'IFS et la fonction `transceive_raw_T1()` permettant d'envoyer des données « brutes » sans transformation APDU/TPDU automatique).

CONCLUSION

Cet article est un « gros morceau », je vous l'accorde, et nous n'avons pas exploré un dixième du standard. Même dans mon implémentation expérimentale, bon nombre de points sont ignorés ou tout bonnement rejetés (comme la convention indirecte). Mon objectif n'est pas, cependant, de créer une bibliothèque complète de traitement et de communication implémentant la totalité du standard ISO/IEC 7816. Ce serait fort intéressant, mais j'ai déjà un travail à plein temps qui m'occupe largement, je pense. Il n'est même pas certain qu'une telle implémentation, purement logicielle, soit réellement une bonne idée, car la stabilisation du code et sa vérification coûteraient bien plus cher, pour un projet sérieux, que de simplement utiliser un composant fournissant tout cela clé en main, comme le SEC1210 de Microchip. L'objectif premier est de comprendre ces technologies et d'arriver à déléguer certaines opérations à une applet placée dans une Java Card dont nous maîtrisons le fonctionnement. En cela, je pense que le but est atteint et que les codes de démonstration, pour RP2040 et ESP32, mis à disposition dans



Certaines smartcards et Java Cards disposent d'une double interface, contacts + NFC/RFID. Dans ce cas, il est parfaitement possible d'oublier toutes les problématiques soulevées dans cet article et de simplement utiliser un module à base de NXP PN532, disposant de nombreuses bibliothèques pour presque toutes les plateformes et MCU.

mon GitLab peuvent constituer une base, certes perfectible, pour votre propre projet.

Nous avons également mis en parallèle les développements RP2040 et ESP32. Ceci nous a permis de constater que le SDK Pico, bien qu'à première vue plus simple d'utilisation que l'ESP-IDF, possède quelques limitations pénibles. L'ESP32, pour sa part, mise sur la souplesse d'utilisation, mais nécessite pour chaque fonctionnalité beaucoup plus de code, et donc de digestion de documentation. Fort heureusement, dans les deux cas, lesdites documentations sont riches en explications et parfaitement intelligibles. Au final, le choix de l'une ou l'autre plateforme sera sans doute plus une question de préférences personnelles qu'autre chose.

À noter tout de même qu'une approche plus simple peut également être envisagée, consistant à plutôt se concentrer sur les capacités NFC de certaines UICC. Là, inutile de tenter d'implémenter tout ou partie d'ISO/IEC 7816 puisqu'il suffit de reposer sur le support d'une puce comme la PN532 de NXP, disposant déjà de bibliothèques stables dédiées pour la plupart des environnements (Pico SDK, ESP-IDF, Arduino, etc.) et pour laquelle il existe des modules tout faits et peu coûteux. Nous en parlerons peut-être dans un prochain article... **DB**

RÉFÉRENCES

[1] https://github.com/killergeek/nard

[2] https://lab.flipper.net/apps/seader

[3] https://github.com/ANSSI-FR/cardstalker

[4] https://github.com/ANSSI-FR/Open-ISO7816-Stack

[5] https://wookey-project.github.io/

[6] https://github.com/hydrabus/hydrafw

[7] https://www.etsi.org/

[8] https://www.etsi.org/deliver/etsi_ts/102200_102299/102221/15.00.00_60/ts_102221v150000p.pdf

[9] https://arduinodiy.wordpress.com/2012/05/02/using-mosfets-with-ttl-levels/

[10] https://ww1.microchip.com/downloads/en/AppNotes/01370A.pdf

[11] https://github.com/LudovicRousseau/pcsc-tools

[12] https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf

[13] https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf

[14] https://www.freertos.org/Embedded-RTOS-Queues.html

[15] https://en.wikipedia.org/wiki/Answer_to_reset

[16] https://smartcard-atr.apdu.fr/

[17] https://gitlab.com/0xDRRB/picosmart

[18] https://gitlab.com/0xDRRB/esp32smart

[19] https://web.archive.org/web/20090623030155/http://cheef.ru/docs/HowTo/SW1SW2.info

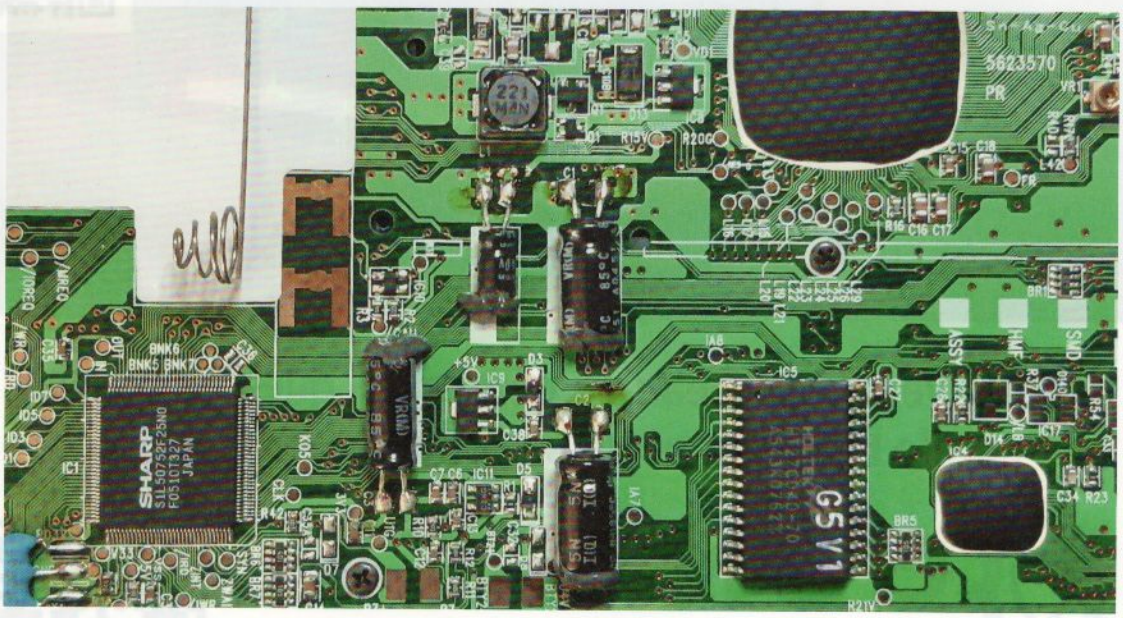
[20] https://www.smartjac.biz/support/main-menu?view=kb&kbartid=3

[21] https://www.eftlab.com/knowledge-base/complete-list-of-apdu-responses

SHARP PC-G850V : LE DERNIER DES ORDINATEURS DE POCHE

Denis Bodor

Les années 90 et 2000... Une période de l'histoire de l'informatique pleine d'énergie et de potentiel qui commençait à peine à se réaliser. C'est le début du Web, le septembre éternel [1], la diversité des machines et des architectures, la période de gloire d'USENET et de l'IRC, l'âge d'or des stations de travail inabordables pour le commun des mortels, et bien sûr, celui des calculatrices programmables, ou plus exactement des ordinateurs de poche. Le Sharp PC-G850V(S) est sans doute celui qui aura fermé la marche de cette belle époque et une machine qui vaut toujours le coût d'être possédée, utilisée et programmée...



– Sharp PC-G850V : le dernier des ordinateurs de poche –

Bien sûr, les calculatrices scientifiques programmables n'ont pas disparu, mais elles ne tiennent plus aujourd'hui que le rôle qui leur était originellement destiné. La génération dont fait partie le Sharp PC-G850V était « autre chose », née d'une période où le smartphone était un doux fantasme, les PC portables encombrants, lourds et peu agréables à utiliser, et où les calculatrices lorgnaient du côté de l'informatique ultraportable avec un concept dont le nom était aussi explicite qu'il puisse l'être : l'ordinateur de poche ou *pocket computer* en anglais. Des machines programmables, souvent en BASIC, capables de conserver les (pluriel) programmes utilisateur en mémoire, interfaçables avec des ordinateurs de bureau (à l'époque, il n'y avait pas que des PC x86), pouvant utiliser des supports externes et, du point de vue pédagogique, destiné à l'apprentissage réel de la programmation, en BASIC, en assembleur, tantôt en Lisp et, plus rarement encore, en C.

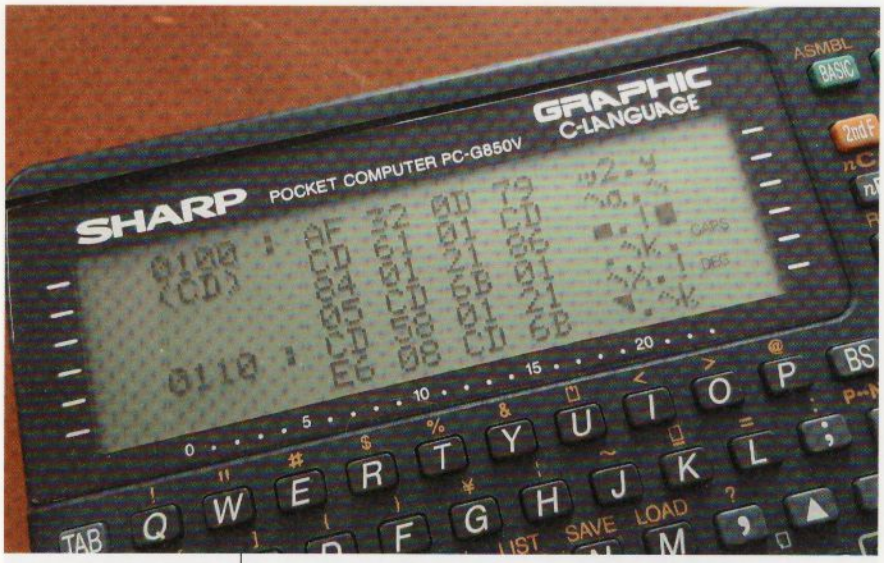
L'ordinateur de poche Sharp PC-G850V a été commercialisé en 2001 (2009 pour la version « VS »),

Le pocket computer Sharp PC-G850V est relativement imposant, il faut l'avouer, mais je suppose que les poches des utilisateurs étaient probablement plus grandes dans les années 90 et 2000 qu'aujourd'hui.

et cela exclusivement pour le marché japonais. Il représente, en quelque sorte, la génération la plus avancée de ce type de machines, et est l'héritier d'une longue lignée d'ordinateurs, pour certains bien plus connus mondialement comme le Sharp PC-1500, également vendu par Tandy (équivalent européen de RadioShack aux USA) sous la désignation TRS-80 PC-2 (alors qu'il n'avait pas grand-chose à voir avec un TRS-80). Sharp n'est, bien entendu, pas le seul fabricant de ce type de machines à cette période. Hewlett-Packard et Casio distribuent également différents modèles, mais la catégorie entière finit par s'éteindre avec l'arrivée de PC et Mac portables plus abordables et pratiques, laissant le reste du marché à la domination des « simples » calculatrices graphiques et des organisateurs de poches de type Palm Pilot (voir Hackable 42 **[2]**) et Psion.

1. SHARP PC-G850V

Cet ordinateur de poche est une créature unique, créé pour un marché qui l'était tout autant. Construit autour d'un processeur 8 bits compatible Z80A, avec 64 Kio de ROM et 32 Kio de RAM, la bête est assez massive avec ses 196 par 95 par



Comparé aux autres machines équivalentes du marché à l'époque, le PC-G850V possède un gros avantage (en plus du compilateur C embarqué) : un écran de 6 lignes de 24 caractères, bien contrasté et lisible.

20 mm (il faut de grandes poches) et ses quelque 300 grammes, mais propose un écran LCD avec un agréable contraste de 6 lignes de 24 caractères (ou 48 x 144 pixels en mode graphique). La machine est alimentée par 4 piles AAA, ou une alimentation externe de 6 V et est relativement confortable à utiliser avec ses quelque 80 touches proposant à la fois un pavé numérique à droite et un vrai clavier QWERTY sous l'écran.

Mais le plus intéressant n'est pas tant l'aspect matériel que les logiciels en ROM. L'ordinateur, en plus de fonctions mathématiques scientifiques classiques et d'un interpréteur BASIC, propose deux autres approches pour la programmation. Nous avons d'une part un assembleur permettant, sur la machine elle-même, de développer du code dans ce langage et de l'assembler pour produire du code machine Z80. Celui-ci peut ensuite être exécuté, mis au point, importé/exporté de la machine (cf. plus loin) et analysé. Si vous avez suivi la série sur l'ordinateur Z80 sur platine à essais dans les numéros précédents, vous comprendrez aisément pourquoi cette machine me semble absolument fascinante.

Mais ce n'est pas tout. Le PC-G850V intègre rien de moins qu'un compilateur C incluant une bibliothèque standard (libc) modeste mais utilisable. On peut ainsi développer en C, **sur la machine elle-même**, compiler le programme et l'exécuter. Attention cependant, ce compilateur ne produit pas du code machine Z80 comme l'assembleur, mais un *bytecode* interprété/traduit à l'exécution. D'après les informations trouvées sur le Web concernant la commercialisation de ce produit, l'objectif premier était de permettre aux étudiants d'apprendre la programmation structurée en C, qui à l'époque était encore considéré un langage de haut niveau.

Pour créer et gérer ses programmes, en BASIC, en assembleur ou en C, le PC-G850V dispose d'un éditeur de texte, gérant bien entendu la modification du code, mais faisant également office de gestionnaire de fichiers et de convertisseur de données. Les programmes BASIC, en effet, ne sont généralement pas ce qu'ils semblent être dans leur représentation purement textuelle, et le programme lui-même est généralement condensé dans un format plus économe en mémoire.

Cerise sur le gâteau, et à mon plus grand étonnement, en plus de tout cela vous trouverez également, parmi ces déjà riches fonctionnalités, un assembleur et

un programmeur pour PIC de Microchip. Oui, vous avez bien lu, le PC-G850V vous permet de développer du code pour les microcontrôleurs PIC16F627, PIC16F83, PIC16F84 et PIC16F84A et de programmer leur mémoire ! Je n'ai pas testé cette fonctionnalité, n'ayant plus depuis longtemps ce genre de composants sous la main, mais ceci me rappelle bien des souvenirs du fabuleux cours de Bigonoff sur le sujet (toujours en ligne **[3]**).

Cette programmation de microcontrôleurs est rendue possible par l'utilisation d'une interface à 11 broches au pas de 2,54 mm présente sur la tranche gauche de l'ordinateur. Celle-ci peut fonctionner dans différents modes (SIO, SSIO/PWM, GPIO ou PIC) et permet également la connexion avec une imprimante série ou un PC pour le transfert de données. De l'autre côté de la machine se trouve un autre connecteur, de 40 broches sur un circuit double face, c'est le « system bus » qui, comme son nom l'indique donne accès à tous les signaux d'une architecture Z80 : 8 lignes de données, 16 lignes d'adresses, /M1, /MREQ, IORQ, etc. Autant de désignations qui vous rappelleront sans doute quelque chose...

Pour en finir avec la partie « présentation » et avant d'attaquer quelque chose de plus pratique et amusant, il est important de noter que, du fait de la commercialisation initiale restreinte au Japon, le seul moyen de se procurer ce matériel aujourd'hui est un passage obligé auprès de vendeurs japonais sur eBay. On trouve des PC-G850V pour une centaine d'euros très facilement, parfois avec le port gratuit et généralement en excellent état. Beaucoup de vendeurs ont d'excellentes notes et ont déjà vendu une quantité impressionnante de ces machines. Assurez-vous toutefois qu'il s'agit bien de modèles testés et que les photos montrent un écran allumé et présentant au moins une ligne complète de caractères. Ceci vous assurera l'absence d'éventuelles colonnes verticales de pixels désactivés, caractéristiques d'un problème de connexion entre le circuit et l'afficheur (c'est tantôt réparable, mais généralement assez délicat/pénible).

2. UTILISATION DU PC-G850V

Je ne vais pas excessivement entrer dans le détail ici, étant donné qu'il existe un manuel pour la machine **[4]** et que si votre japonais n'est pas assez bon, une version anglaise a été réalisée par des passionnés **[5]**. On y trouve absolument tout le nécessaire pour utiliser le matériel, de l'interface aux mathématiques en passant par le BASIC, une introduction au C, une liste d'instructions pour le Z80, un *mapping* mémoire, et bien d'autres choses.

Ce qui nous intéresse dans la suite de cet article tourne autour du développement, du C, de l'assembleur, du moniteur et surtout de l'échange de données entre l'ordinateur et un PC (ou un SBC sous GNU/Linux).

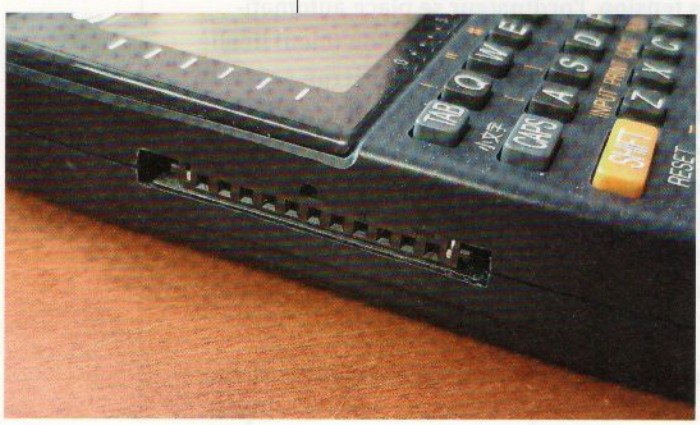
À la mise sous tension, l'ordinateur se place automatiquement en mode exécution BASIC. Un bouton (bleu) libellé « TEST » permet de basculer sur l'éditeur et l'écran présente différentes entrées de menu. Pour les utiliser, il suffit d'entrer la lettre correspondant à la première de chaque mot (sans validation). Dans le cas de l'éditeur par exemple, on utilisera « S » pour accéder à « Sio », présentant alors une autre liste de mots, et ainsi de suite. Revenir en arrière d'un cran passe par l'utilisation de la touche « ON », alors désignée sous le nom « BREAK » dans le manuel.

L'éditeur permet de sauvegarder et de charger des fichiers dans le tampon actif. C'est ce qui se trouve dans ce tampon

qui est utilisé pour compiler (si c'est du C) ou assembler le code. Voyez cela comme si GCC ou GAS travaillaient toujours avec ce qui se trouve présentement dans votre Vi, et non des fichiers. Pour appeler l'assembleur ou le compilateur, on utilisera les fonctions « secondaires », mentionnées en orange sur le clavier, accessibles soit via la touche « SHIFT » maintenue enfoncée en même temps que la touche cible, soit via la touche « 2nd F », puis la touche cible.

Enfin, le moniteur, qui permet de manipuler les données en mémoire et le code machine, est accessible via le BASIC (mise sous tension ou touche « BASIC ») en utilisant l'instruction `MON`. L'écran présente alors le texte « MACHINE LANGUAGE MONITOR » ainsi qu'un prompt (`*`) et vous permet d'entrer des commandes comme `G` pour *go* (sauter à une adresse), `D` pour afficher une zone mémoire, `BP` pour spécifier un *break point*, etc.

Une interface de 11 broches est accessible sur le côté de la machine, cachée derrière une petite trappe. Celle-ci permet non seulement une communication série et un fonctionnement en GPIO avec PWM, mais joue également le rôle de programmeur de PIC Microchip !



Par défaut, après réinitialisation ou premier démarrage de l'ordinateur, le moniteur ne dispose pas de mémoire dédiée, le système allouant par défaut la totalité de la mémoire au BASIC et aux fichiers. Pour allouer de la mémoire au code machine en RAM, on utilisera la commande `USER` suivi de la quantité de mémoire souhaitée en hexadécimal sur 4 positions. Vous pouvez répéter la commande pour changer cette valeur ou utiliser `USER00FF` pour désallouer la totalité de la mémoire (voyez ça comme une valeur « magique »). `USER` seul vous affichera la place mémoire allouée, qui commence toujours à `0x0100` (ou plutôt ici à `"0100H"`).

Notez bien que dans les 30179 octets de RAM à votre disposition (commande `FRE` en mode BASIC) sont partagés l'espace fixe alloué au moniteur pour le code machine, celui consommé dynamiquement par le code actuellement dans l'éditeur, et enfin par les fichiers de code et de données stockés en mémoire. À vous de jongler entre tout cela, sachant qu'il est, en principe, possible d'étendre la mémoire avec 32 Kio supplémentaires via le bus système (signal CERAM2).

De manière générale enfin, en cas de message d'erreur, utilisez la touche « CLS » (rouge) pour accuser réception et revenir au fonctionnement standard.

3. CHARGER ET SAUVEGARDER DES DONNÉES

Voilà qui nous amène précisément à l'objet de cet article avec une petite exploration qui pourrait vous être utile en dehors de ce contexte. Éditer du C ou de l'assembleur sur un écran de 8 cm par 3 cm, avec seulement

24 colonnes et 6 lignes, n'est pas ce qu'on peut considérer être le summum de l'ergonomie. Mieux vaut alors s'orienter sur un développement sur PC avec le confort habituel puis échanger des données avec le G850V. Plusieurs options de développement peuvent être envisagées :

- rédiger le code C dans un éditeur, transférer sur G850V et compiler ;
- faire de même avec une source en assembleur pour obtenir un code machine directement sur l'ordinateur de poche ;
- assembler sur PC avec z80asm ou l'assembleur SDCC et transférer sur le G850V pour exécution dans le moniteur ;
- et enfin, plus hypothétiquement, car cela demande davantage de recherches et d'expérimentations (note de moi du futur : en fait, pas du tout, c'est très simple, cf. la fin de l'article), développer et *cross-compiler* avec SDCC pour ensuite transférer et exécuter le code machine sur le G850V.

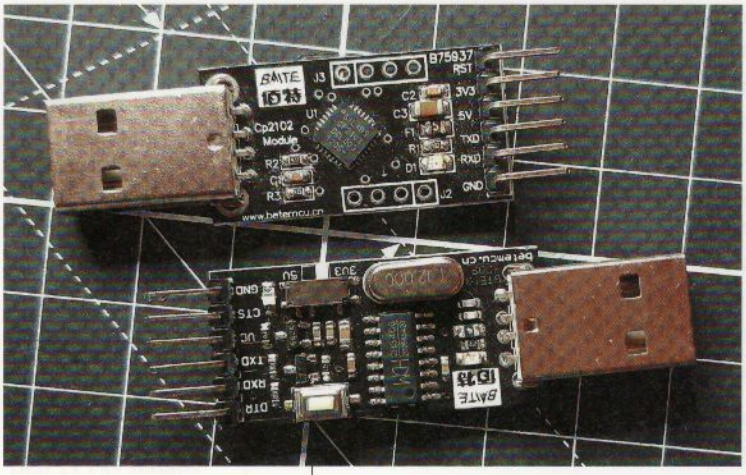
Dans les quatre cas, il nous faut une solution pour échanger des données, et cela tombe bien puisque l'interface 11 broches en mode SIO (*serial I/O*) est spécialement prévue pour cela. Il est possible d'utiliser des outils comme Minicom pour échanger des données, malgré quelques subtilités que nous allons voir dans un instant, ou de développer un petit outil dédié qui, pour changer des habitudes, sera écrit en Go. Mais avant cela, nous devons savoir comment interfacer physiquement le PC-G850V avec le PC ou SBC en USB.

Les adaptateurs USB/série utilisant une puce FTDI comme ce FT232RF peuvent être reconfigurés pour en changer les caractéristiques, dont l'inversion des niveaux logiques sur les lignes RX, TX, RTC, CTS, DTR, etc.

Le connecteur sur la tranche de l'appareil est broché ainsi en mode SIO (avec 1 situé en direction du mot « SHARP » au-dessus de l'écran) :

- 1 : non connecté ;
- 2 : VCC (+5 V) ;
- 3 : Masse ;
- 4 : RTS (*Request To Send*), pour le contrôle de flux matériel, cette ligne en sortie indique qu'on est prêt à recevoir des données ;
- 5 : DTR (*Data Terminal Ready*), pour les modems, inutilisé ici ;
- 6 : RX, réception de données ;
- 7 : TX, envoi de données ;
- 8 : CD (*Carrier Detect*), pour les modems, inutilisé ici ;
- 9 : CTS (*Clear To Send*), pour le contrôle de flux matériel, cette ligne en entrée nous indique si on peut envoyer des données ;
- 10 : DSR (*Data Set Ready*), pour les modems, inutilisé ici ;
- 11 : CI (*Call Indicator*), pour les modems, inutilisé ici.

Naturellement, on aurait tendance à penser qu'il suffit de connecter un adaptateur USB/série (utilisant des niveaux logiques 0/5 V) ainsi : masse sur masse, RX sur TX et TX sur RX, et ceci fonctionnera sans problème pour le transfert de sources et de texte depuis l'éditeur. C'est le genre de brochage que l'on trouve généralement sur le Web, que ce soit avec un adaptateur USB/série, une carte



Un adaptateur construit autour d'un contrôleur comme le CP2102 de Silabs (en haut) ou le CH340 de WCH (en bas) ne pourra pas, semble-t-il, être reconfiguré pour être adapté aux niveaux logiques du PC-G850V. Ce n'est pas grave, il suffit de faire transiter chaque signal via un sextuple inverseur comme un 74×14 ou un 74×04.

Arduino ou un ESP8266/ESP32 pour ajouter une connectivité Wi-Fi ou Bluetooth. Ce type de montages met également RTS et CTS au niveau haut via une résistance de rappel de 10 kΩ reliée à la broche 2 (+5 V).

Malheureusement, ceci ne s'avère pas suffisant lorsqu'on souhaite utiliser pleinement les fonctionnalités d'échange via le port série. En effet, contrairement à l'éditeur de texte, dont la configuration impacte le fonctionnement général de la communication, le moniteur n'arrive pas à traiter les données reçues suffisamment rapidement pour se passer de contrôle de flux. Celui-ci accepte des données série au format Intel Hex (ou ihex) incluant, pour chaque ligne, une adresse mémoire et une somme de contrôle devant être vérifiée avant que le code machine ne soit inscrit en RAM. Cette procédure prend du temps et le G850V agit alors sur sa sortie RTS pour demander une pause (environ 5 ms) à l'émetteur avant d'accepter un nouveau lot de données. Si cette pause n'est pas respectée et que l'émetteur poursuit l'envoi, les données sont corrompues, la somme de contrôle invalide et le transfert échoue.

En résumé, sans contrôle de flux RTS/CTS, l'éditeur de texte fonctionne dans les deux sens, le moniteur peut envoyer des données (commande `W`), mais il ne peut pas en recevoir (commande `R`). Ceci m'a fait perdre un temps considérable, au point de finalement sortir l'analyseur logique pour clairement identifier le problème. Tout cela parce que la plupart des codes, documentations et projets en lien avec le PC-G850V ignorent totalement le moniteur et se concentrent uniquement sur l'éditeur de texte qui n'est, à mon sens, pas plus important que la possibilité de charger du code binaire directement en mémoire.

3.1 Interface matérielle et configuration

Vous l'aurez compris, si on veut utiliser la liaison série dans l'éditeur et le moniteur, ceci implique d'utiliser RTS/CTS et donc, de toute façon, d'avoir à gérer un contrôle de flux matériel. Autant généraliser la configuration à l'ensemble des échanges et effectivement configurer ce contrôle de flux à la base. Paramétrer la liaison série se fait directement dans l'éditeur via les menus « Sio » (« S ») et « Format » (« F »). Là, une série d'éléments sont listés, qu'on pourra ajuster avec les flèches de direction gauche/droite. Pour enregistrer un paramètre, il faudra utiliser la touche de validation, car dans le cas contraire, en passant simplement au paramètre suivant (flèche vers le bas), l'ancienne valeur sera restaurée.

La configuration choisie ici sera :

- « baud rate » : 9600 (maximum) ;
- « data bit » : 8 ;
- « stop bit » : 1 ;
- « parity » : « odd » (impaire) ;
- « end of line » : « LF » (`\n` ou `0x0a`) ;
- « end of file » : « 1A » (`0x1A` correspondant au caractère `SUB` ou CTRL+Z) ;
- « line number » : « no » ;
- « flow » : « RS/CS ».

Je préfère ici utiliser une parité, c'est une habitude qui m'a bien rendu bien des services par le passé, en particulier avec du matériel ancien, car cela ajoute un mécanisme supplémentaire pour garantir l'intégrité des échanges. Le marqueur de fin de ligne, ici « LF » (*Line Feed*), peut également être configuré sur « CR » (*Carriage Return* ou `\r`) ou « CR+LF » (`\r\n`) mais « LF » est le standard UNIX. Utiliser `0x1a` comme marqueur de fin de fichier est la configuration par défaut, c'est le caractère qui signifiera la fin d'un envoi, que ce soit du PC vers le G850V ou dans le sens inverse. Vous pouvez éventuellement préférer `0x04` (CTRL+D), correspondant au caractère EOT pour *End Of Transmit*, mais je laisse l'adaptation de ce qui va suivre à votre bon soin dans ce cas. Enfin, « line number » influe sur la présence d'une numérotation des lignes, que ce soit en importation ou en exportation. À « no », les lignes de texte provenant du G850V ne seront pas numérotées et la machine n'attendra pas non plus de numérotation dans le texte réceptionné. À « yes », cette numérotation est présente à l'exportation et obligatoire à l'importation, même pour un code C où ces numéros ne sont pas utilisés et n'ont pas de sens.

Ce format de communication, 9600 8O1 avec contrôle de flux matériel, devra être respecté côté PC, mais il nous reste un problème important à régler : les niveaux logiques sont inversés avec cette machine. Ici, un +5 V pour RX ou TX signifie `0` et 0 V indique un `1` logique. De la même manière, alors que les signaux de contrôle de flux sont normalement /RTS et /CTS, avec `1` étant une mise à la masse et `0` correspondant à un niveau haut (+5 V), ici, c'est l'inverse. Vous ne pouvez donc pas simplement relier RX à TX, TX à RX, CTS à RTS et RTS à CTS, cela ne fonctionnera simplement pas.

Pour inverser les niveaux logiques, deux solutions peuvent être envisagées : soit vous utilisez un adaptateur construit autour d'un circuit intégré FTDI comme le FT232R et vous pouvez reconfigurer le composant, soit vous inversez les niveaux à l'aide d'un circuit logique complémentaire comme un 74×04 ou un 74×14, qui sont tous deux des sextuples inverseurs (le 74×14 intégrant des *triggers* de Schmitt, alias « bascules à seuil »). Un composant comme le 74HC14 fera parfaitement l'affaire et son utilisation est simplissime : il suffit de l'alimenter en +5 V et de relier chaque signal sur une entrée « A » pour obtenir, sur la sortie « Y » correspondante, un signal inversé.

Tant que l'adaptateur fournit les broches RTS et CTS, ce qui n'est pas toujours le cas avec des modèles avec seulement 4 ou 6 broches (Vcc, masse, RX, TX, CTS et DTR, parfois 3v3 et 5 V ou RST), la conversion est possible. On préférera cependant une puce FTDI (plutôt qu'un CH340 ou CP2102), économisant un composant, même si cela oblige à quelques manipulations complémentaires.

La configuration d'un FT232R réside en EEPROM et l'opération n'est donc pas définitive. Un outil officiel de configuration, appelé FT_PROG, est proposé sur le site du fabricant **[6]**, mais celui-ci n'est ni *open source*, ni disponible pour autre chose que Windows (avec installation obligatoire du *framework* Microsoft .NET 4.0 en prime). Fort heureusement, une alternative

libre existe et est très certainement disponible pour votre distribution GNU/Linux (vérifié pour Debian et Raspberry Pi OS), ainsi que pour FreeBSD, OpenBSD et NetBSD : `ftdi_eeprom` (reposant sur la libftdi du même auteur [7]).

`ftdi_eeprom` n'est pas un programme que je qualifierais d'agréable à utiliser puisque son fonctionnement est relativement alambiqué. La première chose à faire avant de l'utiliser, et après avoir connecté l'adaptateur à la machine, sera de décharger le module noyau assurant le support du périphérique avec la commande `sudo rmmod ftdi_sio` (la version 1.5 de la libftdi est censée détacher le pilote du périphérique automatiquement). Ceci fait, vous pourrez lister les périphériques USB avec `lsusb` pour repérer les identifiants qui lui correspondent (normalement, `0x0403:0x6001`).

Il faudra ensuite créer un fichier de configuration (`ftdi.conf`) contenant :

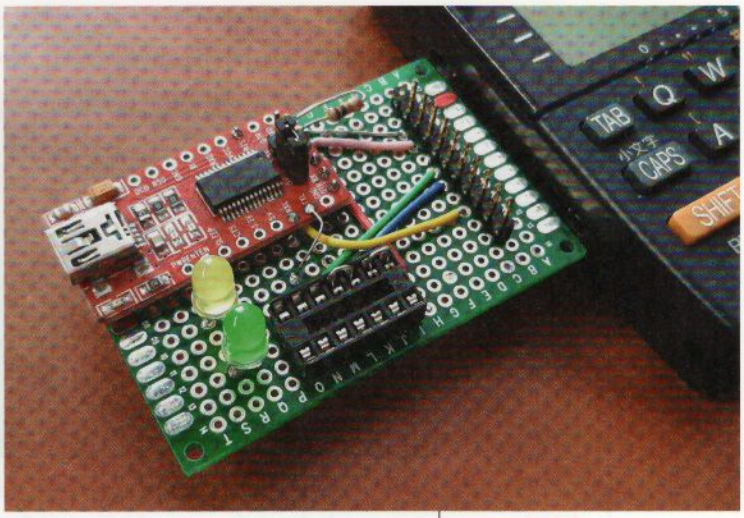
```
filename=FTDIeeprom.orig
vendor_id=0x0403
product_id=0x6001
```

On retrouve là les deux identifiants USB ainsi qu'un nom de fichier permettant de stocker le contenu de l'EEPROM que nous allons récupérer de suite avec :

```
$ ftdi_eeprom --verbose --read-eeprom ftdi.conf
FTDI eeprom generator v0.17
(c) Intra2net AG and the libftdi developers
    <opensource@intra2net.com>
FTDI read eeprom: 0
EEPROM size: 128
VID:     0x0403
PID:     0x6001
Release: 0x0000
Bus Powered:  90 mA USB Remote Wake Up
Manufacturer: FTDI
Product:      FT232R USB UART
Serial:       A9ED1BZ3
Checksum      : 3721
Internal EEPROM
Oscillator: Internal
Enable Remote Wake Up
PNP: 1
Channel A has Mode UART VCP
C0 Function: TXLED
C1 Function: RXLED
C2 Function: TXDEN
C3 Function: PWREN
C4 Function: SLEEP
FTDI close: 0
```

Ceci est la configuration actuelle de la puce et vous trouverez un fichier `FTDIeeprom.orig` dans le répertoire courant, contenant une sauvegarde.

L'étape suivante consistera à modifier/compléter le fichier `ftdi.conf` pour définir une nouvelle configuration. Un exemple de fichier de configuration est normalement présent dans le système (`/usr/share/doc/ftdi-eeprom/example.conf.gz`) pour vous aider dans cette tâche. Il n'est pas forcément nécessaire de spécifier toutes les informations de configuration et vous pourriez parfaitement vous satisfaire de modifier le fichier ainsi :



Réaliser un adaptateur est grandement facilité par le pas de 2,54 mm du connecteur 11 broches. Notez la présence d'un socket 14 broches pour un 74hc14, initialement utilisé avant de finalement opter pour la solution consistant à reconfigurer la puce FTDI.

```
filename=FTDIeeprom.new
vendor_id=0x0403
product_id=0x6001
invert_txd=true
invert_rxd=true
invert_rts=true
invert_cts=true
```

Ici, nous n'avons fait qu'ajouter les quatre lignes/paramètres permettant d'inverser les niveaux logiques pour TX, RX, RTS et CTS (en plus de changer le nom du fichier). Mais l'utilisation de cette configuration utilisera des valeurs par défaut choisies par l'outil lui-même, dont un nom de fabricant, un nom de produit et un numéro de série ne correspondant en rien à ceux d'origine. Ce n'est pas un problème en soi, mais ce n'est pas nécessairement une approche souhaitable. Si ceci ne vous convient pas, utilisez la sortie de la précédente commande, ainsi que le fichier d'exemple, pour composer un fichier de configuration correspondant à la configuration récupérée avec nos quatre lignes ajoutées.

Une fois votre configuration adaptée à vos préférences, vous pourrez générer une nouvelle image du contenu de l'EEPROM avec :

```
$ ftdi_eeprom --verbose --build-eeprom ftdi.conf
FTDI eeprom generator v0.17
(c) Intra2net AG and the libftdi developers <opensource@intra2net.com>
FTDI read eeprom: 0
EEPROM size: 128
Used eeprom space: 86 bytes
Writing to file: FTDIeeprom.new
FTDI close: 0
```

Un nouveau fichier `FTDIeeprom.new` sera créé, correspondant à cette configuration, mais vous pouvez tout aussi bien flasher directement le composant avec :

```
$ ftdi_eeprom --verbose --flash-eeprom ftdi.conf
FTDI eeprom generator v0.17
(c) Intra2net AG and the libftdi developers
    <opensource@intra2net.com>
FTDI read eeprom: 0
EEPROM size: 128
Used eeprom space: 86 bytes
FTDI write eeprom: 0
Writing to file: FTDIeeprom.new
FTDI close: 0
```

Notez que `--flash-eeprom` comme `--build-eeprom` produisent un nouveau fichier `FTDIeeprom.new` sur la base de votre `ftdi.conf`, ce qui n'a pas vraiment de sens pour moi. Quoi qu'il en soit, vous pouvez éventuellement remodifier le fichier de configuration pour ajuster le nom du fichier image et refaire une passe de lecture de l'EEPROM pour valider vos changements. Dans le cas qui nous intéresse ici, ce que vous voulez voir est la ligne :

```
Inverted bits: TXD RXD RTS CTS
```

Notez que les changements apportés ne seront effectifs qu'après déconnexion et reconnexion de l'adaptateur, ce qui rechargera également le pilote dans le noyau, vous fournissant à nouveau une entrée dans `/dev/`.

`ftdi_eeprom` est un outil pour le moins perturbant. Pourquoi ne pas avoir simplement choisi de spécifier le fichier image en option sur la ligne de commande et utiliser une autre option pour créer une configuration à partir de cette image, ou produire une image à partir d'une configuration existante ? Le tout avec une option pour simplement lire l'EEPROM ou l'écrire ? Le comportement actuel est un choix des développeurs et ne râlons pas trop, tout de même. Disposer de cette bibliothèque et de cet outil, sous GPLv2 et supportant une belle collection de puces FTDI, est une aubaine en soi (et un tour de force des développeurs, étant donné la maigre documentation officielle par FDTI). Et puis, on peut toujours réécrire `ftdi_eeprom` si on a une meilleure idée (non, pas forcément en Rust !).

Un conseil : pensez à bien marquer votre adaptateur ainsi modifié, car il ne fonctionne clairement plus de façon standard et vous risqueriez de perdre énormément de temps à déboguer un faux problème en tentant de l'utiliser avec un autre projet.

3.2 Utilisation de Minicom

À présent que nous avons une interface matérielle en état de fonctionner et une configuration du PC-G850V adaptée, nous devons nous pencher sur l'aspect purement logiciel. Vous l'aurez compris via les éléments que je viens de détailler qu'un lot de données échangé avec l'ordinateur de poche, ce que je qualifierais maintenant de « fichier », se compose de lignes

terminées d'un caractère `\n` et que cet ensemble se termine avec un caractère `0x1a` ou CRTL+Z. Pour échanger des fichiers avec le G850V, nous devons donc prendre en considération ces deux caractéristiques.

Avant d'entamer la création d'un outil pour ces transferts, nous pouvons débuter les expérimentations avec un utilitaire de communication série comme Minicom. Celui-ci n'est pas le seul utilisable, mais certainement le plus populaire en environnement Unix, car réellement complet. Il vous faut en effet un outil vous offrant une excellente maîtrise des paramètres de communication, tout en offrant une richesse suffisante en fonctionnalités.

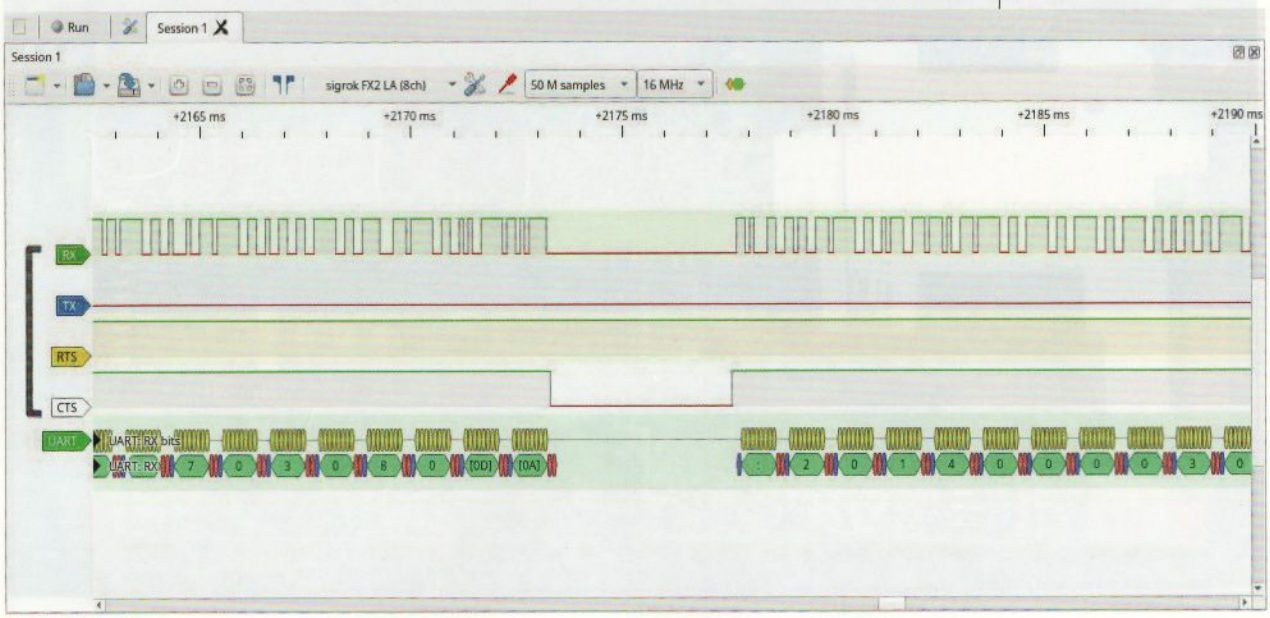
Pour configurer Minicom pour nos besoins, nous pouvons soit passer par les différents menus accessibles via la séquence CTRL+A puis Z, soit aller au plus simple en éditant directement un fichier de configuration, à placer dans votre `$HOME` et ayant pour nom `.minirc.` suivi d'un nom de profil (ou `dfl` pour *default*) : `~/.minirc.g850v` par exemple. Pour le PC-G850V, notre configuration contiendra :

```
pu baudrate             9600
pu bits                 8
pu parity               O
pu stopbits             1
pu rtscts               Yes
pu localecho            Yes
pu addcarreturn         Yes
```

Avec dans l'ordre :

- vitesse 9600 b/s ;
- 8 bits de données ;
- parité impaire ;
- 1 bit de stop ;
- contrôle de flux matériel ;
- écho local des saisies (ce qu'on tape apparaît à l'écran en plus d'être envoyé) ;

Voilà précisément le problème qui m'aura fait perdre énormément de temps. La ligne CTS (côté adaptateur) est mise à la masse par le RTS du PC-G850V durant la réception d'un fichier Intel Hex. Cette pause imposée par la machine doit être respectée, car dans le cas contraire, le transfert échouera. En d'autres termes, le contrôle de flux matériel (RTS/CTS) est obligatoire pour « uploader » du code dans le moniteur.



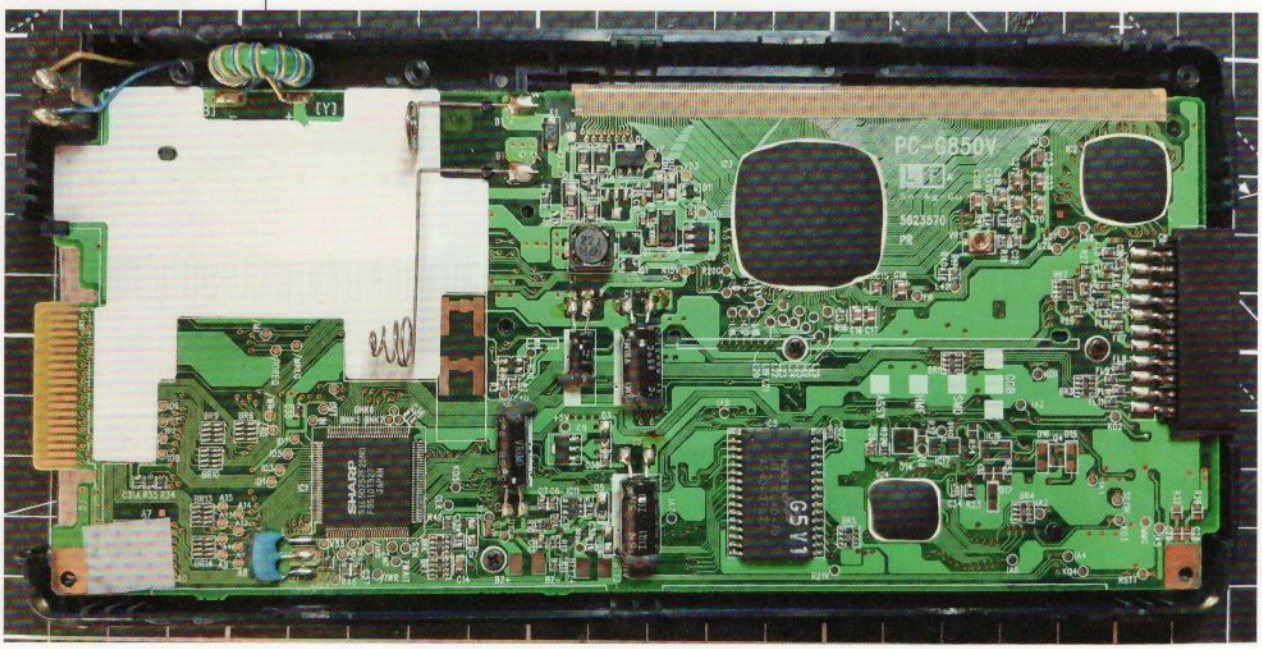
• et on ajoute automatiquement un caractère `\r` à l'écran à chaque ligne reçue pour éviter un effet d'escalier.

Il suffira alors d'appeler Minicom en précisant le port à utiliser via `-D` (`/dev/ttyUSB0`, par exemple) et le profil, ici `g850v`. Automatiquement, la configuration en question sera utilisée et les paramètres en accord avec ceux choisis côté PC-G850V. Pour vérifier le bon fonctionnement de l'interface, rendez-vous dans l'éditeur de texte, composez un code quelconque, revenez au menu avec « ON » puis utilisez « Sio » et « Save ». Sur l'écran Minicom doit alors apparaître le code en question, bien formaté et sans numéros de ligne.

L'opération inverse nécessite une étape supplémentaire. Placez l'éditeur en réception via « Sio » et « Load », collez ou tapez quelques lignes de code en validant normalement chaque ligne, puis terminez la transmission avec un caractère ASCII `SUB` (`0x1a`). Pour cela, rien de plus simple puisqu'il est possible dans un terminal (Xterm ou RXVT) d'envoyer ce caractère de contrôle directement, avec CTRL+Z. Ce raccourci terminera donc la transmission.

Dans le moniteur (touche BASIC, puis `MON`), le fonctionnement doit être identique. Vous pouvez envoyer des données binaires avec `W` suivi d'une adresse de départ, virgule, une adresse de fin. `W0100,011F` nous affichera par exemple dans Minicom :

```
:10010000101200000110504320202020202020202029
:1001100020001900000000000FB636F75636F751D
:00000001FF
```



L'inspection du circuit interne de l'ordinateur de poche révèle la présence de plusieurs condensateurs électrolytiques qui devront, par définition, être remplacés à un moment ou un autre.

Nous avons là les 32 octets (`0x20`) présents en mémoire entre `0x0100` et `0x011f`, encodés au format Intel Hex (ou ihex). Inversement, nous pouvons, comme avec le code source précédent, copier/coller de l'Intel Hex dans Minicom, après avoir placé le moniteur en réception avec `R`, puis terminer avec CTRL+Z. Le format ihex intègre directement l'adresse de destination à chaque ligne (voir **[8]**), inutile de la préciser pour `R` (mais c'est également possible).

Nous pouvons pousser un brin plus loin, car le fait de copier/coller, dans un sens comme dans l'autre, n'est pas ce qu'il y a de plus agréable, en particulier avec de gros volumes de code ou de données ihex. Minicom intègre de base un système d'échange de données reposant sur l'utilisation d'outils externes manipulant des fichiers. Pour les transferts ASCII, par opposition à des choses comme Zmodem ou Kermit, un utilitaire est livré avec Minicom : `ascii-xfr`. La configuration de ces outils est accessible via la configuration (CTRL+A, « Z » et « Protocoles de transfert »).

Vous remarquerez, dans la configuration par défaut, les options `-dsv` utilisées avec `ascii-xfr` : `s` pour *send* (envoi), `v` pour le mode verbeux et surtout, `d` pour utiliser CTRL+D comme marqueur de fin de transmission plutôt que CTRL+Z, mais ce marqueur n'est en réalité pas utilisé (option `-e`). Pour adapter cette commande à nos besoins, nous devons utiliser `-sven`, avec `e` pour envoyer le marqueur (CTRL-Z par défaut) et `n` pour ne pas convertir les CR+LF en LF (et inversement). Encore une fois, nous pouvons changer cette configuration dans l'interface ou ajouter des lignes à notre `~/.minirc.g850v` :

```
pu pname10          YUNYNasciiG850
pu pprog10          ascii-xfr -sven
pu pname11          YDNYNasciiG850
pu pprog11          ascii-xfr -rven
```

Notez que nous ajoutons également une entrée avec l'option `-r` pour la réception. Ceci fait, les copier/coller ne sont plus nécessaires et on peut utiliser la séquence CTRL+A « S » pour l'envoi et CTRL+A « R » pour la réception avec, dans les deux cas, une sélection du fichier à lire ou écrire. C'est bien plus confortable.

4. CRÉER SON OUTIL EN GO

Utiliser Minicom est une solution fonctionnelle, mais ceci n'est pas optimal. La saisie involontaire de caractères dans le terminal peut provoquer une erreur de transfert, il faut beaucoup de manipulations pour envoyer ou recevoir un simple fichier et ceci est relativement difficile à automatiser (pour une intégration future dans un `Makefile`, par exemple). Pourquoi ne pas nous passer de Minicom et, à présent que nous savons tout ce qu'il faut savoir sur les formats et protocoles utilisés, créer un outil dédié ?

Pour changer un peu du récurrent C, nous allons écrire cela en Go (et aussi parce que les `struct termios` sont un peu pénibles à utiliser). C'est une bonne idée au départ, jusqu'à ce qu'on se rende compte que beaucoup de modules Go ne couvrent pas nos besoins ou, pire encore, annoncent le faire mais ne le font pas. Je parle ici du contrôle de flux matériel, non pris en charge par la plupart du code qu'on trouve en ligne. Et dieu sait que j'en ai testé, au point de devoir utiliser un analyseur logique pour me rendre à l'évidence : le code ne faisait absolument pas ce qu'il était censé faire avec RTS et CTS.

Finalement, le module choisi et surtout réellement utilisable est `github.com/jacobsa/go-serial/serial` de Aaron Jacobs **[9]** qui, en compagnie de `github.com/pborman/getopt/v2` pour la gestion des options en ligne de commande, apporte le résultat attendu. Le code est relativement succin (comparé à un équivalent C), et débute très classiquement avec l'inclusion des modules :

```
package main

import (
    "fmt"
    "log"
    "bufio"
    "github.com/jacobsa/go-serial/serial"
    "github.com/pborman/getopt/v2"
)
```

Ceci fait, nous pouvons attaquer la fonction principale en commençant par gérer les options de la ligne de commande :

```
func main() {
    var filename string
    var devName string

    getopt.Flag(&filename, 'u', "upload to PC-G850")
    getopt.Flag(&devName, 'd', "serial device").Mandatory()

    getopt.Parse()
```

L'objectif ici est de pouvoir spécifier le périphérique série à utiliser avec une option `-d` qui est obligatoire (d'où le `Mandatory()`) et gérer une option `-u` provoquant une écriture vers le G850V, en l'absence de laquelle nous avons une lecture avec un affichage sur la sortie standard.

L'étape suivante consiste à configurer notre liaison série et c'est là que les autres modules testés affichaient leur limitation :

```
    config := serial.OpenOptions{
        PortName: devName,
        BaudRate: 9600,
        DataBits: 8,
        StopBits: 1,
        MinimumReadSize: 0,
        InterCharacterTimeout: 5000,
        ParityMode: serial.PARITY_ODD,
        Rs485Enable: false,
        Rs485RtsHighDuringSend: false,
        Rs485RtsHighAfterSend: false,
        RTSCTSFlowControl: true,
    }
```

Vous reconnaîtrez sans peine les éléments de configuration que nous avons utilisés avec Minicom, avec en plus `InterCharacterTimeout` limitant le temps d'attente à cinq secondes. Si aucun caractère n'est reçu durant cette période, avant que nous mettions fin à la réception, ceci provoquera une erreur. Nous pouvons alors ouvrir le port :

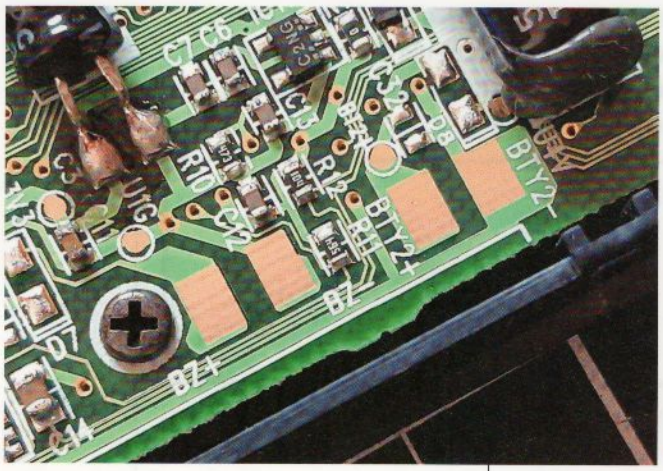
```
    port, err := serial.Open(config)
    if err != nil {
        log.Fatal(err)
    }

    defer port.Close()
```

Et commencer à l'utiliser en gérant tout d'abord l'envoi de données. Le processus est relativement simple puisque nous nous contentons d'ouvrir le fichier, placer son contenu dans un *slice* et compléter du caractère `0x1a`, avant de tout écrire sur le port série :

```
    if filename != "" {
        content, err := ioutil.ReadFile(filename)
        if err != nil {
            log.Fatal(err)
        }

        fmt.Println("Uploading to PC-G850...")

        // add EOF
        content = append(content, '\x1a')

        port.Write(content)
        if err != nil {
            log.Fatal(err)
        }

        fmt.Printf("%d bytes sent\n", len(content))
```

De l'autre côté de la condition `if`, nous avons la lecture, qui est encore plus simple :

```
    } else {
        reader := bufio.NewReader(port)
        data, err := reader.ReadBytes('\x1a')

        if err != nil {
            log.Fatal(err)
        }

        fmt.Println(string(data[:len(data)-2]))
    }
}
```



Par défaut, le PC-G850V dispose de routines permettant la génération de sons, mais n'inclut pas de périphérique audio. On pourra cependant ajouter un buzzer piézo-électrique puisqu'un emplacement est prévu à cet effet sur le circuit imprimé. À droite, nous avons également ce qui semble être un emplacement pour une batterie de secours.

L'autre connecteur présent sur la tranche du PC-G850V expose l'ensemble des bus et lignes de contrôle du processeur Z80, ainsi que des signaux CEROM2 et CERAM2 permettant d'ajouter de la ROM et de la SRAM supplémentaire.

Ici, nous réceptionnons les données que nous accumulons dans `data` jusqu'à soit obtenir le caractère `0x1a`, soit dépasser le délai d'attente (5 s), ce qui provoquera une erreur. Le contenu de `data`, diminué de `0x1a` et d'un caractère LF qui semble systématiquement ajouté par le CP-G850V, est ensuite affiché sur la sortie standard.

Tout ceci est placé dans un fichier `g850xfer.go` accompagné d'un fichier `go.mod` créé grâce aux commandes `go get "github.com/jacobsa/go-serial/serial"` et `go get github.com/pborman/getopt/v2`. Nous n'avons plus qu'à compiler avec `go build` et notre outil est utilisable avec :

```
./g850xfer -d /dev/ttyUSB0 > fichier.xxx
```

ou

```
./g850xfer -d /dev/ttyUSB0 -u fichier.xxx
```

Le code Go, qui évoluera peut-être dans le futur, est disponible dans mon GitLab **[10]**.

5. POUR FINIR

Il est toujours intéressant de se pencher sur d'anciennes technologies comme celles-ci, non pas nécessairement pour en faire usage au quotidien, mais pour se heurter à des difficultés et résoudre les problèmes. Ici, bien que le Sharp PC-G850V soit clairement une petite merveille encore aujourd'hui, nous avons pu jouer avec la configuration des puces FTDI, gérer le contrôle de flux matériel d'une liaison série, faire un brin d'exploration dans les modules du langage Go et utiliser quelques outils pouvant resservir à l'occasion avec d'autres plateformes.

À présent que les problèmes sont réglés et que nous avons même un utilitaire dédié à disposition, la suite est bien plus ludique. En effet, nous avons là une machine 8 bits Z80 qui nous tend les bras et ne demande qu'à être explorée d'une manière plus... « utile ». Je parle, bien sûr, de commencer à développer sérieusement avec un environnement plus moderne comme SDCC et Z88DK **[11]**. Je vous avoue que je comptais terminer l'article sur ce pseudo-*cliffhanger*, mais je n'ai finalement pas pu résister...

Z88DK semble clairement être une petite merveille et, grâce à Docker, ne nécessite pas d'installation compliquée. Pour preuve, considérons le code C suivant (`hello.c`, qui n'est pas de ma création **[12]**) :

```
#include <graphics.h>
#include <math.h>
#include <stdio.h>

void main() {
    int x, y;
    clg();

    plot(0,24);
    for (x = 0; x < 144; x++) {
        y = (int)(24*(1.0-sin(x/144.0*3.14*8)));
        drawto(x, y);
    }

    // use "enter" to stop
    while (getk() != 10) {};
}
```

Produire un binaire et un Intel Hex est un jeu d'enfant (le téléchargement n'a lieu que lors du premier lancement du conteneur) :

```
$ docker run -v ${PWD}:/src/ --rm -it z88dk/z88dk \
  zcc +g800 -lm -create-app -clib=g850b -ohello.bin \
  hello.c
Unable to find image 'z88dk/z88dk:latest' locally
latest: Pulling from z88dk/z88dk
4abcf2066143: Pull complete
da16ac30c08b: Pull complete
f1956c30cadd: Pull complete
40815aa36945: Pull complete
15091d97b1a8: Pull complete
Digest: sha256:673393b602de40482eacd5f6432d8f
    137954f0946c6282d01633001db655215e
```

```
Status: Downloaded newer image for z88dk/z88dk:latest

$ ls
hello.bin  hello.c  hello.ihx  hello.rom
```

Il suffit ensuite d'aller dans le moniteur pour réceptionner les données (R), puis utiliser notre g850xfer :

```
$ ./g850xfer -d /dev/ttyUSB0 -u hello.ihx
Uploading to PC-G850...
13959 bytes sent
```

Et enfin, se plier d'un G0100 pour voir s'afficher un sublime sinus en mode graphique sur l'écran du PC-850G ! Voilà qui ouvre des possibilités fascinantes en termes de développement et mérite d'être exploré avec bien plus de soins (Z88DK m'a l'air incroyablement riche). Et pourquoi pas, dans le même temps, se pencher également sérieusement sur le fameux connecteur « SYSTEM BUS » ? Mais ça, c'est une tout autre aventure qui trouvera place (ou non) dans les pages d'un futur numéro. **DB**

RÉFÉRENCES

[1] **https://fr.wikipedia.org/wiki/Septembre_%C3%A9ternel**

[2] **https://connect.ed-diamond.com/hackable/hk-042/developper-pour-pda-de-plus-de-20-ans-en-2022-c-est-possible**

[3] **https://www.abcelectronique.com/bigonoff/loadpart1.php?par=3673e**

[4] **https://pockemul.com/wp-content/uploads/2020/05/PC-G850VSEng_V3_0.pdf**

[5] **http://basic.hopto.org/basic/manual/Sharp%20PC-G850V%20JAP.pdf**

[6] **https://ftdichip.com/utilities/**

[7] **https://www.intra2net.com/en/developer/libftdi/**

[8] **https://en.wikipedia.org/wiki/Intel_HEX**

[9] **https://github.com/jacobsa/go-serial**

[10] **https://gitlab.com/0xDRRB/g850xfer**

[11] **https://github.com/z88dk/z88dk/wiki/Platform---Sharp-PC**

[12] **https://ashitani.jp/g850/docs/04_z88dk.html**

RPLL

RENCONTRES PROFESSIONNELLES *DU LOGICIEL LIBRE*

ÉDITION #4

SAVE THE DATE

Lundi 10 juin 2024

À l'hôtel de la Métropole de Lyon

20 rue du Lac, à 15 min de la gare de la Part Dieu

De 9h à 18h

+ 20 EXPOSANTS

ENTREPRISES ET ASSOCIATIONS DU NUMÉRIQUE LIBRE

+ 15 CONFÉRENCES

LOGICIELS, GESTION DES DONNÉES, MAINTENANCE ET SÉCURITÉ

Venez échanger autour de l'écosystème du logiciel libre et de l'Open-source dans la région Auvergne Rhône-Alpes.

INSCRIVEZ-VOUS GRATUITEMENT SUR **WWW.RPLL.FR**

UN ÉVÈNEMENT ORGANISÉ PAR

PLOSS-RA
Entreprises du Numérique Libre en Rhône-Alpes Auvergne

AVEC LE SOUTIEN DE

MÉTROPOLE GRAND LYON

SUIVEZ L'ÉVÈNEMENT
#RPLL2024

MINISTÈRE
DES ARMÉES

Liberté
Égalité
Fraternité

#cyberexpert @DGA

Rejoignez-nous à Rennes, dans les métiers de :
Lutte Informatique Défensive et Lutte Informatique Offensive

DGA Maîtrise de l'information

POSTULEZ en cybersécurité !

CONTACT Envoyez votre candidature

dga-mi-bruz.recrutement.fct@intradef.gouv.fr

DIRECTION GÉNÉRALE DE L'ARMEMENT
DGA Maîtrise de l'information