ÉLECTRONIQUE | EMBARQUÉ | RADIO | IOT

HACKABLE

L'EMBARQUÉ À SA SOURCE

N° 53
MARS / AVRIL 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 - 53 - F: 14,90 € - RD

CPPAP : K92470

HACK / ZIGBEE

Prenons le contrôle d'un routeur ZigBee LIDL pour le libérer des services Apple et Google p.20

DOMOTIQUE / ÉCRAN

Contrôlez la luminosité de votre moniteur automatiquement en fonction de l'éclairage ambiant p.124

REVERSE / BLUETOOTH

Analysez le protocole Bluetooth d'une imprimante d'étiquettes pour l'appliquer en USB p.102

Bus / Interfaces / Open Hardware

Accéder à tous les bus avec un seul outil ?

HYDRABUS
CONNECTEZ-VOUS À TOUT ! p.40

SPI
i2c
SWD
USART
JTAG
1-Wire
CAN
Wiegand

ESP32 / CRYPTO

Découvrez les mécanismes de boot sécurisé et la signature électronique de firmware des ESP32 p.56

ESP32 / MODULES

Développez et distribuez vos propres modules pour l'environnement de développement ESP-IDF p.04

SMARTCARD / NFC

Continuons notre exploration des Java Cards et exploitons les fonctionnalités sans contact p.78

20th ANNIVERSARY EDITION

CITÉ DES SCIENCES ET DE L'INDUSTRIE

5 & 6 JUILLET 2024 - PARIS

HTTPS://LEHACK.ORG/→

LE FRENCH HACKERS' CONFERENCE

ÉDITO

Chers consœurs et confrères journalistes de la presse « technique » et « scientifique », si je vous écris ce mot aujourd'hui, c'est pour vous donner un petit conseil qui pourra peut-être à la fois vous éviter de passer pour des demeurés, mais aussi, accessoirement, de prendre vos lecteurs pour des demeurés d'un calibre similaire, sinon bien supérieur.

Mon conseil est tout simple, quand vous recevez un communiqué de presse d'une *startup* chinoise (ou autre) vantant les mérites d'un produit révolutionnaire, qui n'est pas fabriqué, sans prix annoncé, sans caractéristiques publiées, sans vraies photos de prototype, mais promettant tout de même de révolutionner ceci ou cela... faites une pause, prenez une tisane de camomille et souvenez-vous de vos cours de collège. Vous savez, ceux qui parlaient de volts, de watts et d'énergie.

Vous voulez un exemple ? Que diriez-vous de la batterie *Betavolt* ? Mais siiii. Vous avez écrit à son propos, émerveillés par les 100 microwatts produits pendant 50 ans, avec une tension de 3 volts et une taille de seulement 15 x 15 x 5 mm. Vous avez fantasmé sur son utilisation avec un smartphone, dans l'aérospatiale, pour des petits drones et même pour l'IA (allez comprendre le rapport), en répétant bêtement l'annonce du fabricant promettant une version de 1 watt en 2025.

Vous voulez paraître intelligent en faisant un calcul, même niveau CM2 ? En voilà un, un smartphone peut utiliser jusqu'à 2 watts en activité. C'est 20000 fois plus que ce que peut fournir cette « batterie révolutionnaire » (et surtout imaginaire, alors que d'autres, comme City Labs, produisent déjà depuis des années avec une technologie sensiblement différente (tritium, et non un isotope de nickel)). La densité énergétique étant le point clé du communiqué que vous avez reçu, il est improbable qu'elle augmente. Ceci veut dire de cette « innovation technologique sans précédent » aura donc la forme équivalente à 20000 exemplaires de la soi-disant batterie qui vous a tant excité en rendu 3D. On parle là d'un pavé de 30 cm de côté et de 25 cm de haut (20 x 20 x 50 batteries). Pas sûr que vous voulez de ça dans votre poche, surtout pendant 50 ans... Et non, il n'y a pas de « mais si on divise la demie-vie par... », la physique n'est pas d'accord et la physique a toujours raison.

Pardon ? Quoi ? Ah ! Vous ne vouliez pas informer les gens, vous vouliez juste qu'ils cliquent sur « l'incroyable news » et voient vos pubs ? Oh, désolé. Toutes mes excuses pour le dérangement. Non, non, c'est ma faute ! je pensais qu'on faisait le même métier...

Denis Bodor

Hackable Magazine
est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND
STIMULANT DE MATIÈRE GRISE

BP 20142 - 67602 SELESTAT CEDEX - France
E-mail : lecteurs@hackable.fr -
Service commercial : cial@ed-diamond.com
Sites : hackable.fr - ed-diamond.com
Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor
Réalisation graphique : Kathrin Scali
Régie publicitaire : Tél. : 03 67 10 00 27
Service abonnement : Les Éditions Diamond
BP 20142 - 67602 SELESTAT CEDEX, France, Tél. : 03 67 10 00 20
Impression : Westermann Druck | PVA, Braunschweig, Allemagne
Distribution France :
(uniquement pour les dépositaires de presse)
MLP Réassort : Plate-forme de Saint-Barthélemy-d'Anjou.
Tél. : 02 41 27 53 12
Plate-forme de Saint-Quentin-Fallavier. Tél. : 04 74 82 63 04

Service des ventes : Abomarque - Tél. : 06 15 46 15 88
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution
N° ISSN : 2427-4631
CPPAP : K92470
Périodicité : bimestriel - Prix de vente : 14,90 €

La rédaction n'est pas responsable des textes, illustrations et photos qui lui sont communiqués par leurs auteurs. La reproduction totale ou partielle des articles publiés dans Hackable Magazine est interdite sans accord écrit de la société Les Éditions Diamond. Sauf accord particulier, les manuscrits, photos et dessins adressés à Hackable Magazine, publiés ou non, ne sont ni rendus, ni renvoyés. Les indications de prix et d'adresses figurant dans les pages rédactionnelles sont données à titre d'information, sans aucun but publicitaire. Toutes les marques citées dans ce numéro sont déposées par leur propriétaire respectif. Tous les logos représentés dans le magazine sont la propriété de leur ayant droit respectif.

Suivez-nous sur Twitter
@hackablemag

Alsace

SOMMAIRE

À PROPOS DE HACKABLE...

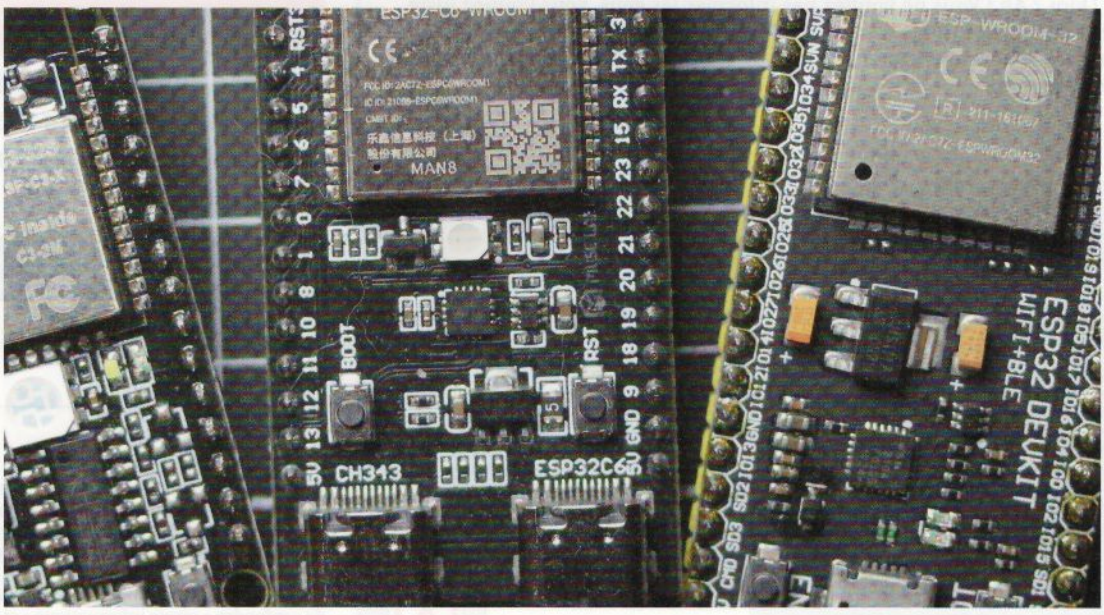
HACKS, HACKERS & HACKABLE

Ce magazine ne traite pas de piratage. Un **hack** est une solution rapide et bricolée pour régler un problème, tantôt élégante, tantôt brouillonne, mais systématiquement créative. Les personnes utilisant ce type de techniques sont appelées **hackers**, quel que soit le domaine technologique. C'est un abus de langage médiatisé que de confondre « pirate informatique » et « hacker ». Le nom de ce magazine a été choisi pour refléter cette notion de **bidouillage créatif** sur la base d'un terme utilisé dans sa définition légitime, véritable et historique.

ESP32 : CRÉER SES COMPOSANTS RÉUTILISABLES AVEC ESP-IDF

Denis BODOR

Vous connaissez certainement l'habituelle routine de développement. On part d'une page vierge et l'on expérimente jusqu'à obtenir un résultat fonctionnel, puis on affine, on nettoie, on sépare le code de test des routines « utilitaires » et l'on fait en sorte d'avoir, dans un ou plusieurs fichiers sources, quelque chose qu'on pourra réutiliser avec un autre projet. Mais, avec l'ESP-IDF, l'environnement de développement pour les ESP32, il est possible de pousser cela plus loin, et de façon plus rationnelle...



L'écosystème ESP32 ne cesse de s'étoffer de nouveaux microcontrôleurs, tantôt plus puissants, tantôt offrant des fonctionnalités très spécifiques (802.15.4 de l'ESP-H2, par exemple). Ces dernières années, nous avons même pu constater qu'une place toujours grandissante était faite à l'architecture RISC-V, en complément de l'historique Xtensa de Tensilica formant la base d'une bonne partie de la famille ESP32 jusqu'alors. C'est non seulement très agréable de voir qu'il existe effectivement une vie en dehors d'ARM, mais surtout de constater que tout ceci est supporté par un unique environnement, avec une approche très peu « usine à gaz », comme c'est le cas chez d'autres fondeurs de MCU et SoC.

Cet environnement, l'ESP-IDF, très facile à prendre en main pour un développeur coutumier de la ligne de commande et des outils « classiques » du monde Unix, s'installe très simplement sur tous les systèmes populaires, Windows, macOS et les distributions GNU/Linux, quelle que soit la plateforme (x86, amd64 ou ARM 32 ou 64 bits). Le *framework* et les bibliothèques accompagnant l'environnement ne sont, certes, pas aussi « accessibles » et pédagogiques qu'Arduino, mais tout à fait dans l'esprit de ce qu'on connaît, par exemple, avec le *Pico C SDK* de Raspberry Pi pour son microcontrôleur RP2040. Ceci à une petite nuance près, il intègre un mécanisme permettant de créer des éléments, détachés d'un projet particulier et aisément réutilisables : les composants.

L'ESP32-C3 est l'un des premiers microcontrôleurs Espressif utilisant un cœur RISC-V.

En séparant le code se chargeant du support d'un élément spécifique de votre projet lui-même, comme le pilotage d'un module ou encore la prise en charge d'un protocole particulier, et en en faisant un composant ESP-IDF, celui-ci deviendra indépendant, comme une bibliothèque Arduino, et pourra avoir sa vie propre. Mieux encore, il deviendra possible de le réutiliser automatiquement en spécifiant simplement la dépendance dans la configuration d'un projet et, bien sûr, sa distribution, qu'elle soit publique ou privée, deviendra excessivement simple grâce à Git.

Pour nous familiariser avec cette fonctionnalité, suivons simplement l'évolution d'un projet plus ou moins standard…

1. UN CODE SIMPLE

Notre projet du moment qui servira de base aux expérimentations sera très simple. La plateforme utilisée importe peu, du moment qu'il s'agit effectivement d'une carte à base d'ESP32. ESP32, ESP32-S2, ESP32-C3, etc., feront tous parfaitement l'affaire, à partir du moment où nous disposons de suffisamment de GPIO pour y connecter des LED. Cinq d'entre elles seront utilisées pour créer une petite animation sans prétention et le code se chargeant de cela sera précisément celui que nous allons « modulariser » et transformer en composant ESP-IDF.

La création d'un nouveau projet peut être faite directement via la commande `idf.py` en utilisant l'option `create-project` suivie d'un nom. Automatiquement, un répertoire de ce nom sera créé, contenant le strict minimum :

```
$ idf.py create-project ledtest
Executing action: create-project
The project was created in /home/denis/SRC/C/ESP32/ledtest

$ cd ledtest/
$ tree
.
├── CMakeLists.txt
└── main
    ├── CMakeLists.txt
    └── ledtest.c
2 directories, 3 files
```

Généralement, je ne garde pas les choses en l'état, préférant un `main.c` en guise de fichier source C principal. Si vous êtes du même avis que moi, renommez simplement `ledtest.c` et ajustez le contenu du fichier `main/CMakeLists.txt` :

```
idf_component_register(
        SRCS "main.c"
        INCLUDE_DIRS "."
)
```

La phase suivante consiste à définir la plateforme visée avec :

```
$ idf.py --list-targets
esp32
esp32s2
esp32c3
esp32s3
esp32c2
esp32c6
esp32h2

$ idf.py set-target esp32
[...]
-- Configuring done
-- Generating done
-- Build files have been written to:
/home/denis/SRC/C/ESP32/ledtest/build
```

Nous pouvons, à ce stade, provoquer une construction, mais le code source C, qui se résume à ceci, n'est pas très intéressant :

```
#include <stdio.h>

void app_main(void)
{

}
```

Complétons donc plutôt ce code avec ce que nous avons initialement prévu :

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"

#define MSDELAY  75

void app_main(void)
{
    int i;

    uint8_t leds[5] = { 12, 14, 27, 26, 25 };

    for (i = 0; i < 5; i++) {
            gpio_reset_pin(leds[i]);
            gpio_set_direction(leds[i], GPIO_MODE_OUTPUT);
            gpio_set_level(leds[i], 0);
    }

    while(1) {
        for (i = 0; i < 5; i++) {
            gpio_set_level(leds[i], 1);
            vTaskDelay(MSDELAY / portTICK_PERIOD_MS);
        }
        for (i = 0; i < 5; i++) {
            gpio_set_level(leds[i], 0);
            vTaskDelay(MSDELAY / portTICK_PERIOD_MS);
        }
    }
}
```

Rien de bien extraordinaire ici, puisque nous nous contentons de configurer les cinq GPIO en sortie avant d'en changer deux fois l'état dans une boucle sans fin, le tout, directement dans `main()`. Je pense qu'il n'y a pas moins modulaire à ce stade, sauf à utiliser chaque GPIO individuellement et non via un tableau de `uint8_t`.

Nous pouvons à présent construire le projet, avec `idf.py build` (ou `idf.py all`) ou, plus efficacement, construire, flasher et suivre l'exécution en une seule commande :

```
$ idf.py -p /dev/ttyUSB1 flash monitor
[...]
Executing action: flash
[...]
Executing "make -j 18 flash"...
[  0%] Built target custom_bundle
[  0%] Generating memory.ld linker script...
[...]
[  4%] Generating project_elf_src_esp32.c
[  4%] Built target _project_elf_src
[  5%] Linking C static library liblog.a
[  5%] Built target __idf_log
[...]
[100%] Linking CXX executable ledtest.elf
[100%] Built target ledtest.elf
[100%] Generating binary image from built executable
[...]
esptool.py v4.7.dev3
Serial port /dev/ttyUSB1
Connecting....
Chip is ESP32-D0WDQ6 (revision v1.0)
Features: WiFi, BT, Dual Core, Coding Scheme None
Crystal is 40MHz
MAC: 30:ae:a4:0b:9e:8c
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 460800
Changed.
Configuring flash size...
Flash will be erased from 0x00001000 to 0x00007fff...
Flash will be erased from 0x00010000 to 0x0003cfff...
Flash will be erased from 0x00008000 to 0x00008fff...
Compressed 26640 bytes to 16684...
Writing at 0x00001000... (50 %)
[...]
[100%] Built target flash
Executing action: monitor
[...]
```

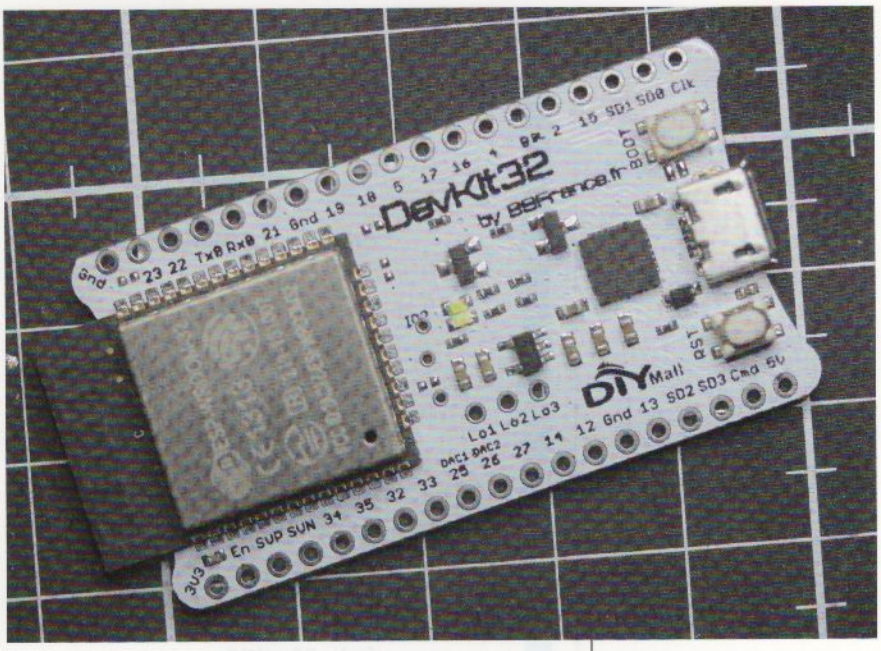
S'en suit donc une avalanche de messages sur la sortie standard, résumant toutes les étapes nécessaires à la programmation de la flash, jusqu'à arriver au « moniteur » série affichant, quant à lui, les éléments provenant de la cible elle-même. Comme nous n'affichons rien depuis le code, sur la sortie série de l'ESP32, la sortie s'arrêtera là. Pour quitter le moniteur, utilisez le raccourci Ctrl+] (Ctrl + AltGr + «) » sur un clavier AZERTY). Bien entendu, ce moniteur écrit en

Python (`idf_monitor.py`) n'est pas la seule option possible pour interagir avec la cible, mais ceci est, à mon sens, plus pratique que d'invoquer manuellement Minicom ou GNU Screen au bon moment.

2. UTILISER LES ÉLÉMENTS DE CONFIGURATION

La première chose à faire pour rendre notre code portable d'une carte à une autre consiste à ne plus désigner explicitement les GPIO dans le code. Bien sûr, nous pourrions définir quelques macros pour simplifier les choses, voire faire cela dans un `main.h`, mais ceci ne réglerait pas réellement le problème : il faudrait toujours éditer le code pour ajuster les valeurs utilisées.

Une autre option est de plutôt reposer sur l'environnement lui-même. En effet, nous pouvons passer cette tâche à la gestion des options d'ESP-IDF et donc à l'interface de configuration qu'il est possible d'invoquer en utilisant `idf.py menuconfig`. Là, il vous est permis d'ajuster énormément de choses, allant des options réseau aux arguments de l'outil de flashage en passant par la configuration du *bootloader*, l'activation de la signature du *firmware*, la répartition de la flash (code, data, OTA) ou encore l'affinement des options de compilation. C'est dans ***Component config*** qu'on trouvera le plus d'options, puisque c'est là que sont configurés chaque composant logiciel et support pour les fonctionnalités matérielles (PSRAM, flash SPI, DAC, ADC, I2S, etc.) et logicielles (LWIP, IPv6, TLS, systèmes de fichiers, etc.). Les choix faits dans cette interface sont ensuite enregistrés dans un fichier `sdkconfig` à la racine du projet et celui-ci est utilisé dans la phase de construction.



Ce devkit ESP32 est longtemps resté un de mes préférés en raison du nombre de GPIO disponibles et de signaux facilement accessibles.

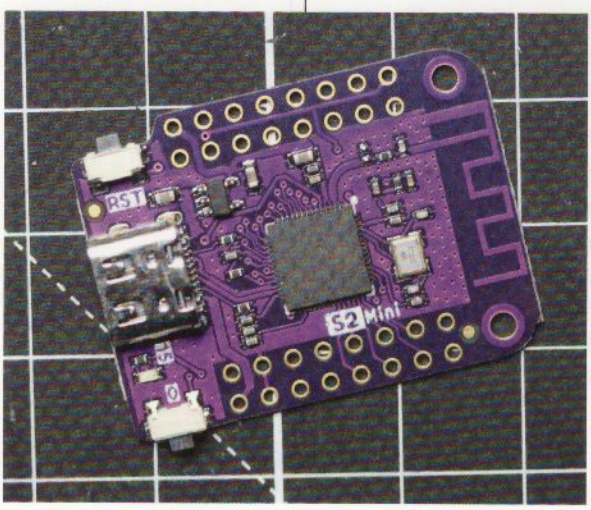
Nous pouvons reposer sur ce mécanisme pour gérer des options pour notre propre code. Pour cela, il nous suffit de créer un fichier `Kconfig.projbuild` dans `main/` contenant :

```
menu "LEDs Demo"
        config LEDSDEMO_LED0
        int "led0"
        default 12
        help
                GPIO for led0
```

```
config LEDSDEMO_LED1
int "led1"
default 14
help
        GPIO for led1

config LEDSDEMO_LED2
int "led2"
default 27
help
        GPIO for led2

config LEDSDEMO_LED3
int "led3"
default 26
help
        GPIO for led3

config LEDSDEMO_LED4
int "led4"
default 25
help
        GPIO for led4

endmenu
```

L'architecture générale du fichier parle d'elle-même puisque nous retrouvons ainsi un nouveau menu `"LEDs Demo"`, supportant cinq options de configuration (`config`), disposant chacune d'un type (ici, entier), d'une valeur par défaut et d'un petit texte d'aide. Les noms des options, comme `LEDSDEMO_LED0` par exemple, seront préfixés de `CONFIG_` et stockés dans le fichier `sdkconfig`. Mais aussi, et surtout, constitueront automatiquement des macros qui pourront être utilisées dans le code.

Peu pratique à utiliser sur platine à essai, ce module ESP32-S2 présente l'avantage d'avoir un connecteur USB-C, ce qui est encore relativement rare sur les modules les moins coûteux. Notez également l'absence de convertisseur USB/série.



Ainsi, notre tableau `leds[]` devient maintenant :

```
uint8_t leds[5] = {
    CONFIG_LEDSDEMO_LED0,
    CONFIG_LEDSDEMO_LED1,
    CONFIG_LEDSDEMO_LED2,
    CONFIG_LEDSDEMO_LED3,
    CONFIG_LEDSDEMO_LED4
};
```

Et la modification s'arrête là. Nous n'avons rien d'autre à faire si ce n'est reconstruire et flasher le *firmware* comme précédemment.

3. FAIRE DU CODE UN COMPOSANT

Ce petit détour par la configuration via `menuconfig` n'était pas tout à fait innocent, car il nous a permis de voir que bon nombre de fonctionnalités disponibles pour la plateforme sont en réalité déjà des composants (ou *components* en anglais et dans la documentation Espressif). Votre code lui-même, placé dans `main/`, est d'ailleurs également un composant et est traité comme tel par l'environnement de développement. Ce que nous voulons à présent, c'est extraire de notre `main.c` tout ce qui est relatif au pilotage des LED et ne conserver que la partie concernant la boucle principale.

Nous allons donc créer deux fonctions, une pour initialiser les GPIO (`ledanim_init()`) et une pour activer les LED (`ledanim_do()`) selon un motif que nous utilisons déjà et prenant en argument le délai utilisé après chaque activation/désactivation. Mais avant cela, demandons à l'IDF de créer un composant pour nous, comme nous l'avons fait pour le projet lui-même :

```
$ mkdir components
$ cd components/
$ idf.py create-component ledanim
Executing action: create-component
The component was created in
/home/denis/SRC/C/ESP32/ledtest/components/ledanim
```

Notre arborescence ressemble maintenant à ceci :

```
$ tree -I build/ -I *.old
.
├── CMakeLists.txt
├── components
│   └── ledanim
│       ├── CMakeLists.txt
│       ├── include
│       │   └── ledanim.h
│       └── ledanim.c
├── main
│   ├── CMakeLists.txt
│   └── main.c
└── sdkconfig
5 directories, 7 files
```

Comme précédemment, le contenu de `components/ledanim` est généré pour nous et consiste en un minimum de code :

```
#include <stdio.h>
#include "ledanim.h"
void func(void)
{
}
```

Nous n'avons qu'à compléter cela en répartissant le contenu de notre `main.c` avec, dans `ledanim.c` :

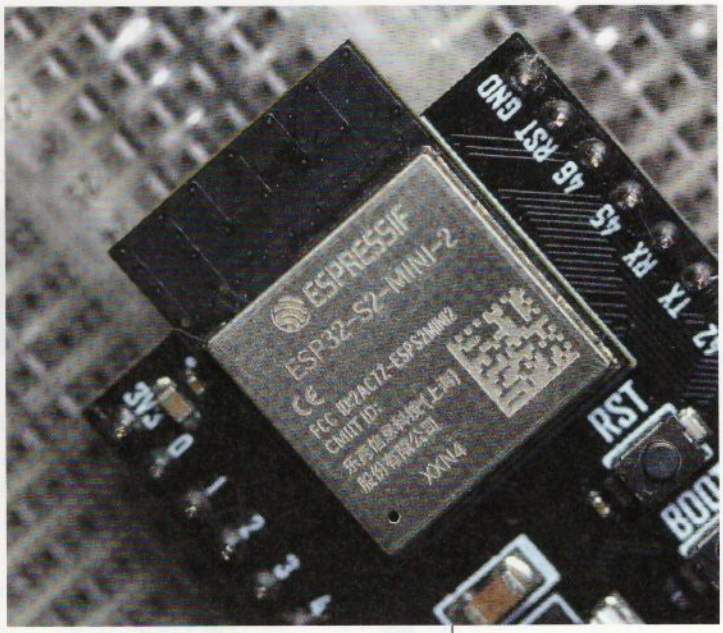
```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "driver/gpio.h"
#include "ledanim.h"

uint8_t leds[5] = {
    CONFIG_LEDANIM_LED0,
    CONFIG_LEDANIM_LED1,
    CONFIG_LEDANIM_LED2,
    CONFIG_LEDANIM_LED3,
    CONFIG_LEDANIM_LED4
};

void ledanim_init(void)
{
    int i;

    for (i = 0; i < 5; i++) {
        gpio_reset_pin(leds[i]);
        gpio_set_direction(leds[i], GPIO_MODE_OUTPUT);
        gpio_set_level(leds[i], 0);
    }
}

void ledanim_do(TickType_t delay)
{
    int i;

    for (i = 0; i < 5; i++) {
        gpio_set_level(leds[i], 1);
        vTaskDelay(delay / portTICK_PERIOD_MS);
    }
    for (i = 0; i < 5; i++) {
        gpio_set_level(leds[i], 0);
        vTaskDelay(delay / portTICK_PERIOD_MS);
    }
}
```

Et dans `include/ledanim.h` :

```
#ifndef LEDANIM_H
#define LEDANIM_H

void ledanim_init(void);
void ledanim_do(TickType_t delay);

#endif /* LEDANIM_H */
```

Remarquez que les macros utilisées pour les GPIO ont changé de noms et pour cause, nous n'avons plus ici une configuration pour le projet mais pour le composant. Il convient donc d'adapter la configuration en déplaçant/renommant le fichier `main/Kconfig.projbuild` en `components/ledanim/Kconfig` et en modifiant les mots-clés derrière chaque `config` pour les faire correspondre aux noms des macros choisies (sans `CONFIG_` bien sûr).

Nous en profitons pour créer un nouveau `main/Kconfig.projbuild` pour gérer un paramètre supplémentaire, le temps d'attente après chaque opération sur les LED, passé en argument de `ledanim_do()` :



Ce module ESP32-S2-MINI-2 est celui d'un devkit très récent. Très compact, il se caractérise, entre autres, par le format de boîtier utilisé : VFQFN. Ceci rendra relativement difficile une éventuelle permutation de modules d'un devkit à l'autre, sauf à disposer de l'équipement de dé-soudage/soudage adéquat.

```
menu "LEDs Animation Demo"
        config LEDSDEMO_MSDELAY
        int "msDelay"
        default 75
        help
                Delay in milliseconds for leds animation
endmenu
```

Et, enfin, nous ajustons notre `main.c`, débarrassé du code supporté par ailleurs et faisant usage du paramètre de configuration supplémentaire :

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "ledanim.h"

void app_main(void)
{
    printf("IDF Version: %s\n", IDF_VER);

    ledanim_init();

    while(1) {
        ledanim_do(CONFIG_LEDSDEMO_MSDELAY);
    }
}
```

Ceci fait, un petit coup de `idf.py menuconfig` nous permettra de constater qu'une entrée ***LEDs Animation Demo*** est bien présente et permet de régler ***msDelay***, mais nous trouvons également une entrée ***LED Animation*** dans ***Component config***, où nous pouvons ajuster les GPIO à utiliser.

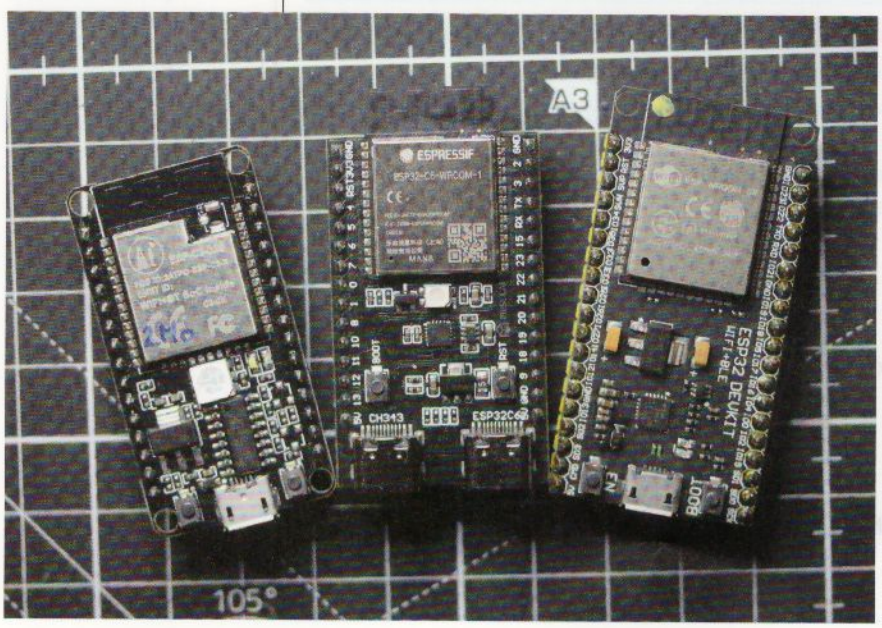
Si vous tentez une construction à ce stade, l'opération échouera avec le message :

```
Compilation failed because ledanim.c (in "ledanim" component)
includes driver/gpio.h, provided by driver component(s).
However, driver component(s) is not in the requirements list of "ledanim".
```

Notre composant utilise, en effet, les GPIO et nous devons spécifier cette dépendance explicitement, ce qui n'est pas le cas pour `main/`, le composant principal. Pour ce faire, nous n'avons qu'à ajouter une simple ligne dans `components/ledanim/CMakeLists.txt`, qui devient :

```
idf_component_register(
        SRCS "ledanim.c"
        INCLUDE_DIRS "include"
        REQUIRES driver
)
```

Les ESP32 arrivent sous toutes les formes et toutes les déclinaisons. Ici, nous avons des devkits, de gauche à droite, équipés d'ESP32-C3 (RISC-V), d'ESP32-C6 (RISC-V) et d'ESP32 (Xtensa LX6), tous supportés de manière égale par l'ESP-IDF.



Les composants en question sont généralement ceux mis à disposition par l'environnement Espressif lui-même. Notez qu'un certain nombre de dépendances communes à tous les composants n'ont pas besoin d'être explicitement référencées, car ils le sont automatiquement : `cxx`, `newlib`, `freertos`, `esp_hw_support`, `heap`, `log`, `soc`, `hal`, `esp_rom`, `esp_common`, `esp_system`, `xtensa`/`riscv`. Pour savoir ce que vous devez spécifier avec `REQUIRES`, c'est très simple : il s'agit du nom du répertoire dans lequel se trouve le fichier d'en-tête que vous incluez dans le code. Ici, `driver/gpio.h` et donc `driver`. `freertos/task.h` ne nécessite pas de mention puisque `freertos` est une dépendance implicite.

Cette modification apportée, la construction, la programmation et l'exécution se passent sans problème. Mais nous n'en avons pas fini...

4. AJOUTER LE NÉCESSAIRE POUR LA DISTRIBUTION

À la bonne heure, nous disposons maintenant d'un composant que nous pouvons copier dans n'importe quel projet et qui dispose, de plus, de sa propre configuration indépendante de celle du projet lui-même. Mieux encore, nous pouvons utiliser Git pour gérer les révisions du code et, si nous spécifions `components/` dans le `.gitignore` du projet, nous pouvons même faire de `components/ledanim` un dépôt Git indépendant.

Mais ce qui serait encore plus agréable, c'est d'éviter de copier le composant nous-mêmes d'un projet à l'autre. Ne serait-ce pas fantastique de n'avoir qu'à simplement spécifier le nom du composant quelque part et que celui-ci soit automatiquement cloné d'un dépôt Git distant avant la construction (ou la configuration avec `menuconfig`) ?

Ça tombe bien, parce que l'ESP-IDF permet précisément de faire cela. Il suffit de rendre notre composant réellement distribuable en lui ajoutant un manifeste qui en décrit les caractéristiques. Pour cela, une fois n'est pas coutume, nous faisons appel à `idf.py` :

```
$ idf.py create-manifest --path=components/ledanim
Executing action: create-manifest
Created "/home/denis/SRC/C/ESP32/ledtest/
  components/ledanim/idf_component.yml"
```

Le fichier créé à l'emplacement spécifié est, encore une fois, une base qu'il conviendra d'adapter à ses besoins :

```
## IDF Component Manager Manifest File
dependencies:
  ## Required IDF version
  idf:
    version: ">=4.1.0"
  # # Put list of dependencies here
  # # For components maintained by Espressif:
  # component: "~1.0.0"
  # # For 3rd party components:
  # username/component: ">=1.0.0,<2.0.0"
  # username2/component2:
  #   version: "~1.0.0"
  #   # For transient dependencies `public` flag can be set.
  #   # `public` flag doesn't have an effect dependencies
  #   #  of the `main` component.
  #   # All dependencies of `main` are public by default.
  #   public: true
```

Dans notre cas, ceci deviendra donc :

```
dependencies:
  idf:
    version: ">=4.1.0"
```

```
description: Simple animation with 5 leds
url: https://gitlab.com/0xDRRB
version: 0.0.1
license: BSD-2-Clause
```

La documentation officielle **[1]** décrit parfaitement l'usage de chaque directive et nous pourrions, par exemple, restreindre l'usage de ce composant à une cible donnée (`targets`) ou encore l'agrémenter de tags (`tags`) en décrivant les fonctionnalités. Notez que `license` utilise un identifiant tiré de la liste fournie par le site *Software Package Data Exchange* **[2]** et que les choix sont nombreux (même si `BSD-2-Clause` ou éventuellement `GPL-2.0-only` devraient être suffisants).

Nous pouvons maintenant sereinement donner à notre composant un dépôt Git distant, qu'il soit public (sur GitLab, par exemple) ou privé (sur une machine accessible en SSH). Ceci fait, nous n'avons plus besoin de `components/` au sein de notre projet et nous pouvons maintenir ce code séparément. Le projet lui-même devra référencer cette dépendance avec un fichier placé dans le répertoire de notre composant principal (`main/`). C'est `main/idf_component.yml` :

```
dependencies:
  ledanim:
    git: ssh://serveur.tld/chemin/ledanim.git
```

Par défaut, la racine du dépôt Git sera utilisée, mais vous pouvez également choisir de stocker tous vos composants au même endroit et préciser quel répertoire devra être utilisé, dans ce dépôt, avec `path:`, ou encore explicitement désigner une branche, un tag ou un *hash* de *commit* avec `version:`. Cet ajout effectué, notre arborescence de projet ressemble à présent à ceci :

```
$ tree
.
├── CMakeLists.txt
├── main
│   ├── CMakeLists.txt
│   ├── idf_component.yml
│   ├── Kconfig.projbuild
│   └── main.c
└── sdkconfig
```

Et notre `.gitignore` contiendra :

```
*~
*.old
build/
dependencies.lock
managed_components/
```

En utilisant `idf.py` dans ce projet tout propre (`fullclean`) et tout neuf, avec `menuconfig` par exemple, nous pourrons effectivement voir que la dépendance est traitée comme il se doit :

```
[...]
Processing 2 dependencies:
[1/2] idf (5.1.2)
[2/2] ledanim (aedb610283f08dfbfa6620dc8bb2a5a15e6224f1)
[...]
```

Le dépôt sera cloné et son contenu placé dans un sous-répertoire `managed_components/`. Un fichier `dependencies.lock` sera également créé pour garder une trace des dépendances du projet et il vous sera possible, à tout moment, de mettre à jour les composants avec `idf.py update-dependencies`. Notez toutefois que cette vérification est exécutée automatiquement par le gestionnaire de composants à chaque construction ou reconfiguration (`idf.py reconfigure`).

Enfin, il peut être également intéressant de savoir qu'il est possible de désactiver totalement le gestionnaire de composants en définissant la variable d'environnement `IDF_COMPONENT_MANAGER` à `0`.

CONCLUSION

J'aime beaucoup l'ESP-IDF, c'est propre, reposant sur des outils fiables, *open source* et connus, et la logique utilisée est cohérente et bien structurée. C'est tellement agréable, à mon sens, qu'après avoir longuement travaillé avec l'environnement Espressif, utiliser quelque chose comme STM32CubeMX pour un autre projet, et surtout manipuler le code ainsi généré (quand GNU Make est supporté) me donne envie de pleurer. Je ne plaisante qu'à moitié, c'est une véritable souffrance de voir (et subir) une telle différence entre les deux environnements de développement.

Bien sûr, il s'agit presque de deux mondes distincts, mais je pense qu'ESP-IDF pourrait être une excellente source d'inspiration pour STM (et d'autres). Le SDK Raspberry Pi Pico, quant à lui, est assez proche de cette philosophie moins orientée « *hors de l'IDE alambiqué, point de salut* » et ce genre de choses peut grandement influencer le choix d'une plateforme. Certes, dans un contexte industriel, la souplesse d'utilisation d'un environnement n'est, de loin, pas la priorité puisque des considérations économiques et tantôt géopolitiques entrent en jeu. En effet, Espressif Systems est un acteur chinois majeur dans le domaine des nouvelles technologies, ce qui peut être un problème, en particulier au regard de la situation vis-à-vis de Taiwan que seule la Chine considère comme une province et non un pays démocratique à part entière. Il n'en reste pas moins que, d'un point de vue purement technique (et ça tombe bien, parce que c'est tout ce qui nous intéresse ici), Espressif a vraiment de quoi séduire... **DB**

RÉFÉRENCES

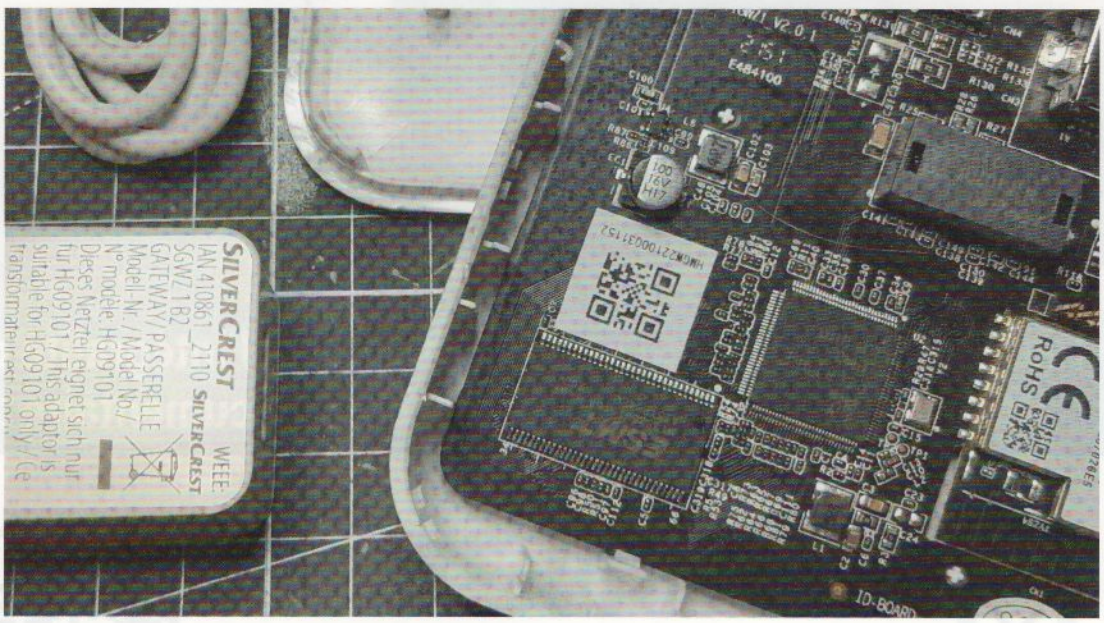
[1] **https://docs.espressif.com/projects/idf-component-manager/en/latest/reference/manifest_file.html**

[2] **https://spdx.org/licenses/**

JOUONS AVEC UNE PASSERELLE ZIGBEE LIDL

Denis BODOR

J'aime bien LIDL, on y trouve des produits qu'on n'a pas ailleurs. De délicieuses olives farcies aux amandes, des pistaches décortiquées non salées, de la litière agglomérante pas chère, parfois des Guinness West Indies Porter... et des passerelles domotiques Ethernet/Zigbee à 20 €. Ai-je besoin d'une telle passerelle ? Non. Je n'ai pour l'instant aucun périphérique Zigbee dans mon installation domotique. Ce que j'ai, en revanche, c'est de la curiosité. Celle de savoir ce que renferme ce produit et éventuellement, d'apprendre comment je pourrais en faire un autre usage. C'est parti pour un week-end de chasse au trésor...



– Jouons avec une passerelle ZigBee LIDL –

Vingt euros pour 24 ou 48 heures d'amusement, d'exploration, de découverte, d'apprentissage, de frustration, et avec un gros *shot* de dopamine ou deux à la fin, ce n'est pas cher payé finalement. Certaines personnes paient davantage, à l'heure, pour des distractions avec une déception presque toujours garantie (*tousse* Marvel *tousse*). L'objectif ici est de « se faire les dents » sur un matériel récent, basé certainement sur un SoC exotique et implémentant peut-être des mécanismes et/ou des outils non déjà connus. Et quoi de mieux qu'un produit qui affiche clairement et un peu trop fièrement qu'il est compatible avec les classiques geôliers du *cloud* que sont « Hey Google » et « Apple HomeKit », alors que domotique et *cloud* sont, pour moi, deux choses qui ne vont pas ensemble ! La démarche étant avant tout autoéducative, je n'ai pas pris la peine de chercher sur le Net s'il existait un *firmware* alternatif, du moins pas avant d'avoir obtenu une victoire ou deux.

1. PRISE EN MAIN ET ÉTAT DES LIEUX

L'objet, désigné sous le modèle HG09101 (version 07/2022), se présente sous la forme d'un carré blanc en plastique de 9 cm de côté et de 2 cm d'épaisseur, disposant d'un connecteur RJ45 pour le réseau filaire (Ethernet), de deux LED intégrées, d'un bouton *reset* accessible via un petit accessoire métallique (fourni) et d'un connecteur USB-C libellé « 5V 1A ». Le produit est livré avec son bloc d'alimentation 5V/1A avec un port USB-A femelle, un câble USB-A vers USB-C, un câble Ethernet plat, deux vis et chevilles, et le petit accessoire métallique faisant office de trombone pour le *reset*.

La passerelle ZigBee LIDL/SilverCrest est d'un aspect relativement sobre et est peu encombrante. Ceci peut être un ajout très intéressant à une installation domotique existante, à condition bien sûr de la libérer du cloud, pour que votre lieu de vie reste aussi privé qu'il mérite de l'être...

Je n'ai pas même mis le matériel sous tension une première fois et suis directement passé au démontage, qui n'a pas été simple. Le boîtier est clipsé et non collé, mais demande de l'insistance. Le but est idéalement d'utiliser la plateforme pour une application maison et de préférence qui ne ressemblera pas à une boîte de sardines ouverte par un homme de Néandertal. Dedans, un unique PCB nous donnant déjà un certain nombre d'indications intéressantes : un SoC Realtek 8196E à cœur MIPS à 400 MHz **[1]**, 32 Mio de SDRAM DDR ESMT M13S2561616A **[2]**, une flash SPI 16 Mio GigaDevice GD25Q128E **[3]** et un module Zigbee IEEE 802.15.4.

Différents points de test sont présents ainsi qu'un emplacement pour 6 connecteurs au pas de 2,54 mm. Si l'on prend la peine de sortir la carte du boîtier, simplement retenue par deux ergots en plastique, on a l'agréable surprise de constater qu'une série de 4 points de test sont placés à proximité et sont marqués respectivement « RX », « TX », « GND » et « 3V3 ». Nous n'avons alors qu'à souder un connecteur, suivre les pistes entre les points de test et les broches, et nous y connecter avec un adaptateur USB/série 3,3 volts (RX, TX et GND). Après plusieurs tentatives avec Minicom ou GNU Screen, en variant la vitesse de communication, on arrive à obtenir une console en 38400 b/s 8N1 :

```
Booting...
DDR1:32MB
---RealTek(RTL8196E)at 2021.07.29-21:33+0800 v3.4T-pre2 [16bit](400MHz)
P0phymode=01, embedded phy
check_image_header  return_addr:05010000 bank_offset:00000000
no sys signature at 00010000!
P0phymode=01, embedded phy
---Ethernet init Okay!
tuya:start receive production test frame ...
Jump to image start=0x80c00000...
decompressing kernel:
Uncompressing Linux... done, booting the kernel.
done decompressing kernel.
start address: 0x80003780
Linux version 3.10.90 (zhangpc@embed) (gcc version 4.6.4
  (Realtek RSDK-4.6.4 Build 2080) ) #1 Thu Jul 29 21:36:28 CST 2021
CPU revision is: 0000cd01
[...]
SPI flash(GD25Q128) was found at CS0, size 0x1000000
boot+cfg offset=0x0 size=0x20000 erasesize=0x10000
linux offset=0x20000 size=0x1e0000 erasesize=0x10000
rootfs offset=0x200000 size=0x200000 erasesize=0x10000
tuya-label offset=0x400000 size=0x20000 erasesize=0x10000
jffs2-fs offset=0x420000 size=0xbe0000 erasesize=0x10000
5 rtkxxpart partitions found on MTD device flash_bank_1
Creating 5 MTD partitions on "flash_bank_1":
0x000000000000-0x000000020000 : "boot+cfg"
0x000000020000-0x000000200000 : "linux"
0x000000200000-0x000000400000 : "rootfs"
0x000000400000-0x000000420000 : "tuya-label"
0x000000420000-0x000001000000 : "jffs2-fs"
PPP generic driver version 2.4.2
[...]
Sending discover...
Sending discover...
Sending discover...
nameserver 114.114.114.114
Sending discover...

tuya-linux login:
```

Un *login*, voilà qui est rassurant. Actuellement, de moins en moins de périphériques de ce type, chinois ou non, fournissent un shell sans autre forme de procès. Les temps changent. Forcément, nous tentons le classique `root`/`root` et quelques autres variations connues, mais :

```
tuya-linux login: root
Password:
Login incorrect
```

Nous n'avons pas d'accès facile, mais les messages de *boot* nous apprennent déjà un certain nombre de choses : il s'agit bien d'un système Linux, le *bootloader* n'est clairement pas U-Boot, la solution semble construite sur la plateforme Tuya **[4]** et la structure du stockage repose d'une part sur SquashFS (lecture seule) et d'autre part sur JFFS2 (peut-être un OverlayFS ?). Nous voyons également quelques mentions de BusyBox et le démarrage de plusieurs scripts shell chargés de la partie « applicative » du produit.

À ce stade, deux approches sont possibles, soit trouver un moyen d'obtenir un shell de manière logicielle, ce qui revient techniquement à trouver une faille dans le système, soit tout simplement accéder plus brutalement au système de fichiers en lisant directement la flash SPI GD25Q128E. Le fait que le composant soit en boîtier SOP8 (SOIC8), et non quelque chose de plus pénible comme du QFN ou du BGA est un avantage pour nous, car les outils et techniques utilisables sont bien plus accessibles.

2. ACCÈS À LA FLASH

Parmi les accessoires indispensables pour ce type d'expérimentation et en dehors des classiques convertisseurs USB/série, analyseurs logiques pas chers et multimètres, nous trouvons le programmeur CH341A. Initialement destiné à la manipulation des BIOS de PC, cet outil USB permet de lire et d'écrire une vaste gamme de mémoires flash SPI ou i²c, en utilisant par exemple l'utilitaire `flashrom` sous GNU/Linux. Le CH341A se trouve très facilement en ligne et une version étoffée d'accessoires complémentaires (adaptateur SOIC8/DIP8, convertisseur 1,8V et pince SOIC8) vous coûtera aux alentours de 20 €.

L'approche la moins intrusive (ou destructive), et la première à essayer avec ce type d'accessoire consiste à utiliser la pince permettant de venir se connecter au composant, sans avoir à le retirer du circuit. Il arrive que cette technique fonctionne, mais généralement, le composant *in situ* ayant nombre de ses lignes connectées à d'autres, même hors tension, ne répond souvent pas correctement. C'est le cas ici où l'utilitaire `flashrom` détecte bien une flash SPI, mais n'arrive pas à l'identifier :

```
$ flashrom --programmer ch341a_spi
flashrom unknown on Linux 6.1.0-10-amd64 (x86_64)
[...]
Found Generic flash chip "unknown SPI chip (RDID)"
  (0 kB, SPI) on ch341a_spi.
[...]
```

Je m'attendais très honnêtement à trouver quelque chose comme un ESP32 (intégrant une MAC) ou éventuellement un MCU STM32 dans le produit. Mais non, il s'agit d'un SoC Realtek avec un cœur MIPS, le RTL8196E.

Dans le doute et avant de passer à des mesures plus drastiques, on prendra cependant le temps de tester une version plus récente en récupérant les sources GitHub **[5]** et en les compilant (un simple `make` suffit). Mais :

```
$ ./flashrom --programmer ch341a_spi
flashrom 1.4.0-devel (git:v1.2-1391-ga21be915)
   on Linux 6.1.0-10-amd64 (x86_64)
[...]
Found Generic flash chip "unknown SPI chip (RDID)"
  (0 kB, SPI) on ch341a_spi.
```

Même motif, même punition, le `flashrom` 1.4.0 de développement réagit exactement comme celui de Debian 12.1 (1.3.0). Mais en listant les composants supportés (`flashrom -L`) par chacune des deux versions, on constate qu'effectivement la flash SPI GD25Q128E n'est prise en charge que par la version la plus récente (`flashrom -L | grep -i GD25Q128E`). Nous restons donc sur cette version, mais devons finalement retirer la flash du circuit. Dessouder un composant SOIC8 est possible avec un fer, de la tresse et beaucoup de flux, à condition d'être excessivement délicat, mais la solution la plus aisée reste la station à air chaud, en protégeant les composants à proximité et en faisant très attention à nos mouvements avec la pince brucelle (en particulier vis-à-vis des deux minuscules résistances toutes proches). Ceci fait, on nettoiera l'emplacement à la tresse et on placera le composant récupéré dans l'adaptateur SOIC8/DIP8 livré avec le CH341A.

Et cette fois, `flashrom` détecte effectivement le type de flash :

```
$ ./flashrom --programmer ch341a_spi
[...]
Found GigaDevice flash chip "GD25B128B/GD25Q128B" (16384 kB, SPI) on ch341a_spi.
Found GigaDevice flash chip "GD25Q127C/GD25Q128E" (16384 kB, SPI) on ch341a_spi.
Found GigaDevice flash chip "GD25Q128C" (16384 kB, SPI) on ch341a_spi.
Multiple flash chip definitions match the detected chip(s):
  "GD25B128B/GD25Q128B", "GD25Q127C/GD25Q128E", "GD25Q128C"
Please specify which chip definition to use with the -c <chipname> option.
```

Comme le précise la sortie, plusieurs modèles de flash correspondent et nous devons spécifier celui qui nous intéresse pour lire l'ensemble de la mémoire et stocker le contenu dans un fichier `flash.bin` :

```
$ ./flashrom --programmer ch341a_spi -c GD25Q127C/GD25Q128E \
--progress -r ~/HACK/LIDLrouteur/flash.bin
[...]
Found GigaDevice flash chip "GD25Q127C/GD25Q128E"
   (16384 kB, SPI) on ch341a_spi.
Reading flash... [READ] 0% complete... [READ] 1% complete...
[READ] 2% complete... [READ] 3% complete...
[...]
[READ] 99% complete... [READ] 100% complete... done.
```

L'opération est relativement lente, et sans l'option `--progress`, on pourrait presque penser qu'il y a un problème. Mais non, au final, nous obtenons effectivement un fichier de 16 Mio que nous prenons soin de mettre en sécurité, en cas de problème ou d'erreur de manipulation.

3. EXTRACTION DES DONNÉES

À présent que nous avons une image du contenu de la flash à disposition, nous pouvons en extraire les informations afin de savoir exactement comment est structuré le système. Nous avons déjà une idée de l'organisation de la mémoire, puisque ceci est directement affiché lors du démarrage :

```
[...]
Creating 5 MTD partitions on "flash_bank_1":
0x000000000000-0x000000020000 : "boot+cfg"
0x000000020000-0x000000200000 : "linux"
0x000000200000-0x000000400000 : "rootfs"
0x000000400000-0x000000420000 : "tuya-label"
0x000000420000-0x000001000000 : "jffs2-fs"
[...]
```

Les deux parties qui nous intéressent ici sont `rootfs`, la partition racine en lecture seule et `jffs2-fs`, un système de fichiers en lecture/écriture contenant sans doute des données et/ou l'application chargée de la connexion avec les services du *cloud* domotique Google et Apple. Pour extraire cela, nous faisons appel à l'incontournable `binwalk` **[6]** et commençons par nous renseigner sur ce qui est détecté dans l'image :

```
$ binwalk flash.bin

DECIMAL   HEXADECIMAL   DESCRIPTION
--------------------------------------------------------------------
4800      0x12C0        LZMA compressed data, properties:
                          0x5D, dictionary size: 8388608 bytes,
                          uncompressed size: 77920 bytes
141336    0x22818       LZMA compressed data, properties:
                          0x5D, dictionary size: 8388608 bytes,
                          uncompressed size: 3555200 bytes
2097152   0x200000      Squashfs filesystem, little endian,
                          version 4.0, compression:xz,
                          size: 918722 bytes, 167 inodes,
                          blocksize: 131072 bytes,
                          created: 2038-01-29 00:53:20
4325376   0x420000      JFFS2 filesystem, big endian
7155132   0x6D2DBC      Zlib compressed data, compressed
7155276   0x6D2E4C      Zlib compressed data, compressed
[...]
```

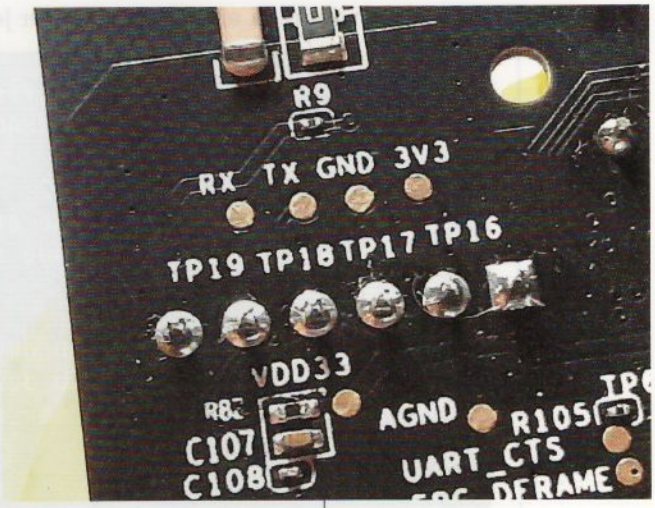
Partant de cette analyse, plusieurs solutions s'offrent à nous. Nous pouvons utiliser `dd` (ou `dcfldd`) pour découper `flash.bin` en plusieurs morceaux en fonction des adresses (ou *offsets*) affichées par `binwalk`, ou nous pouvons demander directement à `binwalk` d'extraire tout cela pour nous, avec `binwalk -e flash.bin`. Ce faisant, nous nous retrouvons avec un sous-répertoire `_flash.bin.extracted` contenant une masse conséquente de fichiers. Chacun d'eux est préfixé de *l'offset* auquel a été trouvée l'information dans `flash.bin`, mais tous ne sont pas pertinents, sachant qu'il y a un certain nombre de faux positifs. Les deux qui nous intéressent sont `200000.squashfs` et `420000.jffs2`, les deux éléments que nous souhaitions extraire.

Un sous-répertoire `squashfs-root/` est également peut-être présent avec le contenu du système de fichiers SquashFS que `binwalk` a traité pour nous. Dans la sortie affichée par `binwalk`, deux mentions sont importantes :

- « `Symlink points outside of the extraction directory[...] changing link target to /dev/null for security purposes` » : en extrayant les données SquashFS, `binwalk` a détecté des liens symboliques pointant sur des fichiers ou répertoires désignés de façon non relative aux données extraites. Comme ces liens peuvent correspondre à des éléments de votre propre système, `binwalk` les a remplacés par des liens vers `/dev/null`.
- « `No such file or directory: 'jefferson'` » : *Jefferson* est un outil initialement développé par Stefan Viehböck et actuellement maintenu par *ONEKEY* **[7]**. Il permet l'extraction des données des systèmes de fichiers JFFS2, et comme pour SquashFS (via `unsquashfs`),

`binwalk` est capable d'utiliser cet outil s'il est présent. Ce n'est pas le cas ici, avec une Debian 12, et un paquet n'est pas même disponible pour la distribution.

Il n'est pas certain que `squashfs-root/` ait été extrait pour vous, car vous n'avez peut-être (probablement pas ?) installé `unsquashfs`. Quoi qu'il en soit, la quantité de fichiers produite par `binwalk` est plus une nuisance qu'autre chose et nous décidons donc de ne conserver que `200000.squashfs` et `420000.jffs2` pour les copier dans le répertoire parent (au côté de `flash.bin`).



Le verso de la carte affiche un marquage explicite pour les points de test situés non loin du connecteur à souder. Quatre de ces six broches correspondent à un port série faisant office de console pour le système (38400 b/s 8N1).

Nous pouvons alors utiliser directement `unsquashfs` (paquet Debian `squashfs-tools`) pour extraire le contenu de `200000.squashfs` :

```
$ unsquashfs 200000.squashfs
Parallel unsquashfs: Using 16 processors
150 inodes (53 blocks) to write
[============================|] 203/203 100%
```

Ainsi, nous obtenons un sous-répertoire `squashfs-root/` contenant le système de fichiers racine de la passerelle. Attention, ici les liens symboliques pointent vers leur cible d'origine, soyez prudent dans vos éditions.

Pour JFFS2, nous devons installer manuellement Jefferson qui, comme `binwalk`, est un outil écrit en Python. Pour éviter de perturber inutilement l'installation de notre système avec des éléments non pris en charge par le système de gestion de paquets, nous pouvons utiliser le mécanisme d'environnement virtuel de Python pour procéder à l'installation via le gestionnaire *pip*. Pour cela, nous créons un environnement propre à Jefferson et procédons à l'installation :

```
$ python3 -m venv /kkpart/jeff

$ source /kkpart/jeff/bin/activate
(jeff)$ pip install jefferson
Collecting jefferson
  Using cached jefferson-0.4.5-py3-none-any.whl (8.8 kB)
Collecting click<9.0.0,>=8.1.3
[...]
Successfully installed click-8.1.7
cstruct-5.3 jefferson-0.4.5 lzallright-0.2.4
```

Nous n'avons plus, ensuite, qu'à lancer Jefferson, puis désactiver l'environnement virtuel :

```
(jeff)$ jefferson 420000.jffs2 -d jffs2
dumping fs to /home/denis/HACK/LIDL/jffs2 (endianness: >)
Jffs2_raw_inode count: 63
Jffs2_raw_dirent count: 62
writing S_ISREG NcpUpgrade.ota
writing S_ISREG app_upgrade.sh
writing S_ISDIR avahi
[...]
writing S_ISREG rcdDb/sub_dev_ddi.rdb
writing S_ISREG rcdDb/sub_dev_schm.rdb
writing S_ISREG rcdDb/s_dev_schm_ver.rdb

(jeff)$ deactivate
$
```

Nous obtenons le sous-répertoire `jffs2/` et pouvons maintenant procéder à une petite analyse/inspection du système.

4. INSPECTONS LE SYSTÈME

Commençons par la partie stockée en SquashFS puisqu'il s'agit clairement du système de fichiers racine. La première chose à regarder est la façon dont démarre le système et donc l'*init*. Dans `etc/init.d/` nous trouvons un unique fichier `rcS` qui détaille tout ce que nous avons besoin de savoir, à commencer par la ligne :

```
mount -t jffs2 /dev/mtdblock4 /tuya
```

Le JFFS2 est monté dans `/tuya` et c'est là qu'ensuite, deux scripts sont exécutés : `tuya_net_start.sh` puis `tuya_start.sh`. Nous y avons naturellement accès, via le contenu obtenu dans `jffs2/`. Le premier script configure le réseau, vérifie la présence d'un fichier `/tuya/backup_flag` qui conditionne ce qui semble être une restauration d'une sauvegarde système et enfin appelle `ssh_monitor.sh`. Ce script implémente un mécanisme pour limiter les tentatives d'accès via SSH en ajoutant des temporisations en fonction du nombre d'échecs d'authentification. C'est aussi ce script qui lance le binaire `tuyadropbear`, qui est n'est autre que Dropbear SSH, un serveur SSH léger pour l'embarqué, certainement modifié par les concepteurs du système (un binaire `/bin/dropbear` est également présent, mais d'une taille différente). Ce script lance un serveur SSH sur le port 2333. C'est amusant de constater que `/bin/dropbear` est exécuté depuis `rcS`, mais tué (`killall dropbear`) dans `ssh_monitor.sh` pour être remplacé par une instance de `tuyadropbear`.

Vient ensuite `tuya_start.sh` qui est clairement là pour déclencher l'exécution de l'applicatif premier du produit, à savoir la passerelle ZigBee. Le script est plus touffu et alambiqué que `tuya_net_start.sh`, mais relativement compréhensible en prenant le temps nécessaire.

Tout ceci est très intéressant, mais ne règle pas notre problème. Si nous voulons avoir la main sur le système, nous devons pouvoir interagir et donc utiliser un shell, que ce soit localement via la console série ou à distance, puisqu'un serveur SSH est effectivement lancé au démarrage. Dans `etc/`, on trouve également un lien symbolique `etc/passwd` pointant vers `/tuya/config/passwd` et, effectivement, dans `jffs2/config/`, nous avons `passwd` et `passwd-` contenant, tout deux :

```
root:.LruAzPuiB8rk:0:0:root:/:/bin/sh
```

Nous pourrions nous amuser à chercher, par force brute, le mot de passe correspondant, avec *John The Ripper* par exemple, mais ceci n'apporterait rien de très intéressant. `/etc/passwd` n'est qu'un lien vers `/tuya/config/passwd` qui se trouve sur un système de fichiers en lecture/écriture.

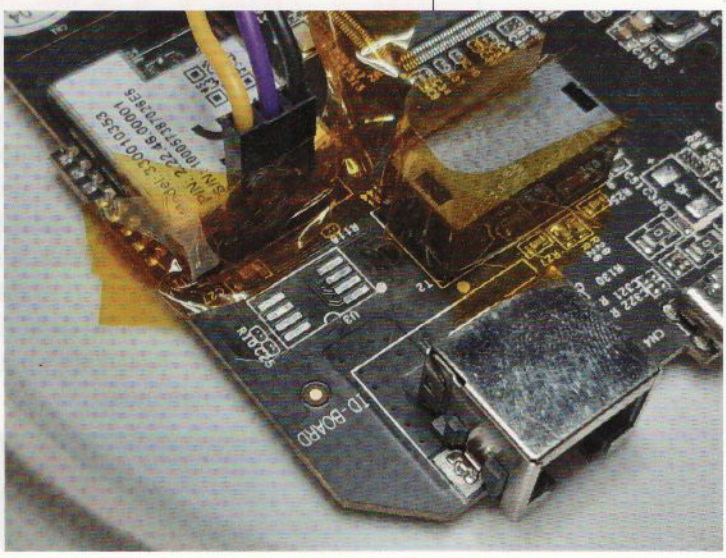
5. APPROCHE NAÏVE, CORRECTION ET SHELL ROOT

Pour adapter l'authentification et définir un nouveau mot de passe, nous pouvons donc simplement substituer ce mot de passe chiffré (« hashé » en fait) par un autre que nous générons nous même :

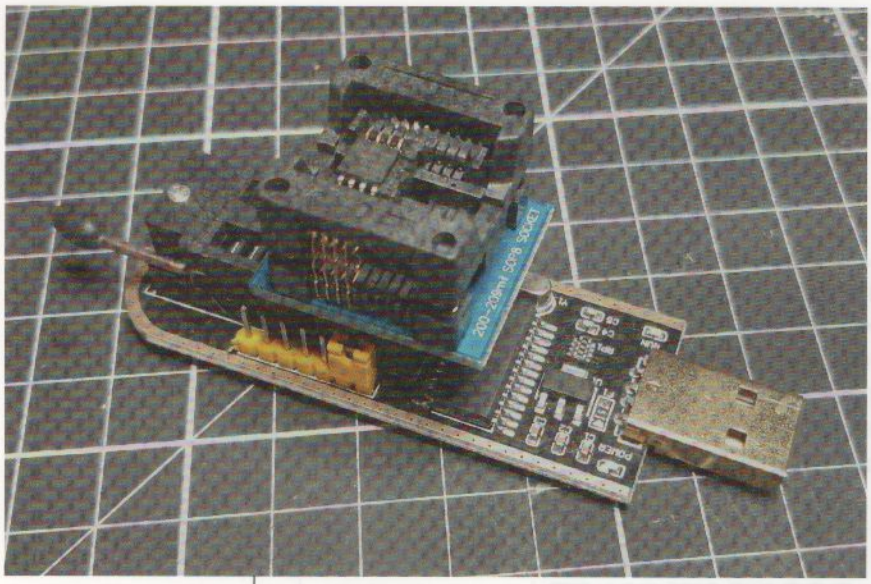
```
$ mkpasswd -m descrypt coucou
y8TBLoShJHp8w
```

Nous éditons alors `/tuya/config/passwd` et remplaçons simplement « `.LruAzPuiB8rk` » par « `y8TBLoShJHp8w` ». Il ne reste ensuite plus qu'à recomposer une nouvelle image, la flasher dans le GD25Q128E et le ressouder sur la carte...

Pas si vite ! Mieux vaut ne présager de rien et se prémunir de toute éventualité de devoir à nouveau dessouder le composant. Ce n'est pas simplement pour s'éviter du travail inutile, mais également pour limiter les risques d'endommager le matériel, puisque les soudures/dessoudage à répétition riment souvent avec décollage de *pads*. La solution que j'ai adoptée ici consiste à souder huit fils fins sur l'emplacement



Retirer la flash SPI GigaDevice GD25Q128E n'est pas très difficile avec une station à air chaud, mais il faut prendre garde à ne pas abîmer les composants aux alentours. Il en va de même pour le nettoyage des pads et la resoudure du composant, en particulier vis-à-vis de la proximité de R10 et C25.



Le CH341A est un outil polyvalent et économique qu'il est possible de trouver un peu partout en ligne pour environ 20 €, livré avec un jeu d'accessoires, dont un adaptateur SOIC8 vers DIP8, ici monté sur l'emplacement ZIF du périphérique.

de la flash pour y connecter un *socket* DIP8. Ainsi, grâce à l'adaptateur SOIC8/DIP8 livré avec le CH341A, il devient possible d'alterner facilement entre programmation et test.

Nous devons réintégrer nos changements dans l'image avant programmation de la flash et ceci passe par une succession d'étapes, plus ou moins inverses à celles que nous avons suivies pour arriver à obtenir le contenu du système. La première étape consiste donc à recréer un système de fichiers JFFS2 à partir du répertoire `jffs2/`.

Pour cela, il nous suffit d'utiliser la commande `mkfs.jffs2` du paquet `mtd-utils`, mais il y a une subtilité. `binwalk` nous indique une organisation en *big endian* (car le processeur MIPS est *big endian* par opposition au x86/amd64 qui est *little endian*) et nous devons faire attention aux permissions sur les fichiers et répertoires qui, dans `jffs2/`, appartiennent tous à l'utilisateur courant. Les options pour gérer respectivement ces points sont `-b` (*big endian*) et `-q` qui attribue la propriété de tous les fichiers à `root` et supprime les permissions en écriture pour *group* et *other*.

Mais ce n'est pas tout. L'étape suivante sera de réassembler le tout pour obtenir une image de précisément 16777216 octets. Si nous utilisons simplement `mkfs.jffs2 -d ./jffs2 -b -q -o new_jffs2.bin`, nous obtiendrons un fichier qui n'est pas un JFFS2 de la taille originale, mais de moins de 3 Mio (au lieu de 12 Mio). Ça ne marchera pas.

Fort heureusement, `mkfs.jffs2` dispose d'une option pour cela, c'est `-p` (ou `--pad`) suivi de la taille que nous voulons obtenir. Les fichiers présents seront alors intégrés dans le système de fichiers, puis celui-ci sera complété avec un bourrage (*padding*) pour arriver à la taille spécifiée. La question est de savoir quelle valeur passer à l'option. Pour cela, à la fois la sortie de `binwalk` et les messages issus du noyau apportent la réponse. Le JFFS2 débute à l'adresse, ou *offset*, `0x000000400000` (`0x00420000`) et finit avec la fin de la flash à `0x000001000000` (`0x01000000`). Nous avons donc 0x1000000 - 0x420000 = 0xbe0000, ou en décimal 12451840 octets, et pouvons lancer notre commande :

```
$ mkfs.jffs2 -d ./jffs2 -b -q \
-p12451840 -o new_jffs2.bin
```

Et nous obtenons un fichier `new_jffs2.bin` de 12451840, exactement comme `420000.jffs2` (que nous aurions tout aussi bien pu prendre comme référence ici, certes). Nous n'en avons pas fini, car nous devons maintenant assembler un morceau de `flash.bin` avec `new_jffs2.bin` et devons commencer par découper la partie nécessaire. Là encore, les positions sont connues, puisque nous avons besoin des 0x420000 premiers octets de `flash.bin`, que nous séparons avec `dd` :

```
$ dd if=flash.bin of=debut.bin bs=1c count=4325376
4325376+0 enregistrements lus
4325376+0 enregistrements écrits
```

`debut.bin` contient maintenant le *bootloader*, le noyau, le SquashFS et le mystérieux « tuya-label » de 128 Kio. Nous pouvons alors concaténer les deux fichiers pour créer `reflash.bin` :

```
$ cat debut.bin new_jffs2.bin > reflash.bin
```

Notre image fait, bien entendu, 16777216 octets, comme `flash.bin` et n'avons plus qu'à l'utiliser avec `flashrom` pour l'enregistrer dans le GD25Q128E. Pour cela, la même syntaxe s'applique que pour la lecture, en remplaçant simplement `-r` par `-w` et en spécifiant le bon fichier. L'opération est encore plus lente que la précédente, car l'outil procède à une lecture de sécurité avant d'effacer la flash, écrit les données et relit le tout pour vérification. Une fois ceci fait, on place le composant dans son adaptateur SOIC8/DIP8 sur le *socket* soudé avec des fils à la carte et on alimente le tout. Le système va alors démarrer comme précédemment (sauf erreur de connexion/soudure de la flash) et...

```
[...]
tuya-linux login: root
Password:
Login incorrect
```

Mais ?!

C'est là que le week-end prend une tournure très pénible, car lorsqu'on pense naïvement toucher au but pour être ainsi frustré, on sait que ce n'est que le début des problèmes. Après quelques heures de tentatives diverses comme :

- essayer un autre mode de chiffrement du mot de passe ;
- ne pas utiliser de mot de passe du tout ;
- créer le `/tuya/config/group` qui était absent ;
- lire et relire les scripts dans `/tuya` ;
- etc.

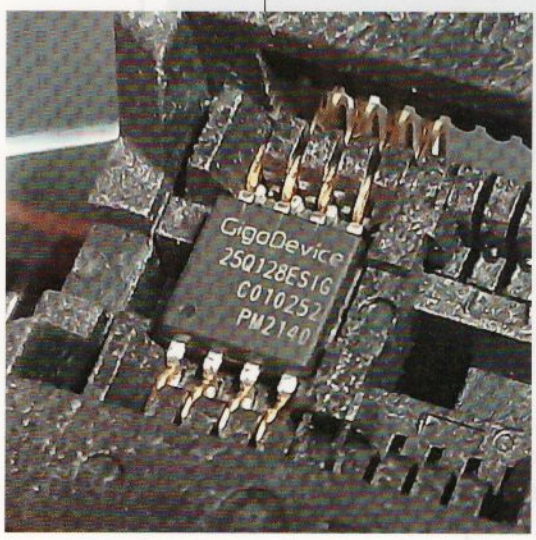
Enfin, l'illumination arrive et on finit par glisser un `cat /etc/passwd` dans un `denis.sh` placé dans `/tuya` et appelé en toute fin de `tuya_start.sh`, juste avant l'invocation de `tuya_start_children.sh` (alias `$TY_START_CHILDREN_SHELL`). Et là, ô surprise, dès le nouveau *boot*,

le contenu du fichier n'est plus le nôtre, mais est remplacé par un autre mot de passe *hashé*, qui est même différent de « `.LruAzPuiB8rk` ». Quelque chose change le mot de passe automatiquement lors du *boot* et ce quelque chose est probablement appelé par les scripts dans `/tuya`.

En guise d'ultime tentative avant d'envisager des mesures plus agressives comme changer le contenu du `inittab` du SquashFS (ce qui n'aurait mené à rien, cf. la fin de l'article), pourquoi ne pas simplement voir ce qui se passe en renommant `tuya_start.sh` en `tuya_start.sh.orig` et `denis.sh` en `rtuya_start.sh`, couplant ainsi littéralement l'herbe sous le pied du script démarrant l'applicatif ? Et...

```
tuya-linux login: root
Password:

Tuya Linux version 1.0
Jan  1 00:00:17 login[121]: root login on 'console'

# ls
bin   etc   init  mnt   root  sys   tuya  var
dev   home  lib   proc  sbin  tmp   usr

# cat /proc/cpuinfo
system type             : RTL8196E
machine                 : Unknown
processor               : 0
cpu model               : 52481
BogoMIPS                : 398.13
tlb_entries             : 32
mips16 implemented      : yes
```

La flash SPI GigaDevice GD25Q128E de 16 Mio contient l'ensemble du système de la passerelle : le bootloader, le noyau, la partition racine en SquashFS et la partition « applicative » en lecture/écriture utilisant le système de fichiers JFFS2.



En effet, bien que n'ayant aucune trace de la chaîne « `passwd` » dans les contenus des scripts, nous avons un changement de mot de passe opéré par l'applicatif (probablement `tyZ3Gw` ou `tuyadropbear`) et le simple fait d'en empêcher l'exécution règle le problème. Nous avons la main sur le système, aussi bien en console que via SSH (puisque Dropbear est actif) et avons quelque 9 Mio à disposition en JFFS2 pour notre usage personnel. Notez que l'option `-oHostKeyAlgorithms=+ssh-rsa` est nécessaire lors de l'appel au client `ssh` afin d'utiliser des algorithmes de signature qui ne sont normalement plus utilisés, car considérés comme obsolètes.

Nous pouvons faire un brin de ménage dans les scripts et le contenu de `/tuya` et même éventuellement intégrer nos propres éléments sans avoir à réutiliser `flashrom`. Le serveur SSH ne propose pas de SFTP

pour une utilisation avec `scp`, mais il est possible de jouer avec `cat` pour transférer des fichiers ainsi : `cat squanchy.jpg | ssh -p2333 -oHostKeyAlgorithms=+ssh-rsa root@192.168.0.69 "cat > /tuya/squanchy.jpg"`. On peut aussi modifier `tuya_net_start.sh` et/ou `ssh_monitor.sh` pour n'utiliser que `/bin/dropbear` et non `/tuya/tuyadropbear`. Les possibilités sont presque infinies...

6. EXTRAIRE LE NOYAU

Ceci n'est pas vraiment capital, mais en fonction de que ce vous avez l'intention de faire du matériel, cela pourra toutefois être intéressant à savoir, peut-être même pour un autre usage ou projet. À ce stade, nous avons séparé le début de l'image initiale de la partie JFFS2, mais qu'en est-il si nous souhaitons découper davantage, et en particulier extraire le noyau qui, comme le précise `binwalk`, est compressé en LZMA ?

Nous avons, à la fois via `binwalk` et les messages du noyau, les adresses relatives au début de la flash et donc, en principe, la taille de l'élément. Mais en réalité, il s'agit juste de partitions MTD ou, en d'autres termes, de zones de flash réservées à cet usage. Ceci ne veut pas dire que le noyau occupe réellement autant de place.

Le noyau se trouve à un offset de 141336 octets et sa taille décompressée, obtenue des méta-données LZMA, est de 3555200 octets. Techniquement, SquashFS commence en 0x00400000, donc la zone du noyau s'étend donc de 0x00200000 à 0x00400000, soit 0x200000 octets, ou 2097152 en base 10. Nous pouvons donc commencer par extraire cette partie avec :

```
$ dd if=flash.bin of=kernel.lzma skip=141336 bs=1 count=2097152
2097152+0 enregistrements lus
2097152+0 enregistrements écrits
2097152 octets (2,1 MB, 2,0 MiB) copiés, 1,78455 s, 1,2 MB/s
```

Et effectivement, nous pouvons obtenir des informations sur l'image LZMA :

```
$ lzmainfo kernel.lzma
kernel.lzma
Uncompressed size:          3 MB (3555200 bytes)
Dictionary size:            8 MB (2^23 bytes)
Literal context bits (lc):  3
Literal pos bits (lp):      0
Number of pos bits (pb):    2
```

Mais si nous tentons de décompresser :

```
$ unlzma kernel.lzma
unlzma : kernel.lzma:
  Les données compressées sont corrompues
```

Le problème avec LZMA est qu'il est possible d'identifier le début des données compressées, ainsi que de récupérer les informations de l'en-tête, mais que nous ne savons pas où elles se terminent. Et c'est précisément ce qui dérange `unlzma`. En regardant de plus près ce que nous avons en main, nous nous rendons compte que plusieurs parties des données contiennent des séries d'octets identiques que `hd` condense visuellement avec une ligne contenant un simple `*` :

```
$ hd kernel.lzma | grep -B 5 -A 5 "^*"
0011deb0  ae be 5a c0 f9 6d 9b 07  13 e9 4f 10 83 32 07 bf  |..Z..m....O..2..|
0011dec0  d8 cb 97 2d 57 a0 59 33  ac 6c 1e 4c 32 c6 9d e7  |...-W.Y3.l.L2...|
0011ded0  f5 53 f7 15 d1 54 ca 8d  25 31 6a 82 07 71 28 e3  |.S...T..%1j..q(.|
0011dee0  88 a0 76 98 ac bf 00 00  60 1c 85 c3 00 00 00 00  |..v.....`.......|
0011def0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
0011dff0  00 00 00 00 00 00 00 00  fd a8 ff ff ff ff ff ff  |................|
0011e000  ff ff ff ff ff ff ff ff  ff ff ff ff ff ff ff ff  |................|
*
001dd7e0  ff ff ff ff ff ff ff ff  68 73 71 73 a7 00 00 00  |........hsqs....|
001dd7f0  00 0e 0d 80 00 00 02 00  08 00 00 00 04 00 11 00  |................|
001dd800  e0 00 01 00 04 00 00 00  c3 15 00 00 00 00 00 00  |................|
001dd810  c2 04 0e 00 00 00 00 00  ba 04 0e 00 00 00 00 00  |................|
001dd820  ff ff ff ff ff ff ff ff  0c f8 0d 00 00 00 00 00  |................|
```

Ceci peut être indicateur d'une fin de données compressées, suivies d'un bourrage et de « vide » (`0xff` en flash). Attention, cela dépend de l'alignement utilisé et de la taille précise des données, qui peuvent parfaitement se finir à un endroit avec peu ou pas de bourrage. Mais ça se tente...

Ici, nous voyons qu'entre 0x0011dee0 (1171168) et 0x0011e000 (1171456), nous avons une belle quantité de `0xff`. C'est peut être un indice intéressant et nous pouvons écrire quelques lignes de shell pour brutalement et sauvagement essayer toutes les possibilités :

```
for ((i=1171168; i<=1171456; i++))
do
  echo $i
  dd if=flash.bin of=kernel_$i.lzma skip=141336 bs=1 count=$i
  unlzma kernel_$i.lzma
done
```

Inutile de suivre l'exécution de la boucle, tout ce qu'il nous faut, c'est un peu d'espace disque et, si l'opération réussie, nous trouverons un noyau décompressé suffixé de l'*offset* correspondant :

```
$ ls kernel_*
[...]
kernel_1171171.lzma
kernel_1171172.lzma
```

```
kernel_1171173.lzma
kernel_1171174
kernel_1171175.lzma
kernel_1171176.lzma
kernel_1171177.lzma
[...]
```

Le noyau compressé se termine donc à l'*offset* 1171174 (0x0011DEE6) et, à toutes fins utiles, nous pouvons refaire la manipulation manuellement :

```
$ dd if=flash.bin of=kernel.lzma skip=141336 bs=1 count=1171174
1171174+0 enregistrements lus
1171174+0 enregistrements écrits

$ unlzma kernel.lzma

$ ls -l kernel
-rw-r--r-- 1 denis denis 3555200 21 janv. 08:13 kernel
```

Et voilà, un beau noyau tout décompressé.

7. REGARDONS CE QUE D'AUTRES ONT FAIT...

Chercher des informations sur le SoC RTL8196E n'est pas très agréable. Cette plateforme existe depuis quelque temps déjà, mais le support OpenWrt, par exemple, sur toute la famille RTL819x est quasi inexistant ou, du moins, inutilisable. Il semblerait que les informations sur ce SoC ne soient pas facilement accessibles, sans parler de la vétusté du SDK disponible sur SourceForge **[8]**.

Mais c'est au détour d'un message sur le forum de Home Assistant qu'on trouve la perle rare **[9]**. Là, un certain Paul Banks dit très modestement « *If I get time, I plan to hack the Tuya / Lidl Smart Home hub to integrate with Home Assistant.* », en pointant l'état de ses travaux décrits sur son blog : **https://paulbanks.org/projects/lidl-zigbee**.

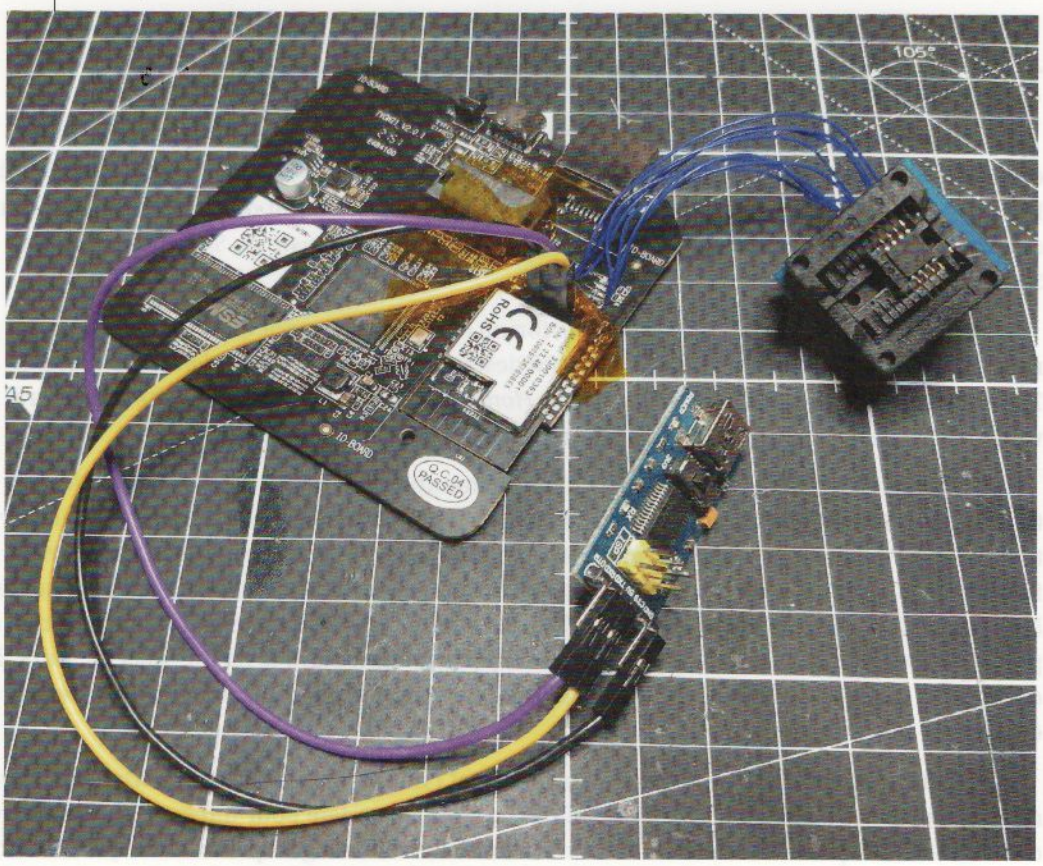
Cette passerelle ZigBee n'est pas de la même génération que celle en vente actuellement, tout en étant vraiment très similaire (seule la SDRAM semble différente), mais le plus important est que Paul est allé beaucoup plus loin que moi, et ce, de manière moins... barbare. La finalité est la même, mais il est intéressant de voir qu'une autre approche, comme utiliser le *bootloader* pour obtenir le contenu de la flash ou obtenir un accès `root` en décodant le mot de passe généré à partir d'une clé intégrée au SoC, permet d'atteindre le même objectif : faire de ce matériel une passerelle libérée du *cloud* (c'est rassurant de voir que je ne suis pas le seul à être de cet avis radical).

La partie la plus intéressante est celle concernant l'image SquashFS, et en particulier le *hack* du constructeur consistant à utiliser certains champs de l'en-tête pour intégrer une somme de contrôle vérifiée par le *bootloader*. Pour créer une nouvelle image à partir de ce qui est extrait

par `unsquashfs`, il ne suffit donc pas simplement d'utiliser `mksquashfs`. Un ajustement de l'image est nécessaire grâce à un script Python écrit par Paul Banks **[10]**. Ce qui pourrait de résumer rapidement par les commandes suivantes :

```
$ dd if=flash.bin of=toutdebut.bin bs=1c count=2097152
$ unsquashfs 200000.squashfs
$ vi squashfs-root/etc/inittab
$ mksquashfs squashfs-root newrootfs.bin -comp xz -noappend
$ python3 rootfs_tool.py build newrootfs.bin newrootfsOK.bin
$ dd if=newrootfsOK.bin of=newrootfsOK.img bs=2228224 conv=sync
$ cat toutdebut.bin newrootfsOK.img new_jffs2.bin > reflash2.bin
```

Ici, l'opération consistait simplement à changer `inittab` pour supprimer le *login* et donner un shell directement (pour simple test). Mais ceci ouvre encore plus de possibilités, dans le sens où il devient possible de changer la partition racine et commencer à intégrer les éléments sans avoir à reposer sur les scripts initiés par un appel à `tuya_start.sh` depuis `/etc/init.d/rcS`. On pense naturellement à un BusyBox un peu plus étoffé en fonctionnalités.



Pour mitiger les conséquences physiques de déconnexions/reconnexions répétées de la flash durant la phase d'expérimentation, la solution la plus simple consiste à utiliser un socket DIP8 soudé, via des fils, à l'emplacement sur la carte, pour ne pas risquer d'endommager les pistes. Une option encore plus sûre aurait été d'utiliser deux adaptateurs SOIC8/DIP8 pour éviter au maximum les contraintes mécaniques.

Paul fournit même une procédure complète pour l'intégration du produit, « décloudifié », dans une installation Home Assistant **[11]** comme *bridge* ZigBee, à l'aide d'un simple binaire de sa création **[12]**, à ajouter dans la partie JFFS2. D'après le log de *commit*, ceci a pris plus de temps qu'un week-end, mais le résultat est impressionnant, car c'est une complète conversion d'un produit pro-Apple et pro-Google en quelque chose de respectueux de la vie privée, et réellement utilisable sereinement dans son installation. Chapeau bas, monsieur Banks !

8. POUR FINIR

Le but ici n'était pas vraiment de vous détailler comment prendre la main sur cette passerelle spécifique, mais de vous entraîner dans ma petite expédition pour vous détailler l'un des cheminements possibles pour arriver à un tel résultat, ainsi que de présenter les outils à votre disposition pour y arriver. La logique aurait voulu qu'on fasse une investigation préalable sur le Web, mais si tel avait été le cas, nous serions tombés sur le blog de Paul Banks qui a déjà synthétisé tout cela et nous n'aurions pas eu le plaisir de chercher et de trouver notre chemin. Bien au contraire, les billets sur le blog de Paul permettent d'avoir une autre approche, d'abord complémentaire, mais surtout plus aboutie.

Peut-être que tout ceci vous aura donné envie, à vous aussi, de vous essayer à ce genre d'exercice. Peut-être avec un routeur, une passerelle, une caméra connectée ou n'importe quel appareil contenant un SoC. On apprend énormément ainsi et parfois on a des surprises, bonnes ou terrifiantes (surtout avec les caméras chinoises), mais dans tous les cas, c'est toujours plus satisfaisant que d'aller voir un Marvel à la sauce Disney.... **DB**

RÉFÉRENCES

[1] **https://www.realtek.com/en/products/communications-network-ics/item/rtl8196e**

[2] **https://esmt.com.tw/upload/pdf/ESMT/datasheets/M13S2561616A(2S)_%20-40~85.pdf**

[3] **https://www.gigadevice.com/product/flash/product-series/spi-nor-flash/gd25q128e**

[4] **https://www.tuya.com/**

[5] **https://github.com/flashrom/flashrom**

[6] **https://github.com/ReFirmLabs/binwalk**

[7] **https://github.com/onekey-sec/jefferson/**

[8] **https://sourceforge.net/projects/rtl819x/**

[9] **https://community.home-assistant.io/t/hacking-the-silvercrest-lidl-tuya-smart-home-gateway/270934**

[10] **https://paulbanks.org/download/files/lidl-zigbee/rootfs_tool.py**

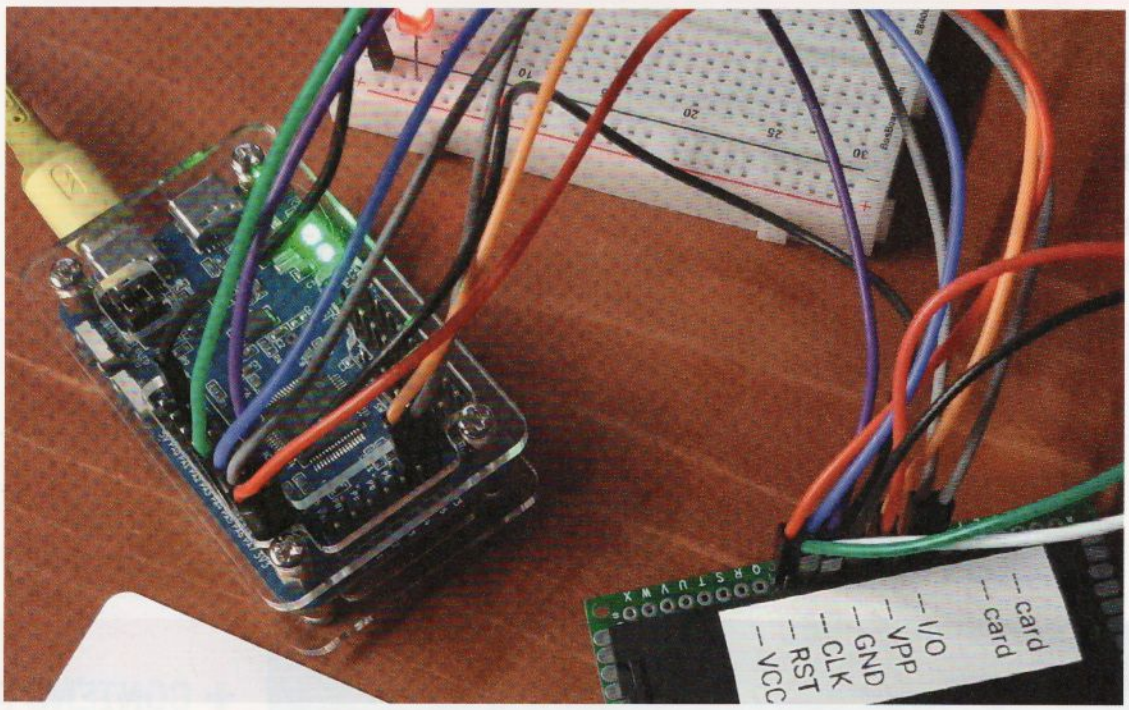
[11] **https://paulbanks.org/projects/lidl-zigbee/ha/**

[12] **https://github.com/banksy-git/lidl-gateway-freedom**

HYDRABUS : UN OUTIL POUR TOUS LES BUS

Denis BODOR

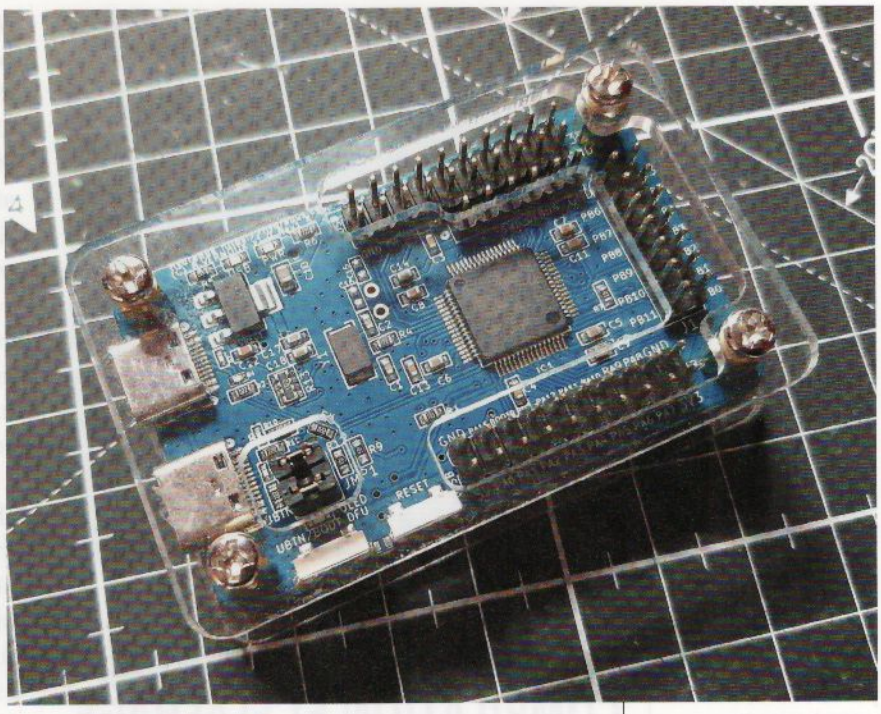
Vous connaissez la routine : un convertisseur USB/série, un analyseur logique, un petit bout de code sur un Arduino ou un ESP8266 pour vérifier un capteur i²c, une sonde JTAG, un programmeur de flash CH341A ou TL866A, un adaptateur CAN/USB, un programmeur SWD... Et forcément, le matériel qu'il vous faut n'est pas sous la main. Hydrabus est un projet open hardware et open source, cousin du Bus Pirate, qui règle ce problème. Un seul outil pour plein d'usages, un vrai couteau suisse des bus et interfaces en tous genres.



Hydrabus

– Hydrabus : un outil pour tous les bus –

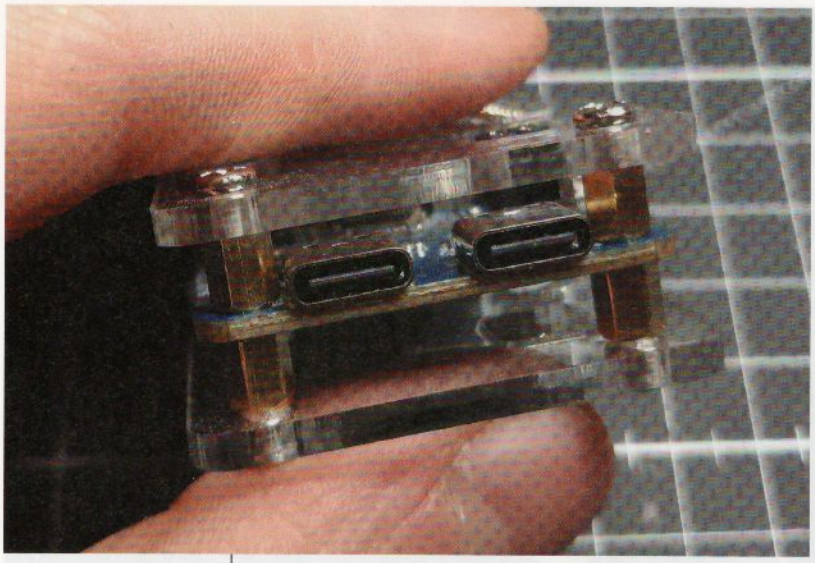
Hydrabus, et son *firmware* HydraFW, ne sont pas nouveaux, puisque le projet a presque 10 ans maintenant. Mais la révision rev1.5 vient tout juste d'arriver, avec enfin une connectique USB-C et est disponible à l'achat chez DigiKey (~65 €). Il n'en fallait pas plus pour que je finisse par craquer et jette mon dévolu sur la bête, en particulier en constatant une possibilité de connexion pour une *smartcard* ISO7816 (dada du moment, vous avez peut-être remarqué).



L'Hydrabus se présente sous une forme très compacte et, en dehors du nombre de broches disponibles, rien ne laisse deviner la quantité de fonctionnalités que concentre ce petit périphérique.

L'Hydrabus est l'œuvre de Benjamin Vernoux, c[illegible] créateur du récepteur SDR Airspy avec Youssef Touil, auteur de SDR# (SDRSharp), et repose sur un microcontrôleur STM32F405. Il se présente sous la forme d'un circuit de 4 par 6 cm, pris en sandwich entre deux morceaux de polycarbonate transparent découpés au laser et fournit d'une part une connectivité USB-C, et de l'autre une myriade de connecteurs permettant de s'interfacer avec différents bus, interfaces et protocoles : SPI, i²c, CAN, UART, 1-wire, JTAG, sortie PWM, DAC, ADC, etc. L'idée derrière le matériel est simple : fournir une interface unique, en USB, pour se connecter à tous les bus et interfaces, sans avoir recours à un utilitaire spécifique (mode interactif via port série USB 115200 8N1), mais tout en permettant une utilisation via des programmes (mode binaire) écrits en Python, par exemple. Un autre périphérique du même type est le Bus Pirate de *Dangerous Prototypes*, longtemps resté en pause depuis la version 3 et revenant aujourd'hui, semble-t-il, en v5 (précommandes), mais dont les caractéristiques ne m'inspirent pas du tout (écran couleur, 18 LED RGB… Quel intérêt en pratique ?).

L'Hydrabus est réellement difficile à décrire tant il sait faire de choses. C'est à la fois un périphérique interactif avec lequel dialoguer pour manipuler des données sur ces interfaces, mais également une sorte de caméléon capable de remplacer un programmeur SWD pour votre devkit STM32, un adaptateur USB/série, un sniffeur i²c/SPI, un programmeur de flash SPI, un analyseur logique, un générateur PWM, un programmeur ISP pour AVR/Arduino, etc. Le tout en un seul et unique petit périphérique. C'est un peu aux bus et interfaces ce que le Flipper Zero est au *pentesting* RF et NFC.



Cette nouvelle et dernière révision du matériel intègre deux ports USB-C, en remplacement des « vieux » et maintenant pénibles connecteurs micro-USB-B.

1. TOUR DU PROPRIÉTAIRE, COMPILATION ET FLASHAGE

Comme dit précédemment, au cœur de l'Hydrabus se trouve le microcontrôleur STM32F405 (ARM Cortex-M4) à 168 MHz avec 1 Mio de flash, 192 Kio de SRAM, USB OTG, support SD/SDIO et une belle collection d'interfaces (série, SPI, i²c, etc.) comme c'est souvent le cas chez ST. L'Hydrabus, en dehors de connecteurs, des deux ports USB-C et d'une tripotée de composants passifs se limite à cela, si ce n'est sur le dessous où trouve place un emplacement pour micro SD. Deux cavaliers sont présents, permettant de rendre active la LED utilisateur (en plus de celle pour l'alimentation et l'USB) ainsi que le bouton utilisateur. Enfin, un bouton *reset* est intégré, permettant bien entendu le *reset*, mais aussi, et surtout, en cas d'utilisation avec le bouton utilisateur, de démarrer l'Hydrabus en mode DFU pour mise à jour du *firmware* (USB1 uniquement).

À la connexion sur PC ou Mac, via l'un des deux ports USB, le périphérique apparaît comme une interface série permettant de se connecter et d'utiliser l'interface interactive (115200 b/s 8N1). Une fois connecté avec GNU Screen ou Minicom par exemple, une invite `>` permet de saisir des commandes, dont `help` pour obtenir l'aide. L'interface maintient un historique et permet la complétion (avec TAB) ainsi que l'utilisation d'une version abrégée des commandes (`e` pour `exit` par exemple, ou `r` pour `read`). La structure de l'interface est une sorte d'arborescence où, à la racine, vous pouvez entrer dans un mode ou un autre en tapant la commande correspondante (`spi`, `i2c`, `uart` par exemple). L'invite change alors pour indiquer le mode et éventuellement le périphérique utilisé. La sortie d'un mode se fait via `exit`. Nous verrons le reste de la syntaxe, assez astucieuse, en explorant les différents bus et interfaces.

Côté logiciel, nous avons l'utilisation de ChibiOS (ou ChibiOS/RT) qui, à l'instar de FreeRTOS et Zephyr, est un système d'exploitation temps réel pour microcontrôleurs. Celui-ci est assez courant sur les plateformes STM32 et permet, en plus des facilités classiques qu'offre un OS, de fournir une couche d'abstraction matérielle et une bibliothèque standard facilitant le développement. Ici, la couche d'abstraction est de plus renforcée par les classiques bibliothèques du *framework* STM32Cube typiques de la plateforme (`src/drv/stm32cube/stm32f4xx_hal/` dans les sources). Il ne s'agit donc pas de code *bare metal* comme on pourrait s'y attendre, mais ceci ne change finalement pas grand-chose d'un point de vue utilisation.

Toujours côté logiciel, mais via le mode binaire (appelé BBIO) cette fois, nous avons le module Python `pyHydrabus`, permettant comme son nom l'indique de profiter des fonctionnalités de l'Hydrabus dans son code Python. Ceci permet de créer des outils reposant entièrement sur le périphérique pour, par exemple, manipuler des flashs SPI, lire des capteurs ou dialoguer avec une *smartcard*. Un certain nombre d'exemples sont fournis avec les sources du *firmware*, dans le répertoire `contrib/`, et vous pourrez installer facilement le module dans un environnement virtuel de Python avec `python3 -m venv /kkpart && source /kkpart/bin/activate && pip install pyHydrabus`, pour ensuite utiliser ces scripts (et quitter l'environnement avec `deactivate`).

Mais la première chose à faire à la réception de votre Hydrabus sera de mettre à jour son *firmware*, dont le développement est toujours très actif. Celui-ci est disponible en version stable (0.11) sur GitHub **[1]**. L'archive ZIP mise à disposition contient, entre autres, le fichier `hydrafw.dfu` que vous pourrez flasher sous GNU/Linux avec `dfu-util` (paquet du même nom sous Debian/Raspian) ou avec DfuSe (fourni dans l'archive) sous Windows. Pour cela, passer l'Hydrabus en mode DFU en le connectant tout en appuyant sur le bouton UBTN (celui à côté des cavaliers) ou en appuyant sur UBTN puis sur le bouton RESET. Le périphérique doit apparaître avec les ID USB `0483:df11` par opposition au démarrage standard avec les ID `1d50:60a7`. Vous pourrez alors flasher le *firmware* avec `dfu-util -i 0 -a 0 -d 0483:df11 -D hydrafw.dfu` (un `sudo` peut-être nécessaire selon votre configuration).

Bien entendu, si vous voulez profiter pleinement des dernières nouveautés et correctifs, vous devrez opter pour la version en cours de développement (branche *master*) que vous pourrez cloner depuis GitHub. Une chaîne de compilation ARM AArch32 *bare-metal* devra être disponible sur le système, soit via votre système de gestion de paquets (`gcc-arm-none-eabi` + `binutils-arm-none-eabi`), soit via un téléchargement sur le site *ARM Developer* **[2]**. Ceci fait, placez-vous dans le sous-répertoire `src/` des sources clonées et utilisez simplement `make` (éventuellement avec l'option `-j`). Vous obtiendrez le fichier `build/hydrafw.dfu` que vous pourrez alors flasher exactement comme le *firmware* stable 0.11. En cas de problème, assurez-vous que votre installation Python dispose du module `IntelHex` (paquet Debian `python3-intelhex`) nécessaire au script `dfu-convert.py` pour convertir le fichier EFL en image DFU. Un ticket est actuellement ouvert sur GitHub pour supprimer totalement les dépendances à Python dans le processus de construction.

Quelle que soit la méthode utilisée, vous pouvez vérifier la version flashée avec :

```
> show system
HydraFW (HydraBus) v0.11-12-g56989e0-dirty 2024-01-30
sysTime: 0x00022fe8.
cyclecounter: 0x04ef1649 cycles.
cyclecounter64: 0x0000000004ef1658 cycles.
10ms delay: 1680027 cycles.

MCU Info
DBGMCU_IDCODE:0x10076413
CPUID:        0x410FC241
Flash UID:    0x480042 0x51315014 0x20333341
Flash Size:   1024KB
```

```
Kernel:       ChibiOS 5.1.0
Compiler:     GCC 12.2.1 20221205
Architecture: ARMv7E-M
Core Variant: Cortex-M4F
Port Info:    Advanced kernel mode
Platform:     STM32F405 High Performance with DSP and FPU
Board:        HydraBus 1.0
Build time:   Jan 31 2024 - 05:20:09
```

2. LES DIFFÉRENTS BUS

Nous n'allons pas ici faire le tour de l'ensemble des bus et interfaces disponibles, faute de place dans le magazine, mais aussi parce que je n'ai pas tout le matériel nécessaire, comme en particulier un matériel interfacé en CAN (typiquement, un périphérique ODB/ODB2 appelé « voiture »). L'idée est avant tout de comprendre le principe de fonctionnement de l'interface utilisateur puisque, entre les différents modes, les similarités sont importantes et surtout délibérées. Commençons par quelque chose de courant, le bus SPI.

2.1 SPI

Hydrabus propose, aussi bien sur GitHub que sur le site du projet, un diagramme présentant l'usage des quelque 57 broches disponibles. Cependant, pour une activité « de terrain », ceci n'est pas utile, car l'interface textuelle offerte par le *firmware* fournit également l'information. Ainsi, après être passé en mode SPI et avoir sélectionné le second périphérique (`spi2`), vous pouvez tout simplement afficher les broches utilisées :

```
> spi
Device: SPI1
GPIO resistor: floating
Mode: master
Frequency: 320khz (650khz, 1.31mhz,
   2.62mhz, 5.25mhz, 10.50mhz, 21mhz, 42mhz)
Polarity: 0
Phase: 0
Bit order: MSB first

spi2> device 2
Note: SPI parameters have been
      reset to default values.

spi2> show pins
CS:   PC1 (SW)
SCK:  PB10
MISO: PC2
MOSI: PC3
```

Pour tester ces fonctionnalités, nous utiliserons ici une flash de 16 Mio Macronix MX25L12835F, initialement flashée avec un *firmware* pour un routeur LIDL (cf. article dans le présent numéro, et non, permuter la flash n'a pas fonctionné). Pour accéder au contenu du composant, nous le connectons ainsi (via un adaptateur ZID SIOC8/PDIP8) :

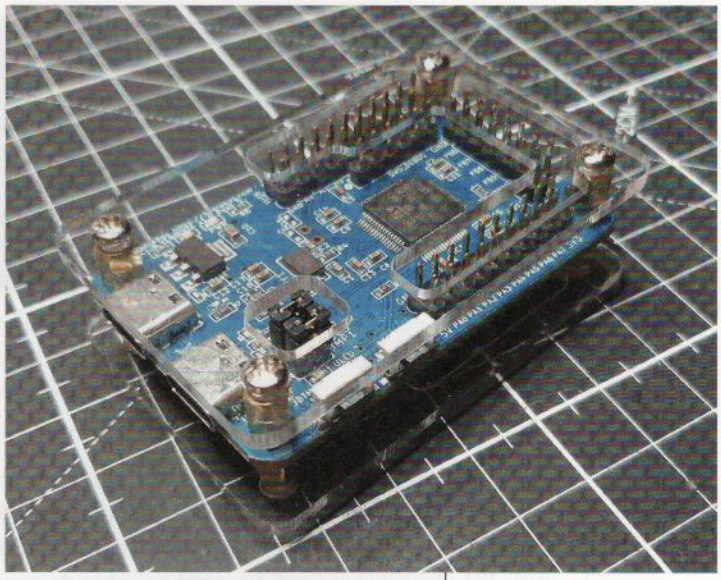
- CS (1) -> CS (PC1) ;
- SO/SIO1 (2) -> MISO (PC2) ;
- SI/SIO0 (5) -> MOSI (PC3) ;
- SCLK (6) -> SCK (PB10) ;
- VCC (8) -> 3V3 ;
- GND (4) -> GND.

Pour dialoguer avec le composant, nous avons plusieurs commandes qui sont en réalité des « mots » destinés à composer une séquence. C'est un point qui peut être perturbant au départ, mais une fois le principe assimilé, ceci est relativement intuitif. Ainsi, la commande `read` (ou `r` en version abrégée) seule n'a pas réellement de sens, pas plus que le fait de pouvoir répéter la commande avec un double point (`:`) suivi d'une valeur.

En regardant la *datasheet* du MX25L12835F, nous voyons qu'il est possible d'interroger le composant sur son identité en écrivant 0x9f sur le bus après avoir sélectionné (ligne /CS) le composant. Lire l'identité revient donc à passer /CS à la masse, envoyer 0x9f, lire deux octets sur le bus et repasser /CS à l'état haut. En « langage » Hydabus, ceci nous donne :

```
spi2> [ 0x9f r:2 ]
/CS ENABLED
WRITE: 0x9F
READ: 0xC2 0x20
/CS DISABLED
```

`[` et `]` permettent de mettre respectivement /CS à l'état bas et haut (on peut aussi utiliser `cs-on` et `cs-off`). Nous obtenons en retour, effectivement, les octets `0xc2` et `0x20` correspondant au MX25L12835F (*datasheet* table 6 page 26).



À défaut de boîtier, l'Hydrabus arrive en kit, avec deux plaques de polycarbonate protégeant le circuit imprimé et retenues par des entretoises. Cela n'empêchera pas la poussière de se glisser partout, mais donne un aspect plus « fini » et robuste à l'ensemble.

Pour lire les données en mémoire, la flash SPI Macronix dispose de la commande 0x03 à faire suivre de trois octets permettant de spécifier une adresse de départ (entre 0x000000 et 0xffffff). Toute lecture qui s'en suit parcourt l'espace de stockage en incrémentant l'adresse automatiquement à chaque lecture. Lire les 8 premiers octets de la flash revient donc à faire :

```
spi2> [ 0x03 0x00 0x00 0x00 r:8 ]
/CS ENABLED
WRITE: 0x03 0x00 0x00 0x00
READ: 0x0B 0xF0 0x00 0x04 0x00 0x00 0x00 0x00
/CS DISABLED
```

Mais nous pouvons également opter pour `hd` (comme *hexdump*) pour obtenir quelque chose de plus lisible avec davantage d'octets lus et surtout mis en forme :

```
spi2> [ 0x03 0x00 0x00 0x00 hd:64 ]
/CS ENABLED
WRITE: 0x03 0x00 0x00 0x00
0B F0 00 04 00 00 00 00  00 00 00 00 00 00 00 00  |  ................
00 00 40 21 40 88 60 00  00 00 00 00 3C 01 B8 00  |  ..@!@.`.....<...
00 01 78 25 8D EE 00 00  00 00 00 00 00 0E 70 25  |  ..x%..........p%
3C 01 81 96 34 21 E0 00  00 01 78 25 15 CF 00 0A  |  <...4!....x%....
/CS DISABLED
```

D'autres commandes/caractères sont disponibles pour configurer et/ou exécuter des opérations. `&` et `%`, par exemple, permettent respectivement d'introduire des pauses en microsecondes ou en millisecondes. Là aussi, on peut faire suivre ces directives d'un double point suivi d'un nombre de répétitions (`%:75` signifie donc « pause de 75 ms »).

Ce type d'opération sur un bus SPI fonctionnera avec n'importe quel composant compatible, mais lorsqu'on a affaire à une flash, le plus simple est d'utiliser un outil dédié, plutôt que de s'amuser à afficher ainsi le contenu. Pour cela, on utilise souvent un CH341A avec `flashrom`, mais l'Hydrabus est parfaitement capable de remplacer un tel matériel, puisqu'il implémente le protocole *serprog* (*serial programming*) pouvant justement être utilisé avec `flashrom`. *Dumper* le contenu de notre MX25L12835F dans un fichier est aussi simple que ça :

```
$ flashrom --programmer serprog:dev=/dev/ttyACM0,spispeed=2M \
-c "MX25L12833F/MX25L12835F/MX25L12845E/MX25L12865E/MX25L12873F" \
--progress -r /tmp/image.bin
[...]
[READ] 29% complete...
[READ] 30% complete...
[READ] 31% complete...
[...]
[READ] 100% complete...
done.
```

2.2 i²c

Tout comme pour SPI, accéder et donc tester un composant sur le bus i²c est un jeu d'enfant, à partir du moment où on a saisi le principe de fonctionnement de l'interface et que l'on connaît le bus en question. i²c est moins intuitif que SPI, mais la lecture approfondie des *datasheets* vient à bout des ambiguïtés. Notre victime pour ce bus sera une RTC Analog Devices DS3231 (initialement Dallas, puis Maxim-Dallas (filiale) et depuis 2021 Analog Devices, on a presque du mal à suivre), très classique, en particulier pour servir de RTC pour un SBC comme une Pi.

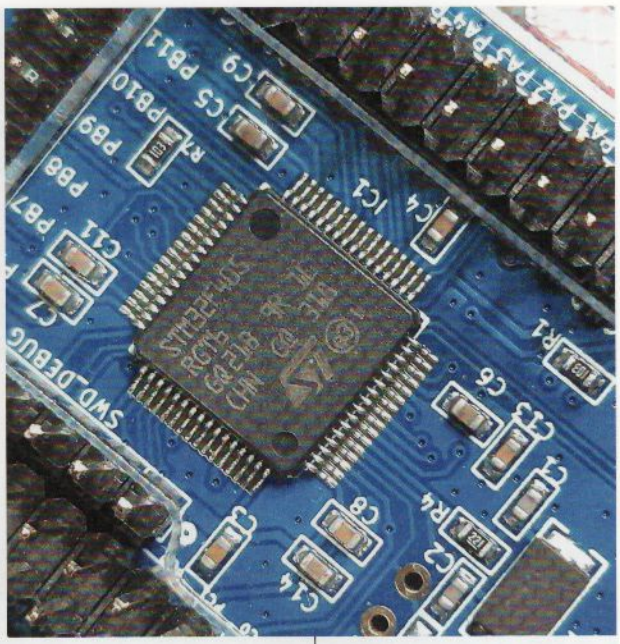
Comme précédemment, après entrée dans le mode `i2c`, on s'enquière du brochage :

```
i2c1> show pins
SCL: PB6
SDA: PB7
```

Puis on procède aux connexions. Notez que pour des vitesses élevées sur le bus, l'utilisation de résistances de tirage (*pull-up*) est recommandée sur SDA (données) et SCL (horloge). Ici, les symboles [et] ont une signification différente puisqu'il s'agit d'indiquer une condition *start* et *stop*. Sans entrer dans le détail du protocole, le bus i²c est bidirectionnel et la ligne de données est contrôlée à la fois (alternativement) par le maître et l'esclave. Le contrôle du changement d'état de SDA permet de régler le problème de collision en régissant qui « à la main » sur la ligne.

Un composant i²c possède une adresse sur le bus et, après connexion, nous pouvons nous enquérir de la présence de périphérique avec :

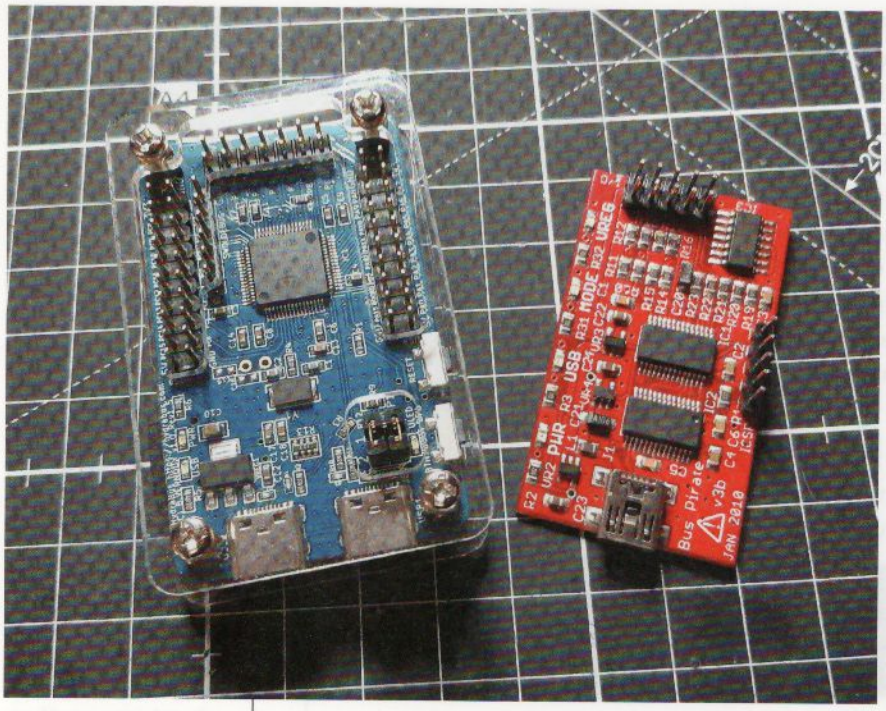
```
i2c1> scan
Device found at address
0x68 (0xd0 W / 0xd1 R)
```



L'Hydrabus est construit autour du microcontrôleur STM32F405 qui se classe parmi les « gros » de la famille STM32. Un F7 n'aurait sans doute rien apporté de plus au concept, si ce n'est faire augmenter inutilement le coût et donc le prix de vente.

Nous obtenons l'adresse de notre RTC et l'Hydrabus à la gentillesse de nous indiquer les « adresses » à utiliser pour communiquer avec le composant. En effet, en i²c, l'adresse sur 7 bits, 0x68 par exemple, est utilisée en la complétant par un huitième bit (de poids le plus faible) indiquant une écriture (0) ou une lecture (1). Notre 0x68 devient donc `0xd0` et `0xd1` selon l'opération à appliquer. Ceci nous permet directement, en reposant sur la *datasheet* du DS3231, de récupérer la valeur contenue dans le registre 0x00, contenant le nombre de secondes de l'heure courante (bits 0 à 3 pour les secondes et 4 à 6 pour les dizaines de secondes) :

```
i2c1> [ 0xd0 0x00 [ 0xd1 r:1 ]
I2C START
WRITE: 0xD0 ACK 0x00 ACK
I2C START
WRITE: 0xD1 ACK
READ: 0x34 NACK
I2C STOP

i2c1> [ 0xd0 0x00 [ 0xd1 r:1 ]
I2C START
WRITE: 0xD0 ACK 0x00 ACK
I2C START
WRITE: 0xD1 ACK
READ: 0x35 NACK
I2C STOP
```



La rencontre de deux générations, avec à gauche l'Hydrabus et à droite un vieillissant (dernier commit il y a 9 ans) Bus Pirate v3b basé sur un Microchip PIC24FJ6 complété d'un convertisseur série/USB FT232RL (USB mini A).

Notre ligne de commande signifie ici : condition *start*, adresse d'écriture `0xd0`, écriture du numéro de registre, condition *start*, lecture à l'adresse de lecture, lecture de 1 octet, condition *stop*. Comme le veut le protocole, chaque opération est ponctuée d'un accusé de réception (`ACK` pour *acknowledge*), sauf la fin, marquée par un « non-accusé-réception » (`NACK`).

2.3 UART

En plus de la connexion via le ou les ports USB-C fournissant une interface série à 115200 b/s, l'Hydrabus dispose de deux USART supplémentaires, respectivement accessibles via les broches PA10/PA9 et PA2/PA3. Pourquoi diable accéder en mode série à un périphérique qui se connecte en mode série ? La réponse est premièrement le fait de pouvoir remplacer sans problème un convertisseur USB/série oublié ou introuvable, mais la seconde est une facilité d'utilisation. Non seulement nous pouvons détecter le débit utilisé, mais pouvons également basculer au besoin sur un mode *bridge* où ce n'est plus avec l'Hydrabus qu'on dialogue, mais avec la cible elle-même.

Comme pour les autres modes, `show pins` indique les broches à utiliser et `device 1` ou `2` basculera sur l'une ou l'autre interface disponible. Ici, c'est PA9/PA10 qu'on utilisera, et ce avec le routeur qui fait l'objet d'un autre article dans le présent numéro. La détection de débit ne se fait pas par l'intermédiaire de l'USART, mais via un compteur de fréquences utilisant PC6 (une autre fonctionnalité de l'Hydrabus). On peut alors procéder de deux manières, soit d'abord tester le débit avec PC6 puis connecter PA9 pour établir la connexion, soit brancher PC6 et PA9 ensemble directement. Les manipulations seront alors :

```
uart1> scan
Estimated baudrate : 38347

uart1> show
Device: UART1
Speed: 9600 b/s
```

```
Parity: none
Stop bits: 1

uart1> speed 38400
Final speed: 38408 b/s(0.03% err)

uart1> bridge
Interrupt by pressing user button.

Sending discover...
tuya-linux login: root
Password:
Sending discover...
Tuya Linux version 1.0
Jan  1 00:01:09 login[121]: root login on 'console'
#
```

Comme vous pouvez le constater, `scan` détecte bien la vitesse, même si elle est approximative, mais ne change pas la configuration. C'est `speed` qui doit alors être utilisé avant de passer en mode *bridge*. On se retrouve effectivement avec une console parfaitement utilisable sur le routeur et on pourra quitter ce mode en appuyant sur le bouton UBTN et revenir à l'interaction classique avec l'Hydrabus, pour des échanges en notation hexadécimale par exemple.

Mais ce n'est pas tout. Nous avons également à disposition, et c'est valable pour d'autres modes/bus (SPI, i²c, etc.), un déclencheur capable de réagir à l'arrivée d'une chaîne de caractères ou d'octets bien spécifiques. Ce mécanisme de *trigger*, s'il est déclenché, passera automatiquement la broche PB3 à l'état haut. L'intérêt de cette fonctionnalité est de pouvoir déclencher une action externe à l'Hydrabus, par exemple pour entamer une capture avec un analyseur logique ou un oscilloscope. Le *trigger* s'utilisera ainsi :

```
uart1> trigger filter "GD25Q128"
Current trigger data :
47 44 32 35 51 31 32 38     |  GD25Q128

uart1> trigger start
Interrupt by pressing user button.

uart1>
```

Ici, la condition sera de recevoir la chaîne de caractères `"GD25Q128"` et il ne s'agit que d'un simple exemple en rapport avec la cible testée, mais on imagine sans peine les possibilités que cela ouvre. En particulier sachant que des déclencheurs complexes de ce type sont encore inaccessibles en utilisant des outils comme PulseView. Une amélioration possible pour l'Hydrabus serait la possibilité de définir plusieurs conditions qu'on pourrait combiner à l'aide d'un *ET* ou d'un *OU* logique. En parlant d'amélioration, la détection du format de données (parité et bit(s) de stop) serait également fort plaisante, même si 8N1 représente généralement 90 % des cas.

3. ANALYSEUR LOGIQUE, SMARTCARDS, JTAG, SWD

Regroupons ici des points et fonctionnalités que je me permettrai de qualifier de secondaires, car même si elles sont effectivement présentes et utiles dans certaines situations, on peut les voir comme des solutions de secours ou des bonus. L'une d'entre elles concerne la compatibilité avec le protocole SUMP créé il y a bien longtemps par Michael Poppitz pour un analyseur logique **[3]** de sa création et devenu, depuis, un standard dans le monde *open source*. SUMP est utilisé par le regretté *Openbench Logic Sniffer* et quelques autres projets, dont l'Hydrabus.

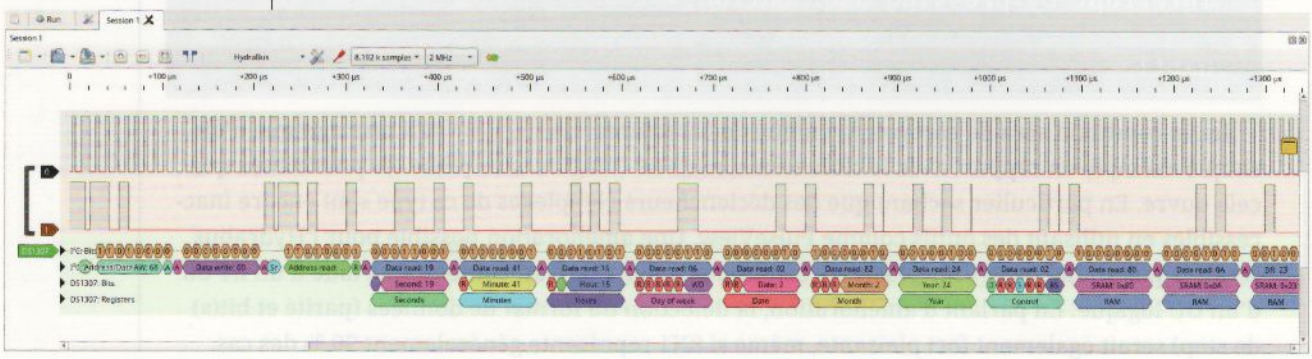
Il vous sera donc possible de capturer des signaux sur 16 canaux (broches PC0 à PC15) avec une fréquence d'échantillonnage allant jusqu'à 2 MHz. SUMP étant parfaitement connu d'outils comme PulseView, la simple sélection d'un périphérique « Openbench Logic Sniffer & SUMP compatibles (ols) » dans la configuration et un scan de périphérique, après sélection du bon port série, vous donnera accès aux fonctionnalités. Seul problème, la taille des données capturées se limite à 8 kiloéchantillons sur 8 canaux et à 4 sur 16. C'est peu, mais suffisant, comme le montre la capture ci-contre. Et surtout, c'est mieux que rien si l'on n'a pas son module Cypress FX2 sous la main.

Dans le même ordre d'idées, nous avons la compatibilité avec OpenOCD permettant de fonctionner comme une sonde JTAG ou une interface SWD. Il ne sera pas même nécessaire d'utiliser une version spécifique de l'outil (contrairement au Picoprobe **[4]**) puisque l'Hydrabus est tout simplement compatible sur ce point avec le Bus Pirate déjà pris en charge par OpenOCD depuis longtemps.

Un simple fichier de configuration, inspiré de celui livré avec les sources du *firmware* fera l'affaire :

```
source [find interface/buspirate.cfg]
buspirate port /dev/ttyACM0
transport select swd
source [find target/stm32f4x.cfg]
reset_config srst_nogate connect_assert_srst
```

Hydrabus permet également une utilisation en analyseur logique, avec Pulseview, grâce à sa compatibilité OLS/SUMP. Les performances sont très modestes, mais si on ne dispose d'aucun autre outil, cela peut parfois vous sortir une épine du pied.



On peut ensuite directement l'utiliser très classiquement avec :

```
$ openocd -f ./hydrabusSWD.cfg
[...]
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : Buspirate SWD Interface ready!
Info : This adapter doesn't support configurable speed
Info : SWD DPIDR 0x2ba01477
Info : [stm32f4x.cpu] Cortex-M4 r0p1 processor detected
Info : [stm32f4x.cpu] target has 6 breakpoints, 4 watchpoints
Info : starting gdb server for stm32f4x.cpu on 3333
Info : Listening on port 3333 for gdb connections
Info : [stm32f4x.cpu] external reset detected
```

Pour ensuite accéder à l'interface avec Telnet et/ou GDB :

```
$ telnet 127.0.0.1 4444
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Open On-Chip Debugger
> targets
    TargetName     Type       Endian TapName            State
--  -------------- ---------- ------ ------------------ -------
0* stm32f4x.cpu   cortex_m   little stm32f4x.cpu       reset
```

Ceci a fonctionné parfaitement avec ma NUCLEO-F401RE (ainsi qu'une *Blue Pill*) même si j'ai fini par prendre pour habitude de brancher l'Hydrabus en USB juste avant d'invoquer OpenOCD, pour éviter tantôt d'étranges problèmes de connexion. Avec une Raspberry Pico (`rp2040.cfg`) en revanche, rien à faire, et ce pour une raison que je n'ai pas encore réussi à clairement identifier. Mais, comme pour l'analyseur logique, on peut raisonnablement considérer que ceci n'est pas une fonctionnalité cruciale, car, lorsqu'on *debug*, on utilise généralement des outils dédiés dans un environnement maîtrisé. Ici, l'Hydrabus est davantage un accessoire d'appoint ou de secours en cas de problème.

Et enfin, nous avons mon petit centre d'intérêt du moment, qui m'a d'ailleurs poussé à proposer une contribution au projet : les *smartcards*. Un grand nombre de microcontrôleurs STM32 proposent cette interface qui n'est autre qu'une liaison série asynchrone half duplex utilisant une unique ligne de données.

```
> smartcard
Device: SMARTCARD1
Speed: 9600 b/s
Parity: even
Stop bits: 1.5
```

```
Convention: normal
Prescaler: 12 / 3.50mhz

smartcard1> atr
Timing information:
Fi=372, Di=4, 93 cycles/ETU
37634 bits/s at 3.50mhz, 53763 bits/s for fMax=5 MHz)
3B F8 13 00 00 81 31 FE  45 4A 43 4F 50 76 32 34  |  ;.....1.EJCOPv24
31 B7                                             |  1.
```

Comme vous pouvez le voir, le *firmware* procède à un décodage partiel de l'ATR (*Answer to Reset*), la « signature » de la carte et, comme pour les autres interfaces, différents éléments de configuration et commandes sont à votre disposition. Notez que `[` et `]` prennent encore une fois un sens différent dans ce contexte, puisqu'il s'agit de contrôler la ligne de *reset* de la carte. Il est possible d'envoyer (`write`) et recevoir (`read`) des données, mais comprenez bien qu'il s'agit là d'une communication au plus bas niveau et non d'un simple échange d'APDU.

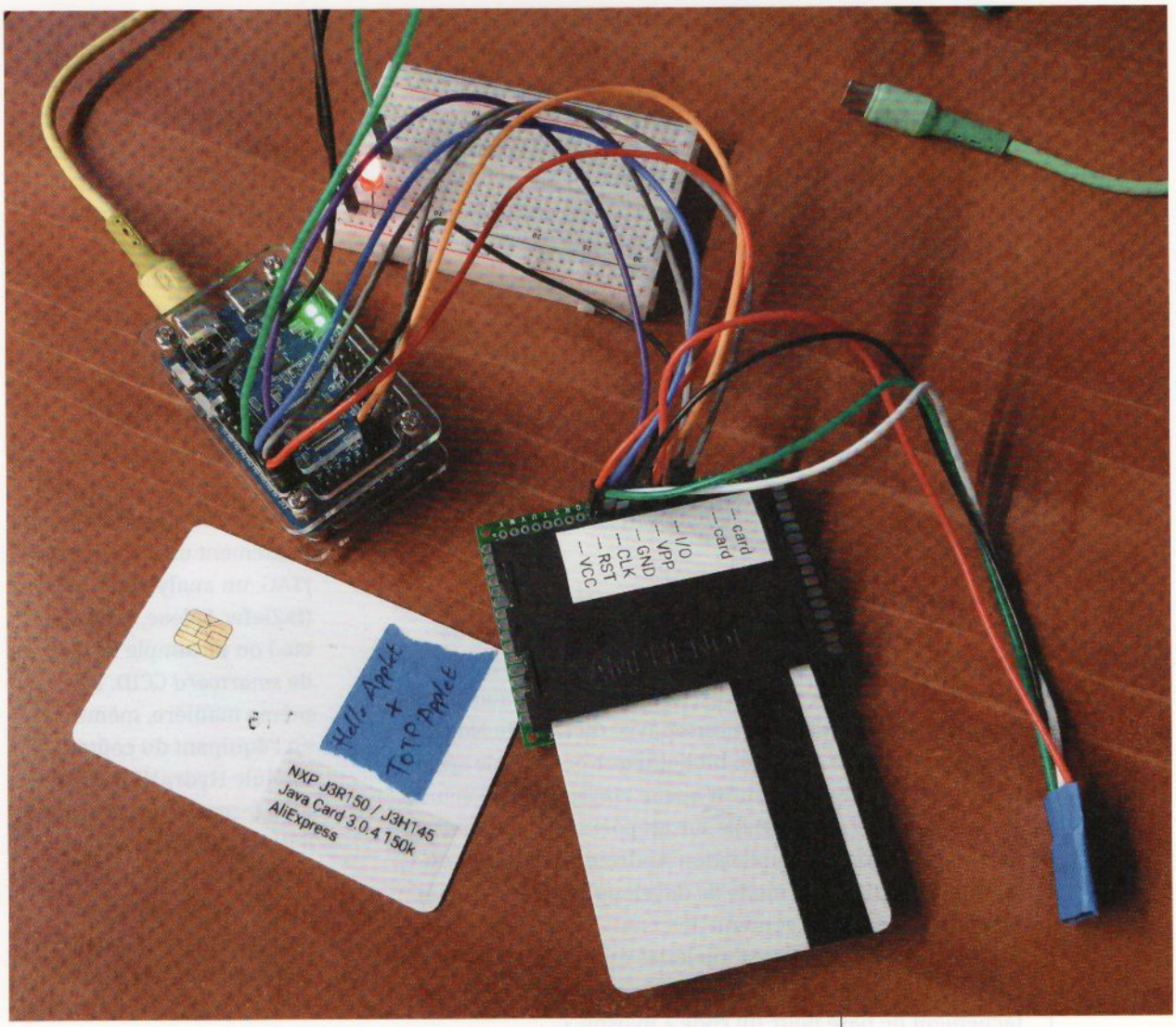
Ainsi, il est possible de réitérer la réception de l'ATR avec :

```
smartcard1> ]%[r:18
RST DOWN
DELAY: 1 ms
RST UP
READ: 0x3B 0xF8 0x13 0x00 0x00 0x81 0x31 0xFE 0x45
  0x4A 0x43 0x4F 0x50 0x76 0x32 0x34 0x31 0xB7
```

Ou encore sélectionner l'application de la carte via, en encodant manuellement l'APDU en TPDU (ici une NXP J2A081 en T=1) :

```
smartcard1> 0x00 0x00 0x0f 0x00 0xa4 0x04 0x00 0x0a 0xf2
  0x76 0xa2 0x88 0xbc 0xde 0xad 0xbe 0xef 0x01 0x94 % r:6
WRITE: 0x00 0x00 0x0F 0x00 0xA4 0x04 0x00 0x0A 0xF2 0x76
  0xA2 0x88 0xBC 0xDE 0xAD 0xBE 0xEF 0x01 0x94
DELAY: 1 ms
 READ: 0x00 0x00 0x02 0x90 0x00 0x92
```

Le code tel qu'il existe pour l'instant nécessite une lecture du nombre exact d'octets à recevoir et c'est une limitation que mon patch vise à corriger (PR en attente de *merge* à ce jour) en introduisant une commande `tread` utilisant un *timeout* configurable et retournant les octets lus avant ce délai, même si leur nombre n'est pas celui demandé. À noter également, la broche PA5, correspondant à VCC pour la *smartcard*, n'est pas forcément en mesure de fournir le courant nécessaire à la carte (du moins selon ISO7816), mieux vaut donc connecter directement 3V3 ou éventuellement passer par un MOSFET ou une solution similaire.



Une importante différence entre le Bus Pirate et l'Hydrabus concerne la présence de l'interface smartcard incluse de base dans de nombreux MCU STM32. Le support est relativement rudimentaire dans le firmware Hydrabus, mais suffisant pour tester une carte et/ou échanger quelques APDU.

4. MODE BINAIRE ET SCRIPTS PYTHON

Comme nous venons de le voir, interagir avec l'Hydrabus se fait via une interface série prenant en charge des commandes qu'on qualifierait de textuelles. Pour automatiser l'utilisation du périphérique, cependant, on préférera un autre mode d'utilisation : le mode binaire ou BBIO. Là, il ne s'agit plus de texte, mais d'entiers 8 bits qui sont utilisés pour donner des ordres à l'Hydrabus. Pour entrer en mode binaire, il suffit d'envoyer 20 octets à zéro (0x00) et le périphérique confirmera le changement en retournant la chaîne « `BBIO1` ». Ceci fait, la logique de fonctionnement est la même avec, pour chaque mode, non pas une commande, mais un octet spécifique à utiliser. 0x01 pour le SPI, 0x03 pour l'UART, 0x07 pour *smartcard*, etc.

Une fois le mode changé, on retrouve les mêmes opérations qu'en mode textuel, mais déclinées sous forme de valeur 8 bits, qu'il s'agisse de commandes ou du réglage de paramètres de configuration. En mode SPI par exemple, l'octet `0x6s` réglera la vitesse du bus, où `s` détermine la fréquence souhaitée avec `0` 320 kHz, `1` 650 kHz, `4` 5,25 MHz, `6` 21 MHz, etc. Chaque instruction possède ou non des arguments qui sont spécifiés sous la forme d'octets qui suivent l'octet de commande.

Les valeurs retournées sont également dépendantes du mode et/ou de la commande. Pour les instructions réglant des paramètres, 0x01 est généralement la valeur qui confirme la bonne exécution de l'opération. Pour les commandes retournant des données, typiquement une lecture sur un bus ou une interface, ce 0x01 de confirmation peut également précéder les données. Référez-vous au wiki GitHub du projet **[5]** pour connaître le détail.

Ce mode binaire est spécifiquement prévu pour l'écriture de programme reposant sur l'Hydrabus et évite de devoir gérer des chaînes de caractères. Le plus souvent, ce seront des scripts Python qui reposeront sur ce mode, mais rien ne vous empêche de l'utiliser avec du Lua, du Node.JS ou même du C/C++. Aucune bibliothèque ou module spécifique n'est nécessaire, il suffit que le langage puisse échanger des données à 115200 b/s sur un port série (on pourrait même envisager une application Android en ajoutant un module BT/BLE). Si ce mode de développement vous intéresse, jetez un œil au contenu de `contrib/` dans les sources du projet, vous y trouverez quelques exemples pour les bus SPI, i²c, CAN et même pour les *smartcards*. De quoi servir facilement de base pour un code « maison »...

5. POUR FINIR

En parlant de l'Hydrabus autour de moi, certaines personnes, qui se reconnaîtront, n'ont pas manqué de me faire remarquer que ce périphérique n'est, je cite, « qu'un simple *rip off* du Bus Pirate sur base STM32 ». Il est vrai que les similarités sont importantes, au point que certains protocoles (AVR ISP et JTAG/SWD par exemple) sont ouvertement identiques et compatibles. Pour ma part, je n'irai pas jusqu'à dire que c'est un « *rip off* », mais plutôt une déclinaison du concept introduisant quelques nuances. Le Bus Pirate n'a plus réellement de développement actif, la dernière version étant la v3b, la v4 n'ayant jamais abouti et la v5 étant pour le moins surprenante, pour dire les choses gentiment **[6]**. Je vois donc cela davantage comme un *fork* utilisant un MCU récent, que comme un *rip off*.

Hydrabus est un outil générique, « à tout faire », mais ne remplacera jamais totalement une vraie sonde JTAG, un analyseur logique (fx2lafw, Saleae, LA2016, etc.) ou un simple lecteur de *smartcard* CCID. De la même manière, même en l'équipant du coûteux module HydraNFC, vous n'en ferez pas un Proxmark 3 RDV4. Il n'en reste pas moins que c'est un outil qui pourra s'avérer très utile dans le sens où il peut tout faire, ou presque, et est parfait dans une utilisation « nomade ».

Ce matériel, malgré ses déjà nombreuses fonctionnalités, a énormément de potentiel. La présence du support SD/MMC à lui seul laisse envisager des choses intéressantes, comme la collecte et l'enregistrement de mesures et de données sur le long terme, via les bus, interfaces et ports (ADC) déjà supportés.

Un système de *log* existe, mais concerne l'interface elle-même, et non, par exemple, la création d'un fichier de données brutes (CVS, JSON, etc.). Et le concept est intéressant également dans l'autre direction, où un fichier présent sur une SD pourrait service de source pour une diffusion de longue durée. L'ajout d'autres interfaces peut également être envisagé, comme le pilotage/test de LED adressables ou l'émission de signaux infrarouges.

Ce qui nous amène justement au point le plus important concernant ce matériel : c'est aussi une plateforme de développement STM32 équipée d'un MCU avec des ressources conséquentes. Rien ne vous empêche donc d'en faire « une super Blue Pill » ou de vous en servir de base pour un nouveau projet. Mais je pense qu'en termes de développement, le mieux que vous puissiez faire est tout simplement de contribuer au projet, que ce soit sur le *firmware* lui-même ou via des applications annexes. **DB**

RÉFÉRENCES

[1] **https://github.com/hydrabus/hydrafw/releases**

[2] **https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads**

[3] **https://www.sump.org/projects/analyzer/**

[4] **https://github.com/raspberrypi/picoprobe**

[5] **https://github.com/hydrabus/hydrafw/wiki**

[6] **https://buspirate.com**

Chez votre marchand de journaux !

Et sur ed-diamond.com

NOUVEAU !

LINUX PRATIQUE N°142

FRAIS DE PORT OFFERTS !*

* Offre valable sur les publications en kiosque pour toute livraison en France Métropolitaine.

Également disponible en version lecture numérique Flipbook HTML5**

** L'offre Flipbook HTML5 est réservée aux clients particuliers.

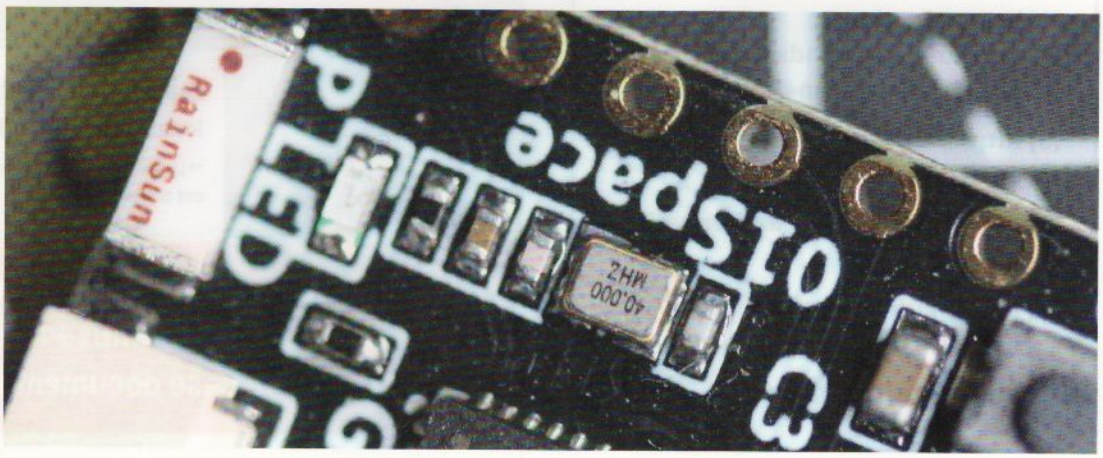
Retrouvez ce nouveau numéro, ainsi que l'intégralité de Linux Pratique sur notre base documentaire :

CONNECT
LA DOCUMENTATION TECHNIQUE DES PROS DE L'IT
connect.ed-diamond.com

ARDUINO + ESP-IDF + OTA + SECURE BOOT : LE MEILLEUR DES DEUX MONDES

Denis BODOR

Lorsqu'il s'agit d'utiliser un microcontrôleur de la famille ESP32, deux approches sont généralement raisonnablement envisageables. La première consiste à utiliser l'environnement dédié proposé par Espressif, ESP-IDF, et la seconde à reposer sur un niveau d'abstraction supérieur via le framework, et implicitement, l'environnement Arduino. L'une assure une maîtrise totale du développement et l'autre, une plus grande facilité d'implémentation du projet que vous avez en tête. Le coût du choix de l'une ou de l'autre solution est, pour la première, une complexité accrue et pour la seconde, un environnement relativement limité, sinon simpliste. Il existe cependant une voie du milieu qu'il est important de considérer...



– Arduino + ESP-IDF + OTA + Secure Boot : le meilleur des deux mondes –

Nous avons déjà soulevé cette problématique dans un précédent numéro (Hackable 24 **[1]**), mais nous étions en 2018 et depuis, l'environnement Espressif a subi au moins deux mises à jour majeures, changeant radicalement l'approche à adopter. En parallèle à cela, l'environnement Arduino a également évolué, en proposant une solution entièrement en ligne de commande (voir Hackable 46 **[2]**), fiable et complémentaire à l'utilisation d'un IDE plus que modeste (que ce soit en version 2.x ou 1.8.x). Ce à quoi s'ajoute la solution PlatformIO, offrant encore une autre voie pour développer, soit avec l'ESP-IDF, soit le *framework* Arduino.

Pourquoi donc alors revenir sur le sujet aujourd'hui alors que nous avons l'embarras du choix en termes de préférences de développement ? Tout simplement, car dans certaines situations, aucun environnement seul n'apporte de réponse à la fois simple et efficace. L'une de ces situations est la suivante : vous souhaitez protéger efficacement votre *firmware* et, dans le même temps, reposer sur le mécanisme de mise à jour OTA (*Over The Air*). Là, nous avons besoin des mécanismes présents dans ESP-IDF pour signer le *firmware* mais qui sont absents du support Arduino, et n'avons pas forcément envie de réimplémenter tout le système de mise à jour OTA via UDP d'Arduino, ou de déployer une infrastructure plus complexe (serveur HTTPS) en se basant sur les exemples ESP-IDF **[3]**. Ce que nous voulons, c'est produire un *firmware* signé, dans un contexte de *boot* sécurisé, tout en bénéficiant de la mise à jour « facile » avec le socle Arduino (`espota.py`).

Dans ce scénario, chaque environnement apporte sa contribution. Nous disposons à la fois de la facilité d'Arduino et de la sécurité du socle plus « bas niveau » de l'ESP-IDF. Ceci devient possible dès lors que l'on comprend que le support ESP32 dans Arduino utilise en réalité l'ESP-IDF en coulisse ou, plus exactement, que le support lui-même **est** l'ESP-IDF, agrémenté d'un composant logiciel permettant d'utiliser le *framework* Arduino. Oui, c'est tordu, mais le support Arduino pour les ESP32 est en réalité un support Arduino pour l'environnement Espressif. Mieux encore, il est possible d'utiliser ce support sans avoir recours à l'IDE Arduino (GUI ou CLI) en faisant appel au composant « espressif/arduino-esp32 » référencé dans le « *Espressif IDF Component Registry* » **[4]**.

1. UTILISER LE COMPOSANT ARDUINO DANS ESP-IDF

Nous partirons ici du principe que vous avez une installation fonctionnelle de l'environnement ESP-IDF. Ceci s'installe très facilement, que ce soit sur une machine GNU/Linux, macOS ou Windows, avec une préférence bien logique pour la première option (x86, amd64 ou ARM importe peu). L'un des points très appréciables de cet environnement est de ne pas perturber votre système et de s'intégrer sans avoir recours à une véritable installation nécessitant des permissions super-utilisateur. Tout se passe par la modification de certaines variables d'environnement, en appelant simplement un script `export.sh` placé dans le répertoire où se trouve l'ESP-IDF. Définir un alias avec quelque chose comme `alias initidf = source ~/chemin/export.sh`, directement dans votre `~/.bashrc`, vous facilitera grandement la vie. À la simple invocation de cette commande, vous voici avec l'ESP-IDF près à servir et à se plier à vos désirs.

Pour faire connaissance avec le composant Arduino, commençons par créer un nouveau projet en invoquant la principale commande de l'environnement et en définissant notre plateforme cible :

```
$ cd ~/chemin/kkpart
$ idf.py create-project arduidf
$ cd arduidf
$ idf.py set-target esp32
```

Nous nous retrouvons avec une arborescence comprenant, entre autres, un sous-répertoire `main/` contenant `arduidf.c` et `CMakeLists.txt`. Un code source C n'est pas adapté à Arduino qui utilise principalement C++, nous commençons donc par renommer le fichier source en `arduidf.cpp` et par ajuster le contenu du fichier `CMakeLists.txt` utilisé par CMake dans le processus de construction :

```
idf_component_register(
        SRCS "arduidf.cpp"
        INCLUDE_DIRS "."
)
```

La gestion de composants ESP-IDF permet d'ajouter des éléments permettant de prendre en charge aussi bien du matériel spécifique (comme des écrans et des capteurs) que des briques logicielles (mDNS, accès ChatGPT, WolfSSL, etc.). Un site officiel référence la plupart de ces composants et c'est là que nous trouvons notre « Arduino core » **[4]**. Une simple commande permet de l'ajouter à notre projet :

```
$ idf.py add-dependency "espressif/arduino-esp32^3.0.0-alpha2"
Executing action: add-dependency
Created "/home/denis/SRC/C/ESP32/plop/main/idf_component.yml"
Successfully added dependency
  "espressif/arduino-esp32^3.0.0-alpha2" to component "main"
Done
```

Ceci aura pour effet de créer le fichier `main/idf_component.yml`, dont le contenu se résume à :

```
dependencies:
  espressif/arduino-esp32: "^3.0.0-alpha2"
  idf:
    version: ">=4.1.0"
```

Dès lors, toute opération sur le projet (construction, configuration, reconfiguration, etc.) s'assurera automatiquement de la présence et du maintien à jour de ce composant, sans que nous ayons à intervenir. Exemple :

```
$ idf.py reconfigure
[...]
Processing 4 dependencies:
[1/4] chmorgan/esp-libhelix-mp3 (1.0.3)
```

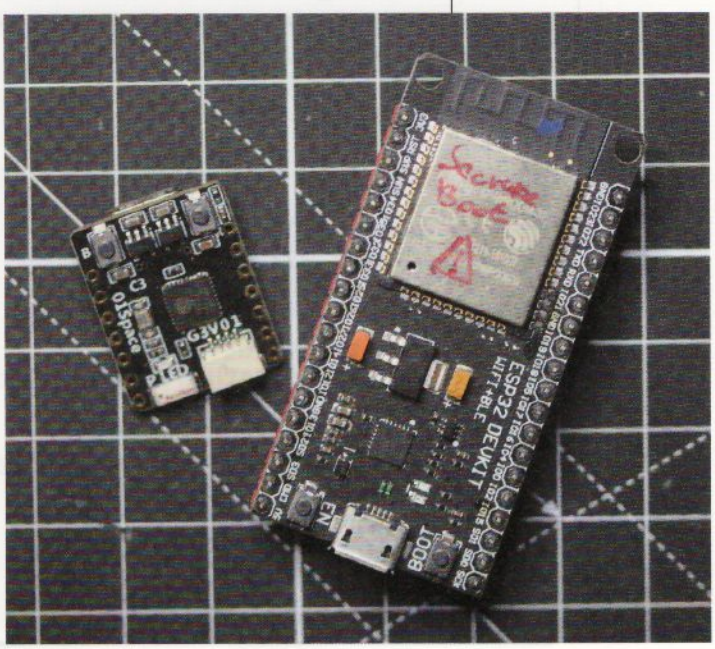
```
[2/4] espressif/arduino-esp32 (3.0.0-alpha2)
[3/4] espressif/mdns (1.2.2)
[4/4] idf (5.1.2)
[...]
```

Notez que les dépendances vers « chmorgan/esp-libhelix-mp3 » et « espressif/mdns » proviennent directement de l'ajout de « espressif/arduino-esp32 », et sont parfaitement gérées de la même manière.

Mais nous n'en avons pas fini avec la configuration. En effet, ajouter un composant ne signifie pas que la configuration du projet sera automatiquement adaptée en conséquence et l'Arduino core nécessite qu'un certain nombre de fonctionnalités soient activées/ajustées. Ceci contraste avec le support ESP32 pour Arduino où l'ESP-IDF sous-jacent a déjà été configuré pour nous.

Pour adapter la configuration, nous utilisons `idf.py menuconfig` et naviguons dans les menus pour modifier les points suivants :

- ***Arduino Configuration → Autostart Arduino setup and loop on boot*** (`AUTOSTART_ARDUINO`) : ceci permet un comportement similaire à celui des croquis Arduino avec les fonctions `setup()` et `loop()` (cf. plus loin dans l'article).
- ***Component config → mbedTLS → TLS Key Exchange Methods → Enable pre-shared-key ciphersuites*** (`MBEDTLS_PSK_MODES`) et ***Component config → mbedTLS → Enable PSK based ciphersuite modes*** (`MBEDTLS_KEY_EXCHANGE_PSK`) : certaines fonctionnalités intégrées au *framework* Arduino nécessitent la présence de protocoles cryptographiques de la famille TLS-PSK pour fonctionner (typiquement `WiFiClientSecure`). Cette option active ces protocoles.
- ***Component config → ESP System Settings → Watch CPU1 Idle Task*** (`ESP_TASK_WDT_CHECK_IDLE_TASK_CPU1`) doit être **désactivé** (actif par défaut) sous peine d'avoir des avertissements (voire des *reboots*) de la part du *watch dog* (WDT). Celui-ci, qui constitue un mécanisme de sécurité permettant de s'assurer que le code ne reste jamais bloqué dans une boucle, ne doit être actif **que** pour le cœur 0 où la pile réseau fonctionne. Le cœur 1, dédié aux croquis utilisateur, doit permettre l'exécution de code potentiellement bloquant (voir le ticket dans ce sens sur GitHub **[5]**).



Voici les pauvres victimes de ces expérimentations. À droite, un devkit ESP32 très courant et à gauche, un minuscule module à base d'ESP32-C3 avec cœur RISC-V.

- ***Serial flasher config → Flash size*** (`ESPTOOLPY_FLASHSIZE`) **→** ***4 MB*** (`ESPTOOLPY_FLASHSIZE_4MB`) ou plus : pour pouvoir utiliser l'OTA, la plateforme doit disposer d'au moins 4 Mio de flash et la configuration du projet doit refléter le volume effectivement présent dans le matériel. Utiliser une configuration spécifiant un volume de flash inférieur à la réalité n'est théoriquement pas un problème, le *bootloader* vous affichera simplement un avertissement. En revanche, spécifier davantage de flash qu'il n'en existe conduira forcément à un crash.
- ***Partition Table → Partition Table → Factory app, two OTA definitions*** (`PARTITION_TABLE_TWO_OTA`) : pour pouvoir utiliser la mise à jour OTA, le schéma de partitionnement doit comprendre des zones dédiées à cet effet. Cette option est identique aux choix qu'il est possible de faire dans l'IDE Arduino concernant l'utilisation de la flash des ESP32.

Note : il est probable que `idf.py menuconfig` vous affiche une erreur avec un message spécifiant que `CONFIG_FREERTOS_HZ` (la fréquence du *tick* système de FreeRTOS) ne possède pas une valeur compatible avec le composant « arduino-esp32 ». Si tel est toujours le cas (ESP-IDF 5.1 et « arduino-esp32 » 3.0.0-alpha2), éditez simplement `sdkconfig`, cherchez `CONFIG_FREERTOS_HZ` et changez la valeur de `100` à `1000`. Ceci aura pour effet de modifier la période entre les interruptions système de 10 ms à 1 ms et donc d'avoir la précision nécessaire pour la gestion des temporisations, au détriment des performances. Enregistrez et relancez `idf.py menuconfig`, et tout devrait rentrer dans l'ordre.

Une fois tout ceci dûment configuré, votre projet sera prêt et vous pourrez débuter le développement dans le fichier `arduidf.cpp` qui tiendra lieu de fichier `.ino` dans un environnement Arduino. Notre code de démonstration sera relativement modeste, du moins par rapport à un équivalent reposant uniquement sur ESP-IDF et consistera simplement en une implémentation « maison » d'une mise à jour OTA *over* UDP avec protection par mot de passe. Celui-ci débute avec la fonction de gestion de l'OTA, justement :

```
#include <Arduino.h>
#include <WiFi.h>
#include <ArduinoOTA.h>

unsigned long previousMillis = 0;

void confOTA() {
    // Port 3232 (défaut)
    ArduinoOTA.setPort(3232);

    // Hostname défaut
    ArduinoOTA.setHostname("arduidf");

    // Mot de passe pour OTA
    ArduinoOTA.setPassword("123456");

    // Lancé au début de la MaJ
    ArduinoOTA.onStart([]() {
            Serial.println("/!\\ MaJ OTA");
            });
```

```
    // Lancé en fin MaJ
    ArduinoOTA.onEnd([]() {
            Serial.print("\n/!\\ MaJ terminee ");
            });

    // Lancé lors de la progression de la MaJ
    ArduinoOTA.onProgress([](unsigned int progress, unsigned int total) {
            Serial.printf("Progression: %u%%\r", (progress / (total / 100)));
            });

    // En cas d'erreur
    ArduinoOTA.onError([](ota_error_t error) {
            Serial.printf("Erreur[%u]: ", error);
            if (error == OTA_AUTH_ERROR)
                Serial.println("OTA_AUTH_ERROR");
            else if (error == OTA_BEGIN_ERROR)
                Serial.println("OTA_BEGIN_ERROR");
            else if (error == OTA_CONNECT_ERROR)
                Serial.println("OTA_CONNECT_ERROR");
            else if (error == OTA_RECEIVE_ERROR)
                Serial.println("OTA_RECEIVE_ERROR");
            else if (error == OTA_END_ERROR)
                Serial.println("OTA_END_ERROR");
            else Serial.println("Erreur inconnue");
            });

    // Activation fonctionnalité OTA
    ArduinoOTA.begin();
}
```

Puis implémente très classiquement les fonctions `setup()` :

```
void setup() {
    int count = 0;

    Serial.begin(115200);

    Serial.print(F("Connexion Wifi AP..."));
    WiFi.mode(WIFI_STA);
    WiFi.begin("monAPwifi", "mot2passeWIFI");
    while(WiFi.status() != WL_CONNECTED && count<=16) {
        delay(500);
        Serial.print(".");
        count++;
    }
    if(count>16) {
        Serial.println("Erreur connexion Wifi ! Reboot...");
        ESP.restart();
    }
```

```
    Serial.println("ok");
    Serial.print(F("Mon adresse IP: "));
    Serial.println(WiFi.localIP());

    // configuration OTA
    confOTA();
}
```

et `loop()` :

```
void loop() {
    unsigned long currentMillis = millis();

    if(currentMillis - previousMillis >= 5000) {
        previousMillis = currentMillis;
        Serial.println("plop");
    }

    ArduinoOTA.handle();
}
```

Nous pourrons ensuite construire le *firmware* et flasher la plateforme en invoquant `idf.py -p /dev/ttyUSB1 flash monitor` où `ttyUSB1` désigne le port série où est connectée la carte via USB. Une fois ce code fonctionnel, comme le montrera très certainement la sortie du moniteur série invoquée automatiquement (grâce à `monitor`), nous pourrons voir l'ESP32 se connecter en Wi-Fi et boucler paisiblement sur l'affichage régulier de la chaîne `"plop"`. En coulisse cependant, nous sommes en attente d'une connexion pour la mise à jour OTA et, pour ce faire, nous devrons utiliser le script Python normalement intégré au support ESP32 pour Arduino. Celui-ci n'est pas intégré à l'ESP-IDF ni aux éléments du composant « arduino-esp32 » (il devrait), mais nous pouvons le récupérer directement depuis l'environnement Arduino (`~/.arduino15/packages/esp32/hardware/esp32/3.0.0-alpha2/tools/espota.py` par exemple) ou via le dépôt GitHub du projet **[6]**. Ce script `espota.py` pourra être placé à la racine de notre projet et invoqué directement pour procéder à la mise à jour :

```
$ ./espota.py -i arduidf.local -a 123456 -f build/arduidf.bin
Sending invitation to arduidf.local
Authenticating...OK
Uploading.............................
[...]
.........................
```

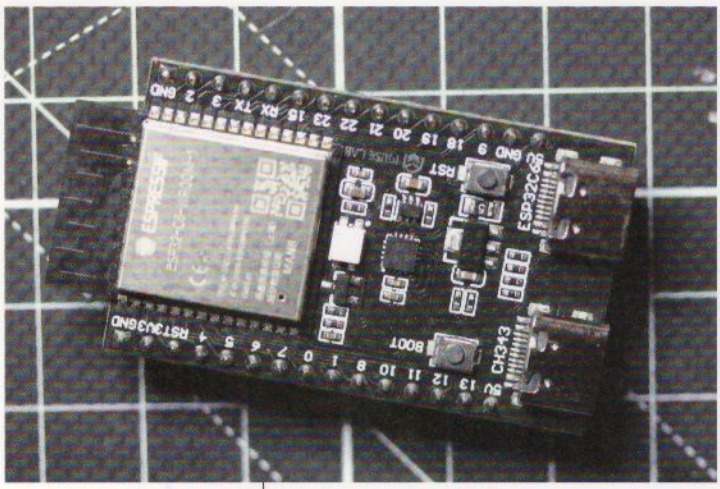
Les arguments utilisés sont, respectivement, le nom d'hôte de l'ESP32 (`-i`), le mot de passe OTA (`-a`) et le binaire à utiliser (`-f`), produit lors de la construction (avec `idf.py all`, par exemple). Et bien entendu, si tous ces paramètres sont corrects, nous procéderons à la mise à jour du *firmware* exactement comme nous pourrions le faire depuis l'IDE Arduino. Nous nous retrouvons donc avec un environnement plus « technique », propre à l'ESP-IDF, mais avec la facilité d'utilisation du *framework* Arduino.

Avant d'attaquer la suite, bien plus intéressante, prenons le temps de jeter un œil à ce qui se passe en coulisse et ce qui fait la différence, en termes de code, entre un projet ESP-IDF standard et un autre, utilisant le composant « arduino-esp32 ». Pour cela, il suffit de se pencher sur le fichier `managed_components/espressif__arduino-esp32/cores/esp32/main.cpp`, qui est le vrai point d'entrée (ou presque) du code exécuté. En en retirant les éléments non critiques, nous avons une tâche FreeRTOS appelant `setup()` puis bouclant (`for`) sur des appels successifs à `loop();` :

```
void loopTask(void *pvParameters)
{
    if(shouldPrintChipDebugReport()){
        printBeforeSetupInfo();
    }

    setup();

    if(shouldPrintChipDebugReport()){
        printAfterSetupInfo();
    }

    for(;;) {
        if(loopTaskWDTEnabled){
            esp_task_wdt_reset();
        }
        loop();
        if (serialEventRun) serialEventRun();
    }
}
```

Pour exécuter cette tâche, le vrai *main* FreeRTOS est implémenté juste après :

```
extern "C" void app_main()
{
    loopTaskWDTEnabled = false;

    initArduino();

    xTaskCreateUniversal(
      loopTask,
      "loopTask",
      getArduinoLoopTaskStackSize(),
      NULL,
      1,
      &loopTaskHandle,
      ARDUINO_RUNNING_CORE);
}
```



De plus en plus de devkits « moyen de gamme » proposent désormais une connectivité USB-C. Celui-ci, basé sur ESP32-C6, offre même un connecteur pour le convertisseur USB/série CH343 et un second directement relié à l'ESP32 pour l'USB OTG.

Ceci peut constituer un point de départ intéressant, car il vous est parfaitement possible d'implémenter cela directement dans votre projet et donc de désactiver l'exécution automatique (`AUTOSTART_ARDUINO`) dans la configuration (`menuconfig`). Est-ce une bonne idée ou non ? À vous de voir, mais c'est une option parfaitement viable et proposée par le composant.

2. BOOT SÉCURISÉ : CE QU'IL N'EST PAS POSSIBLE D'AVOIR DANS L'IDE ARDUINO

Bien ! Nous avons la base de l'expérimentation et pouvons développer « en Arduino » avec l'ESP-IDF. Il est temps à présent d'ajouter ce pour quoi nous faisons tout cela et d'aborder le concept de *secure boot*. Le principe de base est relativement simple, tout comme l'objectif lui-même : ne permettre l'exécution d'un *firmware* **que** s'il est dûment autorisé. Si le *firmware* en question n'est pas considéré d'une origine sûre ou que le contenu de la flash est altéré dans le but de modifier le comportement de l'ensemble, alors l'exécution ne sera pas déclenchée. Pour arriver à un tel résultat, un mécanisme de signature est utilisé afin d'obtenir une chaîne de confiance : le code en ROM valide l'authenticité du *bootloader* et celui-ci vérifie la validité de l'application (le code utilisateur stocké en flash). Chaque vérification valide l'exécution de l'étape suivante, d'où la notion de chaîne.

Pour valider le code avant exécution, une signature électronique est utilisée, reposant sur une clé stockée sous la forme de « fusible électronique » (*eFuse*) au sein du microcontrôleur. Cette clé ne peut être écrite qu'une seule et unique fois, d'où la désignation de « fusible », et n'est jamais lisible. Elle est utilisée, en compagnie des données à vérifier, pour produire un condensé ou *hash* cryptographique qui sert de signature et est stockée avec les données. Si le moindre octet change dans les données ou que celles-ci sont simplement remplacées dans leur totalité, la signature ne correspond plus et l'exécution n'a pas lieu. Pour mettre à jour les données de façon valide, le seul moyen est d'utiliser la clé pour produire une signature correspondante valide, assurant que seules les personnes en possession de ce secret (la clé) peuvent produire un *firmware* exécutable. On parle alors de *boot* sécurisé ou *secure boot* en anglais.

Dans le cas de la famille de microcontrôleurs ESP32, deux versions de *boot* sécurisé existent. Le *Secure Boot V1* pour les ESP32 (tout court) avant la révision v3.0 (alias ECO3) de la puce, et le *Secure Boot V2* pour cette révision et les suivantes. Vous pouvez trouver la révision de votre microcontrôleur ESP32 en utilisant :

```
$ esptool.py -p /dev/ttyUSB1 chip_id
esptool.py v4.7.0
[...]
Detecting chip type... ESP32
Chip is ESP32-D0WDQ6 (revision v1.0)
[...]
```

Nous avons ici une révision v1.0 (ECO1) ne supportant donc pas *Secure Boot V2*. V1 et V2 diffèrent sur plusieurs points mineurs, dont l'algorithme de signature utilisé ou par certaines limitations spécifiques, mais l'architecture générale reste identique. *Secure Boot V1* utilise exclusivement ECDSA alors que *Secure Boot V2*, selon le microcontrôleur, peut utiliser RSA-PSS et/ou ECDSA. En condensant la documentation Espressif Systems, on peut résumer la compatibilité avec *Secure Boot V2*, par modèle d'ESP32, à la liste suivante :

- ESP32 : rev v3.0 (ECO3) et suivantes en RSA-PSS ;
- ESP32-S2 : toutes versions, rev v0.0 (ECO0) et v1.0 (ECO1) en RSA-PSS ;
- ESP32-S3 : toutes versions, rev v0.0 (ECO0), v0.1 (ECO1) et v0.2 (ECO2) en RSA-PSS ;
- ESP32-C2 : toutes versions, rev v1.0 (ECO1) et v1.1 (ECO2) en ECDSA ;
- ESP32-C3 : rev v0.3 (ECO3) et suivantes en RSA-PSS ;
- ESP32-C6 : toutes versions rev v0.0 et v0.1 (ECO1) en RSA-PSS ou ECDSA ;
- ESP32-H2 : toutes versions, rev v0.0, v0.1 (ECO1) et v0.2 (ECO2) en RSA-PSS ou ECDSA ;
- ESP32-P4 : toutes versions, rev v0.0 pour l'instant en RSA-PSS ou ECDSA.

L'évolution de V1 à V2 découle d'un changement physique de la puce (*wafer-level changes*) provoqué par la découverte de deux failles de sécurité (CVE-2019-17391 et CVE-2019-15894) concernant la possibilité, par un attaquant, de passer outre la vérification du *boot* sécurisé par injection de faute, voire de récupérer la clé (avec suffisamment de tentatives). L'une de ces failles (CVE-2019-17391) ne dispose d'aucune contre-mesure si ce n'est d'utiliser un ESP32 plus récent (ECO3 ou plus), ce qui est un gros problème. Notez que ces failles n'ont strictement rien à voir avec l'algorithme utilisé et que ECDSA (256 bits) offre un niveau de sécurité tout à fait équivalent à RSA-PSS (3070 bits). L'omniprésence de RSA-PSS sur les nouveaux microcontrôleurs (C2, S2, S3, etc.) est due à un gain de vitesse dans la vérification des signatures, en contrepartie d'une taille de clé plus importante.

Dans cet article, je me concentrerai cependant sur *Secure Boot V1* et l'ESP32 (tout court), car c'est indéniablement le microcontrôleur le plus courant de la famille et très probablement celui que vous avez dans vos tiroirs. Je toucherai bien entendu un mot à propos du *Secure Boot V2* plus loin, lorsque nous nous serons fait les dents sur la première version, avec des microcontrôleurs moins récents. La raison est toute simple : ces manipulations, mal exécutées, peuvent potentiellement rendre votre matériel inutilisable, donc autant prendre des risques avec des ESP32 que vous ne regretterez pas de perdre avant de passer aux choses sérieuses, quand les principes seront parfaitement assimilés.

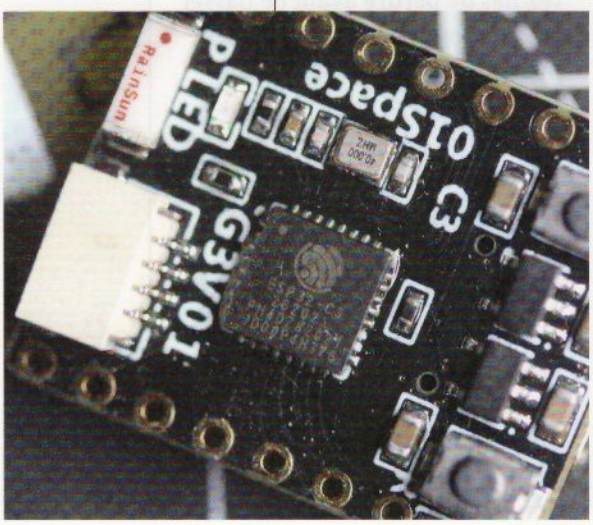
3. CONFIGURATION DU BOOT SÉCURISÉ

Vous l'avez compris, le *bootloader* est signé par une clé stockée en eFuse et lui-même intègre une clé publique permettant de vérifier la signature de l'application dont il est censé déclencher

l'exécution. Nous avons donc deux clés, une pour la signature du *bootloader*, vérifiée par la ROM et une pour l'application utilisateur.

Deux modes d'activation du *boot* sécurisé sont disponibles et celui utilisé par défaut repose sur le fait qu'au premier démarrage du *bootloader* configuré avec cette fonctionnalité, celui-ci génère sa clé aléatoirement grâce au générateur de nombres aléatoires intégré au microcontrôleur. Ceci fait, la clé en question est stockée en eFuse en désactivant la lecture et l'écriture directes, puis utilisée pour générer un condensé qui est ajouté en flash juste avant le *bootloader*. C'est la signature du *bootloader*. Celui-ci active alors définitivement la fonctionnalité en eFuse et éventuellement désactive certaines autres fonctionnalités considérées comme peu sûres (JTAG, par exemple). Les démarrages qui suivront provoqueront une vérification de la signature conditionnant l'exécution du *bootloader*.

Ceci signifie une chose très importante : dans ce mode d'activation, nous n'avons aucune connaissance de la clé et n'avons aucun moyen de la récupérer. Il nous est donc impossible de générer une signature valide ou, en d'autres termes, de mettre à jour le *bootloader*. La mise à jour de l'application est toujours possible puisque le *bootloader* intègre la clé publique permettant de vérifier la signature, mais ça s'arrête là.



Ce microcontrôleur ESP32-C3 possède un cœur RISC-V et utilise Secure Boot v2. Notez l'absence de composants dédiés pour la flash et pour l'USB, tout est directement géré nativement par le SoC.

L'autre mode d'activation consiste à choisir nous-mêmes la clé qui sera stockée en eFuse et utilisée pour la signature du *bootloader*, nous permettant alors des mises à jour si nécessaire. Ce n'est pas la méthode la plus sûre, mais nous gardons un peu plus la main sur le développement. Cette clé connue sera dérivée de celle permettant la signature de l'application sous la forme d'un condensé SHA-256 de sa partie privée. ECDSA repose en effet sur un chiffrement asymétrique, nous avons en réalité une paire de clés : une clé privée pour signer et une clé publique pour vérifier une signature. C'est cette dernière qui est embarquée dans le *bootloader* et qui ne nécessite aucune protection particulière. La clé privée, elle, doit être tenue strictement secrète et gardée en lieu sûr (cf. plus loin à propos de la signature à distance).

C'est ce second mode d'activation que nous allons utiliser ici et la première chose à faire est donc de générer une paire de clés ECDSA avec un outil présent dans tous les systèmes dignes de ce nom, `openssl` : `openssl ecparam -name prime256v1 -genkey -noout -out cle_de_signature.pem`. Le fichier obtenu, au format standard PEM, est temporairement conservé à la racine du projet et sera mis en sécurité plus tard. Nous pouvons désormais nous pencher sur la configuration avec `idf.py menuconfig` et nous rendre dans la section ***Security features*** pour ajuster/activer :

- ***Enable hardware Secure Boot in bootloader*** (`SECURE_BOOT`) pour utiliser le *boot* sécurisé ;
- ***Secure bootloader mode*** (`SECURE_BOOTLOADER_MODE`) sera réglé sur ***Reflashable*** (`SECURE_BOOTLOADER_REFLASHABLE`) pour nous permettre de mettre à jour le *bootloader* par la suite grâce à une clé connue, par opposition à ***One-time flash*** (`SECURE_BOOTLOADER_ONE_TIME_FLASH`), déclenchant la génération aléatoire au premier *boot* ;
- ***Sign binaries during build*** (`SECURE_BOOT_BUILD_SIGNED_BINARIES`) est activé par défaut et le reste, puisque nous voulons effectivement que l'application soit également signée ;
- ***Allow potentially insecure options*** (`SECURE_BOOT_INSECURE`) est également activé puisque nous sommes en phase de développement et ceci nous permet de conserver l'utilisation du JTAG si nécessaire, via l'activation de ***Allow JTAG Debugging*** (`SECURE_BOOT_ALLOW_JTAG`) ;
- et enfin, ***Secure boot private signing key*** (`SECURE_BOOT_SIGNING_KEY`) nous permet de spécifier notre fichier contenant la paire de clés ECDSA (`cle_de_signature.pem` ici), relativement à la racine du projet.

À noter que l'entrée ***Hardware Key Encoding*** (`SECURE_BOOTLOADER_KEY_ENCODING`) dépend de la plateforme visée. Certains ESP32 stockent en eFuse une version tronquée à 192 bits (3/4) de la clé de 256 bits dérivée (SHA-256) de la clé privée ECDSA. Vous pouvez connaître l'encodage utilisé par votre ESP32 en utilisant la commande `esptool.py chip_id` et en regardant la ligne « *Features* » où « *Coding Scheme None* » indique l'utilisation d'une clé de 256 bits non tronquée. Absolument aucun des ESP32 en ma possession n'utilise un encodage 3/4 et j'en possède de toutes les « époques ». Je suppose donc que ceci doit être relativement rare.

Une fois cette configuration faite et comme l'activation ne générera pas de clé automatiquement, il convient de construire le *bootloader* et, ce faisant, générer la clé dérivée de `cle_de_signature.pem`, avec `idf.py bootloader`. Il est cependant très probable que cela ne fonctionne pas, car l'ajout de ces fonctionnalités, en plus d'autres, occupant déjà un certain volume de flash (`BOOTLOADER_LOG_LEVEL_INFO` par exemple) fait que la taille du *bootloader* dépasse l'espace normalement alloué et écrase une partie de la table des partitions placée à l'adresse 0x8000. Un message d'erreur relativement explicite signale donc le problème :

```
"Error: Bootloader binary size 0x9dc0 bytes is
too large for partition table offset 0x8000.
Bootloader binary can be maximum 0x7000 (28672)
bytes unless the partition table offset is
increased in the Partition Table section of
the project configuration menu."
```

La table des partitions se trouve à 0x8000, mais le message nous indique que la taille maximum du *bootloader* est de 0x7000. Ceci est parfaitement normal puisque le binaire du *bootloader* est préfixé des 4096 octets (0x1000) de condensé formant la signature du binaire en question. Il faut donc une table de partition placée avec un décalage correspondant à la taille du *bootloader* plus 4096. Ici, 0xb000 fera parfaitement l'affaire et nous devrons alors retourner dans la configuration, menu ***Partition Table*** et ***Offset of partition table*** (`PARTITION_TABLE_OFFSET`) pour préciser cette valeur. Ceci ajusté, `idf.py bootloader`

nous génère enfin une image du *bootloader*, mais également le fichier contenant la clé dérivée : `build/bootloader/secure-bootloader-key-256.bin`. Celle-ci devra alors être inscrite manuellement dans l'eFuse prévu à cet effet avec :

```
$ espefuse.py -p /dev/ttyUSB1 burn_key secure_boot_v1 \
build/bootloader/secure-bootloader-key-256.bin
espefuse.py v4.7.0
Connecting....
Detecting chip type... ESP32

=== Run "burn_key" command ===
Sensitive data will be hidden (see --show-sensitive-info)
Burn keys to blocks:
- BLOCK2 -> [?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??
             ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ?? ??]
        Reversing the byte order
        Disabling read to key block
        Disabling write to key block

Burn keys in efuse blocks.
The key block will be read and write protected

Check all blocks for burn...
idx, BLOCK_NAME,        Conclusion
[00] BLOCK0             is not empty
  (written ): 0x00000004000000000000003500008000002a30aea40b9e8c00000000
  (to write): 0x00000000000000000000000000000000000000000000000000020100
  (coding scheme = NONE)
[02] BLOCK2             is empty, will burn the new value
.
This is an irreversible operation!
Type 'BURN' (all capitals) to continue.
```

Ceci est une étape critique, car l'opération est irréversible. L'outil `espefuse.py` vous présente les modifications qui seront apportées et vous demande de confirmer en saisissant le mot « `BURN` » et en validant. Deux eFuses sont modifiés, `BLOCK2` contiendra la clé et `BLOCK0` sera ajusté avec les bits 8 et 17 « brûlés » (`WR_DIS.BLOCK2` et `RD_DIS.BLOCK2` dans `esp_efuse_table.csv`) qui correspondent, respectivement, à la désactivation de l'écriture et de la lecture du contenu du `BLOCK2` afin de protéger la clé s'y trouvant. En confirmant l'opération, les eFuses sont alors inscrits définitivement :

```
BURN
BURN BLOCK2 - OK (write block == read block)
BURN BLOCK0 - OK (all write block bits are set)
Reading updated efuses...
Successful
```

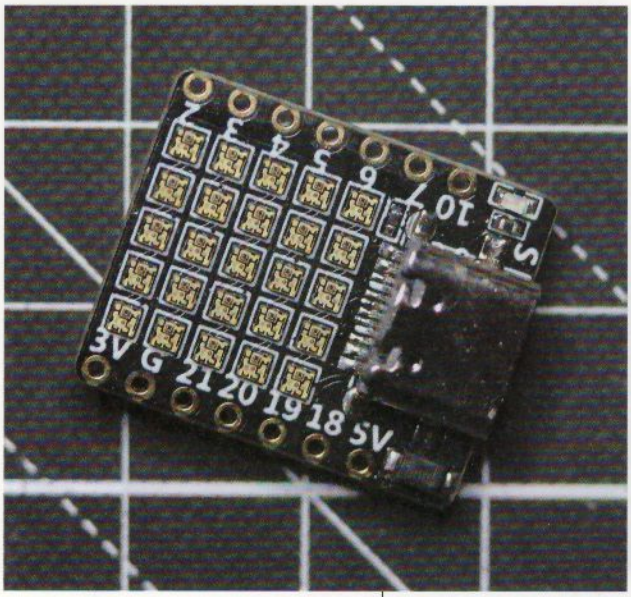
À présent que la clé est programmée, nous pouvons manuellement faire de même pour le *bootloader* lui-même avec :

```
$ esptool.py --port=/dev/ttyUSB1 \
  --after=no_reset write_flash \
  --flash_mode dio --flash_freq 40m
  --flash_size 4MB 0x1000
  build/bootloader/bootloader.bin
[...]
Configuring flash size...
Flash will be erased from 0x00001000 to 0x0000afff...
Compressed 40384 bytes to 25304...
Wrote 40384 bytes (25304 compressed) at 0x00001000
   in 2.7 seconds (effective 119.6 kbit/s)...
Hash of data verified.
Leaving...
Staying in bootloader.
```

La commande utilisée est directement affichée lors de la construction avec `idf.py bootloader`, tout comme celle concernant la programmation de la clé. Il faut simplement l'adapter en précisant le port série à utiliser. Vous remarquerez la présence de l'option `--after=no_reset`, empêchant un redémarrage automatique de la plateforme. Ceci nous permet de lancer le moniteur série avec `idf.py -p /dev/ttyUSB1 flash monitor` pour pouvoir observer ce premier démarrage en mode sécurisé, tout en écrivant l'application en flash :

```
[...]
W (860) secure_boot_v1: Using pre-loaded secure boot key in EFUSE block 2
I (867) secure_boot_v1: Generating secure boot digest...
I (895) secure_boot_v1: Digest generation complete.
I (895) boot: Checking secure boot...
I (895) efuse: Batch mode of writing fields is enabled
I (899) secure_boot_v1: blowing secure boot efuse...
I (905) secure_boot: Read & write protecting new key...
W (911) secure_boot: Not disabling JTAG - SECURITY COMPROMISED
I (918) secure_boot: Disable ROM BASIC interpreter fallback...
I (924) efuse: BURN BLOCK0
I (939) efuse: BURN BLOCK0 - OK (all write block bits are set)
I (939) efuse: Batch mode. Prepared fields are committed
I (942) secure_boot_v1: secure boot is now enabled for bootloader image
[...]
```

Nous constatons qu'effectivement la clé est chargée depuis `BLOCK2` et utilisée pour calculer et vérifier le condensé. Ceci fait, le nouveau mode de *boot* est définitivement activé par l'écriture de nouveaux bits dans `BLOCK0` (bit 196, alias `ABS_DONE_0`) et tout démarrage futur nécessitera la présence d'une signature valide. Notez au passage le message concernant le JTAG que nous souhaitons encore actif pour le moment, mais qui est considéré, de base, comme un niveau de sécurité insuffisant. Durant cette première étape, l'image utilisée pour le *bootloader*



Le module ESP32-C3FH4-RGB utilisé pour les tests de boot sécurisé version 2 dispose de 25 LED adressables. Amusant à première vue, mais de peu d'intérêt dans le cadre du présent article, et peut-être même de façon générale (d'où son utilisation pour les tests).

est `build/bootloader/bootloader.bin`, car l'activation du *boot* sécurisé va ajouter la signature calculée avec la clé en eFuse, mais il ne sera plus possible, par la suite, de faire de même. Si vous ajustez des paramètres, comme réduire le niveau de verbosité du *bootloader* par exemple, vous devrez flasher le fichier `build/bootloader/bootloader-reflash-digest.bin` et non `bootloader.bin`.

Nous pouvons à présent continuer de développer notre code (ou croquis) et le charger en flash comme nous le faisions précédemment avec `idf.py flash`. Ceci ne pose aucun problème puisque l'image sera automatiquement signée et le binaire porte exactement le même non (ici, `build/arduidf.bin`). `idf.py flash` ne tentera jamais d'écrire le *bootloader* si le *boot* sécurisé est actif sur la cible.

Une version non signée est également présente dans `build/`, c'est `arduidf-unsigned.bin` et nous pouvons très simplement faire un essai avec `esptool.py -p /dev/ttyUSB1 --after=no_reset write_flash --flash_mode dio --flash_freq 40m --flash_size 4MB 0x20000 build/arduidf-unsigned.bin`. Et effectivement, si nous surveillons le démarrage avec un `idf.py monitor` juste après :

```
[...]
I (521) esp_image: Verifying image signature...
E (521) secure_boot: image has invalid signature
  version field 0xffffffff (image without a signature?)
E (525) esp_image: Secure boot signature verification failed
I (531) esp_image: Calculating simple hash to check for corruption...
W (806) esp_image: image valid, signature bad
[...]
```

L'application n'est pas exécutée, car la signature n'est pas valide. Ceci est un *firmware* illégitime et le message du *bootloader* est assez amusant, je trouve : « *image valid, signature bad* » (vilaine signature, vilaine !). Il est d'ailleurs assez intéressant de comparer les deux binaires, avec un outil comme Binwalk par exemple (`binwalk -W arduidf-unsigned.bin arduidf.bin | tail`), et d'ainsi constater une différence en fin de fichier, là où la signature est ajoutée. Si nous

procédons de même avec le *bootloader* (`binwalk -W bootloader.bin bootloader-reflash-digest.bin | less -R`), c'est au début du fichier que l'on constate un décalage de 0x1000 octets (4 Kio), précisément là où commence le binaire dans une version sécurisée.

« Et l'OTA ? » demanderez-vous timidement. Eh bien, ceci fonctionne exactement de la même manière que précédemment, car nous flashons la version signée de l'application, avec `python3 espota.py -i arduidf.local -a 123456 -f build/arduidf.bin`, et le résultat sera donc strictement identique. Vous avons touché au but puisque nous avons à présent un projet disposant des facilités offertes par le *framework* Arduino, tout en interdisant l'exécution d'un *firmware* qui ne provient pas d'une source autorisée (nous).

4. PROCÉDER À LA SIGNATURE SUR UNE AUTRE MACHINE

Ici, nous sommes partis du principe que la personne procédant au développement était la même que celle en charge de la sécurité, censée signer les *firmwares*. Mais il arrive que ceci ne soit pas le cas pour des raisons évidentes de sécurité. L'ESP-IDF prend cette séparation de responsabilité en charge en permettant la signature « distante » (*remote* dans la documentation) ou, plus exactement, par un tiers.

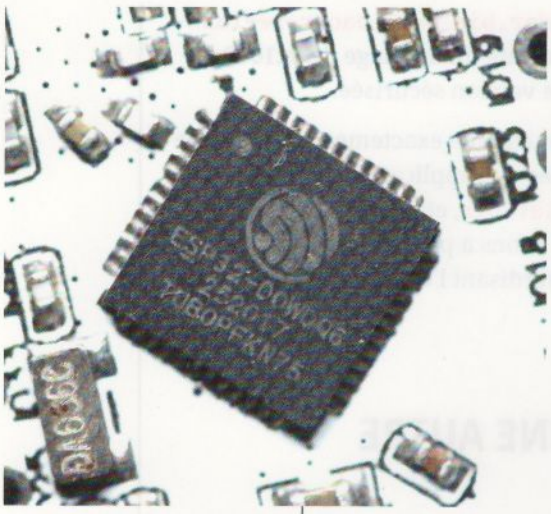
Le fichier PEM contenant la paire de clés ECDSA ne peut alors pas être partagé entre le développeur et le signataire. La partie publique doit être présente dans le *bootloader*, mais la partie privée reste hors de portée du développeur. L'astuce consiste donc à extraire la clé publique du fichier `cle_de_signature.pem` à l'aide d'un script Python livré avec l'environnement :

```
$ espsecure.py extract_public_key \
--keyfile cle_de_signature.pem \
cle_de_verification.bin
```

Ce nouveau fichier, `cle_de_verification.bin` est une simple représentation binaire de la clé et pourra être transmise au développement sans avoir à utiliser un canal sécurisé. Celui-ci devra alors ajuster la configuration du projet, en activant effectivement le *boot* sécurisé (`SECURE_BOOT`), mais en désactivant l'option ***Sign binaries during build*** (`SECURE_BOOT_BUILD_SIGNED_BINARIES`). Ceci change l'objet et le nom de l'entrée suivante du menu qui devient ***Secure boot public signature verification key*** (`SECURE_BOOT_VERIFICATION_KEY`) et permettra de préciser le fichier `cle_de_verification.bin`, relativement à la racine du projet.

Une fois le développement abouti, le résultat compilé sera transmis au signataire, qui utilisera :

```
$ espsecure.py sign_data --keyfile cle_de_signature.pem \
--version 1 binaire_du_developpeur.bin \
--output binaire_signe.bin
espsecure.py v4.7.0
Signed 982960 bytes of data from binaire_du_developpeur.bin
```



Le bon vieux ESP32 (tout court) possède également des fonctionnalités de boot sécurisé, mais deux vulnérabilités limitent grandement son utilisation pour des projets réellement critiques (même si l'exploitation des failles est relativement difficile). Il sera cependant parfaitement adapté pour appréhender cette technologie et se faire la main à peu de frais.

L'option `--output`, suivie du fichier de destination, peut être omise pour signer le binaire du développeur « sur place ». Le résultat pour alors être renvoyé au développeur, ou à une personne en charge de mettre à jour le *firmware*, que ce soit via USB (avec `esptool.py`) ou en OTA. Ce type de mécanisme permet de séparer les responsabilités en ayant autant d'acteurs que de tâches à effectuer et donc de renforcer encore davantage la sécurité.

Il est même possible de construire, sur cette base, une véritable PKI susceptible de supporter toute une flottille de périphériques ESP32, possédant chacun des clés spécifiques. L'ESP-IDF cependant ne met pas à disposition de solutions de ce type, mais ceci n'en reste pas moins parfaitement envisageable.

5. QUELQUES MOTS À PROPOS DE SECURE BOOT V2

Avant toute chose, il est important de préciser que le mécanisme général de sécurisation via une chaîne de confiance reposant sur des signatures est similaire en tout point au *Secure Boot V1*. Seuls des détails d'implémentation comme les structures de données et les algorithmes utilisés changent. On retrouve ainsi l'entrée ***Security features*** et le ***Enable hardware Secure Boot in bootloader*** dans la configuration, ainsi que l'option permettant de signer automatiquement les binaires durant la construction.

Le développeur doit également spécifier le fichier au format PEM contenant la paire de clés. Sauf quelques rares exceptions (comme ESP32-C6, l'ESP32-H2 et le futur ESP32-P4) qui permettent de choisir entre les algorithmes ECDSA (en version *Secure Boot V2*) et RSA-PSS, c'est généralement ce dernier qui sera utilisé. La commande pour générer le fichier PEM change et devient :

```
$ openssl genrsa -out cle_de_signature.pem 3072
```

Le mode `SECURE_BOOTLOADER_REFLASHABLE` disparaît puisqu'il n'y a plus de clé générée automatiquement et inconnue du développeur. C'est un condensé SHA-256 de la clé publique qui est stocké en eFuse(s) (oui, pluriel, voir ci-après). En revanche, un autre mécanisme de protection du *bootloader* existe sous la forme de trois entrées du sous-menu ***UART ROM download mode*** (`SECURE_UART_ROM_DL_MODE`) :

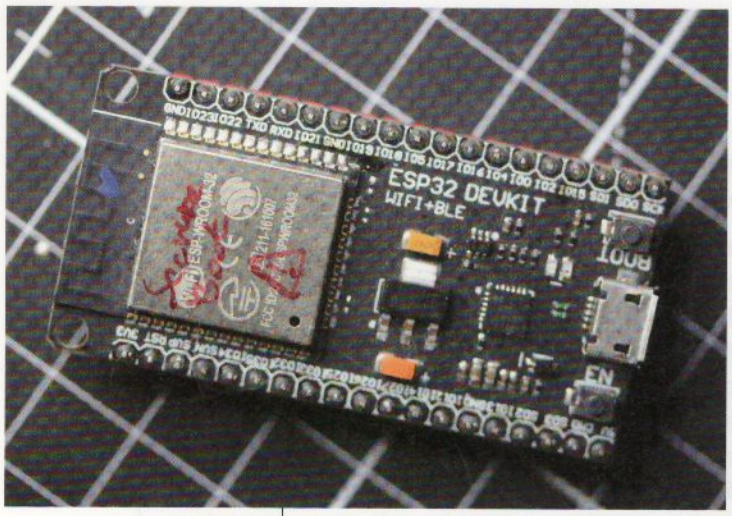
- ***Permanently disabled*** (`SECURE_DISABLE_ROM_DL_MODE`) : le code en ROM permettant la mise à jour et l'utilisation de `esptool.py` et `espefuse.py` est tout simplement désactivé, y compris si GPIO0 est à la masse au *boot*.

Ce mode est destiné à être utilisé sur les microcontrôleurs inclus dans des périphériques en production.

- ***Permanently switch to Secure mode*** (`SECURE_ENABLE_SECURE_ROM_DL_MODE`) : ce mode est une version réduite en fonctionnalités se limitant aux opérations essentielles pour les mises à jour de *firmware*, mais interdisant, par exemple, l'inspection de la mémoire et des registres, ou encore l'utilisation du *flasher stub* de l'outil `esptool.py`. Pour rappel, le *flasher stub* ou *stub loader* est un petit bout de code, envoyé par `esptool.py` puis exécuté par la cible pour se substituer au code en ROM. Dans ce mode, il est toujours possible d'utiliser `esptool.py`, mais vous devrez ajouter l'option `--no-stub` pour utiliser uniquement le code en ROM (qui techniquement est le premier *bootloader*, parfois appelé FSBL, pour *First Stage Boot-Loader*, sur certaines plateformes). Le fait de ne pas utiliser le *stub loader* quand la cible est en *boot* sécurisé est directement pris en charge par `idf.py flash`.
- ***Enabled*** (`SECURE_INSECURE_ALLOW_DL_MODE`) : correspond au fonctionnement standard que vous connaissez déjà en programmant vos ESP32 et est celui à préférer en phase de développement, ou plutôt en fin de phase de développement, puisque le reste du temps, mieux vaut simplement ne pas activer du tout le *boot* sécurisé.

Il convient donc de faire excessivement attention à cette configuration, puisque vous pouvez vous retrouver dans une situation où le microcontrôleur deviendra totalement inutilisable, car le *firmware* ne peut plus être mis à jour du tout via une connexion physique. Un autre point important à propos de la programmation avec `esptool.py` concerne l'USB-CDC. Certaines cartes et certains modules, en particulier ceux utilisant des ESP32 à cœurs RISC-V, n'utilisent pas de convertisseur USB/série (type CP2104 ou CH341), mais reposent sur les fonctionnalités USB-OTG intégrées dans la ROM du microcontrôleur qui émulent un port série ou une interface DFU. La documentation précise qu'après que le *boot* sécurisé ait été activé, la pile USB-OTG de la ROM est désactivée et qu'il n'est plus possible de programmer la flash de cette manière. Ce n'est cependant pas ce que j'ai constaté via mes essais avec l'ESP32-C3. Il était toujours possible de flasher de cette manière tant que l'***UART ROM download mode*** n'était pas réglé sur `SECURE_DISABLE_ROM_DL_MODE`.

Un autre changement majeur par rapport à *Secure Boot V1* est la possibilité de configurer plusieurs clés avec la plupart des ESP32 récents (S2, S3, C3, C6, H2 et P4). Jusqu'à trois clés peuvent être utilisées successivement et le *bootloader* devra être signé avec chacune d'elles, alors que l'image de l'application sera signée par la première. Un mécanisme de révocation entre alors en jeu avec, par défaut, la logique suivante : le *boot* sécurisé vérifie la signature de l'application, si celle-ci ne correspond pas à la première clé publique présente dans le *bootloader*, la seconde est utilisée. Si celle-ci réussit, la première clé devra être révoquée et les *boots* suivants utiliseront la seconde clé directement. Si la vérification échoue, le même principe s'applique avec la troisième clé. Ceci est décrit comme étant l'approche conservatrice dans la documentation et est le comportement par défaut en cas d'utilisation de plusieurs clés. Une approche *agressive* peut également être configurée via ***Enable Aggressive key revoke strategy*** (`SECURE_BOOT_ENABLE_AGGRESSIVE_KEY_REVOKE`) et dans ce cas, c'est l'échec de vérification qui révoquera immédiatement la clé utilisée (ceci n'a pas été testé ici).



Pensez à clairement identifier les plateformes sur lesquelles vous avez activé le boot sécurisé et à garder la ou les clés à disposition et en sûreté, car la configuration est totalement irréversible et dans certains cas, une erreur de manipulation peut rendre le matériel totalement inutilisable.

Notez que, par défaut, si vous configurez une seule clé (emplacement 0), les emplacements vides pour les clés 1 et 2 sont automatiquement révoqués. Ce comportement est modifiable en choisissant l'option ***Leave unused digest slots available*** (`SECURE_BOOT_ALLOW_UNUSED_DIGEST_SLOTS`) dans ***Potentially insecure options*** (`SECURE_BOOT_INSECURE`).

En guise de démonstration, et comme l'utilisation d'une unique clé est relativement similaire au *Secure Boot V1*, nous allons expérimenter autour de cette gestion multiclé. Notre projet sera identique au précédent avec intégration du *framework* Arduino pour le support OTA et reposera sur un module très basique (G3V01, alias ESP32-C3FH4-RGB) utilisant un ESP32-C3FH4 intégrant 4 Mio de flash. Celui-ci ne dispose pas de convertisseur USB/série et le connecteur USB-C est directement relié au microcontrôleur. Il dispose également d'une matrice de 25 LED RGB adressables **[7]** mais ceci n'a aucune importance ici.

Pour débuter, nous générons trois paires de clés RSA avec `openssl` :

```
$ openssl genrsa -out cle_de_signature_0.pem 3072
$ openssl genrsa -out cle_de_signature_1.pem 3072
$ openssl genrsa -out cle_de_signature_2.pem 3072
```

Seul le premier fichier sera renseigné dans la configuration (`SECURE_BOOT_SIGNING_KEY`) puisque l'image de l'application n'utilise qu'une seule signature, alors que le *bootloader* utilise les trois et que les eFuses (emplacements) seront programmés avec les condensés SHA-256 correspondants. Notez qu'il sera nécessaire, comme précédemment, de décaler la table de partition (`PARTITION_TABLE_OFFSET` à `0xb000`) en raison de la taille plus importante du *bootloader*. Une fois la configuration adaptée à nos besoins, nous pouvons procéder à la construction du *bootloader* avec `idf.py bootloader` et celui-ci sera automatiquement signé, mais uniquement avec la première clé (`cle_de_signature_0.pem`) que nous avons spécifiée dans la configuration.

Pour compléter les opérations avant de flasher le *bootloader*, nous devons signer le binaire manuellement avec `espsecure.py`, en utilisant l'option `--append_signatures` :

```
$ espsecure.py sign_data -k cle_de_signature_1.pem \
-v 2 --append_signatures -o signed_bootloader_1.bin \
build/bootloader/bootloader.bin
[...]
```

```
Signature block 0 is valid (RSA).
1 valid signature block(s) already present
  in the signature sector.
1 signing key(s) found.
Signed 36864 bytes of data from
  build/bootloader/bootloader.bin.
  Signature sector now has 2 signature blocks.

$ espsecure.py sign_data -k cle_de_signature_2.pem \
-v 2 --append_signatures -o signed_bootloader_2.bin \
signed_bootloader_1.bin
[...]
Signature block 0 is valid (RSA).
Signature block 1 is valid (RSA).
2 valid signature block(s) already present
  in the signature sector.
1 signing key(s) found.
Signed 36864 bytes of data from
  signed_bootloader_1.bin.
  Signature sector now has 3 signature blocks.
```

Là encore, il n'est pas nécessaire en principe d'utiliser `-o` et de spécifier un nom de fichier en sortie, étant donné qu'en l'absence de cette option, le binaire sera signé « sur place ». Quoi qu'il en soit, nous obtenons `signed_bootloader_2.bin` dans le répertoire courant et cette image comporte effectivement trois signatures, comme le précise la sortie de la dernière commande. Nous pouvons alors passer au flashage :

```
$ esptool.py --chip esp32c3 --port=/dev/ttyACM0 \
--after=no_reset --no-stub write_flash \
--flash_mode dio --flash_freq 80m --flash_size keep 0x0 \
signed_bootloader_2.bin
[...]
Wrote 40960 bytes at 0x00000000 in
  0.6 seconds (570.9 kbit/s)...
Hash of data verified.
Leaving...
Staying in bootloader.
```

Comme avec *Secure Boot V1*, nous restons dans le *bootloader* et nous enchaînons sur le flashage de l'application avec `idf.py -p /dev/ttyACM0 flash monitor`. Le premier démarrage qui s'en suit nous montre qu'effectivement, les condensés des clés publiques sont enregistrés dans les eFuses correspondants (`EFUSE_BLK_KEY0` à `EFUSE_BLK_KEY2`), que l'application est bien détectée comme ayant une signature valide et que le *boot* sécurisé est définitivement activé :

```
I (352) esp_image: Verifying image signature...
I (355) secure_boot_v2: Secure boot V2 is not enabled yet and
  eFuse digest keys are not set
```

```
I (364) secure_boot_v2: Verifying with RSA-PSS...
I (372) secure_boot_v2: Signature verified successfully!
I (375) secure_boot_v2: Secure boot digests absent, generating..
I (394) secure_boot_v2: Digests successfully calculated,
  3 valid signatures (image offset 0x0)
I (394) secure_boot_v2: 3 signature block(s) found
  appended to the bootloader.
I (400) secure_boot_v2: Burning public key hash to eFuse
I (408) efuse: Writing EFUSE_BLK_KEY0 with purpose 9
I (412) efuse: Writing EFUSE_BLK_KEY1 with purpose 10
I (418) efuse: Writing EFUSE_BLK_KEY2 with purpose 11
I (513) secure_boot_v2: Digests successfully calculated,
  1 valid signatures (image offset 0x20000)
I (513) secure_boot_v2: 1 signature block(s) found appended to the app.
I (519) secure_boot_v2: Application key(0) matches with bootloader key(0).
I (526) secure_boot_v2: blowing secure boot efuse...
W (532) secure_boot: UART ROM Download mode kept enabled
  - SECURITY COMPROMISED
W (540) secure_boot: Not disabling JTAG - SECURITY COMPROMISED
W (546) secure_boot: Allowing read disabling of additional efuses
- SECURITY COMPROMISED
I (556) efuse: BURN BLOCK6
I (562) efuse: BURN BLOCK6 - OK (write block == read block)
I (566) efuse: BURN BLOCK5
I (572) efuse: BURN BLOCK5 - OK (write block == read block)
I (575) efuse: BURN BLOCK4
I (582) efuse: BURN BLOCK4 - OK (write block == read block)
I (584) efuse: BURN BLOCK0
I (590) efuse: BURN BLOCK0 - OK (write block == read block)
I (594) efuse: Batch mode. Prepared fields are committed
I (600) secure_boot_v2: Secure boot permanently enabled
```

Suite à cela et en fonction de la configuration choisie, il sera encore possible ou non de programmer le microcontrôleur avec `esptool.py` et/ou `idf.py flash`, mais l'OTA sera toujours utilisable. Notez bien qu'en cas d'erreur dans le code du *firmware* flashé, que ce soit concernant la connexion en Wi-Fi ou dans la gestion de l'OTA, la cible peut devenir totalement inutilisable, car impossible à mettre à jour.

Nous sommes dans une configuration de *boot* sécurisé et c'est là que les choses deviennent très intéressantes. En effet, imaginons que le fichier `cle_de_signature_0.pem` soit compromis et ait été diffusé en dehors du groupe de personnes dignes de confiance. Pour éviter tout risque de voir les *firmwares* de nos installations être mis à jour de façon illégitime, nous pouvons utiliser une nouvelle clé (`cle_de_signature_1.pem`) pour reconstruire le *firmware* et révoquer la clé problématique. Cela n'est toutefois pas automatique et doit normalement être intégré au mécanisme de mise à jour OTA via un appel à la fonction `esp_ota_revoke_secure_boot_public_key()`, qui n'est bien entendu pas présent dans l'implémentation de la gestion OTA du

framework Arduino. Ceci suppose donc une réimplémentation de la gestion OTA, ou du moins une adaptation de ce qui existe pour prendre en charge ce type de mécanisme. Mais si tel est le cas, l'intégration du *framework* Arduino devient sans intérêt et nous pouvons alors reposer entièrement sur l'ESP-IDF seul. Notez également qu'à cette date, une *issue* est ouverte sur le GitHub de l'ESP-IDF, car, semble-t-il, la révocation ne fonctionne pas **[8]**. En l'absence de révocation effective de clés, n'importe quelle clé enregistrée peut être utilisée pour signer un *firmware*, ce qui peut également présenter un intérêt dans certaines situations.

CONCLUSION

Arrêtons-là cette exploration déjà très longue en émettant un avis à propos de ces fonctionnalités. Ce type de mécanismes est indéniablement intéressant, dès lors qu'il s'agit d'avoir des périphériques « dans la nature » et de limiter les risques concernant des mises à jour malveillantes. Les matériels IoT, de manière générale, ne sont pas spécialement réputés en termes de sécurité et sont souvent vus comme des points d'entrée de choix pour attaquer un système ou une infrastructure. Ce genre de solutions, qu'elles soient proposées par Espressif Systems ou un autre constructeur, sont un excellent point de départ, mais la difficulté d'implémentation est un frein évident. Nous venons de le voir, pour arriver à un niveau de sécurité réellement satisfaisant, des ressources et du temps doivent être engagés, mais cet investissement parait généralement secondaire, voire est tout simplement écarté une fois arrivé à un certain stade du projet. Ce que propose Espressif est prometteur, mais, à mon sens, encore insuffisant pour que les considérations vis-à-vis de la sécurité deviennent un automatisme, comme c'est le cas dans d'autres domaines (serveurs, solutions web, etc.). La révocation automatique d'une clé « N-1 », directement par le *bootloader*, serait un plus, tout comme le fait d'avoir, dans le bloc de signature, un court champ textuel en décrivant l'objet. Ce n'est là qu'une paire d'idées, mais le raisonnement global est on ne peut plus clair : pour que la sécurité devienne un réflexe, il est impératif qu'elle soit facile à mettre en œuvre et ne nécessite que peu d'efforts. Pour l'heure, ce n'est toujours pas le cas, mais on progresse... **DB**

RÉFÉRENCES

[1] https://connect.ed-diamond.com/Hackable/hk-024/esp32-developpez-vos-croquis-arduino-sans-l-ide-arduino

[2] https://connect.ed-diamond.com/hackable/hk-046/developper-pour-arduino-en-ligne-de-commandes-pour-de-vrai

[3] https://github.com/espressif/esp-idf/tree/master/examples/system/ota

[4] https://components.espressif.com/components/espressif/arduino-esp32

[5] https://github.com/espressif/arduino-esp32/issues/8913

[6] https://github.com/espressif/arduino-esp32/tree/master/tools

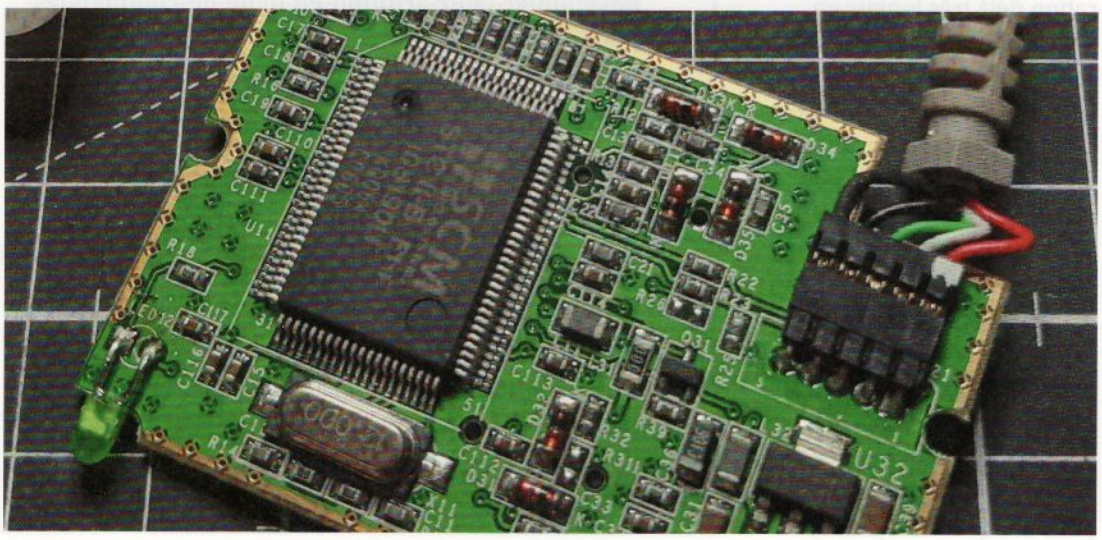
[7] https://github.com/01Space/ESP32-C3FH4-RGB/blob/main/Schematic/ESP32-C3FH4-RGB%20Schematic.pdf

[8] https://github.com/espressif/esp-idf/issues/12851

CONTINUONS NOTRE EXPLORATION DES JAVA CARDS : JCKIT 3.0.4, NFC ET CODE PIN

Denis BODOR

Dans un précédent article, nous avons découvert le monde des smartcards et des Java Cards en particulier, en prenant en main un modèle, certes toujours utilisable, mais relativement ancien (NXP J2A081). Il est temps aujourd'hui de nous mettre à jour, de goûter à davantage de modernité en nous penchant sur une version plus récente des spécifications et aussi d'explorer les aspects « sans contact » de cette formidable technologie. Depuis la dernière fois, une certaine expérience a été acquise et nous approfondirons également ce qu'il est possible de faire, autant avec des smartcards « anciennes » qu'avec des déclinaisons plus actuelles...



Java Cards

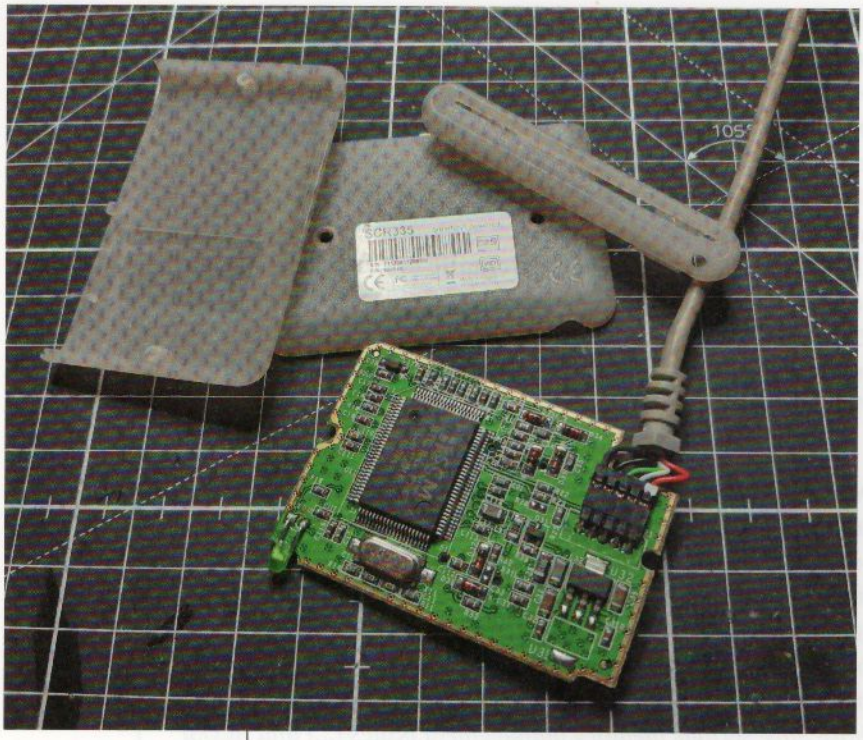
– Continuons notre exploration des Java Cards : jckit 3.0.4, NFC et code PIN –

Je ne vous le cache pas, mettre le doigt dans le monde des *smartcards* et Java Cards est un piège. Ce qui commence par un brin de curiosité se transforme rapidement en une quête exploratoire avec des hauts, mais aussi des bas, en particulier quand Java n'est pas sa langue maternelle. Dans l'article précédent, je vous avais laissé sur une application « *Hello World* » relativement basique et les choses ont évolué depuis. Je vous parlais également de cartes ACS ACOSJ-G qu'il était possible d'acquérir à moindre coût auprès d'un distributeur français du nom de Hitools Access. Ces cartes ont depuis été non seulement réceptionnées, mais aussi utilisées, explorées, analysées et, pour deux d'entre elles, sacrifiées (de façon non délibérée).

Deux problèmes distincts se sont posés avec ces cartes : un avec le revendeur et l'autre avec le constructeur/produit. Les cartes en question sont vendues, sur le site de Hitools Access **[1]**, comme ayant « 95 Ko de mémoire EEPROM pour application de gestion de données ». Ce n'est pas le cas. Les trois modèles de cartes (contact, sans contact et *dual interface*) sont des ACOSJ 1.01 (2015), non des 2.04 (2019) et de ce fait ne disposent que de 40 Kio d'EEPROM. Ceci a été vérifié non seulement via le numéro de version présent dans le message ATR (*Answer To Reset*), mais également via un code exécuté par la carte elle-même qui remonte effectivement (et seulement) 40036 octets de mémoire (de type `MEMORY_TYPE_PERSISTENT`) disponible. Un message a, bien entendu, été envoyé au SAV, sans réponse pour le moment. À noter que les Allemands de **https://www.cardomatic.de/** distribuent également des cartes ACS ACOSJ à double interface, données pour 95 Kio d'EEPROM pour 34 € le lot de 5 (~50 € TTC avec le port) qui, elles, sont effectivement des versions 2.04 avec le volume d'EEPROM promis.

L'autre problème concerne le produit lui-même et, avec du recul, ne m'étonne finalement pas vraiment (ACS fabrique également le lecteur RFID/NFC ACR122U connu pour son *firmware* bogué). Ces Java Cards ont une très mauvaise réputation **[2]**, parfaitement résumée par un commentaire dans un *bug report* **[3]** de l'applet SmartPGP (cf. plus loin dans l'article) : « *I'm asking because it may be a buggy knockoff like the Feitian JavaCOS A22 and ACS ACOSJ. Both of those have serious memory management issues [...] and ACOSJ (speculatively) corrupting the EEPROM all the way until bricking the card).* ». Et effectivement, je ne peux que confirmer les « *serious memory management issues* », car même si la gestion mémoire Java Cards est peu tolérante (en particulier lorsqu'on a la mauvaise idée de faire des `MessageDigest.getInstance()` à répétition), il est normalement toujours possible de se rattraper. C'était le cas avec les NXP J2A081 où une simple suppression de l'applet et du *package* est suffisante, mais pas avec les ACOSJ qui ont fini, deux fois, par tout simplement ne plus répondre aux APDU et ne plus envoyer d'ATR, se transformant, de fait, en simples cartes en plastique (*bricking*).

Ce n'est pas tout. Vous souvenez-vous peut-être que j'aie précédemment fait mention de cartes J3R150 vendues sur AliExpress ? La communication avec la carte est impossible, car la clé n'est pas celle par défaut et la communication avec le vendeur est difficile. Moralité, n'achetez pas n'importe quoi à n'importe qui et gardez à l'esprit que des sites comme AliExpress sont peut-être intéressants pour beaucoup de choses, mais clairement pas lorsqu'on parle de *smartcards* évoluées et modernes. Peut-être que les plus anciennes J2A040 (Java Card 2.2.2) en vente sont originales (*new old stock*), mais je ne tenterai plus ma chance à ce petit jeu, car la leçon est



Un lecteur de smartcards compatible CCID est relativement simple dans sa construction. Un circuit intégré spécialisé, quelques composants passifs et c'est tout. C'est une solution totalement intégrée qu'on retrouve dans presque tous les lecteurs.

payée et retenue ! [Note de dernière minute : le vendeur AliExpress (XCRFID Store) m'a finalement transmis les bonnes clés pour les J3R150 et celles-ci sont maintenant pleinement utilisables, après changement des clés en question et suppression des applets/*packages* encombrant la mémoire.]

Finalement, mes « vieilles » J2A081 ne sont pas si mauvaises, même si l'obligation d'utiliser un JDK 8 est très pénible et que les fonctionnalités ne sont pas au goût du jour (pas de SHA-512 par exemple). Certes, les ACOSJ peuvent être des alternatives aux Java Cards plus coûteuses, mais certainement pas dans un contexte de développement et de mise au point d'application puisque les erreurs de gestion mémoire du développeur sont fatales au produit.

Je vais donc drastiquement corriger ma pseudorecommandation faite en fin du précédent article et vous déconseiller fortement ces cartes et, à défaut d'avoir une réponse doublée d'un geste commercial de la part de Hitools Access, faire de même pour ce distributeur. Pas de problème, en revanche, avec Cardomatic. La commande et la livraison se sont passées sans problème, mais cela reste des cartes ACOSJ d'ACS auxquelles je n'accorde plus aucune confiance.

Mais trêve de blabla et de récit autour de regrettables déboires, passons au sujet qui nous intéresse présentement, à commencer par la dualité contact/sans contact de certaines cartes...

1. UTILISER LE LECTEUR NFC AVEC PC/SC

Le précédent article traitait de Java Cards relativement anciennes, doublées d'un tag NFC DESFire tout aussi ancien. Je dis bien « DESFire » tout court et non DESFire EV1 ou EV2. La partie Java Card n'a ici aucun lien avec la partie NFC et, en utilisant le flash de son smartphone par exemple, on distingue clairement une seconde puce par transparence dans le support en PVC. Ceci ne doit pas être confondu avec des cartes effectivement dites « double interface », où une seule et unique puce, et donc les applets qui s'y trouvent, disposent de deux canaux de communication distincts, avec et sans contact. Nous avons fait connaissance avec l'aspect « contact » la dernière fois, penchons-nous sur la partie NFC, avec de nouvelles cartes.

– Continuons notre exploration des Java Cards : jckit 3.0.4, NFC et code PIN –

Mais nous avons un point à régler. L'utilisation d'outils comme GlobalPlatformPro ne pose aucun problème avec les Java Cards à contact ou à double interface puisque PC/SC prend en charge n'importe quel lecteur CCID directement. Pour les Java Cards sans contact, et donc NFC, nous n'avons pas d'accès direct et aucun moyen de gérer les applications qui s'y trouvent. Par ailleurs, toute la partie *contact less* et les lecteurs NFC type ACR122U, SCL3711 ou ASK/LoGO basés sur la puce NXP PN532/PN533 sont parfaitement pris en charge par la LibNFC (elle-même utilisant la LibUSB), mais les outils pour Java Cards n'en font pas usage, reposant uniquement sur PC/SC.

Nous devons donc établir un lien entre PC/SC et la LibNFC, ou plus exactement utiliser un pilote PC/SC qui se chargera de cette tâche. Et c'est précisément là qu'intervient le code développé par Ludovic Rousseau **[4]** sur la base de celui initialement créé par Frank Morgner **[5]**. Ce code est relativement ancien (dernier *commit* en 2016), mais toujours parfaitement utilisable, bien qu'il ne soit pas intégré dans les distributions GNU/Linux (du moins Debian et consorts). Il sera donc nécessaire de le compiler et de l'installer manuellement.

La procédure est la suivante :

```
$ cd kkpart
$ git clone https://github.com/nfc-tools/ifdnfc.git
$ cd ifdnfc.git

$ libtoolize
libtoolize: putting auxiliary files in '.'.
libtoolize: linking file './ltmain.sh'
[...]

$ aclocal

$ autoheader

$ automake --add-missing
configure.ac:10: installing './ar-lib'
configure.ac:10: installing './compile'
Makefile.am: installing './INSTALL'
[...]

$ autoconf

$ ./configure
checking for a BSD-compatible install... /usr/bin/install -c
checking whether build environment is sane... yes
[...]
ifdnfc has been configured with following options:
Version:                0.1.4
Host:                   x86_64-pc-linux-gnu
Compiler:               gcc
Preprocessor flags:
```

```
Compiler flags:         -g -O2 -std=c99
Preprocessor flags:
Linker flags:
Libraries:
LIBNFC_CFLAGS:
LIBNFC_LIBS:            -lnfc -lusb
PCSC_CFLAGS:            -I/usr/include/PCSC -pthread
PCSC_LIBS:              -lpcsclite
BUNDLE_HOST:            Linux
DYN_LIB_EXT:            so
[...]

$ make
```

Notez que les différents appels aux *GNU autotools*, et même la compilation, peuvent être résumés en une seule ligne : `libtoolize && aclocal && autoheader && automake --add-missing && autoconf && ./configure && make`, mais mieux vaut procéder par étape en cas de problème.

Normalement, ce qui va suivre n'est pas la bonne façon de faire, mais cette approche permet de disposer du pilote PC/SC sans nuire au système de gestion de paquets de la distribution (chose que je déteste). Vous vous ferez donc vilipender dans les logs du démon `pcscd` : « *USB drivers SHOULD NOT be declared in a reader.conf* » (Vilain Denis ! Vilain !). Après compilation, nous obtenons dans `src/.libs/` la bibliothèque dynamique `libifdnfc.so.0.0.0` et dans `src/` le binaire `ifdnfc-activate`. En temps normal, les pilotes PC/SC sont placés dans `/usr/lib/pcsc/drivers` accompagné d'un fichier XML `Info.plist` décrivant différents paramètres selon le type de pilote. Ici, nous allons faire cela de façon manuelle sans avoir à toucher à `/usr/lib`. Nous copions le fichier `libifdnfc.so.0.0.0` dans `/usr/local/lib` et le binaire `ifdnfc-activate` dans un endroit référencé dans le `$PATH`. Pour ce genre d'horreur très « bricolo » (mes propres scripts shell, par exemple), j'ai un répertoire `bin/` dédié dans mon `$HOME` et, bien entendu, `~/bin` est dans le `$PATH`.

Ceci fait, nous devons créer un fichier décrivant ce pseudolecteur qui, en fait, est un pont vers la LibNFC : `/etc/reader.conf.d/ifdnfc`. Celui-ci contient :

```
FRIENDLYNAME    "IFD-NFC"
LIBPATH         /usr/local/lib/libifdnfc.so.0.0.0
CHANNELID       0
```

Nous y référençons directement la bibliothèque et un périphérique géré par la LibNFC sera alors identifiable sous le nom « IFD-NFC ». Il ne nous reste plus, ensuite, qu'à redémarrer le service PC/SC avec `sudo systemctl restart pcscd.service` et tester avec :

```
$ pcsc_scan
PC/SC device scanner
V 1.6.2 (c) 2001-2022, Ludovic Rousseau <ludovic.rousseau@free.fr>
Using reader plug'n play mechanism
```

```
Scanning present readers...
0: IFD-NFC 00 00

Sat Oct 21 17:43:42 2023
Reader 0: IFD-NFC 00 00
  Event number: 0
  Card state: Card removed,
[...]
```

En posant un tag sur le lecteur, rien ne se passe, car par défaut, le lecteur, bien que détecté par PC/SC, n'est pas actif et ne surveille (*polling*) pas les événements comme la détection d'un tag ou son retrait. Il faut alors utiliser `ifdnfc-activate` dans un autre terminal pour activer la détection :

```
$ ifdnfc-activate yes
Activating ifdnfc with "pn53x_usb:003:017"...
IFD-NFC is inactive.
```

Ne prêtez pas attention à la dernière ligne, car dans le premier terminal, ceci vient d'apparaître :

```
Sat Oct 21 17:45:06 2023
Reader 0: IFD-NFC 00 00
  Event number: 1
  Card state: Card inserted,
  ATR: 3B 80 80 01 01

ATR: 3B 80 80 01 01
+ TS = 3B --> Direct Convention
+ T0 = 80, Y(1): 1000, K: 0 (historical bytes)
  TD(1) = 80 --> Y(i+1) = 1000, Protocol T = 0
-----
  TD(2) = 01 --> Y(i+1) = 0000, Protocol T = 1
-----
+ Historical bytes:
+ TCK = 01 (correct checksum)

Possibly identified card (using /usr/share/pcsc/smartcard_list.txt):
3B 80 80 01 01
  ISO 14443 Type B without historical bytes
  Electronic Passport
  Spanish passport (2012)
  Canadian Passport
  Venez_Prox
  Carta nazionale dei servizi
  https://www.agid.gov.it/it/piattaforme/carta-nazionale-servizi
```

Là, c'est un tag ST25 qui est posé sur le lecteur, mais ceci fonctionnera tout aussi bien avec une Java Card sans contact ou double interface. Il est même possible d'échanger des APDU avec le script Perl livré avec `pcsc-tools` :

```
$ scriptor
No reader given: using IFD-NFC 00 00
Using T=1 protocol
Reading commands from STDIN
00a4040007d276000085010100
> 00 a4 04 00 07 d2 76 00 00 85 01 01 00
< 90 00 : Normal processing.
00a4000c02e101
> 00 a4 00 0c 02 e1 01
< 90 00 : Normal processing.
00b0000012
> 00 b0 00 00 12
< 00 12 01 00 11 00 81 00 02 C4 00 4E 44 72 24 1F
FF C4 90 00 : Normal processing.
```

Nous venons tout juste de sélectionner l'application adéquate sur le tag ST25, puis le fichier de configuration ST pour en lire enfin le contenu. Le tout, avec des APDU transitant par PC/SC qui accède au lecteur via la LibNFC. Magique ! Bien entendu, il n'est pas possible d'utiliser le lecteur NFC avec une application LibNFC tant que celui-ci est occupé avec PC/SC. Inutile cependant d'arrêter le service pour autant, il nous suffit de désactiver la prise en charge avec `ifdnfc-activate no` et chacun reste à sa place.

Notez qu'il est même possible de pousser sensiblement plus loin, puisque la LibNFC peut également prendre en charge un lecteur PN532 interfacé via un port série, comme un module destiné à être utilisé avec un microcontrôleur. Connecter un tel module à un convertisseur USB/série puis au PC vous fournira un port série supplémentaire et, puisque ceci n'est pas détectable/identifiable comme c'est le cas en USB, nous devons préciser le type et le port dans `/etc/nfc/libnfc.conf` avec :

```
device.name = "lecteurUSBserie"
device.connstring = "pn532_uart:/dev/ttyU0"
```

Ici, nous sommes sous FreeBSD 13.2, d'où le `/dev/ttyU0`, mais cela fonctionne exactement de la même manière avec GNU/Linux et OpenBSD (non testé avec NetBSD). Dès lors, le lecteur sera accessible par les outils de la LibNFC comme le montre la sortie de la commande :

```
$ nfc-scan-device
nfc-scan-device uses libnfc 1.8.0
1 NFC device(s) found:
- lecteurUSBserie:
    pn532_uart:/dev/ttyU0
```

Et de ce fait, celui-ci sera également utilisable par PC/SC une fois activé comme précédemment. Ceci peut être très pratique comme solution alternative si d'aventure vous n'avez pas votre lecteur USB sous la main. Et personnellement, je trouve absolument fantastique d'ainsi

assembler des briques (USB/série, module PN532, LibNFC, PC/SC, etc.) et de constater que l'*empilage* fonctionne parfaitement. Il paraît que la passion s'éteint lorsqu'on perd la capacité de s'émerveiller. Je pense que je suis tranquille de ce point de vue. À noter que des ports FreeBSD et OpenBSD existent pour ce pilote *ifdnfc* pour PC/SC Lite. Ils sont disponibles dans mes dépôts GitLab **[6][7]**.

À présent, nous sommes en mesure d'utiliser n'importe quelle application capable de se servir du *middleware* PC/SC Lite et ceci inclut, bien sûr, GlobalPlatformPro qui nous permet de gérer les applications de nos Java Cards. Ainsi, une carte ACOSJ sans contact, posée sur le lecteur RFID/NFC, sera parfaitement accédée :

```
$ gp -k 404142434445464748494A4B4C4D4E4F -l
ISD: A000000151000000 (OP_READY)
     Privs:    SecurityDomain, CardLock, CardTerminate,
               CardReset, CVMManagement, TrustedPath,
               AuthorizedManagement, GlobalDelete,
               GlobalLock, GlobalRegistry, FinalApplication

PKG: A0000001515350 (LOADED)
     Applet:   A000000151535041
```

Comprenez bien que, du point de vue de vos applets, la communication avec et sans contact n'a, de base, aucun impact. Ceci signifie donc que vous pouvez développer vos codes et les installer sur la carte pour ensuite communiquer avec cette dernière via un outil basé sur la LibNFC ou même avec votre smartphone ou un Proxmark3. Le pont LibNFC vers PC/SC n'est là que pour nous permettre d'utiliser GlobalPlatformPro.

2. GPSHELL, UNE ALTERNATIVE À GLOBALPLATFORMPRO

GlobalPlatformPro, écrit en Java par Martin Paljak, est certainement l'outil le plus utilisé pour installer et gérer des applets sur une Java Card compatible avec les spécifications GlobalPlatform 2.1.1 et supérieures. Mais ce n'est pas le seul. En effet, un projet bien plus ancien, écrit en C par Karsten Ohme, fournit également une solution sous la forme d'une bibliothèque doublée d'un outil en ligne de commande appelé GPShell. Toujours activement maintenue par son créateur (malgré ce que semble penser une partie de la faune de Stack Overflow), la bibliothèque et l'outil ne sont malheureusement *packagés* pour aucune

Transformer une smartcard en une carte SIM n'est pas très difficile, même sans outillage spécialisé. Cette pauvre J2A081 a été sauvagement charcutée au cutter de façon pas très jolie (ou droite), mais ça fonctionne sans problème.

distribution GNU/Linux et l'installation devra se faire manuellement (compilation et installation). À noter qu'un binaire Windows est en revanche disponible pour toutes les dernières versions (2.3.1 à cette date) sous la forme d'un ZIP directement téléchargeable dans la section *release* du dépôt GitHub **[8]**.

GPShell fonctionne différemment de GlobalPlatformPro dans le sens où il s'utilise normalement avec des scripts dont un certain nombre sont livrés en guise d'exemples avec l'outil. Il peut également être utilisé de façon interactive ou avec un script passé via STDIN, mais l'idée est généralement d'intégrer cela directement dans un processus de construction. Parmi les scripts livrés, nous avons par exemple celui permettant de lister les applets présentes sur une carte (`listGP211.txt`) :

```
# mode GlobalPlatform 2.1.1
mode_211
# affiche APDUs
enable_trace
# initialisation
establish_context
# connexion carte
card_connect
# sélection Applet de gestion
select -AID a000000003000000
# Établissement d'un canal sécurisé (sur une ligne)
open_sc -security 3 -keyind 0 -keyver 0
   -mac_key 404142434445464748494a4b4c4d4e4f
   -enc_key 404142434445464748494a4b4c4d4e4f
# Lister les packages
get_status -element 20
# Lister les applets
get_status -element 40
# Déconnexion
card_disconnect
# Libération des ressources
release_context
```

Un tel script s'utilise soit avec `gpshell listGP211.txt`, soit avec un `cat listGP211.txt | gpshell` et permet de faire l'équivalent d'un `java -jar gp.jar -l` (ou `gp -l` en ayant configuré l'alias qui va bien). Des scripts existent également pour effacer une ou des applets/*packages*, ainsi que pour en installer. Notez que, techniquement, il n'y a pas vraiment de différence entre un tel script pour une Java Card 2.2.2 (J2A081) ou pour une Java Card 3.0.4 (ACOSJ), si ce n'est principalement par l'AID de l'applet de gestion ou ISD (respectivement `a000000003000000` et `A000000151000000`). Les scripts fonctionnant avec une carte ACS ACOSJ incluent `SCP03` dans leurs noms. SCP signifiant *Secure Channel Protocol* et constituant le mécanisme par lequel la communication avec une *smartcard* est rendue résistante à l'espionnage (*overhearing*) et à la falsification (*tampering*), le « 03 » faisant référence à la génération/déclinaison (« 01 », « 02 » ou « 03 »).

Finissons sur ce point en précisant qu'il existe un port OpenBSD pour GPShell, dans le dépôt GitLab précité, étant donné que j'ai rencontré de grandes difficultés à faire fonctionner GlobalPlatformPro avec ce système. Il était plus simple de créer un port pour GPShell que

– Continuons notre exploration des Java Cards : jckit 3.0.4, NFC et code PIN –

de chercher la source du problème ou de contourner celui soulevé par le déplaisant message obtenu lors d'une recompilation de GlobalPlatformPro : « *Sorry, Launch4j doesn't support the 'OpenBSD' OS* ».

3. ÉVOLUTION DE NOTRE PETITE EXPÉRIENCE

3.1 À propos des Java Cards

Mettre innocemment son nez dans le terrier du lapin d'Alice est le début d'une véritable aventure lorsqu'on explore ce domaine. Ce qui paraît simple et évident au premier regard est en réalité une sorte de *multivers* où se côtoient des mondes très différents. Nous avons d'une part celui des développeurs d'applications Java Card, très pros, très « bancaires » et, à l'autre extrême, celui de revendeurs chinois baignant dans un flou artistique entretenu où on ne sait pas vraiment qui fabrique quoi et sous quelle licence. Et entre les deux, nous avons le monde des filières de distribution avec, dans le meilleur des cas, des cartes explicitement coûteuses et dans le pire, le classique bouton « contactez-nous » synonyme de « nos commerciaux vont vous suivre partout jusqu'à la fin de votre vie », ou encore de « NDA sur 17 générations pour pouvoir lire le début d'un sommaire d'un *reference manual* »...

S'y retrouver dans cette galaxie de genres n'est pas chose facile et le jargon à géométrie variable n'aide pas beaucoup, bien au contraire. J'allais dire « commençons par le début », mais il n'y a pas réellement de début ou de fin à ce genre d'exploration. Je vais donc faire les choses dans le désordre et voyez cela comme une liste de choses à savoir ou connaître pour éviter les problèmes, sans pour autant approfondir outre mesure les concepts (sinon l'article ferait 80 pages).

Voici une petite découverte sortie tout droit d'AliExpress. Ce périphérique fait lecteur de smartcard compatible CCID, lecteur de SIM, lecteur de microSD et de SD/MCC. Tout ça pour 5,20 € + 1,89 € de port. Incroyable !

Commençons par le cycle de vie d'une Java Card. Lorsque vous recevez la carte du fabricant, celle-ci se trouve dans l'état *OP_READY* signifiant qu'elle est prête à être utilisée **par le développeur**. Tout est en place pour cela, de l'environnement d'exécution aux structures de données de base ou encore la clé permettant de manipuler son contenu (applets et *packages*). De l'état *OP_READY*, nous pouvons passer, de façon **irréversible**, à l'état *INITIALIZED* dont les fonctionnalités sont en dehors des spécifications GlobalPlatform. *INITIALIZED* est un passage obligé vers le prochain état qui sera également irréversible, mais sa signification est laissée à la discrétion du développeur ou de la structure déployant la solution. En passant de *INITIALIZED* à *SECURED* en revanche, les choses sont différentes puisque cet état est généralement associé à la phase post-émission (suite à la mise en production, si vous voulez) et certaines applications vont vérifier cet état pour valider leur fonctionnement. Une carte en état *SECURED* est donc une carte

prête pour l'utilisateur final. Nous avons ensuite l'état *CARD_LOCKED*, qui est **réversible** et qui bloquera l'exécution de n'importe quelle applet n'étant pas le gestionnaire d'applications. La carte est donc bloquée et inutilisable, généralement pour des raisons de sécurité. Il est cependant possible de revenir à l'état *SECURED* pour remettre la carte en production. Ce qui est irréversible dans le sens le plus strict du terme, en revanche, est le passage au dernier état : *TERMINATED*. Cela aura pour effet de désactiver toutes les fonctionnalités de la carte, y compris le gestionnaire d'applications et donc de la détruire de manière logicielle. Ceci peut être décidé en raison d'une faille de sécurité grave ou de l'expiration pure et simple de la carte, et le passage à *TERMINATED* peut être fait depuis n'importe quel état précédent.

Notez que ces changements d'état nécessitent d'utiliser le gestionnaire d'applications qui, par défaut, dispose de tous les privilèges. C'est ce qui apparaît, sous `Privs`, lorsque vous faites un `gp -l` :

```
ISD: A000000003000000 (OP_READY)
     Privs:    SecurityDomain, CardLock,
               CardTerminate, CardReset,
               CVMManagement
```

ou (ACOSJ Java Card 3.0.4) :

```
ISD: A000000151000000 (OP_READY)
     Privs:    SecurityDomain, CardLock,
               CardTerminate, CardReset,
               CVMManagement, TrustedPath,
               AuthorizedManagement,
               GlobalDelete, GlobalLock,
               GlobalRegistry, FinalApplication
```

L'ISD ou *Issuer Security Domain* est le gestionnaire d'applications et celui-ci dispose des privilèges `CardLock` et `CardTerminate`, se référant au passage vers les états *CARD_LOCKED* et *TERMINATED*. Mais avant de parler d'un autre privilège très intéressant, revenons un instant sur le fameux état *OP_READY*.

On pourrait supposer celui-ci comme étant l'état (au sens large du terme) d'une carte neuve et vierge, mais en réalité, il existe une phase avant celle-ci, appelée prépersonnalisation. Celle-ci a normalement lieu en usine et consiste à initialiser la carte pour la rendre utilisable. Nous sommes ici en dehors des spécifications normales et les constructeurs implémentent cette fonctionnalité de prépersonnalisation comme ils l'entendent, via une « application racine » utilisant des APDU propriétaires. Là où cela peut être un problème, c'est avec les filières d'approvisionnement non officielles (comprendre « AliExpress » et consorts). En effet, un certain nombre de vendeurs proposent des cartes dites « *unfused* » et donc à prépersonnaliser soi-même. Ceci implique d'avoir accès à la documentation du constructeur, souvent disponible qu'après signature d'un accord de non-divulgation (*Non-Disclosure Agreement* ou NDA en anglais), ainsi que d'une clé spécifique, appelée *Transport key*, pour appliquer la procédure. Vous l'aurez compris, ces cartes « *unfused* » ne sont pas directement utilisables (il n'y a pas de gestionnaire d'applications ou de clé) et, je pense, révèle clairement l'origine suspecte du matériel : ce sont très probablement des produits volés en usine (type « tombé du camion »). Vous pouvez tenter votre chance et, en cherchant bien, on trouve des procédures en ligne pour certaines cartes (comme ce billet **https://curriegrad2004.ca/2017/02/dealing-with-unfused-jcop-java-cards-sold-from-aliexpress-or-ebay/**).

Revenons maintenant sur cette notion de privilège qu'une application peut avoir. Je ne vais pas faire le tour ici de la liste complète, parfaitement détaillée dans les spécifications GlobalPlatform **[9]**, mais simplement mettre l'accent sur `CardReset` qui, contrairement à ce que son nom laisse entendre, permet d'impacter sur la sélection implicite d'application. Par défaut, l'application sélectionnée automatiquement (après *reset* justement) est l'ISD. Il est cependant parfaitement possible d'attribuer ce privilège à une autre application et en particulier la vôtre (ou l'une des vôtres). Il ne sera alors plus nécessaire de la sélectionner avant de lui envoyer des APDU. Ceci peut se faire très facilement en ajoutant l'option `--privs CardReset` lors de l'installation de l'applet (ou `-priv 4` avec la commande `install` de GPShell). Une seule application peut disposer de ce privilège et, si celle-ci est supprimée, c'est naturellement l'ISD qui le récupérera automatiquement.

L'application ayant ce privilège possède également la capacité de modifier les octets historiques (*historical bytes*) présents dans l'ATR (message présenté automatiquement par la carte) et permettant de l'identifier. Cette information peut être encodée de façon standard ou propriétaire. Dans le cas des cartes ACOJS, par exemple, ces octets historiques sont `41 43 4F 53 4A 76 31 30 31` (avec le début de l'ATR à `3B 69 00 02`), se traduisant littéralement en la chaîne de caractères ASCII `ACOSJv101` (ou `ACOSJv204` pour une 95k). Pour une J3R150, les octets historiques sont `00 31 C1 73 C8 40 00 00 90 00` et là nous avons un encodage standard au format TLV compact (*Type Length Value* ou type taille valeur en français). Les spécifications décrivent le sens de chaque octet dans ce format et donc le type de service proposé par la carte, ses fonctionnalités, l'encodage de données, etc.

Modifier ces octets historiques n'est pas possible avec le SDK Java Card, pas plus que le fait, pour une applet, de connaître l'état dans lequel se trouve la carte (*OP_READY*, *INITIALIZED*, etc.). Ceci est en dehors des fonctionnalités supportées par le SDK et il est donc nécessaire de faire appel à un *package* Java supplémentaire (une sorte de bibliothèque statique du C ou d'un module en Python). Et cela nous amène donc à parler de l'aspect logiciel de l'expérience...

3.2 À propos du code

Je l'ai dit précédemment, les états *INITIALIZED* et *SECURED* peuvent être vus comme de simples notions « administratives » dans une politique de production d'une solution basée sur des *smartcards*. Une applet sur la carte peut par exemple refuser certaines opérations, car elle n'est pas dans le bon état et donc officiellement prête pour l'utilisateur final. Mais pour que cela fonctionne, encore faut-il que l'applet arrive à déterminer l'état de la carte.

Voici une OpenPGP card v2.0 qui a maintenant une bonne dizaine d'années. Les spécifications ont été mises à jour depuis, en version 3.4, et un nouveau modèle est disponible à l'achat. Mais vous pouvez également utiliser une Java Card 3.0.4 faisant fonctionner l'applet SmartPGP de l'ANSSI sous GPLv2 et vous aurez un résultat parfaitement similaire.

Pour faire cela, nous devons utiliser la classe `GPSystem` du package `org.globalplatform`, qui n'est pas présent dans le SDK. Nous devons donc trouver un moyen d'intégrer un tel *package* à notre projet et la procédure sera identique pour n'importe quel *package* livré sous la forme d'un fichier JAR. Dans le cas de `org.globalplatform`, nous pourrons récupérer le nécessaire directement sur le site de l'organisation de standardisation GlobalPlatform, à la section « *Technology Document Library* » **[10]**. Là, nous avons accès à la *GlobalPlatform Card API*, téléchargeable en différentes versions. Le choix de cette dernière dépendra des fonctionnalités de votre ou vos cartes, ainsi que du SDK et du convertisseur JAR/CAP utilisé. Si vous avez un doute sur cette version, commencez simplement par l'API la plus récente (1.7.1) et descendez jusqu'à ce que ça fonctionne (lors de l'installation de l'applet).

Lorsqu'on n'est pas développeur Java (ou qu'on l'est, mais qu'on repose uniquement sur des environnements intégrés clé en main, type JCIDE), intégrer un tel *package* peut être difficile. Après téléchargement depuis le site (via « *NON-MEMBER DOWNLOAD* »), on se retrouve avec une archive ZIP contenant un `gpapi-globalplatform.jar`, un `globalplatform.exp` (placé dans une arborescence) et une tripotée de documentations concernant les classes et méthodes. Que faire avec tout cela ?

Fort heureusement, Ant et *ant-javacard* **[11]**, que nous utilisons déjà, savent parfaitement prendre cela en charge. Il suffit de légèrement modifier notre `build.xml` :

```
<javacard jckit="ext/sdks/jc222_kit">
  <cap aid="f2:76:a2:88:bc:de:ad:be:ef"
       output="Hello.cap" sources="src" version="1.0">
    <applet class="dev.drrb.javacard.Hello.Hello"
            aid="f2:76:a2:88:bc:de:ad:be:ef:01"/>
    <import jar="gpapi/gpapi-globalplatform.jar"
            exps="gpapi/exports/"/>
  </cap>
</javacard>
```

Il faudra ensuite copier le `gpapi-globalplatform.jar` et toute l'arborescence où se trouve `globalplatform.exp` (qui est un fichier initialement généré par un convertisseur JAR/CAP pour éviter d'utiliser une collection de `.class`) dans un sous-répertoire `gpapi/`, et le tour est joué. Nous pouvons alors modifier notre applet ainsi :

```
[...]
import org.globalplatform.GPSystem;

public class Hello extends Applet {
[...]
   private static final byte HELLOGETSTATE = (byte)0x80;
[...]
    if (CLA != HELLO_CLA) {
       ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
    }
```

```
   switch (INS) {
[...]
         case HELLOGETSTATE:
             getGPstate(apdu);
             break;
         default:
            ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
         }
[...]
   // APDU : 80800000 (oui, j'ai changé la classe de D0 à 80)
   private void getGPstate(APDU apdu) {
      byte[] buffer = apdu.getBuffer();
      buffer[0] = org.globalplatform.GPSystem.getCardState();
      apdu.setOutgoingAndSend((short)0, (short)1);
   }
}
```

Nous disposerons alors d'un nouvel APDU permettant de connaître l'état de la carte via une simple valeur numérique (`0x01` pour *OP_READY*, `0x07` pour *INITIALIZED*, `0x0f` pour *SECURED*, etc.). Bien entendu, voir cela sous forme d'APDU et de réponse n'a d'intérêt que démonstratif, on utilisera plutôt ce mécanisme pour justement autoriser ou non le déclenchement de certaines opérations. Un exemple simple dans notre cas serait de ne permettre la décrémentation et l'incrémentation du compteur que si la carte est prête à être utilisée et donc en état *SECURED*, en testant la valeur retournée par `org.globalplatform.GPSystem.getCardState()`.

Cet exemple, utilisé avec une Java Card 2.2.2 comme la NXP J2A081, nécessitera de mettre la main sur un *package* `org.globalplatform` en version 1.0 existant sous une forme identique pour faciliter les essais sur différentes cartes. J'ai trouvé mon bonheur dans un dépôt GitHub de OpenJavaCard **[12]**, contenant un répertoire `globalplatform-1.0` dédié spécifiquement à cet usage. Avec une carte ACOSJ, la version de l'API le plus important semble être la 1.5, présent en compagnie de la 1.6 et la 1.7 dans le fichier téléchargeable de la 1.7.1. Notez au passage que le format du fichier `.exp` a son importance pour le convertisseur JAR/CAP (version 2.3 ou 2.1). L'archive ZIP de l'API 1.7.1 propose une version 2.3 adaptée sur SDK 3.0.4 dans un sous-répertoire `exports23/` (et non `exports/` qui utilise un format 2.1 pour les plateformes plus anciennes).

On pourra tester le code facilement en changeant l'état d'une carte avec `gp -k 404142434445 464748494A4B4C4D4E4F --initialize-card`, passant de *OP_READY* à *INITIALIZED* (attention, on ne revient pas en arrière). En envoyant l'APDU, on obtient en retour :

```
APDU> 8080000001
=> 80 80 00 00 01
<= 07 90 00 (3)
```

`0x07` est bien `CARD_INITIALIZED` dans l'API **[13]**, alors que pour une carte « neuve » c'est `0x01` et `CARD_OP_READY`. Nous venons d'intégrer un *package* à notre projet et avons accès, à présent, aux fonctionnalités de `org.globalplatform.*`. Notez au passage que cette sortie-écran est celle

Entre les tags NFC, les tokens RFID LF et les smartcards en tout genre, il devient forcément difficile de s'y retrouver dans sa collection. La solution consiste à investir dans ce type d'imprimante (Bluetooth LE) à étiquettes thermiques, disponible par exemple sur Amazon pour environ 30 €. Seul problème, les recharges par lot (5 rouleaux de 4 mètres) coûtent aussi cher que l'imprimante...

d'un projet personnel appelé PCSCapdu **[14]**, petit frère de NFCapdu **[15]** qui, lui, repose sur la libNFC. C'est un simple outil interactif permettant de transmettre des APDU et d'obtenir des réponses, avec support d'alias, une complétion, un historique et, depuis peu, un interpréteur Lua intégré pour écrire des scripts.

3.3 Améliorons un peu notre exemple

Notre petite démonstration détaillée dans l'article précédent consistait à maintenir un compteur interne (de points, de crédits ou autre) et à proposer trois APDU permettant respectivement d'obtenir la valeur du compteur, l'incrémenter et le décrémenter. C'est suffisant pour un premier *Hello World* Java Card, mais on peut aller un peu plus loin en s'approchant de quelque chose de plus réaliste (tout en restant hypothétique, car non sécurisé).

On peut facilement imaginer un tel mécanisme comme un système de crédits pour un distributeur de café, par exemple. Si tel est le cas, la consultation du crédit et même la décrémentation peuvent être libres, mais l'incrémentation doit supposer une forme de vérification. Lorsqu'on parle de carte à puce, ou même de carte SIM, la première méthode d'authentification qui vient à l'esprit est le code PIN (*Personal Identification Number*), une série de chiffres (généralement entre 4 et 8) faisant office de mot de passe pour débloquer des fonctions spéciales ou protégées.

Ceci est tellement commun que le SDK Java Card propose une facilité pour faire ce genre de choses ou plus exactement un type d'objet spécialement dédié à cet usage : `OwnerPIN`. Attention cependant, la mécanique de présentation du code PIN reste la tâche du développeur et ceci signifie donc de définir un nouvel APDU. La vérification du code ainsi que l'habituelle gestion des tentatives

échouées menant à un blocage définitif de l'authentification seront, en revanche, gérées de manière transparente. Pour utiliser cette fonctionnalité, nous commençons par importer et déclarer le nécessaire :

```
import javacard.framework.OwnerPIN;
[...]
   private OwnerPIN pin;
   private byte[] defaultpin = { 1, 2, 3, 4 };
```

On modifie ensuite le constructeur de notre classe `Hello` pour initialiser le code PIN par défaut à l'installation sur la carte :

```
   private Hello() {
      valeur = 10;
      pin = new OwnerPIN((byte)3, (byte)10);
      pin.update(defaultpin, (byte)0, (byte)4);
      register();
   }
```

La méthode `new` de `OwnerPIN`, permettant d'instancier l'objet, prend en argument respectivement le nombre d'essais infructueux autorisés avant blocage (3) et la taille maximum du code PIN (10). Pour utiliser tout cela, nous allons ajouter trois nouveaux APDU exactement comme nous l'avons fait la dernière fois, avec trois nouvelles instructions (octet *INS* de l'APDU) :

- `HELLOPIN` (`0x63`) : soumettant le PIN de l'utilisateur à vérification ;
- `HELLOPTRIES` (`0x62`) : permettant de consulter le nombre de tentatives restantes ;
- `HELLOPINUP` (`0x64`) : pour changer le code PIN.

Ces nouvelles instructions seront gérées comme précédemment, et comme les autres, dans le `switch/case` de `process()` en appelant différentes fonctions. La première est celle concernant la vérification du code PIN soumis :

```
   private void verifyPIN(APDU apdu) {
      byte[] buffer = apdu.getBuffer();
      short dataLen = apdu.setIncomingAndReceive();

      if (pin.check(buffer, (short)ISO7816.OFFSET_CDATA, (byte)dataLen) ==
          false) {
         ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
      }
   }
```

Comme d'habitude, on récupère les données via `buffer[]` avec ici un décalage (*offset*) spécifié par `ISO7816.OFFSET_CDATA`, et on soumet cela directement à la méthode `check` de `pin`. Le code du SDK fait tout le travail en comparant le code PIN soumis avec celui stocké, retourne `true` en cas de correspondance et active alors un drapeau signifiant cet état de fait. Si ce n'est pas le cas, en revanche, `false` est retourné et nous agissons en conséquence en retournant une

erreur, mais en coulisse, le nombre de tentatives restantes est décrémenté. Si celui-ci atteint zéro, `pin.check()` retournera alors toujours `false`. Ce compteur de tentatives est réinitialisé à la valeur par défaut (ici 3) à chaque fois qu'un bon code PIN est validé.

L'état du drapeau peut être vérifié, ailleurs, en utilisant `pin.isValidated()` et nous pouvons alors immédiatement adapter notre fonction d'incrémentation des « crédits » :

```
private void incrementer(APDU apdu) {
   if (pin.isValidated() == false)
      ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);
   if (valeur < 0xff)
      valeur++;
}
```

C'est aussi simple que cela. `incrementer()`, et donc l'APDU correspondant, ne peut plus être utilisé qu'après avoir soumis avec succès le code PIN. Nous enchaînons alors avec la fonction permettant de connaître le nombre de tentatives restantes :

```
private void pintriesleft(APDU apdu) {
   byte[] buffer = apdu.getBuffer();
   buffer[0] = pin.getTriesRemaining();
   apdu.setOutgoingAndSend((short)0, (short)1);
}
```

Sans oublier celle permettant de changer le code PIN :

```
private void changePIN(APDU apdu) {
   byte[] buffer = apdu.getBuffer();
   short dataLen = apdu.setIncomingAndReceive();

   if (pin.isValidated() == false)
      ISOException.throwIt(ISO7816.SW_CONDITIONS_NOT_SATISFIED);

   if (dataLen < 4 || dataLen > 10)
      ISOException.throwIt(ISO7816.SW_DATA_INVALID);

   pin.update(buffer, ISO7816.OFFSET_CDATA, (byte)dataLen);

   if (pin.check(buffer, (short)ISO7816.OFFSET_CDATA, (byte)dataLen)
       == false) {
      ISOException.throwIt(ISO7816.SW_SECURITY_STATUS_NOT_SATISFIED);
   }
}
```

Notez que le fait d'utiliser `pin.update()` a pour conséquence de réinitialiser le drapeau et l'utilisateur n'est alors plus authentifié. Pour éviter les problèmes, puisque cela n'est pas forcément logique, nous revalidons immédiatement le nouveau PIN.

Que se passe-t-il lorsque l'authentification est bloquée ? La réponse est simple : rien. Il n'est plus possible de soumettre un code PIN et la seule solution sera d'effacer et de réinstaller l'applet. Ce n'est pas très pratique et, dans le monde de la téléphonie, on utilise pour cela un code PUK (*Personal Unlocking Key*) qui n'est rien d'autre qu'un second code PIN. Ceci est relativement facile à implémenter ici, puisque tout ce que nous avons à faire se résume à exactement la même chose qu'avec `pin`, à commencer par un `private OwnerPIN puk`. On jonglera ensuite avec `pin` et `puk` tout en ajoutant trois autres APDU à notre collection. On modifiera alors la première condition `if` de `changePIN()` en `if (pin.isValidated() == false && puk.isValidated() == false)` et un code PIN **ou** PUK valide permettra de changer le PIN (et de faire un *reset* du compteur de tentatives, par la même occasion).

3.4 ACOSJ et gestion de mémoire

Lorsqu'on n'a pas (ou plus) l'habitude de programmer en Java, les subtilités du langage nous échappent et cela mène parfois à des problèmes. Dans le cas des Java Cards, disposant de ressources limitées (RAM et EEPROM/flash), les conséquences peuvent être fâcheuses, voire très fâcheuses, si le produit n'est pas de la meilleure qualité. Comme je l'ai dit en début d'article, les cartes ACS ACOSJ jouissent d'une assez triste réputation et ceci s'est concrétisé assez brutalement par deux fois avec les exemplaires en ma possession.

Contrairement au développement Java sur PC, ici, non seulement les types disponibles sont limités, mais nous devons également faire avec l'absence de ramasse-miettes (*garbage collector* en anglais). Pire encore, sans démarche spécifique, les allocations mémoire pour les variables, tableaux et objets se font en EEPROM afin que les données persistent entre les utilisations (comme le code PIN, notre compteur, etc.). Ce n'est que lors de la désinstallation de l'applet qu'une sorte de *garbage collector* entre en jeu et libère les ressources qui ne sont plus utilisées et référencées.

Et c'est précisément là qu'on se heurte à un problème que seule une certaine expérience permet d'éviter : les références statiques à des objets persistants. Ceci tient en peu de choses :

```
public class Hello extends Applet {
   private static byte[] mesdata;
[...]
```

Le problème se pose en déclarant un objet `static` comme un `MessageDigest digest`, par exemple, et en utilisant à répétition `digest = MessageDigest.getInstance()` via un `try/catch` pour déterminer si un algorithme est disponible dans implémentation Java Card utilisée par la puce de la carte. Le fait de déclarer cela en `static`, et `private` qui plus est, fait qu'il n'est plus possible de retrouver la référence à l'objet et certaines implémentations, comme celle d'ACS, sont alors incapables de récupérer la mémoire EEPROM allouée. La carte finit par ne plus répondre aux APDU (J2A081 et ACOSJ) et, dans le cas des ACOSJ, la carte elle-même devient muette et n'envoie plus d'ATR au *reset*. De ce fait, elle se retrouve *brickée* et littéralement bonne pour la poubelle (ou pour étaler de la pâte à braser pour la soudure des CMS).

Comme le précise une réponse éclairée d'un développeur sur StackOverflow **[16]**, la solution est simple : **n'utilisez pas de références statiques à des objets persistants** (« *Remember, static references to persistent objects are EVIL AND DANGEROUS* »).

Mon manque d'expérience évidente avec le langage Java, mais aussi la faible tolérance de l'implémentation ACS, m'a ainsi coûté deux cartes ACOSJ. Ne faites pas la même erreur.

4. ALLEZ PLUS LOIN, GRÂCE AUX PROJETS DES AUTRES

À propos d'expérience, justement, le meilleur moyen d'en gagner est de lire et comprendre le code de développeurs sachant vraiment ce qu'ils font. Et pour cela, rien de plus simple, puisqu'il existe pléthore d'applets *open source* couvrant une vaste gamme de fonctionnalités et de domaines. Un excellent point de départ est « *Curated list of JavaCard applications* » dont je vous parlais la dernière fois **[17]**, mais certaines sont plus utiles ou intéressantes que d'autres (à mon sens).

Nous avons tout d'abord l'applet *SmartPGP* de l'ANSSI **[18]** ne proposant rien de moins que de transformer votre Java Card en carte OpenPGP 3.4 permettant la signature électronique via GnuPG, le chiffrement, l'authentification, etc. Les OpenPGP Cards ne sont pas une nouveauté et il est parfaitement possible d'en acquérir une en ligne sur FLOSS Shop, par exemple (anciennement *Kernel concepts* **[19]**). Mais, en dehors du prix qui est somme tout similaire à une Java Card de bonne facture, l'intérêt est de comprendre le fonctionnement puisque le code développé par l'ANSSI (GPLv2) est une implémentation quasi complète (voir `README.md` du dépôt) des spécifications, couvrant énormément de fonctionnalités disponibles avec une Java Card 3.0.4.

IsoApplet **[20]** que nous avons déjà évoqué dans le précédent article est une solution de chiffrement et de signature électronique compatible OpenSC, proposant une génération de clé sur la carte et le stockage en PKCS#15 pour l'authentification. Comme pour SmartPGP, la version minimum de Java Card à utiliser est 3.0.4, mais une déclinaison « *Legacy* », compatible 2.2.2n est également disponible dans une branche Git distincte. Cette applet sous GPLv2 permettra une utilisation avec n'importe quelle application supportant OpenSC et/ou PKCS#15 (elle est d'ailleurs listée dans les pilotes de cartes de `opensc-tool`).

Dans un tout autre domaine, nous avons ensuite *OpenJavaCard NDEF* **[21]** qui vise cette fois les cartes de type *contactless* ou *dual interface*, pour créer un tag NFC Type 4. Le projet est initialement développé pour Java Card 2.2.2, mais un *Pull Request* existe, proposant de modifier le `build.xml` pour supporter le SDK Java Card jusqu'à 3.1.0 (et donc 3.0.4, par la même occasion). L'intérêt de ce projet, en dehors du fait de se créer un tag NFC de jusqu'à 32 Kio est, bien entendu, d'explorer l'aspect « sans contact » des Java Cards et de comprendre, en pratique, qu'il est parfaitement possible de créer (ou d'émuler) toutes sortes de tags NFC. Notez au passage qu'on trouve, sur AliExpress par exemple, des tags NFC type 4 de 32 Kio, qui sont précisément ce type d'implémentation et très probablement basés sur ce code (bien entendu, les clés permettant la gestion des applets ne sont pas connues avec ces produits et on ne peut donc pas simplement les recycler en plateforme de développement).

Plus original et peut-être pas financièrement intéressant, nous avons « de.fac2 » **[22]**, une implémentation d'un *token* Fido U2F permettant l'authentification multifacteur. Encore une fois, une carte compatible Java Card 3.0.4 sera nécessaire, et ce, en version sans contact (ou *dual*) pour une utilisation avec un smartphone. Les *tokens* U2F, avec les navigateurs web sur PC et Mac,

– Continuons notre exploration des Java Cards : jckit 3.0.4, NFC et code PIN –

se présentent sous la forme de périphériques USB HID et sont utilisés comme tels. Aucun navigateur, pour l'instant, ne supporte PC/SC ou des lecteurs CCID pour ce type d'usage. Comme le précise une note dans le `README.md`, ce projet n'est pas prévu pour une utilisation en production (d'autant que des *tokens* Fido U2F et Fido2 peuvent être trouvés en ligne pour le prix d'une Java Card), même s'il a fait l'objet de certifications (BSI + Fido level 3+) sur une carte Sm@rtCafe Expert 7.0. Si vous voulez savoir comment fonctionne une authentification U2F en interne et en détail, cette applet est clairement la voie à suivre.

Et enfin, nous avons la *Gauss Key Card* **[23]**, qui cette fois fonctionnera avec une Java Card 2.2.2, mais devant disposer de fonctionnalités sans contact. En effet, cette applet implémente un sous-ensemble réduit, mais fonctionnel du protocole utilisé par les véhicules Tesla. Une fois l'application installée dans la carte (d'après l'auteur) et appairage avec la voiture, il devient possible de l'utiliser comme une carte originale pour verrouiller, déverrouiller et démarrer le véhicule. L'objectif de Robert Quattlebaum, le développeur de l'applet, n'est pas de permettre de cloner une carte Tesla, mais simplement de comprendre le protocole. Et comme il le dit lui-même dans le `README.md` : si n'arrivez pas à imaginer pourquoi vous pourriez vouloir utiliser cette applet, alors elle n'est pas pour vous.

Une autre technique pour conserver un semblant d'ordre dans ses cartes et tags consiste à utiliser ce genre de « portes-cartes » de 8, 18 ou 28 emplacements. On en trouve partout, y compris tantôt en grandes surfaces et, en prime, on a droit aux petits chatons tout mignons.

5. POUR FINIR

Jouer avec des Java Card est un passe-temps qui peut coûter cher, sachez-le. Suite à mes expérimentations avec mes « vieilles » NXP J2A081, l'achat hasardeux de trois J3R150 sur AliExpress, la commande d'ACOSJ chez Hitools Access, puis une commande chez Cardomatic pour 5 **vraies** ACOSJ-DI 95K (faite le 05/11, reçu le 09/11), le budget commence à être conséquent, mais le résultat peu satisfaisant. Le problème, voyez-vous,

est celui de la confiance. Non dans le distributeur, mais dans le produit lui-même, car il est particulièrement difficile, en étant dans le flou complet, de savoir si un éventuel comportement étrange est du fait de son propre code, de l'origine douteuse du produit ou de la piètre qualité du matériel.

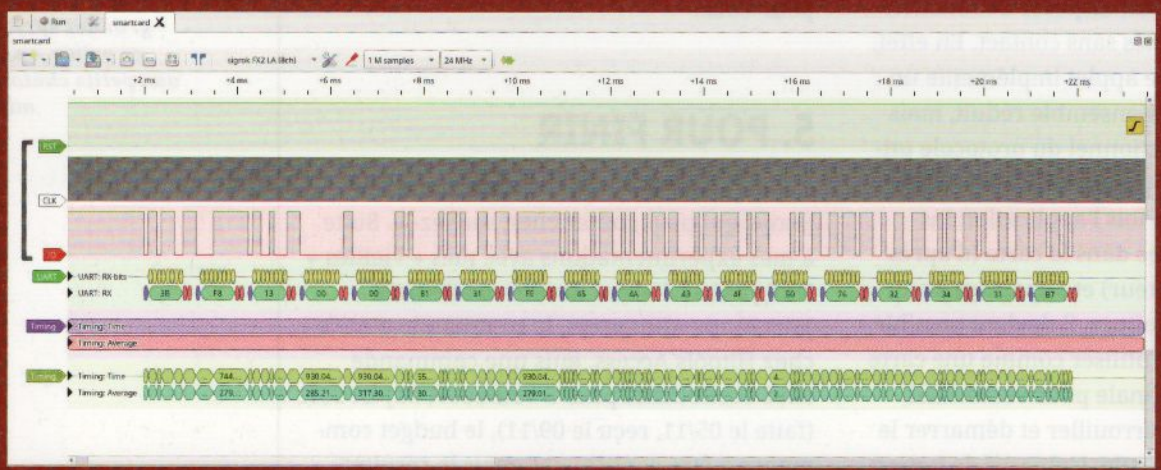
Pour être serein, il faut obligatoirement une base de référence et c'est un problème que je suis loin d'être le seul à avoir, puisque c'est l'objet même d'une *issue* laissée délibérément ouverte pour le projet SmartPGP : « *Sourcing 3.04 Javacards #17* » **[2]**. Suivant les recommandations données dans ses discussions, et comme d'autres avant moi, je me suis donc tourné vers le distributeur *Smartcard Focus* pour des NXP J3H145 Dual Interface Java Card 3.0.4 **[24]**. Là, nous ne sommes plus dans les mêmes budgets, car c'est 19,50 € la carte (HT), plus la livraison UPS à 24 €, plus les frais de douane et la TVA à la réception (merci, le Brexit) pour un total de ~90 € pour deux J3H145. C'est cher, mais c'est le prix de la tranquillité avec quelque chose de testé/recommandé par les utilisateurs de SmartPGP, incluant le support de RSA 4096.

Le bilan et la leçon sont simples, en faisant abstraction des J2A081 déjà en ma possession : au final, cette petite escapade technologique m'a coûté

UNE DERNIÈRE CHOSE...

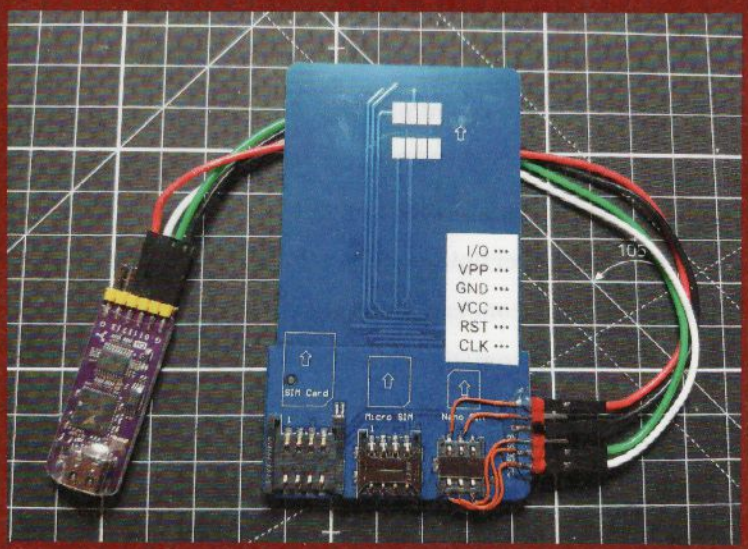
Tout ce que nous avons vu dans cet article, et dans le précédent, est fonctionnel, mais absolument pas sécurisé, alors que c'est là, précisément, l'intérêt des *smartcards* et de la myriade de fonctionnalités cryptographiques intégrées aux Java Cards. La notion de *secure channel* est un domaine à part entière (qui fera peut-être l'objet d'un futur article), mais pour montrer l'importance de ce genre de mécanismes, il suffit de se tourner vers un analyseur logique.

En effet, moyennant un budget tout à fait modeste (~10 €) permettant d'acquérir un clone chinois de Saleae Logic à base de Cypress CY7C68013 (FX2) ou CY7C68013A (FX2LP), voir l'article dans Hackable 43 par exemple [25], ainsi qu'un adaptateur multi-SIM/*smartcard* [26], il est possible d'espionner les communications entre le lecteur et la carte sans le moindre problème.



presque 130 €, certes pour des jours et des jours d'exploration et une masse importante de connaissances acquises, mais c'est un budget qui aurait pu se transformer en davantage de cartes dignes de ce nom si j'avais initialement opté pour l'option qui semblait la moins intéressante financièrement. Car oui, j'étais tombé sur le site de *Smartcard Focus* et oui, j'avais survolé les commentaires GitHub de SmartPGP, mais j'ai préféré jouer et j'ai perdu (en partie, car les J3R150 sont finalement utilisables et semblent originales).

Moralité, les *smartcards* sont comme l'outillage pour le bricolage (DeWalt, Metabo, Makita, etc.) : on finit de toute façon avec le matériel qui est digne de sa réputation (et donc coûteux). La question est juste de savoir combien d'argent on risque de dépenser (dans du Black & Decker de « papa bricole un dimanche par an ») avant de s'en rendre compte. Mais je ne suis pas négatif pour autant, car j'ai découvert un monde dont je ne soupçonnais pas l'existence et l'investissement m'oblige implicitement à creuser d'autant plus le sujet. Après tout, j'ai tout un stock de cartes à présent, des vieilles, des douteuses et des fiables. Si vous ne prenez pas garde, il vous arrivera peut-être la même chose... **DB**



Ceci ne nécessite pas d'application ou d'outil particulier, car Sigrok/PulseView fera parfaitement le travail en décodant le protocole série asynchrone établi via la broche I/O de la carte à puce. Quelques ajustements manuels sont nécessaires pour le débit non standard, déduit de la période des signaux (93 µs = 10752 b/s) et pour le format de données (1 bit de *start*, 8 bits de données, parité paire et 1,5 bit de stop), mais les APDU, dans un sens comme dans l'autre, sont parfaitement lisibles, comme le montre la capture ci-contre présentant l'ATR d'une des cartes J2A081.

Le seul moyen de sécuriser l'ensemble passe par une authentification mutuelle et du chiffrement, et donc par l'utilisation des *packages* et méthodes Java Card dédiés ainsi que, côté PC, des bibliothèques et/ou modules adaptés (OpenSC, pyscard, OpenSSL, etc.).

RÉFÉRENCES

[1] https://www.hitools-access.com/puce-a-contact-5258/acosj-gj1aacsa-123812.html

[2] https://github.com/github-af/SmartPGP/issues/17

[3] https://github.com/github-af/SmartPGP/issues/29

[4] https://github.com/nfc-tools/ifdnfc

[5] https://sourceforge.net/projects/ifdnfc/

[6] https://gitlab.com/0xDRRB/drrbOpenBSDports

[7] https://gitlab.com/0xDRRB/drrbFreeBSDports

[8] https://github.com/kaoh/globalplatform

[9] https://globalplatform.org/wp-content/uploads/2018/06/GPC_Specification-2.2.1.pdf

[10] https://globalplatform.org/specs-library/globalplatform-card-api-org-globalplatform/

[11] https://github.com/martinpaljak/ant-javacard

[12] https://github.com/OpenJavaCard/globalplatform-exports

[13] https://www.win.tue.nl/pinpasjc/docs/apis/gp211/org/globalplatform/GPSystem.html

[14] https://gitlab.com/0xDRRB/pcscapdu

[15] https://gitlab.com/0xDRRB/nfcapdu

[16] https://stackoverflow.com/questions/32764384/cannot-delete-java-card-applet

[17] https://github.com/crocs-muni/javacard-curated-list

[18] https://github.com/github-af/SmartPGP

[19] https://www.floss-shop.de/de/security-privacy/smartcards/13/openpgp-smart-card-v3.4

[20] https://github.com/philipWendland/IsoApplet

[21] https://github.com/OpenJavaCard/openjavacard-ndef

[22] https://github.com/tsenger/de.fac2

[23] https://github.com/darconeous/gauss-key-card

[24] https://www.smartcardfocus.com/shop/ilp/id~879/nxp-j3h145-dual-interface-java-card-144k/p/index.shtml

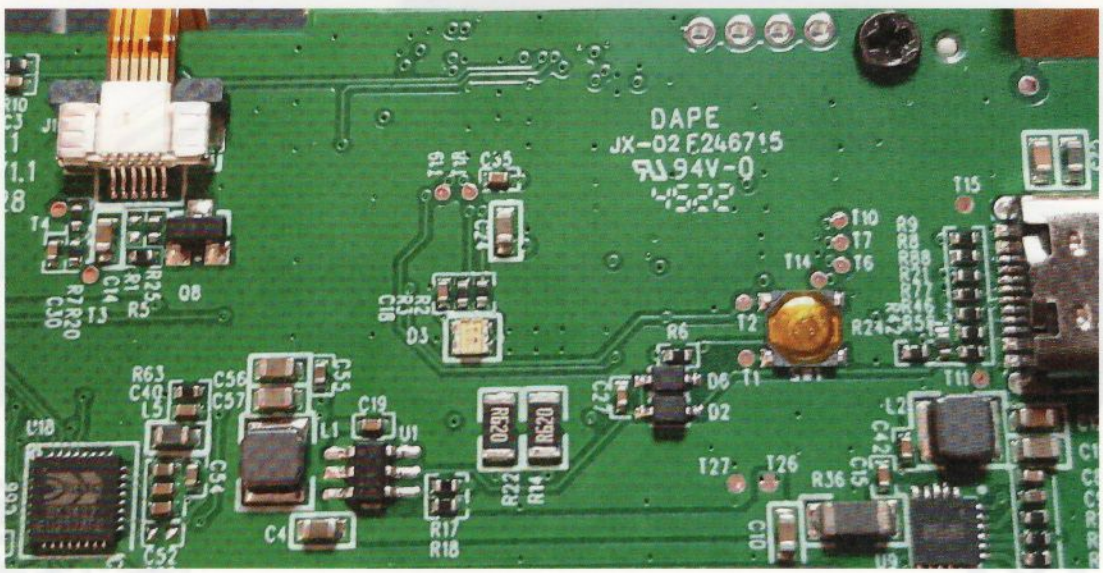
[25] https://connect.ed-diamond.com/hackable/hk-043/instrumentez-votre-analyseur-logique-avec-libsigrok

[26] https://fr.aliexpress.com/item/1005005272744876.html

REVERSE : UTILISER SON IMPRIMANTE À ÉTIQUETTE SANS BLE ET EN USB

Denis BODOR

AliExpress et Amazon sont de véritables mines d'or pour qui est prudent et prend son temps. On trouve ainsi tout un tas de petits accessoires électroniques qui sont non seulement fort pratiques, mais cachent également un petit jeu intégré. Le jeu s'appelle : « oui, mais moi je veux pas l'utiliser comme c'est prévu ». Dans cette catégorie, je vous présente l'imprimante thermique L1-A de MakeID, censée être utilisée en Bluetooth Low Energy (?) avec son smartphone et via l'application du constructeur. « Censée » est le mot important dans cette dernière phrase…



– Reverse : utiliser son imprimante à étiquette sans BLE et en USB –

Jusqu'à dernièrement, j'avais le même problème que tout le monde : l'électronique, la programmation, l'embarqué... c'est facile. Garder un semblant d'ordre lorsqu'on poursuit plusieurs projets à la fois, c'est dur. Surtout quand le ou les sujets en question ne se prêtent certainement pas à cela. Ma motivation initiale concernant l'achat d'une imprimante à étiquettes autocollantes concernait le marquage de cartes, *smartcards* et NFC, qui ont l'horrible particularité d'être, pour la plupart, de simples rectangles blancs tout ce qu'il y a de plus anonyme. Bien sûr, suite à l'achat et en raison de la facilité d'utilisation par rapport à des étiquettes papier standard en feuilles A4 prédécoupées, d'autres usages sont apparus : boîtes de rangement de modules et composants, brochage de montages, produits chimiques, panneaux de commandes, outils divers et variés, boîte aux lettres... tout y passe.

Jusqu'au moment où, n'ayant pas sous la main un bloc d'alimentation USB-C pour recharger l'accu intégré, j'ai branché la bête sur le hub de mon PC GNU/Linux. Et là, surprise !

```
usb 3-2.1.3.1: new full-speed USB device
  number 18 using xhci_hcd
usb 3-2.1.3.1: New USB device found,
  idVendor=09c5, idProduct=0200, bcdDevice= 2.00
usb 3-2.1.3.1: New USB device strings:
  Mfr=0, Product=0, SerialNumber=0
usblp 3-2.1.3.1:1.0: usblp0: USB Bidirectional printer
  dev 18 if 0 alt 0 proto 2 vid 0x09C5 pid 0x0200
```

Ce n'est pas un simple port de recharge, mais une interface permettant d'utiliser l'imprimante via une connexion filaire et l'appareil est directement reconnu comme une imprimante standard USB. Bien entendu, la maigre documentation du matériel n'en fait pas mention et une rapide recherche sur le Web montre qu'il ne semble pas y avoir de pilote ou d'applications dédiées. Certains parlent d'un hypothétique support pour Windows 10, mais pour un modèle différent (PeriPage A6), et de toute façon, il ne s'agit pas seulement de se servir du matériel comme une mini-imprimante standard. Non, la perspective d'utiliser ce matériel, couplé à un SBC Raspberry Pi, par exemple, laisse entrevoir des possibilités très intéressantes : génération de codes, mots de passe temporaires cycliques, génération de code-barres non standard, billetterie, etc. Bref, interfacer cela avec un code maison sans passer par le Bluetooth est un objectif qui mérite d'être poursuivi et atteint.

1. L'IMPRIMANTE L1-A, ALIAS MAKEID L1, ALIAS PERIPAGE A6 ?

L'imprimante en question a été achetée, sans surprise, sur Amazon **[1]** au prix d'environ 30 euros. C'est une imprimante thermique, sans encre ou toner donc, utilisant un rouleau de 16 mm, 12 mm ou 9 mm de large pour 4 mètres de long. Ce consommable autocollant, décliné en plusieurs couleurs (blanc, transparent, vert, jaune, rose, etc.) est vendu en packs de 5 rouleaux au même prix que l'imprimante elle-même (classique). Notez qu'il ne s'agit pas de papier,

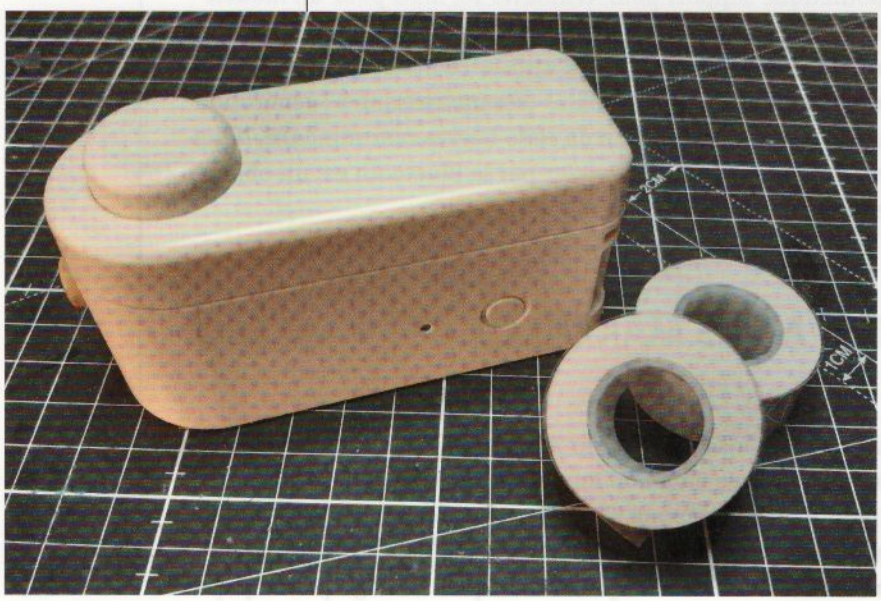
mais de différents types de plastiques laminés, prenant en sandwich une substance thermochromique similaire aux tickets de caisse et de terminaux de paiement (TPE). Le résultat est donné pour être résistant à l'eau, aux corps gras et à l'usure de surface, en prenant bien soin d'oublier de mentionner les solvants et bien entendu la chaleur.

Mais peu importe, le matériel fait son travail correctement pour son prix, une fois l'application Android ou iPhone (non testé) installée sur le smartphone compatible Bluetooth Low Energy (BLE). Notez au passage que l'imprimante arrive avec, dans la documentation, un QRcode à flasher pour installer l'application *MAKEID-Life*, sous la forme d'un fichier APK téléchargé depuis le site du constructeur. Il va de soi qu'il est absolument hors de question de procéder de la sorte et on se tournera alors vers les *stores* classiques **[2] [3]**, faisant un minimum d'analyse et de tri dans ce qui est proposé.

Les manipulations qui vont suivre ont toutes été réalisées sur un smartphone Android Samsung Galaxy A40, ainsi que sur une tablette Samsung Galaxy Tab A (8.0", 2019), les deux sous Android 11. Je ne sais pas s'il est possible de faire de même avec un périphérique iOS, mon dernier produit Apple avait une batterie amovible et je n'aime pas les écosystèmes en vase clos.

L'imprimante est faite pour être utilisée ainsi, en Bluetooth, avec l'application du constructeur qui, bien qu'utilisable pour des impressions ponctuelles et manuelles, est parfaitement incapable de générer et d'imprimer automatiquement du contenu. Pour utiliser le matériel avec un PC ou un SBC, deux options sont envisageables, via Bluetooth ou par la connexion filaire USB-C. La première option nécessite une interface Bluetooth, ce qui est le cas des Raspberry Pi, mais plus rare sur d'autres SBC. Bien sûr, il est toujours possible d'ajouter un *dongle* Bluetooth, mais ceci ne rend pas plus facile la prise en charge via un code « maison » (croyez-moi, j'ai tâté du Bluetooth fût un temps, ce n'est pas amusant).

D'autre part, nous avons la connexion USB qui est générique. Le produit apparaît, en effet, comme un périphérique USB de classe *USB Printer*, d'où le `usblp0` du message du noyau Linux, qui est parfaitement



L'imprimante thermique L1-A, qui permet d'imprimer des étiquettes plastiques autocollantes via votre smartphone, est compacte et de relative bonne facture pour son prix (~30 €). Ce que ne dit pas le descriptif du produit, en revanche, c'est qu'elle cache un petit secret...

standardisé. L'entrée `/dev/usb/lp0` qui apparaît alors pourrait laisser penser qu'il suffit de faire un `echo "coucou" > /dev/usb/lp0` pour obtenir un résultat, mais il n'y a rien de plus faux. Le standard décrit l'interface au niveau USB, mais absolument pas le format des données échangées. Nous avons donc bien un pseudofichier sur lequel écrire, mais nous devons tout d'abord savoir quoi écrire et quel langage utilise effectivement l'imprimante. Et c'est précisément là que les choses deviennent intéressantes...

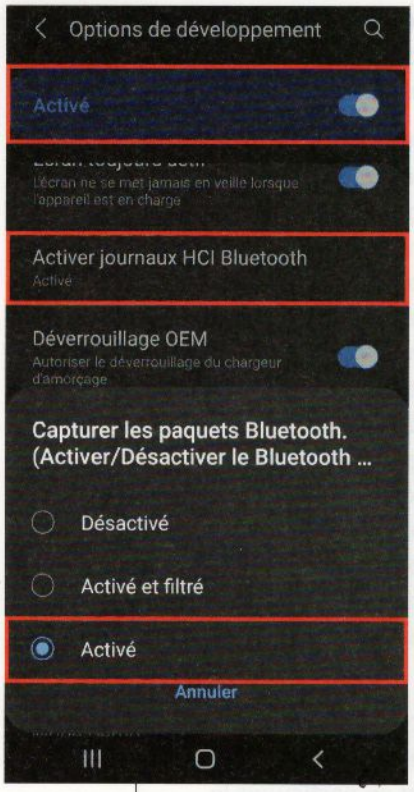
2. ANALYSER LES ÉCHANGES BLUETOOTH

Pour apprendre le « langage » de l'imprimante, en l'absence de documentation ou d'analyse préexistante du protocole, la seule approche possible est d'espionner les communications entre le smartphone et le périphérique, en espérant repérer des transactions significatives entre les deux entités. Noyé dans la masse de données doit, forcément, se trouver un lot de données qui représente l'image qui est imprimée. À nous de le trouver. Notez bien qu'on part ici d'une théorie selon laquelle le protocole est au moins vaguement identique en USB et en BLE (ce qui s'avère finalement être exact, car sinon il n'y aurait pas d'article).

Pour ce faire, nous avons plusieurs options à notre disposition, et deux en particulier. La première qui semble logique et que j'ai naïvement tentée consiste à espionner « de l'extérieur » en *sniffant* la communication au moment de l'impression ou, du moins, dès le lancement de l'application Android. Si vous cherchez sur le Web, le mot *Ubertooth One* **[4]** est sans le moindre doute le premier sur lequel vous allez tomber. L'*Ubertooth* est une plateforme de développement dédiée aux expérimentations autour du BLE et du *Bluetooth Classic* (alias BR/EDR). Créé par *Great Scott Gadgets* et son fondateur Michael Ossmann, également à l'origine du *HackRF One* et du *YARD Stick One*, l'*Ubertooth* permet de capturer et de surveiller les communications Bluetooth. Le problème de ce matériel est son prix, quelque 200 € pour la version officielle produite par NooElec, même s'il existe des versions plus économiques sur AliExpress, par exemple (~70 €).

Dans la même catégorie, mais plus accessible financièrement, une alternative est la plateforme de développement nRF52* de Nordic Semiconductor, et en particulier le *nRF52840 Dongle*. Initialement produit pour être une base générique pour les développements autour des technologies Bluetooth (EDR et LE), Thread, Zigbee, 802.15.4, ANT et 2,4 GHz en général, ce matériel qui se présente sous la forme d'une clé USB peut être *flashé* avec le *firmware* « *nRF Sniffer for Bluetooth LE* » qui, vous l'aurez compris, le transforme en *sniffer* BLE. Le *nRF52840 Dongle* ne coûte qu'une quinzaine d'euros et il existe même une version, créée par Makerdiary, en boîtier plastique pour quelque 25 € **[5]**. La plateforme *reflashée* s'utilise avec un plug-in pour Wireshark permettant de capturer et d'analyser les trames BLE.

Le problème avec ces deux solutions est la faible fiabilité de la technique elle-même. BLE est un protocole relativement complexe au niveau matériel, utilisant plusieurs canaux de fréquences et un certain nombre de mécanismes de sécurité. Même sans chiffrement des communications, capturer efficacement un échange entre un hôte et un périphérique est délicat et repose en partie sur la chance. Pour maximiser la probabilité d'une capture réussie, il faut en principe utiliser non pas un de ces équipements, mais trois : un pour chaque canal d'*advertisement* (37, 38 et 39) sur les 40 que compte le



standard. Ceci en raison du fait que les communications sont effectivement initiées sur ces canaux, en utilisant les adresses MAC physiques des périphériques, mais qu'une des premières phases de l'échange consiste à changer ces adresses aléatoirement. Il faut donc suivre la transaction dans son ensemble pour capturer les informations de manière cohérente, et non sous la forme d'une soupe presque inintelligible.

En réalité, le réflexe « *Ubertooth One* », ou même « *nRF52840 Dongle* » n'est pas le bon. Ceci ne doit être envisagé **que** s'il n'existe pas de possibilité de capturer le trafic directement sur l'une des entités impliquées. Or, justement, nous avons ici un hôte dont nous avons le contrôle : le smartphone Android. Dès lors, nous pouvons, plus facilement qu'on ne le pense, capturer les données de façon très fiable et sans le moindre problème. Il n'est même pas nécessaire de *rooter* le smartphone, puisque ceci est prévu directement dans les outils développeurs. Ce dont nous avons besoin, c'est tout bonnement d'activer la journalisation des communications Bluetooth HCI (*Host Controller Interface*).

Pour cela, la première chose à faire est d'activer les fonctionnalités développeurs sur le smartphone. Rendez-vous dans les paramètres Android, puis ***À propos du téléphone***, ***Information sur le logiciel***, et tapotez au moins sept fois sur ***Numéro de version***. Un message vous signalera alors que le mode développeur a été activé. Toujours dans les paramètres se trouve à présent une nouvelle entrée (généralement, la dernière) : ***Options de développement***. Là, vous verrez un ***Activer les journaux HCI Bluetooth*** qu'il vous suffira d'activer (Figure ci-dessus).

Dès lors, toutes les transactions Bluetooth seront automatiquement enregistrées dans un fichier dédié de la mémoire du smartphone. Assurez-vous également que le ***Débogage USB*** est activé pour pouvoir utiliser, par la suite, l'outil `adb`, l'*Android Debug Bridge* des outils de développement Android.

Ce fichier ne peut malheureusement pas être consulté ou récupéré seul, car placé dans un espace non accessible (`/data`) du système de fichiers du smartphone (s'il n'est pas *rooté*). Pas de problème cependant, puisque nous pouvons télécharger, en USB, l'ensemble du rapport de *bug* Android sous la forme d'un fichier ZIP. Pour cela, commencez par connecter votre smartphone à une machine GNU/Linux (Windows et macOS sont certainement utilisables également) et utilisez la commande `adb` (du paquet éponyme avec Raspbian/Debian et consorts) pour lister les périphériques présents :

```
$ adb devices
* daemon not running; starting now at tcp:5037
* daemon started successfully
List of devices attached
R58NA1R381Z     device
```

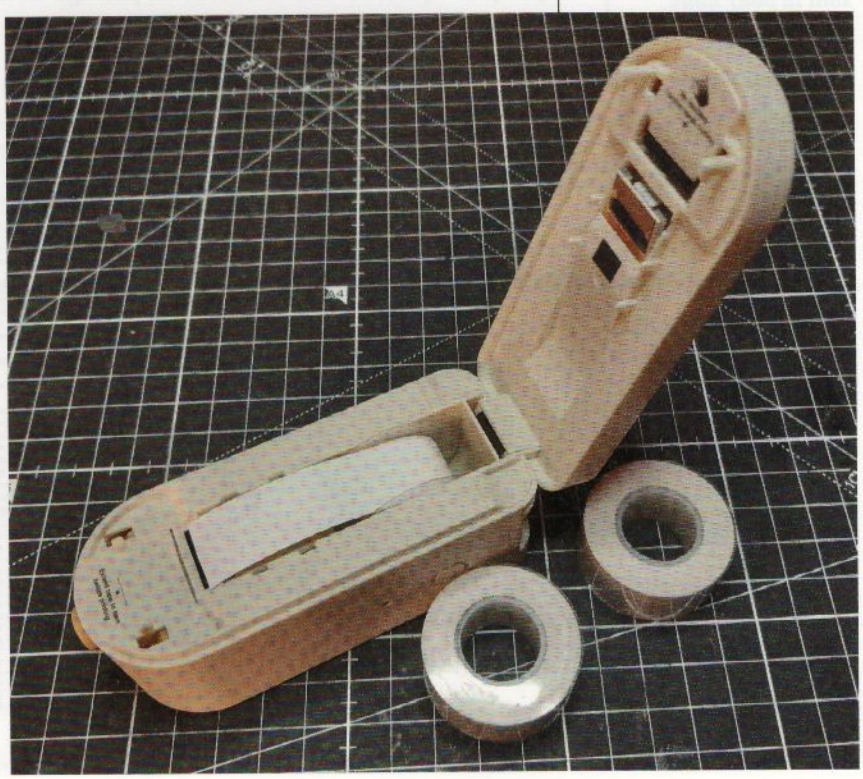
Si c'est la première connexion, vous devrez confirmer l'autorisation sur l'écran du smartphone pour établir la communication. Ceci fait, le serveur `adb` reste en mémoire (jusqu'à un `adb kill-server`) et vous pourrez utiliser directement `adb` pour récupérer le rapport de *bug* :

```
$ adb bugreport
[  8%] generating dumpstate-2024-01-09-11-46-26.zip
...
/data/user_de/0/com.android.shell/files/bugreports/
  dumpstate-2024-01-09-11-46-26.zip:
1 file pulled, 0 skipped. 32.3 MB/s
  (23364946 bytes in 0.690s)
```

L'opération peut prendre un temps conséquent, dépendant de la rapidité du smartphone et de la taille des données à récupérer. Vous obtenez alors, dans le répertoire courant, le fichier `dumpstate-xxxx-xx-xx-xx-xx-xx.zip` horodaté avec la date et heure de l'opération. Vous pourrez alors le décompresser pour extraire le journal qui nous intéresse : `FS/data/log/bt/btsnoop_hci.log`.

Bien entendu, la récupération doit avoir lieu après la capture des données. Pour cela, voici comment je procède :

- désactivation de la capture ;
- désactivation du Bluetooth ;
- fermeture de toutes les applications sur smartphone ;
- activation de la capture ;
- activation du Bluetooth ;
- allumage du périphérique Bluetooth ;
- lancement de l'application cible (MAKEID-Life) ;
- connexion à l'imprimante ;
- impression d'une étiquette avec quelque chose de « reconnaissable » ;
- fermeture de l'application ;
- désactivation du Bluetooth ;
- désactivation de la capture ;
- transfert du rapport de *bug*.



Les étiquettes plastiques sont vendues sous la forme de rouleaux et se déclinent en plusieurs largeurs, couleurs et formats. Le rechargement est simple, et pour les étiquettes « en continu », un outil de coupe est intégré à l'imprimante.

Notez que les fichiers `dumpstate*` peuvent être assez gros et sont conservés par le smartphone dans `/bugreports`. Il peut être intéressant de supprimer le contenu de ce répertoire (via `adb shell`) pour gagner un peu de place, une fois les données transférées sur l'ordinateur.

Le fichier `btsnoop_hci.log` n'est pas un fichier texte, mais un binaire dans un format proche du standard décrit par la RFC 1761 et contenant quelque chose de similaire aux données d'un fichier PCAP (typiquement, une capture de trames et paquets réseau). Et comme pour les PCAP, l'outil par excellence pour lire ce fichier est Wireshark. Celui-ci permet non seulement de voir le contenu du fichier, mais en interprète le contenu en analysant chaque couche de protocole, en utilisant des filtres pour écarter ou sélectionner certains paquets, ou encore en procédant à des recherches sur des données spécifiques. Wireshark est un outil réellement très complet, capable, historiquement, de traiter des captures réseau, mais il peut également prendre en charge les trames USB, AX.25, WCDMA, SIP, i²c, ISDN, etc.

2.1 Analyse des échanges avec l'imprimante et surprise

À l'ouverture du fichier avec Wireshark, on est assailli par une quantité de messages, étant donné qu'absolument tout a été capturé et se présente à nous en bloc (Fig. 1). En défilant tout cela et grâce au travail de décodage de l'outil, on voit des échanges entre l'hôte (Android) et le contrôleur Bluetooth, des scans, des réponses de périphériques, des associations, un passage en mode « *privacy* », le choix d'une adresse aléatoire, etc.

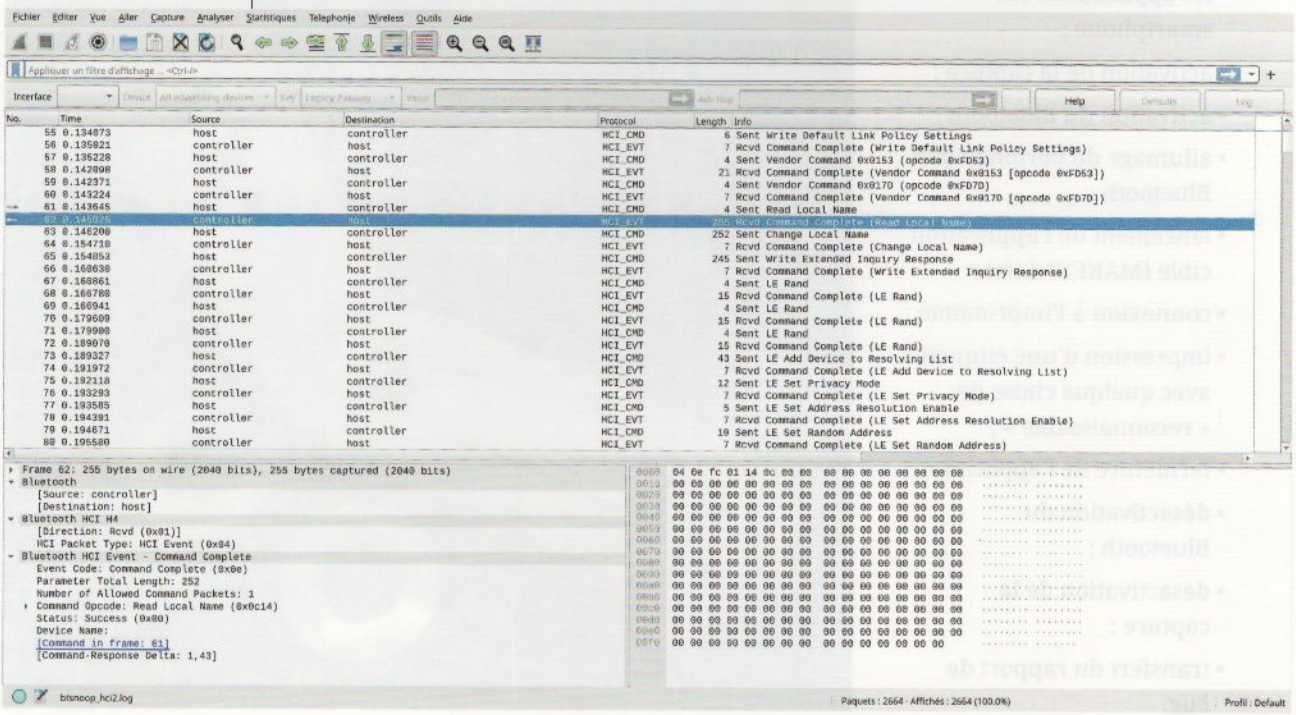
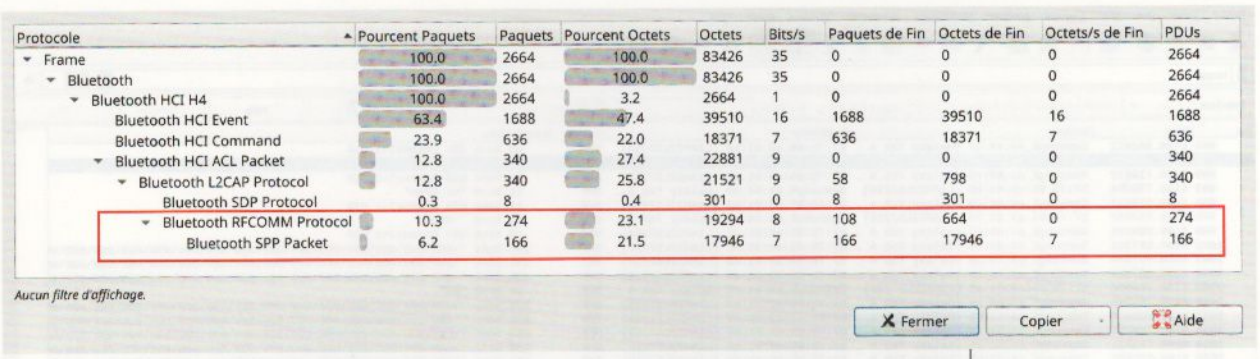


Figure 1 : L'ouverture du fichier de capteur avec Wireshark nous submerge d'informations, mais les filtres sont là pour nous aider.



Protocole	Pourcent Paquets	Paquets	Pourcent Octets	Octets	Bits/s	Paquets de Fin	Octets de Fin	Octets/s de Fin	PDUs
Frame	100.0	2664	100.0	83426	35	0	0	0	2664
Bluetooth	100.0	2664	100.0	83426	35	0	0	0	2664
Bluetooth HCI H4	100.0	2664	3.2	2664	1	0	0	0	2664
Bluetooth HCI Event	63.4	1688	47.4	39510	16	1688	39510	16	1688
Bluetooth HCI Command	23.9	636	22.0	18371	7	636	18371	7	636
Bluetooth HCI ACL Packet	12.8	340	27.4	22881	9	0	0	0	340
Bluetooth L2CAP Protocol	12.8	340	25.8	21521	9	58	798	0	340
Bluetooth SDP Protocol	0.3	8	0.4	301	0	8	301	0	8
Bluetooth RFCOMM Protocol	10.3	274	23.1	19294	8	108	664	0	274
Bluetooth SPP Packet	6.2	166	21.5	17946	7	166	17946	7	166

Aucun filtre d'affichage.

Fermer | Copier | Aide

Figure 2 : La hiérarchie de protocoles donne généralement de bonnes pistes lorsqu'il s'agit de trouver des données brutes.

Tout ceci ne nous avance guère. Noyée dans la masse, nous avons certainement l'information qu'il nous faut et nous pourrions, par exemple, commencer à filtrer très simplement. Pour cela, dans la partie inférieure gauche est présentée la structure d'un paquet, avec chaque couche du protocole clairement détaillée. Là, nous pouvons cliquer un élément qui nous intéresse, puis utiliser le clic droit pour choisir ***Appliquer comme un filtre*** puis, au choix ***Sélectionné*** ou ***Non sélectionné*** pour, respectivement, ne conserver que les paquets correspondants ou, au contraire, les cacher. On peut répéter l'opération à plusieurs reprises en combinant les filtres ainsi spécifiés à l'aide d'opération logiques (*ET* ou *OU*). Le champ sous la barre de boutons se remplit alors d'expressions, comme par exemple `(bthci_acl.src.role == 2) && !(btl2cap.psm == 0x0003)` qu'on finira, à force, par utiliser presque intuitivement en lieu et place des menus.

Personnellement, si je peux éviter ce travail de fourmi consistant à disséquer la capture comme on le ferait avec un poisson en retirant une à une chaque horrible arête (vous avez dit « brochet » ?), je le fais. Je commence donc généralement par avoir un aperçu général en affichant les statistiques de la capture, via le menu ***Statistiques*** et ***Hiérarchie des protocoles*** (Fig. 2).

Cette approche est particulièrement utile pour le Bluetooth, car c'est un gros empilement de protocoles, à l'instar de ce qu'on trouve en réseau. Là, on peut voir clairement jusqu'où descend (ou monte) le terrier du lapin et on constate immédiatement que tout en haut des couches (en bas sur la capture d'écran) nous trouvons des échanges SPP. SPP pour *Serial Port Profile* est un profil Bluetooth souvent confondu ou fusionné avec RFCOMM (*Radio Frequency COMMunication*) alors qu'en réalité ce dernier est le protocole sur lequel repose SPP. L'objet de SPP/RFCOMM est de fournir une connectivité série *over* Bluetooth, elle-même construite sur L2CAP (*Logical Link Control and Adaptation layer Protocol*) qui se charge, entre autres, de découper/assembler les paquets.

Ce que nous avons donc sous les yeux est un échange équivalent à une communication série entre le smartphone Android et l'imprimante, passant par toutes les couches de la pile Bluetooth. Pour dépatouiller tout cela, il existe un filtre magique dans Wireshark. Cliquez simplement dans le fameux champ permettant de saisir des filtres, tapez `btspp` et validez : pouf (Fig. 3, page suivante) ! La magie opère !

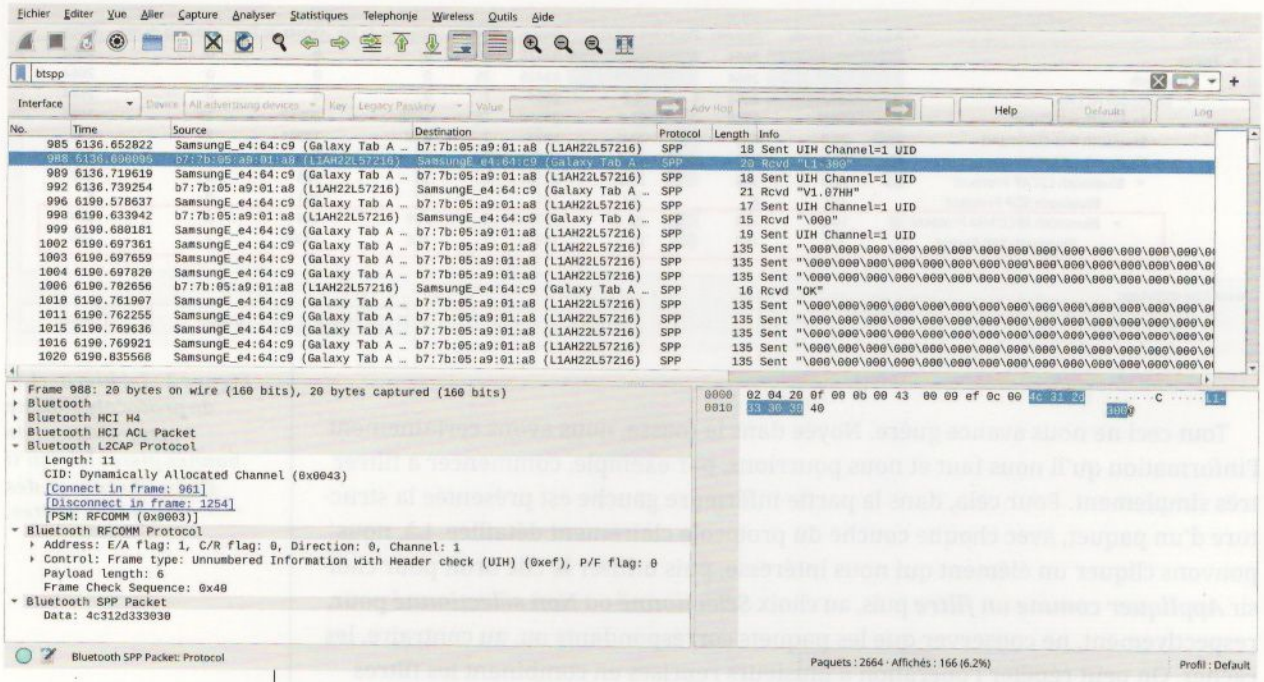


Figure 3 : Le filtre btspp permet de limiter instantanément l'affichage au protocole SPP. Tout devient, d'un coup, bien plus intelligible !

Nous ne voyons plus que les échanges de plus haut niveau et avons une belle surprise. Les échanges ne se font en réalité pas du tout en Bluetooth LE mais en Bluetooth Classic (ou BR/EDR), car il n'existe pas de RFCOMM/SPP au-dessus de L2CAP en BLE qui utilise ATT/GATT pour ce genre de chose. C'est très intéressant, car l'imprimante en question est annoncée comme BLE, et plus exactement comme ayant une « *Interface : Bluetooth BLE/SPP* », ce qui n'a pas de sens en soi.

Chose vaguement surprenante, le périphérique apparaît bien, lors d'un scan BLE, comme Bluetooth Classic avec une RPi :

```
$ hcitool scan
Scanning ...
   B7:7B:05:A9:01:A8  L1AH22L57216
   D0:05:2A:B8:6A:D3  Bouygtel4K

$ hcitool lescan
LE Scan ...
B7:7B:05:A9:01:A8 (unknown)
B7:7B:05:A9:01:A8 L1AH22L57216
F5:2B:47:7F:3B:A8 Ion 200 RT
6A:C2:F6:5E:35:E4 (unknown)
[...]
```

Il y a fort à parier que le contrôleur intégré, voir le microcontrôleur lui-même, supporte de base BLE/BR/EDR et de ce fait qu'il répond naturellement aux demandes dans les deux standards. C'est d'ailleurs une très bonne chose d'avoir

– Reverse : utiliser son imprimante à étiquette sans BLE et en USB –

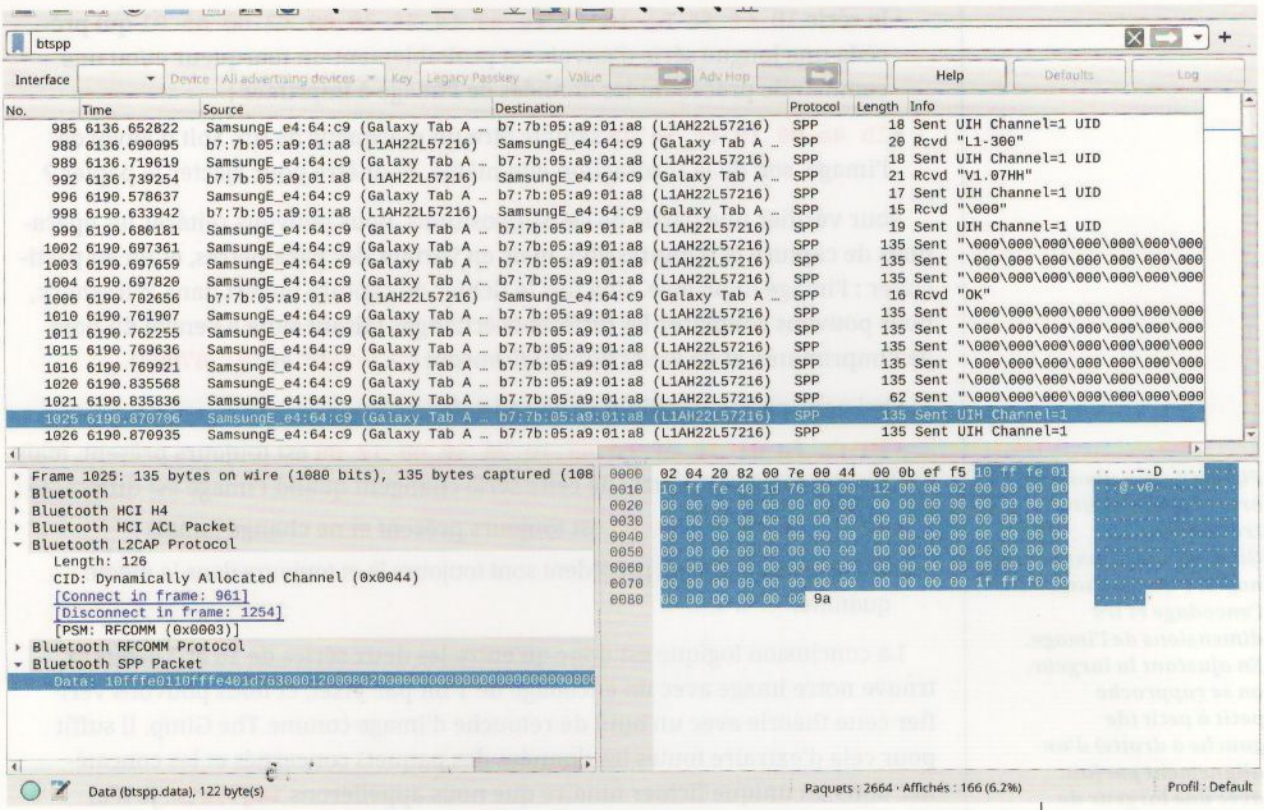


Figure 4 : En réduisant le champ des recherches au profil SPP, nous voyons enfin des échanges qui peuvent nous intéresser.

opté pour la méthode Android, car l'approche *nRF Sniffer* avec un *nRF52840 Dongle* n'aurait donc forcément mené à rien...

Quoi qu'il en soit, nous avons drastiquement restreint le champ de données et voyons enfin quelque chose (Fig. 4). On constate, ne serait-ce que sur la figure 4, que nous avons quelques mentions ASCII relativement explicites. `"L1-300"` et `"V1.07HH"` sont des réponses de l'imprimante aux demandes qui précèdent et sont constituées de seulement 4 octets : `10 ff 20 f0` et `10 ff 20 f1`. Une troisième demande suit, `10 ff 40` avec comme réponse un simple `00`.

S'en suit une tripotée d'envois de `0x00`, en paquet de 122 octets, se concluant par une réponse `"OK"`, puis à nouveau une série de `0x00` en lots de 122 se terminant par un envoi de seulement 48 octets à `0x00`. Et là, juste ensuite, nous tombons sur `10 ff fe 01 10 ff fe 40 1d 76 30 00 12 00 08 02`, suivi d'une très longue série d'envois de données en paquets de 122 octets se terminant par un dernier envoi de seulement 111 octets, principalement composé de `0x00` et se concluant avec `1b 4a 40 10 ff fe 45`.

Les suppositions que nous pouvons faire à ce stade sont les suivantes :

- les données débutant par `10 ff` sont des commandes ;
- les séries d'envois se terminent avec un paquet n'ayant pas 122 octets de données, car le dernier paquet transporte probablement le « reste » qui n'est pas un multiple de 122 ;

- la série `10 ff fe 01 10 ff fe 40 1d 76 30 00 12 00 65 01` qui précède une longue série d'envois est probablement un marqueur et/ou une commande précédant les données de l'image à imprimer ;
- `1b 4a 40 10 ff fe 45` semble être un marqueur de fin. Soit d'envoi de l'image, soit de la transaction elle-même. Peut-être pour éjecter le papier ?

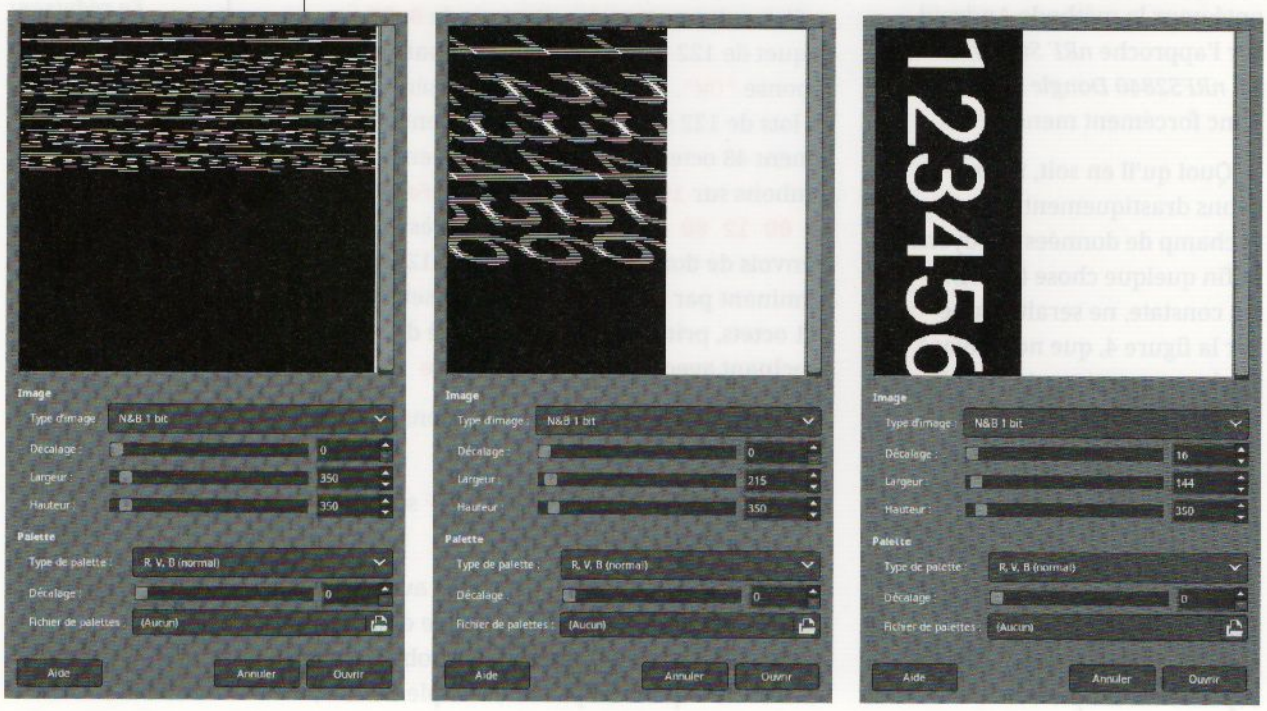
Pour vérifier une partie de ces suppositions, nous pouvons réitérer les opérations de capture et d'impression, mais en variant des paramètres, et un en particulier : l'image imprimée. Une fois le fichier de capture ouvert dans Wireshark, nous pouvons comparer les différents échanges faisant suite à l'envoi du nom de l'imprimante et de la version du *firmware* (`"L1-300"` et `"V1.07HH"`).

Ceci nous permet d'affiner et de constater que :

- `10 ff fe 01 10 ff fe 40 1d 76 30 00 12 00` est toujours présent, mais les deux derniers octets de cette série changent quand l'image est différente ;
- `1b 4a 40 10 ff fe 45` est toujours présent et ne change jamais ;
- les séries de `0x00` qui précèdent sont toujours là et toujours dans la même quantité.

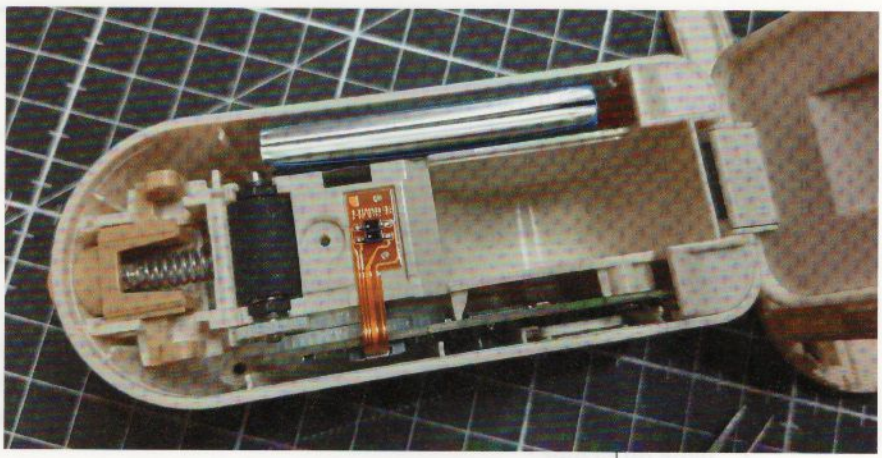
La conclusion logique est donc qu'entre les deux séries de 16 et 7 octets se trouve notre image avec un encodage de 1 bit par pixel, et nous pouvons vérifier cette théorie avec un outil de retouche d'image comme The Gimp. Il suffit pour cela d'extraire toutes les données des paquets concernés et les concaténer dans un unique fichier binaire que nous appellerons `img.data`. Je n'ai

Figure 5 : Des données brutes peuvent être traitées par The Gimp en spécifiant le nombre de couleurs, l'encodage et les dimensions de l'image. En ajustant la largeur, on se rapproche petit à petit (de gauche à droite) d'un alignement parfait, avec une largeur de 144 pixels.



malheureusement pas trouvé de solution rapide permettant de faire cela dans Wireshark et ai été obligé de passer sur chaque paquet, cliquer sur le champ « Data » de la section SPP et utiliser ***Exporter Paquets Octets***... 53 fois ! Il doit exister un moyen, mais excité par la probabilité de vérifier ma théorie, je n'ai pas cherché outre mesure, préférant concaténer ensuite les fichiers à l'aide d'un simple `cat` redirigé vers `img.data`.

Ce fichier pourra être ouvert avec The Gimp qui nous demandera de lui indiquer les caractéristiques de l'image, dont son type (N&B 1 bit ici) et ses dimensions (Fig. 5). Pour aligner les pixels, on influera simplement sur la largeur qui, progressivement, rendra l'image de plus en plus lisible. Il n'est même pas nécessaire de supprimer les « commandes » en début et en fin de données, puisqu'un simple décalage de 16 octets élimine l'information parasite. Au final, on se rend compte qu'effectivement ces données représentent l'image brute, mais également que la largeur de cette dernière est de 144 pixels. En utilisant le décalage de 16 pixels en début d'image, on constate également que les données de fin finissent seules sur une ligne et qu'en ajustant la hauteur de l'image pour les faire disparaître, on trouve donc également le nombre de pixels sur cet axe, ici, 357.



Je n'ai, forcément, pas résisté à la tentation de jeter un œil à l'intérieur du périphérique. Par simple curiosité, mais également parce qu'à un moment ou un autre, l'accu intégré (en haut) rendra l'âme et devra être remplacé.

Revenons donc un instant sur les deux fameux octets changeant à la fin du « message » placé avant les données de l'image. Bien sûr, vous aurez ici le résumé simplifié, mais dans les grandes lignes, après plusieurs essais et ouvertures avec The Gimp, la lumière est apparue. Prenons l'exemple de la capture initiale avec `65 01`. On pourrait penser qu'il s'agit de la quantité d'octets composant l'image, mais 0x6501 équivaut à 25857, alors que le fichier `img.data` fait 6449 octets (6426 pour l'image seule). Cette information n'est peut-être pas 0x6501 mais 0x0165 (*little endian*), ce qui fait 357 en décimal et... oh ! 357 est ce qui semble être la hauteur de l'image ! Ce qui est encodé n'est ni la taille des données ni le nombre de pixels, c'est le nombre de lignes de 144 pixels qui composent l'image. Et effectivement, 357 lignes de 144 pixels nous donnent 51408 bits, soit (357*144)/8 = 6426 octets. Bingo !

Nous pourrions tourner en rond, tester et retester encore pour voir si d'autres éléments varient, mais qu'auriez-vous fait à ma place ? Bien sûr, il fallait tester. Pour cela, inutile de programmer quoi que ce soit, le système dispose des outils nécessaires pour réaliser cela à la main avec un bon éditeur de code (Vim/Neovim donc). Il suffit de créer un fichier contenant tout d'abord 20 lignes de :

```
00000000000000000000000000000000000000
```

suivies de 180 lignes de :

```
000000000000000ffffffffffff0000000000
```

puis à nouveau 20 lignes de :

```
00000000000000000000000000000000000000
```

Chaque ligne est composée de 36 caractères et à raison de deux caractères hexadécimaux par octet, nous avons des lignes de 18 octets, soit 144 bits. Et 20 + 180 + 20 nous donnent 220 lignes, soit 0xdc en hexa, et donc `dc 00` sur 16 bits en *little endian*. Nous ajoutons donc, en première ligne du fichier :

```
10fffe0110fffe401d7630001200dc00
```

Et en dernière ligne :

```
1b4a4010fffe45
```

On enregistre le fichier sous, par exemple, `tmp.hex` et on demande gentiment à `xxd` d'en faire un fichier binaire (les sauts de lignes ne sont pas interprétés) :

```
$ cat tmp.hex | xxd -r -p - > tmp.bin
```

Le fichier obtenu fait 3983 octets, ce qui correspond bien à (220*144)/8+16+7 et il ne nous reste plus qu'à envoyer tout cela à l'imprimante (l'utilisateur courant doit faire partie du groupe `lp`, ou il faut utiliser `sudo`/`doas`) avec :

```
$ cat tmp.bin > /dev/usb/lp0
```

Et, sous nos yeux émerveillés, un petit rectangle noir apparaît magiquement en sortie de l'imprimante. Victoire ! Nous avons imprimé en USB depuis un SBC GNU/Linux.

3. IMPLÉMENTER UN PETIT OUTIL

À présent, nous sommes en mesure de produire une sortie sur l'imprimante, mais passer par l'édition d'un fichier texte contenant des valeurs hexadécimales n'est, bien entendu, pas une solution viable, même pour un *hack*. L'ensemble de la procédure pour faire de même à partir d'une image au format JPEG ou PNG, par exemple, est finalement relativement simple :

- orienter l'image correctement. On part ici du principe que l'image en entrée est en paysage, mais l'imprimante produit une sortie avec des données en portrait ;
- redimensionner l'image en imposant une largeur de 144 pixels et en ayant une hauteur proportionnelle ;

- réduire le nombre de couleurs à 2 (1 bit, blanc/noir), de préférence en utilisant un *dithering* pour simuler d'éventuelles nuances ;
- transformer l'image composée de pixels en un tableau de bits ;
- utiliser la nouvelle hauteur de l'image pour encoder les deux derniers octets du message de départ ;
- ajouter ce message avant les données dans le tableau ;
- ajouter le message de fin après les données ;
- envoyer le tout sur `/dev/usb/lp0` (ou `lp1`, `lp2`, etc., selon le nombre d'imprimantes USB déjà présentes).

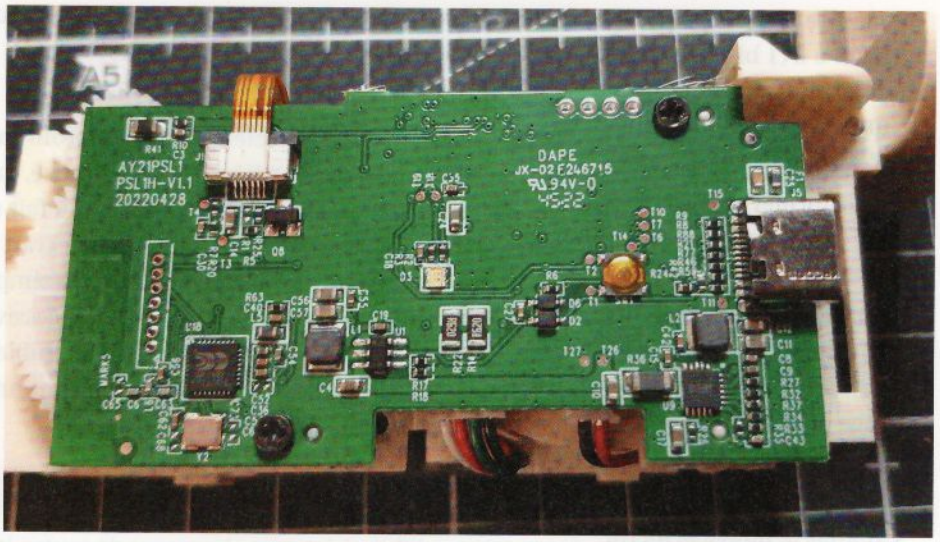
Il y a bien des manières de faire tout cela et on pourrait même envisager de n'utiliser que le shell, en reposant sur les outils en ligne de commande d'*ImageMagick*, `xxd` et `cat`. Cependant, le réflexe de la plupart des programmeurs sera de plutôt d'utiliser un langage comme Python, Node.js ou encore Lua ou Go, proposant des bibliothèques ou modules permettant le traitement d'images. Pour ma part, comme je garde toujours en tête la possibilité de porter 80 % de mes codes sur un microcontrôleur (ESP32, RP2040, STM32, etc.), mon choix se porte presque toujours sur le C. Oui, MicroPython et CircuitPython existent, mais j'ai une allergie aux langages qui utilisent l'indentation comme syntaxe (pourquoi pas aussi les lignes vides, la casse, la police ou la couleur, tant qu'on y est ?). Discutez pas, c'est une allergie, c'est médical et j'ai un mot de mon docteur du code...

Les plus fidèles lecteurs remarqueront sans doute comme un goût de déjà-vu, et pour cause, j'ai effectivement abordé une problématique tout à fait similaire dans le numéro 34 en me penchant sur un écran *e-paper* NFC de chez WaveShare **[6]**. Étant donné la simplicité de la procédure, je ne rentrerai donc pas excessivement dans le détail. D'autant que le code est disponible sur mon GitLab **[7]** et que ce qui est pour l'instant un simple POC tient en seulement 300 lignes de C.

Comme pour l'écran NFC, l'idée consiste à déléguer le traitement de l'image, quel que soit le format, à une bibliothèque se chargeant de toutes les parties pénibles. Cela ajoute une dépendance conséquente, *MagickWand* **[8]** en l'occurrence, mais c'est bien plus facile que de devoir gérer directement la LibJPEG ou la libPNG (ou Cairo qui permet aussi de faire des choses intéressantes).

L'utilisation de *MagickWand* se limite à une unique fonction chargée de lire le fichier graphique et de produire un tableau de `uint8_t` correspondant aux données de l'image, monochrome et dans les bons format et orientation. Ce tableau est ensuite manipulé dans `main()` pour lui adjoindre les messages de début et de fin et encoder le nombre de lignes sur les deux octets en position 14 et 15. Le tout est ensuite écrit sur `/dev/usb/lp0` (ou tout autre chemin passé en argument de l'outil), après ouverture du fichier en `O_RDWR`.

Le prototype de la fonction de lecture et de traitement de l'image source est : `int readimage(char *filename, uint8_t **imgbuf, uint16_t *nline, int rotate)`. Celle-ci prend en argument le nom du fichier à traiter, un pointeur vers l'adresse où se trouve le tableau à remplir, un pointeur vers un entier qui contiendra le nombre de lignes qui composent l'image finale et un paramètre précisant si l'on veut ou non tourner l'image de 90° (c'est une option de la ligne de commande). L'entier retourné par la fonction est soit une valeur négative en cas d'erreur, soit la taille du tableau généré si tout s'est correctement déroulé.



Le circuit imprimé, encore fixé sur le support, montre un contrôleur Bluetooth 5.0 dual mode BK3432 de Beken Corporation avec, sur la gauche, l'antenne et ce qui semble être, au minimum, un jeu de connecteurs pour une interface JTAG. Chose amusante, c'est le PCB qui tient en place les rouages du mécanisme pour avancer le papier...

Je ne vais pas tout lister ici, mais m'en tenir aux éléments que je juge intéressants ou importants, et en particulier la succession d'étapes pour transformer l'image. Les initialisations et libérations des ressources, que ce soit avec *MagickWand* ou `malloc()`/`free()`, sont aussi majoritairement laissées de côté ici (mais pas dans le code !). Tout ceci est dans le POC sur GitLab.

Nous commençons donc par tourner l'image avec :

```
    if (MagickReadImage(mw, filename) == MagickFalse) {
        fprintf(stderr, "Error loading image file!\n");
        DestroyMagickWand(mw);
        MagickWandTerminus();
        return -1;
    }
```

MagickWand propose ici une solution très simple et se charge de tout. Il suffit de préciser le `MagickWand` à cibler et le chemin vers le fichier. Si tout se passe bien, `wm` contient notre image et nous pouvons alors passer à la rotation :

```
if (rotate) {
    background = NewPixelWand();
    PixelSetColor(background,"#ffffff");

    if (MagickRotateImage(mw, background, 90) == MagickFalse) {
        fprintf(stderr, "Error resizing image!\n");
        DestroyPixelWand(background);
        DestroyMagickWand(mw);
        MagickWandTerminus();
        return -1;
    }
    DestroyPixelWand(background);
}
```

Nous avons besoin d'un `PixelWand`, qui est un élément représentant un ensemble de pixels, pour faire office de « fond » lors de la rotation. Avec 90°, 180° et 270°, ceci n'a théoriquement pas d'importance, mais la fonction `MagickRotateImage()` a besoin de cet élément au cas où la rotation laisse des parties « vides » (rotation de 45° par exemple), qui doivent alors être remplies par un fond (`background`).

Suite à cela, nous pouvons récupérer les dimensions de l'image et ajuster la taille en conséquence :

```
    width = MagickGetImageWidth(mw);
    height = MagickGetImageHeight(mw);
    newheight = (height*144)/width;

    if (width != 144) {
        if (MagickResizeImage(mw, WIDTH, newheight,
            LanczosFilter, 1) == MagickFalse) {
            fprintf(stderr, "Error resizing image!\n");
            DestroyMagickWand(mw);
            MagickWandTerminus();
            return -1;
        }
    }
```

Le redimensionnement d'une image est, d'un point de vue graphique et mathématique, une opération complexe et plusieurs algorithmes peuvent être utilisés pour obtenir des résultats plus ou moins qualitatifs. Le choix de l'algorithme se fait ici via le quatrième argument de `MagickResizeImage` et est décrit comme un « filtre » dans la documentation. Les choix possibles sont listés dans l'énumération présente dans `/usr/include/ImageMagick-6/magick/resample.h`. Bien sûr, à moins d'avoir des compétences mathématiques solides dans le domaine, la meilleure façon est de tester. Je trouve que `LanczosFilter` donne de bons résultats en général.

Vient ensuite l'étape de réduction des couleurs qui se passe en deux temps :

```
    if (MagickGetImageColors(mw) != 2) {
        if (MagickPosterizeImage(mw, 2,
                FloydSteinbergDitherMethod) == MagickFalse) {
            fprintf(stderr, "Error posterizing image!\n");
            DestroyMagickWand(mw);
            MagickWandTerminus();
            return -1;
        }

        if (MagickSetImageType(mw, BilevelType) == MagickFalse) {
            fprintf(stderr, "Error setting image to BW!\n");
            DestroyMagickWand(mw);
            MagickWandTerminus();
            return -1;
        }
    }
```

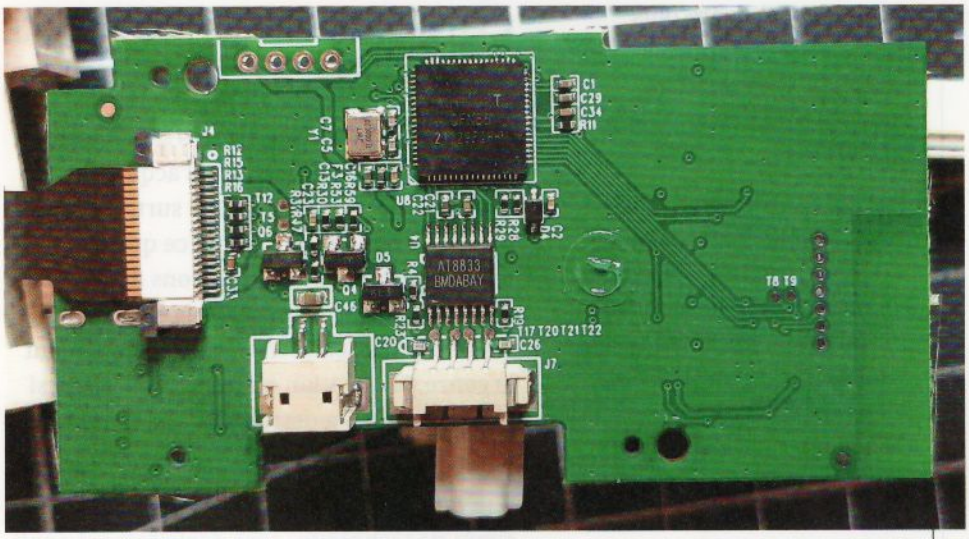
Nous commençons par la réduction de couleurs avec une *posterisation* en précisant la diffusion (ou *dithering*) à utiliser, qui peut être `RiemersmaDitherMethod`, `FloydSteinbergDitherMethod` ou tout simplement `NoDitherMethod`. Puis on change le type de l'image pour passer en `BilevelType`, une image purement monochrome. Notez que si l'image est déjà en deux couleurs, nous évitons soigneusement d'y toucher, exactement comme lors de la phase de redimensionnement, si l'image fait déjà 144 pixels de large (ou de haut avant rotation).

Il ne reste alors plus qu'à transformer cette masse de pixels en données pour l'imprimante, en commençant par créer le tableau où les stocker :

```
width = MagickGetImageWidth(mw);
height = MagickGetImageHeight(mw);

if (height > 0xffff) {
    fprintf(stderr, "Final Image is too big! %lu > 65535!\n", height);
    DestroyMagickWand(mw);
    MagickWandTerminus();
    return -1;
}

if ((buffer = (uint8_t *)malloc((width * height) / 8)) == NULL) {
    DestroyMagickWand(mw);
    MagickWandTerminus();
    fprintf(stderr, "readimage malloc() error!\n");
    return -1;
}

memset(buffer, 0x00, (width * height) / 8);
```

Sachant que nous n'avons que deux octets pour spécifier le nombre de lignes de l'image dans le message de départ, la valeur maximale est donc `0xffff` et c'est par conséquent notre limite, ainsi qu'une source d'erreur de la fonction. Le reste n'est que de l'allocation classique et une mise à zéro de tout le *buffer*, ce qui nous permet de ne nous occuper, ensuite, que des pixels noirs. Et c'est ce que nous faisons dans la foulée à l'aide d'un itérateur :

```
iterator = NewPixelIterator(mw);
for (y = 0; y < height; y++) {
    pixels = PixelGetNextIteratorRow(iterator,&width);
    for (x = 0; x < width; x++) {
        PixelGetHSL(pixels[x], &pixh, &pixs, &pixl);
        if (pixl == 0)  // 0 == no light == print black
            buffer[j] |= (128 >> ( i - (j * 8))); // reversed bits order
        i++;
        if (i % 8 == 0) j++;
    }
}
```



De l'autre côté du circuit, nous trouvons le SoC MH1902T de Megahunt et, dessus, le pont en H DRV8833 de Texas Instruments pilotant le moteur. La présence du MH1902T est surprenante, car c'est un « secure MCU » intégrant un accélérateur cryptographique (DES/TDES/AES 128-192-256, RSA-1024/2048+ECC et SHA-1 à 512), ainsi qu'une interface pour smartcard 7816-3 EMV level-1. Serait-ce un SoC normalement utilisé dans des terminaux de paiement ? Je m'attendais honnêtement à trouver un ESP32 dans ce produit. Le port en haut à gauche semble clairement être une interface série.

Nous concaténons les pixels par groupe de huit permettant de former un octet et reposons sur `PixelGetHSL()` pour obtenir la valeur de chaque pixel (la teinte et la saturation n'ont ici aucune importance). Une fois cette étape passée, il ne nous reste plus qu'à conclure en « remontant » le nombre de lignes de l'image, le pointeur vers les données et retourner en spécifiant la taille de ces dernières :

```
    *nline = height;
    *imgbuf = buffer;

    return j;
}
```

Dans la fonction `main()`, après avoir géré les options de la ligne de commande et appelé `readimage()`, nous n'avons plus qu'à allouer un nouveau tableau et regrouper les données, avant d'encoder le nombre de lignes dans les bons emplacements :

```
    memcpy(outputbuf, header, 16);
    memcpy(outputbuf + 16, imgbuf, imgbuffsize);
    memcpy(outputbuf + 16 + imgbuffsize, footer, 7);
    outputbuf[14] = (uint8_t)(nline & 0xff);
    outputbuf[15] = (uint8_t)((nline >> 8) & 0xff);
```

Le code final propose également de produire une version binaire des données de l'image traitée dans un fichier (pour vérification avec The Gimp) ainsi qu'un PNG correspondant. Créer le premier fichier est l'affaire d'un simple *dump* du tableau avec `open()`/`write()`/`close`, mais pour le

second, c'est encore plus facile, il suffit d'appeler `MagickWriteImage(mw, "nom_fichier.png")`, et le tour est joué.

C'est ce que j'aime dans *MagickWand*. Tout est excessivement simple à utiliser et nous sommes totalement déchargés de toutes les tâches fastidieuses. Même préciser le format du fichier en sortie devient inutile puisque la bibliothèque se base, toute seule, sur l'extension utilisée dans le nom de la sortie. De plus, si l'on est coutumier des outils ImageMagick, on retrouvera sans problème la même « logique » sous la forme de fonctions avec, comme ici, les différentes opérations, parfaitement réalisables à l'identique avec l'outil `convert`.

Ce type d'expérimentation fait forcément des victimes innocentes, sous la forme d'une tripotée d'étiquettes inutiles. Le rectangle noir en bas à gauche est la toute première impression réussie en USB.

CONCLUSION ET FUTURES VICTIMES

Ce genre d'exercice est très satisfaisant, non seulement en raison de l'acquisition de connaissances, mais aussi, et surtout, pour le *shot* de dopamine lorsque ce qui est initialement un lot de suppositions devient une réalité très concrète prenant la forme d'une superbe étiquette. Voilà typiquement le genre de situation où le chemin est aussi important que la destination, sinon plus.

Il s'agissait ici d'une imprimante, avec quelques surprises intéressantes, mais ce genre de techniques sera parfaitement applicable à d'autres matériels. Trouvez son chemin dans des protocoles et des formats peu documentés est, à 80 %, une affaire d'instinct et de suppositions parfois éclairées. Ici, malgré quelques encombres, en particulier pour les deux fameux octets qui ont demandé le sacrifice de bien des centimètres d'étiquette, tout s'est relativement bien passé. Dans d'autres situations, cela peut rapidement tourner au jeu de piste sans fin et à un début de dépression (mais rien qui ne puisse être réglé par une bonne Guinness à température ambiante). Mais là, c'est à chacun de savoir quand jeter l'éponge, car les ressources, comme le nombre d'heures d'une journée, ne sont malheureusement pas infinies, même si l'échec est douloureux.

De plus, même si nous avons atteint un résultat satisfaisant ici, des zones de flou persistent. Il doit exister d'autres « commandes » peut-être utilisables via USB et nous ne savons pas ce que veut vraiment dire `1b 4a 40 10 ff fe 45` (ou `10 ff fe 45` qui ressemble à `10 ff fe 01` et `10 ff fe 40`, mais se trouve en plein milieu d'un message). Et quid du BLE ? L'imprimante est visible via ce protocole, existe-t-il un moyen de l'utiliser ainsi ?

Une autre option est de s'en tenir à ce qu'on a et se tourner vers des mises en œuvre pratiques. Nous pourrions, par exemple, faire de même avec un ESP32-S2 (ou une Pico W ?) pour pouvoir imprimer en Wi-Fi et/ou depuis, et vers, un site distant (*over* VPN éventuellement). Je ne sais pas trop à quoi cela pourrait servir, mais l'idée d'ajouter un mode d'impression et une connectivité non prévue par le constructeur est une idée plaisante.

Nous avons aussi la voie de l'amélioration du code qui, pour l'instant, n'est qu'une sorte de démonstration technologique. Sans pour autant partir dans la création d'une application graphique (Qt, GTK+, etc.), on notera que l'outil *ImageMagick* `convert` sait faire ceci :

```
$ convert -list font
[...]
glyphs: /home/denis/.fonts/euro.ttf
glyphs: /home/denis/.fonts/Florilane Cardillac.ttf
glyphs: /home/denis/.fonts/Heavitas.ttf
[...]
```

Autrement dit, *ImageMagick*, et donc l'API *MagickWand*, sait gérer des polices et donc produire des images à partir de texte. Là, ceci nous ouvre des perspectives **très** intéressantes, car le nombre de polices utilisables sur l'application mobile est assez limité. Bien entendu, à un niveau d'abstraction supérieur, nous pourrions créer un filtre CUPS et ainsi prendre en charge l'imprimante au niveau du système pour la rendre accessible à toutes les applications. Personnellement, je passerai mon tour sur ce point précis, je ne suis pas en très bons termes avec CUPS, ces derniers temps.

Et enfin, la L1-A est **une** des imprimantes de ce type. Il en existe d'autres qui, à mon avis, doivent être technologiquement assez proches. Ceci peut être un passe-temps intéressant et instructif... **DB**

RÉFÉRENCES

[1] **https://www.amazon.fr/dp/B09Y929QKV**

[2] **https://play.google.com/store/apps/details?id=com.wewin.house_print_international**

[3] **https://apps.apple.com/cm/app/makeid-life/id1502057906**

[4] **https://greatscottgadgets.com/ubertoothone/**

[5] **https://makerdiary.com/products/nrf52840-mdk-usb-dongle-w-case**

[6] **https://connect.ed-diamond.com/Hackable/hk-034/ecran-e-paper-nfc-une-histoire-d-exploration-et-de-code**

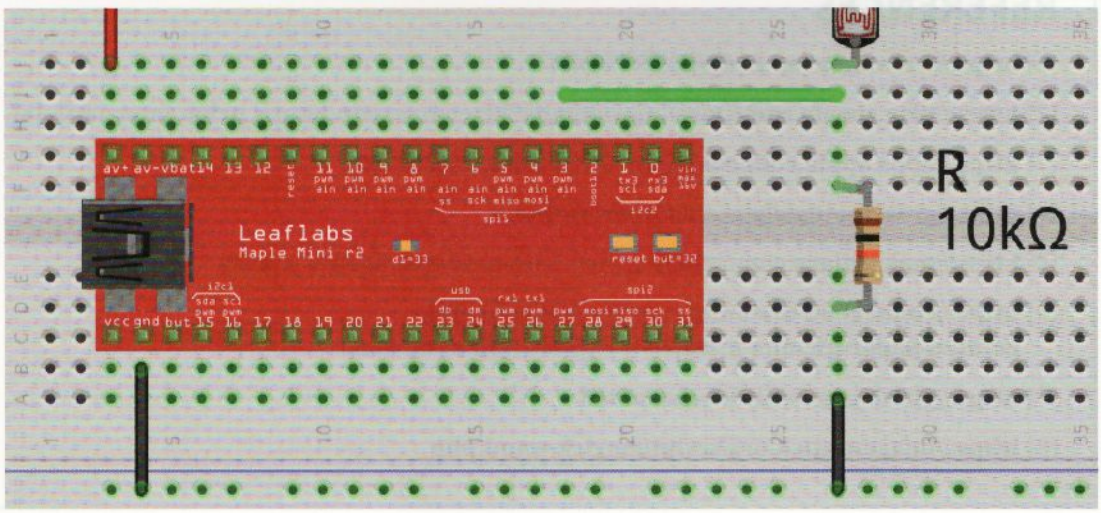
[7] **https://gitlab.com/0xDRRB/makeidcli**

[8] **https://imagemagick.org/script/magick-wand.php**

POURQUOI MON MONITEUR DESKTOP NE FERAIT-IL PAS AUSSI BIEN QUE L'ÉCRAN DE MON SMARTPHONE ?

Pierre BAUDEMONT
Artisan du code

Je suis plutôt sensible à la lumière, à l'éclairement lumineux subi par mes rétines pour être plus précis. Les lux que m'inflige un écran trop zélé ont tôt fait de me faire frénétiquement chercher le bouton ou le menu permettant d'en réduire la luminosité, pour peu que l'écran en question soit le mien. Le problème est qu'alors la variation d'éclairage du lieu de travail rend nécessaires de multiples ajustements au cours de la journée, surtout dans les saisons où elle se fait plus courte.



– Pourquoi mon moniteur desktop ne ferait-il pas aussi bien que l'écran de mon smartphone ? –

Mais dites donc, ajuster sans cesse la luminosité de son écran de la sorte, c'est tout de même bien pénible. Il doit y avoir moyen de faire autrement. C'est alors qu'en regardant son smartphone, on réalise avec perplexité que ce genre d'appareil adapte sa luminosité automatiquement depuis longtemps. Pourquoi diable un moniteur de bureau n'en ferait-il pas autant ? Parce qu'un moniteur de bureau n'a pas vocation à être déplacé, on pense peut-être par raccourci que ses conditions d'utilisation ne changent pas ou peu ? Pourtant ce n'est pas le cas : variation de la lumière extérieure, éclairage « dimmable », ouverture ou fermeture de volets, stores ou rideaux, lampe de bureau, etc., les conditions dans lesquelles on regarde un moniteur peuvent varier considérablement, rien que sur une journée ! J'avoue que l'absence d'une fonctionnalité de ce genre sur les moniteurs m'étonne. Je suppose que je suis plus sensible que la moyenne à la luminosité, d'ailleurs je crois bien n'avoir jamais utilisé un écran réglé « à fond ».

1. INTRODUCTION

Une idée qui vient naturellement à tout bon fainéant^Winformaticien est de chercher à automatiser les tâches fastidieuses et/ou répétitives. Je me suis donc rapidement dit qu'il devait y avoir une solution. C'est en constatant que nos chers ordinateurs sont capables de récupérer des informations sur les moniteurs, comme leur résolution native, que l'espoir est venu. En effet, si de telles informations peuvent être récupérées automatiquement, il doit exister un canal de communication entre les deux.

Quelques recherches hasardeuses plus tard et me voici rendu sur l'article Wikipédia dédié au DDC **[1]**. Le Graal !

2. LE DDC/CI

Pour paraphraser l'article en question, le DDC, pour *Display Data Channel*, est un lien de communication entre l'écran et le périphérique « émetteur » (ici, l'ordinateur) qui l'utilise. À l'origine, il permet au dit périphérique d'accéder à des informations en lecture seule à propos du modèle de l'écran et surtout, des modes d'affichage supportés.

Plus intéressante encore, l'extension DDC/CI, pour *DDC Command Interface*, offre (depuis 1998 !) un canal dans l'autre sens qui permet au périphérique de donner des ordres à l'écran. Ces possibilités ont été exploitées ou étendues par les constructeurs, et il existe par exemple des moniteurs dépourvus de boutons **[2]** et qui se contrôlent entièrement par logiciel, depuis le PC. Quoi qu'il en soit, un jeu de commandes standard existe **[3]** et permet ainsi de faire des choses comme ajuster la luminosité ou le contraste des moniteurs compatibles. Bien que ce ne soit jamais mis en avant (personnellement, je n'ai jamais vu un moniteur avec un *sticker* DDC/CI par exemple), il se trouve que la quasi-totalité des écrans supporte en fait DDC/CI (*a priori* ceux produits à partir de 2016 notamment, selon la version anglaise de l'article Wikipédia).

Cerise sur le gâteau, l'extension a beau être ancienne, elle reste disponible avec les connectiques modernes. On y a donc accès avec un branchement VGA, DVI et HDMI !

3. APPROCHE CONCRÈTE

Une première façon d'aborder la chose est de recourir à un logiciel déjà existant, ce qui permet de plus de s'assurer de la compatibilité du moniteur avec DDC/CI.

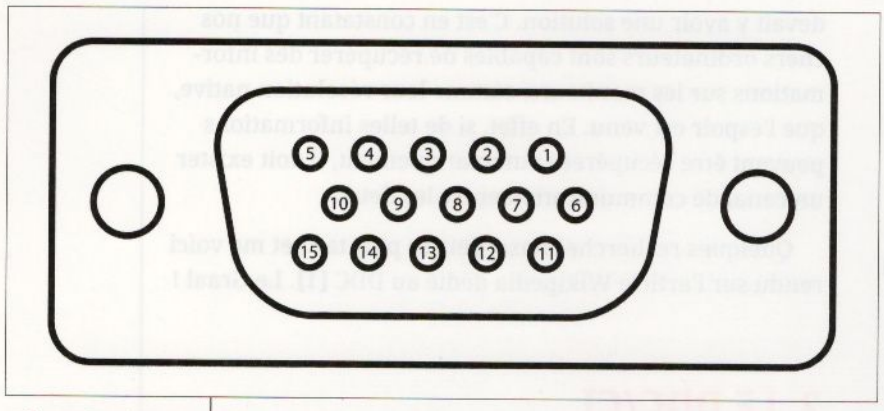


Fig. 1 : le pinout du connecteur VGA femelle. ATTENTION : le brochage du connecteur mâle est symétrique (broche 1 en haut à gauche).

Ainsi, l'outil Monitorian **[4]** pour Windows ou encore `ddccontrol` **[5]** pour GNU/Linux permettent tous deux d'ajuster la luminosité du moniteur de manière logicielle.

Une fois rassuré quant au bon fonctionnement de la chose, on a envie d'aller plus loin. En tout cas, j'ai eu envie de me faciliter la vie jusqu'au bout. Certes, c'est déjà nettement plus commode d'ajuster la luminosité directement depuis son bureau ou la ligne de commande, mais ça reste répétitif.

Pourquoi s'arrêter en si bon chemin ?

En effet, mon moniteur, comme de nombreux modèles, offre toujours un port VGA (D-sub DE-15). En plus d'être plus rarement utilisé de nos jours, ses broches demeurent bien plus accessibles que celles du DVI ou pire, de l'HDMI. Je n'avais par ailleurs aucune envie d'opérer un câble HDMI pour en sortir les fils dont j'aurais eu besoin. Bref, j'ai donc décidé de tenter d'utiliser le DDC/CI depuis le port VGA, qui a le bon goût de pouvoir être partiellement raccordé avec des câbles Dupont.

Qu'en est-il de la connectique ?

Là encore, l'article anglais de Wikipédia **[6]** est d'un grand secours. On peut résumer ce qui nous intéresse avec le petit tableau suivant, en notant au passage que le lien est en fait un bus i²c (mais limité) :

Broche 9	+5V (DDC)
Broche 10	Masse (pour le DDC)
Broche 12	Données (SDA)
Broche 15	Horloge (SCL)

L'emplacement des broches en question est donné par le *pinout* du connecteur **femelle**, voir la figure 1.

Faut-il envisager d'implémenter DDC/CI nous-mêmes ? Là encore, tout bon informaticien va commencer par chercher à ne pas réinventer la roue. Il existe notamment une librairie Arduino, *ddcvcp* **[7]**, composée de 2 petits fichiers sources et qui permet notamment de gérer la luminosité. C'est parfait !

4. LE MATÉRIEL

Les ingrédients sont désormais réunis : port VGA disponible sur un moniteur avec DDC/CI, câbles Dupont, librairie Arduino. Plus qu'à placer une petite carte compatible Arduino en face et le tour sera joué. Enfin, pas tout à fait. N'oublions pas qu'on souhaite réagir à la luminosité ambiante. Ça tombe bien, le montage tout simple réalisé dans l'article « Mesurer l'éclairement avec Arduino » dans Hackable n°37 fait tout à fait ça. Pour rappel, il suffit d'une simple LDR (photorésistance), d'une résistance de 10 kΩ et d'un petit bout de code pour obtenir une mesure, certes imprécise mais cohérente de l'éclairement. Ce montage et

le code associé peuvent sans nul doute être améliorés, mais ce sera bien suffisant pour une preuve de concept.

La fonction à réaliser étant ici extrêmement simple : mesurer en boucle la luminosité ambiante et ajuster en conséquence celle du moniteur, il m'a semblé « overkill » de recourir à une carte à base de STM32. J'avais envie de quelque chose de plus simple, mais aussi d'encore plus réduit, qui puisse à terme, pourquoi pas, être intégré dans un petit boîtier branché directement sur le port VGA, ou relié par câble.

Bienvenue petite carte stm8blue (voir figure 2), plus précisément la *STM8S103 breakout board*. Il s'agit d'une carte minuscule tant par la taille que par le prix, et qui embarque donc un microcontrôleur 8 bits, supporte l'i²c et possède un ADC 10 bits, entre autres fonctionnalités. Bien suffisant pour ce qu'on souhaite faire ici ! À noter cependant que si vous n'en possédez pas, une interface de programmation type ST-Link V2 est nécessaire.

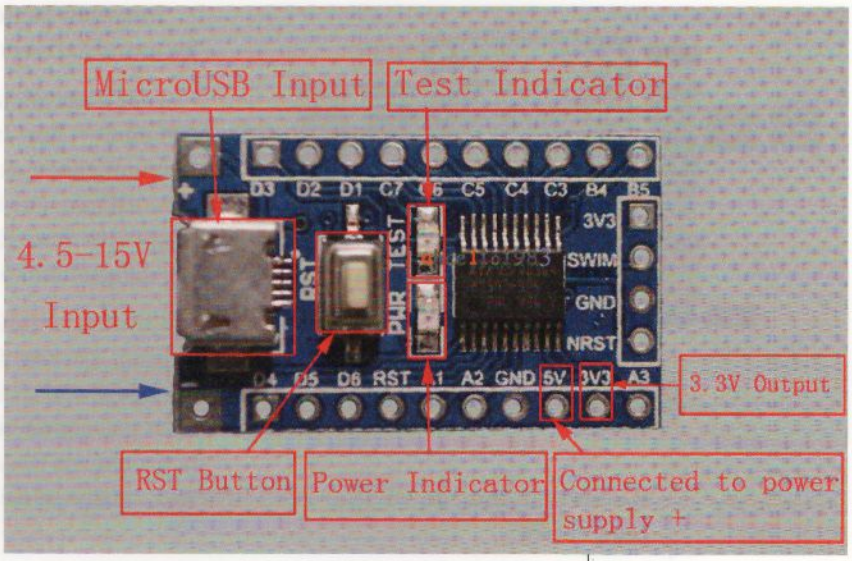


Fig. 2 : la carte stm8blue, le port micro-USB permet d'avoir une idée de l'échelle.

5. STM8BLUE ET ARDUINO

De par ses prestations modestes, cette petite carte n'est pas supportée par défaut par Arduino, mais comme pour la famille STM32, un projet existe pour apporter ce support. Il s'agit en l'occurrence de Sduino **[8]**. Le site du projet donne la marche à suivre pour l'intégrer à Arduino. À l'instar de l'ajout du support des STM32 ou des ESP*, cela consiste à :

- ajouter l'URL **https://github.com/tenbaht/sduino/raw/master/package_sduino_stm8_index.json** à la liste du gestionnaire de cartes supplémentaires dans les préférences de l'IDE ;
- ouvrir le gestionnaire de cartes ;
- chercher et sélectionner Sduino ;
- installer l'extension sélectionnée ;
- depuis le menu outils, spécifier le modèle de carte « STM8S103F3 Breakout Board » parmi la liste des nouvelles cartes « STM8S Boards ».

Et voilà, la quasi-totalité des fonctionnalités d'Arduino s'offre à présent à notre petite stm8blue, avec cependant une limitation qui peut s'avérer rapidement gênante : **Sduino ne supporte que le C, pas le C++**. C'est une limitation documentée par le projet, qui comporte même un guide de migration pour les API C++. À noter que plusieurs librairies sont fournies avec le projet.

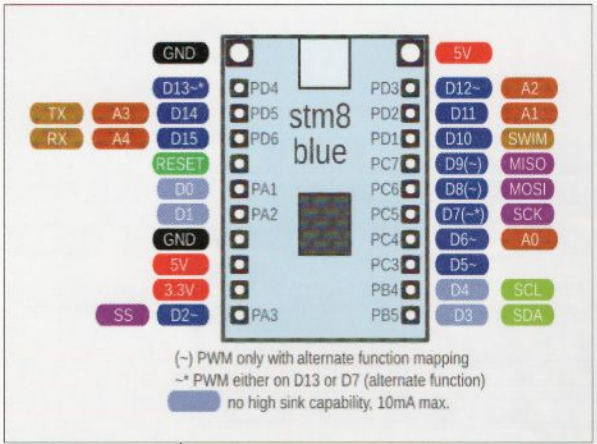


Fig. 3 : le pinout de la stm8blue.

Tout était facile jusqu'à présent, il fallait bien que les ennuis arrivent.

Tout d'abord, la bibliothèque *ddcvcp* présente une API en C++. Fort heureusement, son code est extrêmement concis et la porter en C pur n'aura pas été très difficile. J'ai ainsi tout bonnement récupéré et adapté les deux fichiers sources avant de les incorporer directement à mon croquis.

Ensuite, comme mentionné sur le site du projet Sduino, il arrive fréquemment que le connecteur de débogage soit mal réalisé sur le PCB. Soit la masse n'est pas connectée, soit elle est en court-circuit avec la broche SWIM. Avant de relier la carte au ST-Link, **il convient donc de vérifier ces 2 broches : absence de court-circuit entre elles et masse bien reliée**.

6. MONTAGE ET CODE

Le *pinout* de la carte, en figure 3, montre qu'on n'a pas trop de questions à se poser quant au brochage.

On va ainsi relier :

- la broche GND à la broche 10 du connecteur VGA femelle ;
- la broche PB5 (SDA) à la broche 12 du connecteur VGA femelle ;
- la broche PB4 (SCL) à la broche 15 du connecteur VGA femelle ;
- la broche PC4 à la sortie du diviseur de tension formé par la LDR et la résistance de 10 kΩ.

Inutile de relier le 5 V de la carte à la broche 9 du connecteur VGA femelle.

Pour rappel, la LDR est exploitée en formant un pont diviseur dont la tension de sortie varie selon l'éclairement qu'elle reçoit. La LDR est reliée à la broche 3,3 V de la carte, et la résistance de 10 kΩ à sa masse.

Le code est finalement essentiellement un assemblage des briques que nous avons déjà :

- la version modifiée de la librairie *ddcvcp*, pour la convertir en « C pur » ;
- le code de gestion de la LDR issu de l'article précédent. Il s'agit simplement de la déclaration d'une poignée de constantes et d'un petit calcul. Attention, comme l'ADC est limité à 10 bits au lieu des 12 qu'offrait la Maple Mini, il faut modifier la constante `ADC_LSB` :

```
// coeff de conversion de la valeur de analogRead() en V, ADC 10-bit
const double ADC_LSB = 3.3 / 1023;
```

– Pourquoi mon moniteur desktop ne ferait-il pas aussi bien que l'écran de mon smartphone ? –

Ainsi le fichier principal est concis, le code se contente :

- de mesurer l'éclairement à intervalles réguliers (ici, toutes les 100 ms) ;
- d'en déduire une valeur de luminosité pour le moniteur, entre 0 et 100 ;
- de configurer la luminosité du moniteur avec cette valeur via DDC/CI.

```
#include <math.h> // attention, log() est ln() !
#include "ddcvcp_c.h"

#define INPUT_PIN PC4

// constantes pour le calcul de l'éclairement
const float VCC = 3.3; // 3,3V
const float R = 1e4; // 10kOhms
const float log10_R0 = 6; // log_10(R0)
const float GAMMA = 0.8;
float K;
// coeff de conversion de la valeur de analogRead() en V, ADC 10-bit
const float ADC_LSB = 3.3 / 1023;

void setup() {
    K = logf(10) / GAMMA;

    pinMode(INPUT_PIN, INPUT);

    while (!DDCVCP_begin()) {
        delay(5000);
    }
}

void loop() {
    int U_ADC_raw = analogRead(INPUT_PIN);
    float U_ADC = U_ADC_raw * ADC_LSB;

    float R_LDR = R * (VCC/U_ADC - 1);
    float E = expf(K * (log10_R0 - log10f(R_LDR)));

    int brightness = (int)(E * 1e-3);
    if (brightness > 100) brightness = 100;

    DDCVCP_setBrightness(brightness);
    delay(100);
}
```

La déduction d'une valeur de luminosité est ici très simple, l'essentiel étant déjà de valider le montage. On note aussi le petit *sanity check* sur la valeur de la luminosité afin qu'elle reste dans l'intervalle [0, 100].

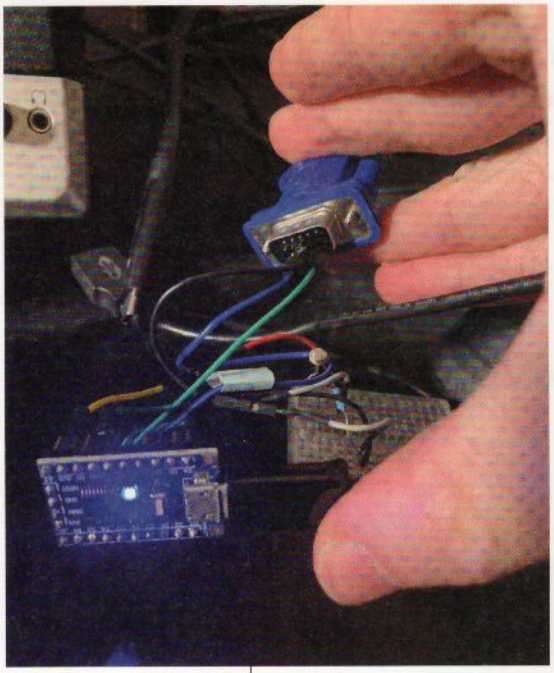


Fig. 4 : Le prototype en cours de fonctionnement. On distingue la LDR au centre de l'image, et l'on voit les câbles Dupont qui relient la stm8blue aux broches utilisées sur le port VGA (ici, un câble mâle qui sort de mon KVM, et ça fonctionne) !

7. PETITS ESSAIS

Une fois le tout relié sur platine à essai, on peut relier les broches dédiées au DDC/CI aux bons emplacements sur le connecteur VGA d'un câble inutilisé. Attention au brochage sur le connecteur mâle !

Une fois le câble relié au moniteur, on peut alimenter le montage. Oui, ça fonctionne ! Si l'on couvre la LDR, le moniteur s'assombrit rapidement. Si on l'éclaire, le moniteur devient plus lumineux. Objectif atteint !

8. LA SUITE

La première chose à améliorer est la déduction d'une valeur de luminosité en fonction de l'éclairement ambiant. Au lieu d'une relation directe comme ici, on pourrait imaginer une table ou encore un lien non linéaire entre les deux.

Surtout, si j'en avais le temps, j'aimerais réaliser un petit boîtier en bois, à défaut de disposer d'une imprimante 3D, afin d'avoir un produit à l'aspect fini, muni d'un connecteur VGA prêt à être branché. Il faudrait pour cela disposer d'une source d'alimentation, le plus simple étant sans doute d'utiliser des piles, le port VGA côté moniteur n'offrant malheureusement pas de courant. **PB**

RÉFÉRENCES

[1] Article Wikipédia à propos de DDC : **https://fr.wikipedia.org/wiki/Display_Data_Channel**

[2] Un exemple est décrit ici : **https://www.techradar.com/reviews/lg-ultrafine-24md4kl-b**

[3] C'est le MCCS, Monitor_Control_Command_Set : **https://en.wikipedia.org/wiki/Monitor_Control_Command_Set**

[4] **https://github.com/emoacht/Monitorian**

[5] **https://github.com/ddccontrol/ddccontrol**

[6] L'article en version anglaise, qui comporte le *pinout* du DDCI/CI au sein du connecteur VGA : **https://en.wikipedia.org/wiki/Display_Data_Channel**

[7] La librairie ddcvcp : **https://github.com/tttttx2/ddcvcp**

[8] Le projet Sduino : **https://tenbaht.github.io/sduino**

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
HACKABLE MAGAZINE


FORUM
IN CYBER
EUROPE
26-28 MARS 2024
LILLE GRAND PALAIS
Parés pour l'IA ?
organisé par
ceis
Forward
avec le soutien de
Région Hauts-de-France
europe.forum-incyber.com