ÉLECTRONIQUE | EMBARQUÉ | RADIO | IOT

HACKABLE

L'EMBARQUÉ À SA SOURCE

N° 42
MAI / JUIN 2022

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

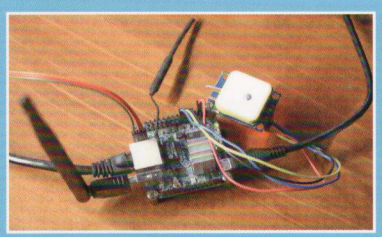
L 19338 - 42 - F: 14,90 € - RD

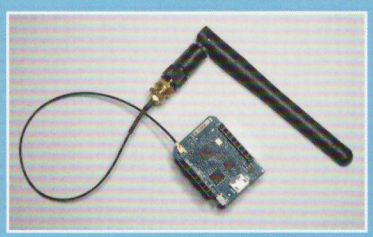
CPPAP : K92470

Wi-Fi / Beacon / Expérimentations / Pi

CRÉEZ DES POINTS D'ACCÈS FANTÔMES AVEC VOS ESP8266 p.74

- Installer les outils d'analyse
- Comprendre le protocole Wi-Fi
- Créer et émettre des trames





#SIDO2022

48H POUR CONCRÉTISER VOTRE TRANSFORMATION DIGITALE !

LYON 8e

SIDO

IoT - AI - ROBOTICS - XR

14 & 15 Septembre 2022

Cité Internationale de Lyon

300 exposants

60 conférences

180 speakers

CRÉER MON BADGE GRATUIT

CODE : P-HASL22

www.sido-lyon.com

RETROUVEZ ÉGALEMENT

SIDO Paris 2e

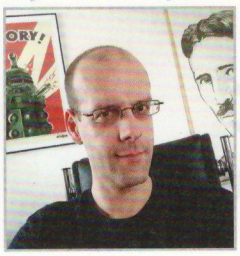
08 > 09 novembre 2022

Palais des Congrès | www.sido-paris.com

UN ÉVÉNEMENT

INFOPRO digital

ÉDITO



Nous avons, en début d'année, fait le choix d'augmenter la fréquence de publication du magazine afin de publier un numéro tous les deux mois, et non plus trois comme précédemment. Ce changement a été fait tout en conservant les autres caractéristiques du magazine et en particulier, la quantité de pages et la masse de contenu présent dans chaque numéro.

Je considère la périodicité bimestrielle comme étant celle naturelle de Hackable, dans le sens où elle permet d'obtenir un flux régulier de contenus techniques liés aux domaines qui sont ceux de la publication. Nous avons maintenant atteint une « vitesse de croisière » et un « tonnage » stable, durable et satisfaisant. Pour autant, cela fait donc également 6 numéros à devoir ponctuellement obtenir avec une plage de disponibilité réduite d'un mois.

Ce qui me conduit tout naturellement à vous parler d'abonnement. Car, au-delà de l'économie que cela constitue et du fait de ne rater aucun numéro, ceci est également une façon de soutenir la publication et un éditeur indépendant ayant fait le choix de vivre grâce à ses lecteurs. Si vous avez reçu le présent magazine dans votre boîte à lettres, parce que vous êtes abonné, je vous en remercie grandement, car c'est sans le moindre doute la plus appréciable marque de confiance que puisse recevoir un rédacteur en chef, un auteur et un éditeur.

Et dans le cas contraire, je me contenterai de faire comme les Youtubeurs en vous recommandant de « cliquer sur le pouce bleu », « d'activer les notifications pour ne rater aucun épisode » et d'envisager de vous « abonner à la chaîne » si le contenu vous plaît. Plus sérieusement, merci à vous aussi, abonné ou non, c'est grâce à vous que nous existons et pouvons continuer à diffuser des articles d'une telle technicité.

Bonne lecture à vous et rendez-vous dès **juillet** pour le prochain numéro !

Denis Bodor

P.-S. - Notez que cet édito ne contient absolument aucune référence à la grande question sur la vie, l'Univers et le reste.

Hackable Magazine
est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND
STIMULANT DE MATIÈRE GRISE

12, place du Capitaine Dreyfus - 68000 Colmar - France
E-mail : lecteurs@hackable.fr -
Service commercial : diamond@abomarque.fr
Sites : www.hackable.fr - www.ed-diamond.com
Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor
Réalisation graphique : Kathrin Scali
Responsable publicité : Tél. : 03 67 10 00 27
Service abonnement : Hackable Magazine / Abomarque
53 Route de Lavaur, 31242 L'Union, Tél. : 03 67 10 00 20
Impression : pva, Druck und Medien-Dienstleistungen GmbH, Landau, 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...

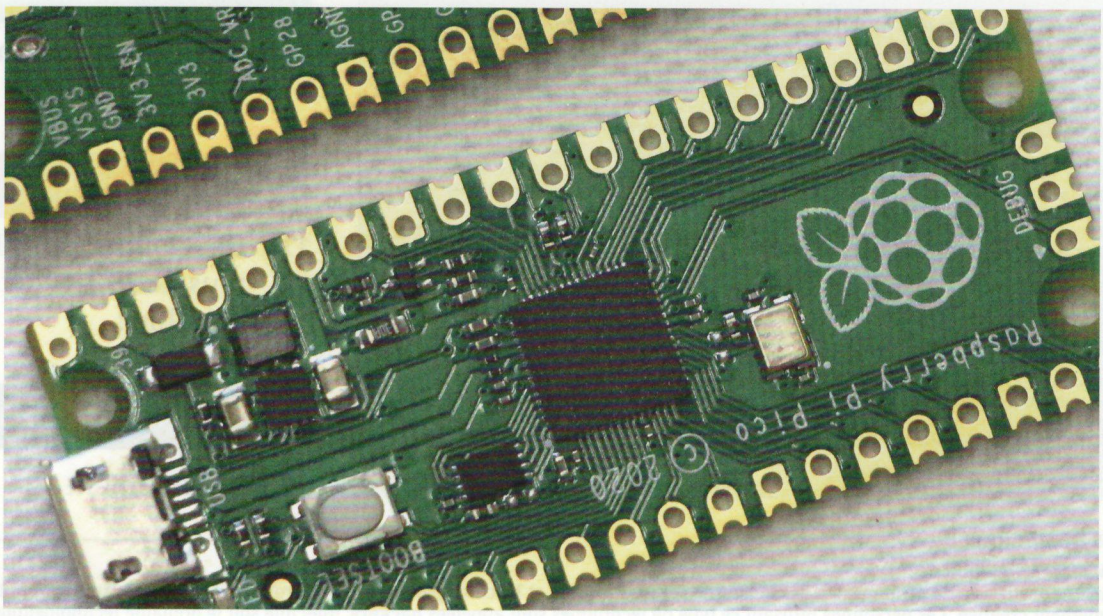
HACKS, HACKERS & HACKABLE

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.

RASPBERRY PI PICO : ÉMULER UNE EEPROM INTERNE POUR LE STOCKAGE DE DONNÉES

Denis Bodor

Les cartes Arduino AVR, ainsi que d'autres plateformes utilisant cet environnement et cet « IDE », proposent de base une EEPROM permettant de stocker une quantité réduite de données survivant entre les phases d'alimentation d'un projet. Dans le cas des AVR, cette EEPROM est physiquement présente dans le microcontrôleur, mais avec un ESP32 par exemple, c'est une zone de la flash qui sera utilisée. Qu'en est-il d'une fonctionnalité similaire avec la Raspberry Pi Pico et son RP2040 ?



Bien que le RP2040 offre un certain nombre de modes de sommeil (*sleep*, *dormant*, *Memory Power Down*), ainsi qu'un réglage fin des horloges, il arrive que l'approche optimale consiste à totalement cesser d'alimenter un projet. Dans ce genre de situations, seule une mémoire de stockage non volatile permettra la rétention des données. Les AVR d'Atmel/Microchip intègrent une zone de flash spécifique, inscriptible depuis le code en C, dédiée à cet usage. D'autres plateformes comme les microcontrôleurs d'Expressif implémentent ce type de fonctionnalités de façon logicielle via l'utilisation d'une partie de la mémoire de stockage NVS (pour *Non-Volatile Storage*, une zone du système de partitionnement de la flash SPI **[1]**) et d'une bibliothèque dédiée (« *Preferences* » dans le support Arduino **[2]**).

Ce genre de mécanismes suppose de pouvoir accéder à la mémoire flash, normalement dédiée au stockage du code binaire et des données en lecture seule, depuis le code en cours d'exécution. Comme nous l'avons vu dans deux précédents articles (sur l'accès aux informations binaires embarquées dans le *firmware* **[3]** et sur l'utilisation conjuguée du PIO, du DMA et de la flash **[4]**), ceci est parfaitement possible très facilement, lorsqu'il s'agit d'une lecture. En effet, les 2 Mo de flash sont mappés, au même titre que les 256 Ko de RAM, dans l'espace d'adressage accessible à votre code, respectivement aux adresses `0x10000000` et `0x20000000`. Bien entendu, ceci fonctionne également avec d'autres cartes utilisant un RP2040, mais disposant de plus de flash (comme la Pimoroni Pico LiPo ou la SparkFun Thing Plus RP2040).

Accéder à cette mémoire est une simple affaire de pointeurs, parfois même sans que vous vous en rendiez compte puisque les `const` seront automatiquement placés, par l'éditeur de liens, en flash et non en RAM. Cependant, accéder en écriture à la flash depuis son code est une autre paire de manches...

1. ÉCRITURE EN FLASH

Lorsque vous programmez votre RP2040, avec `picotool` ou un *firmware* au format UF2, le code binaire sera placé en tout début de flash, mais apparaîtra dans l'espace d'adressage ARM à partir de l'adresse `0x10000000`. Si votre binaire fait 15000 octets, celui-ci sera placé et accessible de l'adresse `0x10000000` à `0x10003A98`. Mais, et ce détail est très important, il sera stocké dans la flash QSPI, de l'adresse `0x00000000` à `0x00003A98`. Ce décalage n'a pas d'importance en lecture, mais il n'en va pas de même pour les fonctions d'effacement et d'écriture de la flash, qui utilisent les adresses physiques et non celles de l'espace d'adressage ARM. Le SDK met d'ailleurs une macro à notre disposition, `XIP_BASE` (`flash.h`), correspondant à ce décalage.

Une mémoire flash NOR comme celle utilisée par le RP2040 ne se manipule pas comme de la RAM. Bien qu'elle soit excessivement rapide et permette l'exécution de code sans transfert en RAM (XIP pour *eXecute In Place*) l'écriture suit une mécanique très spécifique. Il faut l'effacer avant d'y écrire, et ce, par « morceaux ». Une mémoire flash est typiquement divisée et subdivisée :

- la mémoire dans son ensemble ;
- chaque puce qui la compose (*die*) ;
- les *blocks* qui sont l'unité de base pour l'effacement (comprendre la plus grosse unité utilisable pour cette opération) ;

- des demi *blocks* ou *sub-block*, qui sont de plus petites unités d'effacement ;
- les secteurs qui sont la plus petite unité d'effacement (souvent de 4 Ko) ;
- les pages qui sont une unité d'écriture (généralement de 256 octets).

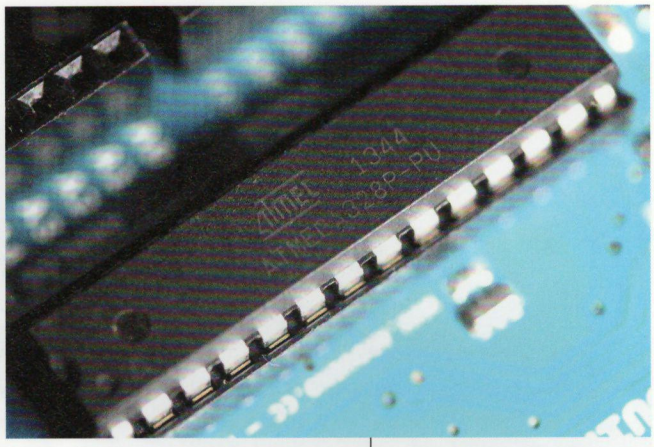
Ceci est une description générique s'appliquant sur une large gamme de composants, mais certains, plus spécialisés, offrent une granularité plus importante ou des fonctionnalités particulières, comme l'effacement des pages par exemple (*page erasable*). Dans ces cas, la nomenclature « *blocks*/secteurs/pages » peut varier selon le type de flash et le fabricant.

Quoi qu'il en soit, on remarque une chose importante : une mémoire flash s'efface grâce à plus petite unité d'effacement et s'écrit via sa plus petite unité d'écriture, qui ne sont pas égales. Dans le cas de la flash QSPI utilisée par la Pico, l'unité d'effacement est `FLASH_SECTOR_SIZE` et celle d'écriture est `FLASH_PAGE_SIZE`, respectivement `(1u << 12)` et `(1u << 8)`, et donc 4096 et 256 octets.

Pourquoi souligner ce point ? Tout simplement parce que les deux fonctions `flash_range_erase(offset, size)` et `flash_range_program(offset, buffer, size)` les utilisent, comme l'indiquent les commentaires dans les prototypes de fonctions (`flash.h`) :

```
/*! \brief  Erase areas of flash
 *  \ingroup hardware_flash
 *
 * \param flash_offs Offset into flash, in bytes, to start
 * the erase. Must be aligned to a 4096-byte flash sector.
 * \param count Number of bytes to be erased. Must be a
 * multiple of 4096 bytes (one sector).
 */
void flash_range_erase(uint32_t flash_offs, size_t count);

/*! \brief  Program flash
 *  \ingroup hardware_flash
 *
 * \param flash_offs Flash address of the first byte to
 * be programmed. Must be aligned to a 256-byte flash page.
 * \param data Pointer to the data to program into flash
 * \param count Number of bytes to program. Must be a
 * multiple of 256 bytes (one page).
 */
void flash_range_program(uint32_t flash_offs,
                         const uint8_t *data, size_t count);
```

Plusieurs éléments très importants sont décrits dans ces commentaires. Que ce soit pour l'effacement ou l'écriture, l'opération doit porter sur un emplacement aligné sur une certaine valeur et avec une quantité de données étant un multiple de la taille d'une unité. On ne peut donc effacer que des secteurs de 4096 octets à un emplacement multiple de 4096, et on ne peut écrire que par paquets de 256 octets à un emplacement multiple de 256. Ceci signifie également que pour écrire une valeur de 256 octets, il faut en effacer 4096 et donc, le cas échéant, réécrire 16 pages de 256 octets.



L'Atmel/Microchip ATmega328P, équipant les maintenant très anciennes Arduino Uno, est un microcontrôleur disposant de 32 Ko de flash pour les programmes, 2 Ko de RAM, mais également de 1 Ko d'EEPROM physique permettant d'enregistrer des données indépendamment du code.

L'argument `flash_offs` est un emplacement **en flash** relatif au début de l'espace d'adressage physique du composant, **qui n'est pas** `XIP_BASE` ! Si nous reprenons notre exemple du début d'article, ces fonctions s'utilisent avec `0x00003A98` et non `0x10003A98`, et plus exactement `0x000040000` pour respecter l'alignement.

D'autres contraintes s'ajoutent à cela. Il ne faut pas que les opérations soient perturbées, au risque d'avoir une corruption de données ou un comportement imprévisible du code. On pourrait penser qu'avec un code relativement simple, ceci ne pose pas de problème, mais en fonction des bibliothèques utilisées, vous n'êtes pas nécessairement maître de l'exécution du code. L'utilisation du port série sur USB, par exemple, génère des interruptions susceptibles de compromettre la bonne marche des effacements/écritures. Il est donc recommandé de suspendre les interruptions lors des phases critiques avec :

```
uint32_t ints = save_and_disable_interrupts();
// faire des trucs
restore_interrupts(ints);
```

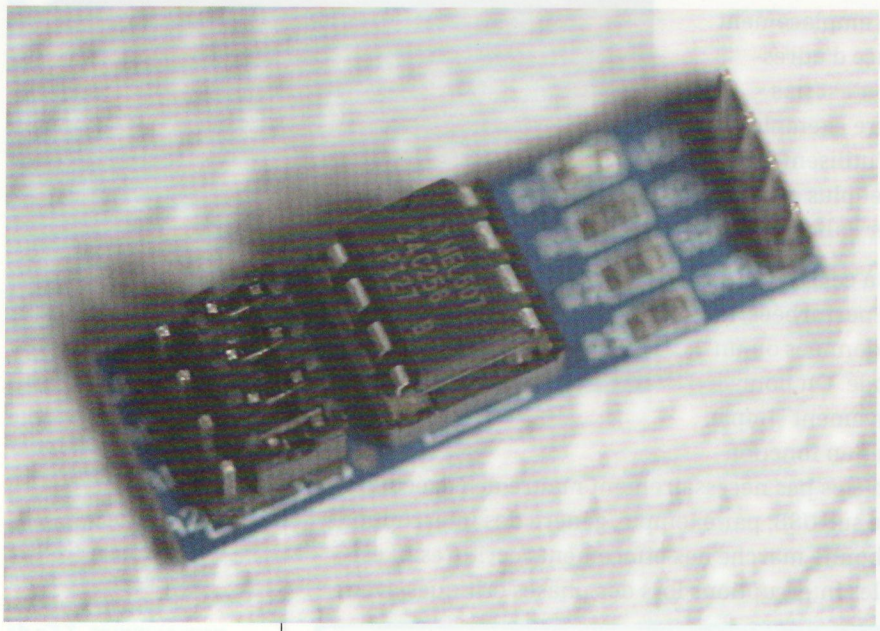
Ces deux fonctions, comme celles concernant l'accès à la flash, nécessiteront l'utilisation de deux bibliothèques spécifiques. Vous devrez donc ajuster le `CMakeLists.txt` de vos projets en précisant :

```
target_link_libraries(${CMAKE_PROJECT_NAME}
                      pico_stdlib
                      hardware_flash
                      hardware_sync
                      )
```

Ce n'est pas tout. Il ne vous est pas possible, sans effort considérable, d'effacer un morceau de code en cours d'exécution. On pourrait s'imaginer déclarer un tableau de 4096 octets en `const` pour qu'il se retrouve dans `.rodata` (et donc en flash), l'aligner sur un multiple de 4096 (avec `__attribute__((aligned(4096)))`) et via un simple pointeur, effacer et réécrire la région. Mais ceci ne fonctionnera pas. Il vous faudrait placer le code en question en RAM, désactiver le XIP, pointer les fonctions utilisées en ROM, accéder à la flash et redémarrer l'ensemble avec le *watchdog*. C'est possible, mais relativement pénible.

D'autres développeurs optent pour une approche plus risquée consistant à compiler le projet, puis à juger où se termine le binaire avec :

```
$ arm-none-eabi-objdump -t main.elf | grep __flash_binary_end
10008bf0 g   .ARM.attributes  00000000 __flash_binary_end
```



Une solution courante lorsque le microcontrôleur ne dispose pas de solution de stockage indépendante est d'avoir recours à une EEPROM externe comme cette 24c256 interfacée en i2c et proposant quelques 32 Ko d'espace. Son adresse étant configurable, il est possible d'en utiliser jusqu'à 8 sur un même bus, augmentant le volume à 256 Ko.

Le symbole indiquant que le code se termine en `0x10008bf0`, on peut alors simplement décider d'utiliser `0x10009000` ou `0x1000a000` comme emplacement pour un enregistrement en flash. Ceci fonctionne, mais implique plusieurs opérations et le risque est bien réel de voir le code grossir et finir par atteindre cette région.

Personnellement je préfère, entre la version pénible et la version « bricolo », une approche mitoyenne que j'estime être un bon compromis (sans pour autant être parfaite)...

2. ÉMULATION D'EEPROM... OU PRESQUE

Lire et écrire en flash depuis un programme, nous l'avons vu, n'est pas un problème si l'on respecte les usages adaptés. Choisir l'emplacement où écrire (et lire) demande une certaine prudence. L'approche que je choisis est de « bloquer » un emplacement qui, si la fonctionnalité est utilisée, sera spécialement dédié. Nous pouvons, en effet, réserver un espace de la flash à cet usage.

Le placement des différents éléments d'un code binaire dans une mémoire est l'une des tâches de l'éditeur de liens. Celui-ci, une fois la phase compilation ayant produit les objets, se charge de réunir cela en un tout cohérent pour former un binaire. Cette tâche n'est pas entièrement automatique et repose en grande partie sur un script utilisé par l'éditeur de liens. Celui livré par défaut dans le SDK de la Pico se trouve dans le sous-répertoire `src/rp2_common/pico_standard_link`. Là, on trouve :

- `memmap_default.ld` : le script utilisé par défaut avec un code classique stocké et exécuté en flash (XIP) ;
- `memmap_copy_to_ram.ld` : exécute le code en RAM après copie depuis la flash ;
- `memmap_no_flash.ld` : pour les binaires purement en RAM (le code doit être chargé à chaque *reboot*) ;
- `memmap_blocked_ram.ld` : ne semble pas documenté, mais le script ne diffère de celui par défaut qu'en précisant une adresse de base pour la RAM à `0x21000000` (il s'agirait donc de rendre la RAM inutilisable ?).

Ces scripts sont automatiquement utilisés par `pico_set_binary_type()` lorsqu'on précise le type de binaire souhaité dans `CMakeLists.txt`, respectivement avec `PICO_DEFAULT_BINARY_TYPE` (ou sans rien préciser), `PICO_COPY_TO_RAM`, `PICO_NO_FLASH` ou `PICO_USE_BLOCKED_RAM`, défini à 1 (par exemple avec `set(PICO_COPY_TO_RAM,1)`).

Le script liste diverses choses comme le point d'entrée du programme, les mémoires en présence et leurs caractéristiques, les différentes sections utilisées et quels éléments du code y trouvent place. Le script par défaut débute ainsi :

```
MEMORY
{
    FLASH(rx) : ORIGIN = 0x10000000, LENGTH = 2048k
    RAM(rwx)  : ORIGIN = 0x20000000, LENGTH = 256k
    SCRATCH_X(rwx) : ORIGIN = 0x20040000, LENGTH = 4k
    SCRATCH_Y(rwx) : ORIGIN = 0x20041000, LENGTH = 4k
}

ENTRY(_entry_point)

SECTIONS
{
    .flash_begin : {
        __flash_binary_start = .;
    } > FLASH

    .boot2 : {
        __boot2_start__ = .;
        KEEP (*(.boot2))
        __boot2_end__ = .;
    } > FLASH
[...]
```

`MEMORY` est ce qui nous intéresse et nous retrouvons là la flash, la RAM ainsi que les *scratch registers* X et Y utilisés par les machines à état des deux PIO du RP2040. Sur cette base, la partie `SECTIONS` dispatche les éléments contenus dans les fichiers objets dans les différentes zones mémoires. Et c'est précisément là que nous pouvons agir.

En effet, le script spécifie non seulement l'adresse, mais également la taille de la flash. Si nous voulons être certains de réserver une partie de celle-ci comme zone de stockage, il nous suffit de l'indiquer ici. Il n'est bien entendu pas question de modifier ce fichier là où il se trouve, mais procédons à une copie que nous plaçons dans le répertoire de notre projet, sous le nom `memmap_custom.ld`. Ceci fait, nous commençons par procéder à quelques calculs pour nous simplifier la vie et utilisons les valeurs résultantes :

```
__EEPROM_LENGTH__       = 4k;
__REAL_FLASH_LENGTH__   = 2048k;
__FLASH_START__         = 0x10000000;
__FLASH_LENGTH__        = __REAL_FLASH_LENGTH__ - __EEPROM_LENGTH__;
__EEPROM_START__        = __FLASH_START__ + __FLASH_LENGTH__;
```

```
MEMORY
{
    FLASH(rx) : ORIGIN = __FLASH_START__, LENGTH = __FLASH_LENGTH__
    EEPROM(r) : ORIGIN = __EEPROM_START__, LENGTH = __EEPROM_LENGTH__
    RAM(rwx)  : ORIGIN = 0x20000000, LENGTH = 256k
    SCRATCH_X(rwx) : ORIGIN = 0x20040000, LENGTH = 4k
    SCRATCH_Y(rwx) : ORIGIN = 0x20041000, LENGTH = 4k
}
```

La flash se terminera maintenant à 4096 octets de sa valeur initiale et une zone supplémentaire, `EEPROM`, a fait son apparition. Si plus tard nous souhaitons plus de mémoire de stockage, nous n'aurons qu'à augmenter `__EEPROM_LENGTH__` (sans oublier qu'il doit s'agir d'un multiple de 4096). Cette modification nous permettra donc d'utiliser 4096 octets à partir de l'adresse `0x101ff000` (`0x001ff000` pour la flash elle-même) et par la même occasion d'exclure cette zone de l'espace utilisable pour le code.

Pour utiliser ce script en lieu et place de celui par défaut, une simple modification de notre `CMakeLists.txt` est suffisante :

```
set_target_properties(
        ${CMAKE_PROJECT_NAME}
        PROPERTIES PICO_TARGET_LINKER_SCRIPT
                ${CMAKE_SOURCE_DIR}/memmap_custom.ld
        )
```

Nous n'avons pas besoin de spécifier l'adresse dans notre code puisque l'éditeur de liens est parfaitement capable de faire ce travail pour nous. Nous n'avons qu'à référencer les symboles du script dans notre code.

3. PASSONS AU CODE !

Ce qui va suivre ne se veut pas être un code directement réutilisable dans vos projets, mais une simple démonstration mettant en évidence le principe de fonction. Il restera à votre charge de rendre cela modulaire, d'écrire des fonctions dédiées et, bien entendu, d'adapter le tout aux informations que vous voulez stocker.

Commençons par les fichiers d'en-tête et les variables globales :

```
#include <stdio.h>
#include <stdlib.h>
#include <pico/stdlib.h>
#include <hardware/flash.h>
#include <hardware/regs/rosc.h>
#include <tusb.h>

// données "miroir"
uint8_t *data;
```

```
// zone eeprom
uint8_t *eeprom;

// Infos de l'éditeur de liens
// adresse de la zone EEPROM
extern uint32_t __EEPROM_START__[];
uint32_t start_of_eeprom = (uint32_t)__EEPROM_START__;
// Taille de la zone
extern uint32_t __EEPROM_LENGTH__[];
uint32_t eeprom_length = (uint32_t)__EEPROM_LENGTH__;
```

Notez l'inclusion de `flash.h` et de `regs/rosc.h` nécessaires à la fonction que nous allons voir dans un instant. `tusb.h` est là pour fournir la fonction `tud_cdc_connected()` qui nous permettra d'attendre une connexion USB avant d'envoyer les messages sur le port série, puisque nous utilisons cette fonctionnalité, en spécifiant, dans notre `CMakeLists.txt` :

```
# enable usb output, disable uart output
pico_enable_stdio_usb(${CMAKE_PROJECT_NAME} 1)
pico_enable_stdio_uart(${CMAKE_PROJECT_NAME} 0)
```

Récupérer les valeurs qui nous intéressent tient en peu de chose. Nous déclarons une « variable » externe (qui n'est pas un *array*, mais une variable de l'éditeur de liens) et nous utilisons la valeur pour initialiser une variable globale. L'adresse de la zone est alors dans `start_of_eeprom` et sa taille dans `eeprom_length`. Ces deux variables pourront alors être utilisées sans problème dans le reste du code. Notez que `__EEPROM_START__` et `__EEPROM_LENGTH__` ne sont pas réellement des variables au sens C du terme, il est plus juste de voir cela comme des symboles exportés par l'éditeur de liens.

`data` est notre pointeur vers les données en RAM et `eeprom` vers la zone en flash. L'opération finale consistera alors à simplement copier ce qui se trouve à `*data` vers `*eeprom`. Avant de nous attaquer à cela, nous avons besoin de données à placer en RAM qui serviront de base, mais également de point de comparaison pour vérifier que l'écriture a bel et bien fonctionné. Pour ce faire, nous pourrions simplement initialiser toutes ces données à l'aide d'un simple compteur bouclant sur 8 bits et nous fournissant un lot de `uint8_t` (`unsigned char`). Cependant, il serait mieux d'éviter des valeurs trop prédictibles, limitant la pertinence de cette vérification.

Nous allons donc initialiser les données de façon aléatoire et *stdlib* met pour cela à disposition la fonction `rand()`. Cependant, pour bien faire les choses, le générateur de nombre pseudoaléatoire doit être initialisé avec une *graine*, qui devra être différente à chaque exécution. Nous avons donc besoin d'une fonction dédiée pour cette initialisation et passer à `srand()` une valeur déjà aléatoire :

```
// graine aléatoire pour rand()
void seed_random_from_rosc()
{
    uint32_t random = 0;
    uint32_t random_bit;
```

```
    volatile uint32_t *rnd_reg =
       (uint32_t *)(ROSC_BASE + ROSC_RANDOMBIT_OFFSET);

    for (int k = 0; k < 32; k++) {
        while(1) {
            random_bit = (*rnd_reg) & 1;
            if(random_bit != ((*rnd_reg) & 1)) break;
        }
        random = (random << 1) | random_bit;
    }
    srand(random);
}
```

Cette fonction utilise l'état de l'oscillateur (*ring oscillator*) interne du RP2040 et est définie comme étant une source fiable pour une valeur aléatoire, sauf si l'oscillateur est stoppé ou fonctionne sur une fréquence harmonique de celle du bus (ce qui n'est pas le cas par défaut). Obtenir un `unsigned int` pouvant être utilisé en argument de `srand()` se résume à un jonglage de bits.

Tout est maintenant prêt pour `main()` qui débute avec les fonctions habituelles du SDK Pico :

```
int main()
{
    // initialisation
    gpio_init(PICO_DEFAULT_LED_PIN);
    gpio_set_dir(PICO_DEFAULT_LED_PIN, GPIO_OUT);
    stdio_init_all();

    // on attend la connexion USB UART
    while (!tud_cdc_connected()) {
        sleep_ms(250);
    }

    printf("Zone EEPROM à 0x%08x (taille=%u)\n",
           start_of_eeprom, eeprom_length);
    eeprom = (uint8_t *)start_of_eeprom;
```

J'ai délibérément ajouté la prise en charge de la LED présente sur la carte afin d'avoir un témoin visuel de l'exécution, ou plus exactement de la terminaison du code. Nous sommes en USB/série, ce qui signifie que le périphérique USB « simulé » par la Pico n'existe qu'après la mise sous tension et que le temps que dure l'exécution du code. Une boucle `while()` infinie est indispensable, même pour un simple test, et autant y ajouter un petit clignotement de LED. C'est aussi pour cette raison que nous conditionnons l'exécution du reste du code à la connexion en USB avec `tud_cdc_connected()`.

Nous pouvons alors passer à la suite en allouant la mémoire pour le *buffer* en RAM, initialisant le PRNG et en remplissant le *buffer* de données aléatoires :

```
// allocation buffer RAM
if((data=malloc(eeprom_length*sizeof(uint8_t)))==NULL) {
    printf("Erreur malloc() !!!\n");
    while(1) { ; }
}

// graine PRNG
seed_random_from_rosc();

// Remplissage données RAM
for(int i=0; i<eeprom_length; i++) {
    data[i] = (rand() & 0xff);
}
```

Nos données sont prêtes et nous pouvons les inscrire en flash comme nous l'avons décrit précédemment avec les fonctions fournies par `flash.h` :

```
// effacement & écriture
printf("Effacement et programmation flash\n");
printf("De 0x%08x (RAM) vers 0x%08x (Flash 0x%08x). Taille=%u.\n",
        data, eeprom, (uint32_t)eeprom-XIP_BASE, eeprom_length);
uint32_t ints = save_and_disable_interrupts();
flash_range_erase((uint32_t)eeprom-XIP_BASE, eeprom_length);
flash_range_program((uint32_t)eeprom-XIP_BASE,
        (const uint8_t *)data, eeprom_length);
restore_interrupts(ints);
printf("Fait.\n");
```

Les données dans notre EEPROM émulée peuvent être accédées comme si elles étaient en RAM en utilisant tout simplement le pointeur `*eeprom`. Ceci nous permet donc de simplement boucler sur toute la zone et de comparer le contenu pointé par `data` à celui de la flash :

```
// comparaison
printf("Vérification.\n");
int error = 0;
for(int i=0; i<eeprom_length; i++) {
    if(eeprom[i] != data[i]) error++;
}

if(error)
    printf("%d erreur(s)\n", error);
else
    printf("Aucune erreur.\n");
```

Pour l'amusement et pour confirmer que chaque exécution crée bien un jeu de données différent, un affichage partiel des valeurs sera du plus bel effet :

```
    for(int i=0; i<256; i++) {
        printf("%02x ", eeprom[i]);
        if(((i+1) % 16) == 0)
            printf("\n");
    }
```

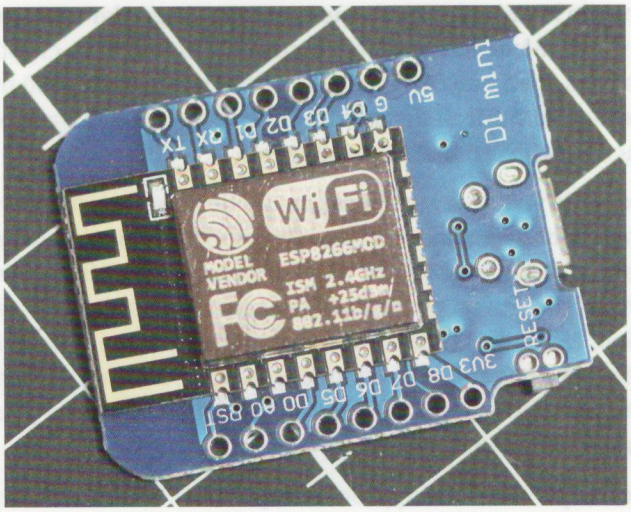
Enfin, nous finissons avec la classique boucle infinie :

```
    // boucle infinie
    while(1) {
        gpio_put(PICO_DEFAULT_LED_PIN, 1);
        sleep_ms(250);
        gpio_put(PICO_DEFAULT_LED_PIN, 0);
        sleep_ms(750);
    }

    return 0;
}
```

Une fois le code compilé et chargé dans la Pico, une connexion (115200 8N1) sur le port série devrait provoquer l'affichage de quelque chose comme ceci :

```
Zone EEPROM à 0x101ff000 (taille=4096)
Effacement et programmation flash
De 0x20001ce0 (RAM) vers 0x101ff000 (Flash 0x001ff000).
Taille=4096.
Fait.
Vérification.
Aucune erreur.
a9 93 7c 9f d9 55 c3 f4 bc 57 2f 64 85 59 3c a7
54 19 9c f2 93 c7 5c 04 34 d3 c1 d5 ba 2f 22 99
be 4a 46 35 41 19 f2 dd 18 f9 5a d6 14 eb bb 77
4e 5e 25 bc 3e 36 89 4c f2 94 a2 20 91 28 b1 d5
4b e2 7b 17 f9 b9 8d f2 07 b7 36 f7 d2 dd 82 44
11 3c d0 6a 35 cb c7 c2 ea 61 9a c9 10 2d 18 07
bb 2c e8 13 84 b6 5e a2 f2 96 f6 ab d3 81 ef 58
29 46 05 b4 ea 37 e7 1e 9f fb 9c ac 41 d5 ce 4e
83 6a 73 9c a7 92 7d 40 ec ed 61 0c 29 48 01 63
1e 39 69 84 36 56 e6 7b 8e 0d b9 49 af df 29 8d
ca 7f 2b ad 71 e5 c5 bb 1a 55 a7 0b a9 91 90 fd
9a a6 7b 5a e2 bb db 8d 1b 9c 75 e7 a9 7f 12 6f
08 18 5f a2 42 66 4c 60 16 9b 36 09 b0 78 86 22
90 a4 24 a6 25 44 f7 e9 73 72 1d ad 8e ab be 62
b1 dd b8 1a ce fa d7 98 59 a5 91 7a ed a2 0f b1
60 79 4e 2c 33 a3 72 3a 71 92 1f bf 0c 6b 61 92
```

Nous disposons maintenant d'une zone de stockage de 4096 octets pouvant survivre aux redémarrages, comme c'est le cas avec l'AVR d'un Arduino, mais en beaucoup mieux.



Les ESP8266 et ESP32 utilisent une flash série dont une partie peut également être utilisée comme une émulation d'EEPROM, mais on préférera, selon le projet, reposer sur SPIFFS proposant un système de fichiers complet et structuré.

4. IL NE S'AGIT PAS D'UNE VRAIE EEPROM

Ce que nous venons de voir est en réalité une pratique relativement courante et beaucoup d'implémentations précisent qu'il faut rester prudent. En effet, une flash destinée au stockage de code n'est pas faite pour des cycles d'écriture intensifs comme celles équipant, par exemple, les disques SSD, les clés USB ou les cartes MMC/SD.

Dans le cas de la Raspberry Pi Pico cependant, deux choses doivent être considérées avant de tirer des conclusions similaires. Premièrement, la flash QSPI Winbond W25Q16JV équipant la Pico officielle affiche des performances intéressantes. La *datasheet* **[5]** précise en page 4 « *Min. 100K Program-Erase cycles per sector* ». Cette valeur, parfois notée « *P/E cycles* » pour un support SD, est donnée en milliers ou dizaines de milliers par secteur, mais il faut prendre en considération la présence d'un FTL (*Flash Translation Layer*), une interface placée avant la flash elle-même, chargée de l'allocation des données, des corrections d'erreurs, du cache, de la gestion des blocs défectueux, etc.

Il est donc difficile de comparer les deux types de support, mais nous pouvons tout simplement ne considérer que les informations de la *datasheet*, tout à fait comparables à celles d'une EEPROM classique. 100000 P/E cycles sont une valeur relativement importante, même en considérant que ce sera toujours le même secteur qui sera utilisé. De plus, et c'est le second point, il s'agit ici d'une flash QSPI externe au microcontrôleur et rien ne nous empêche de la remplacer le cas échéant.

5. POUR FINIR

Il reste du travail à faire sur cette base, mais l'essentiel est fait. En adaptant relativement simplement le script de l'éditeur de liens, il est possible d'attribuer plus ou moins d'espace de la flash à cette utilisation spécifique. Étant donné le volume de notre disposition de base (2 Mo) et sachant que d'autres variations autour du RP2040 offrent jusqu'à 16 Mo de flash, il est parfaitement possible d'allouer bien plus d'espace, voire de travailler avec plusieurs zones. Ceci permettrait un fonctionnement en *double buffer* où une mise à jour se ferait dans une zone

différente de la précédente, permettant ainsi de garder un historique ou une copie des données. Il serait même envisageable d'utiliser des fonctions cryptographiques pour chiffrer les informations, bien que ceci serait d'un effet limité, le RP2040 ne disposant pas de mécanisme de protection (il suffirait de faire un reverse du code en flash pour avoir un bon point d'entrée si la fonctionnalité est mal implémentée).

Du point de vue de la structuration des données, un gros travail reste à faire, mais celui-ci sera totalement dépendant de votre projet. Il suffit pour cela de « mapper » votre ou vos structures dans le *buffer* en RAM, qu'on écrira en flash ou qu'on rafraîchira depuis la flash. Enfin, nous avons ici ce qu'on pourrait considérer comme la couche la plus basse en termes d'implémentation. L'écriture de fonctions dédiées peut être tout aussi intéressante que pédagogique. En particulier en prenant en compte la taille de la zone en multiples de 4096 octets et donc en effaçant et ne réinscrivant que les données nécessaires. On pourrait même aller jusqu'à l'écriture d'une bibliothèque dédiée.

Ceci est cependant une approche risquée, car si cette technique est possible à cette date, rien ne garantit qu'elle le restera. Graham Sanderson a d'ailleurs ajouté un ticket (*issue*) sur le GitHub du SDK Pico **[6]** soulevant un problème lié aux plateformes ayant plus de 2 Mo de flash.

En effet, même si les en-têtes dans `pico-sdk/src/boards/include/boards` listent des macros spécifiques à chaque carte à base de RP2040 (utilisables via un `cmake ../ -DPICO_BOARD=` suivi du nom de la *board*), rien ne change au niveau du script de l'éditeur de liens (on aura toujours que 2 Mo de flash). Ce que propose Graham consiste à composer un script à partir d'un *template* au moment de la construction du *firmware*, mais cette fonctionnalité n'est planifiée (pour l'instant) pour aucune future version (*milestone*) précise du SDK. Son message décrit la technique consistant à copier le script comme « très friable » (« *very brittle* »), ce qui laisse penser qu'un changement aura lieu tôt ou tard et que vous devrez alors adapter cette technique.

Il n'en reste pas moins que cette petite exploration du SDK est très instructive et confirme que la Pico n'est pas une plateforme vraiment comme les autres et à plus d'un tour dans son sac. Plus on fouille, plus on est sous son charme... **DB**

RÉFÉRENCES

[1] **https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/storage/nvs_flash.html**

[2] **https://github.com/espressif/arduino-esp32/tree/master/libraries/Preferences**

[3] **https://connect.ed-diamond.com/Hackable/hk-038/pico-acceder-aux-informations-binaires-depuis-votre-code**

[4] **https://connect.ed-diamond.com/contenu-premium/raspberry-pi-pico-pio-dma-et-memoire-flash**

[5] **https://www.winbond.com/resource-files/w25q16jv%20spi%20revh%2004082019%20plus.pdf**

[6] **https://github.com/raspberrypi/pico-sdk/issues/398**

UNE HORLOGE ARDUINO : QUELQUES TECHNIQUES INTÉRESSANTES

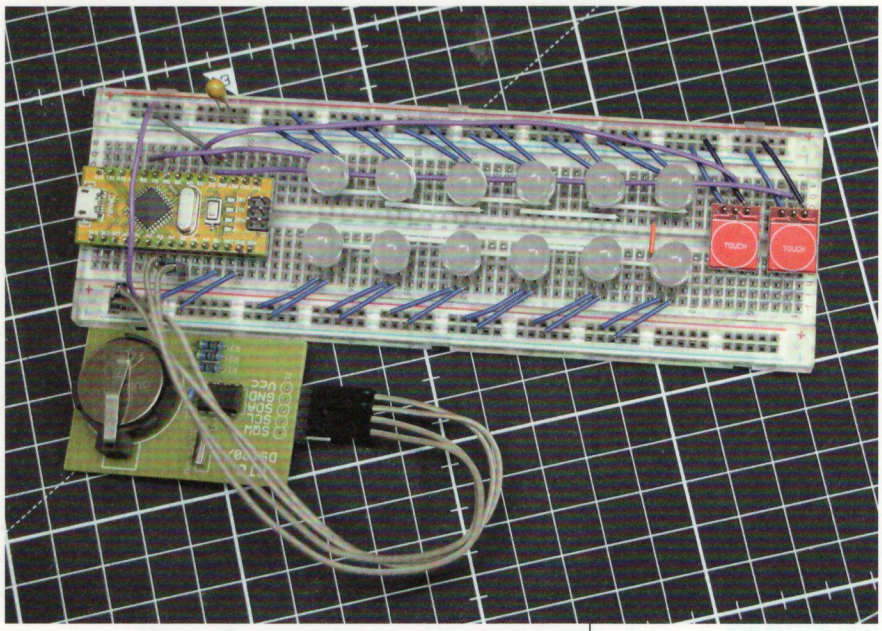
Denis Bodor

Il y a de petits projets de week-end qui paraissent anodins à première vue, mais qui, une fois dans les méandres du code, révèlent des problématiques aussi intéressantes que les solutions qui les accompagnent. L'objet du présent article n'est pas réellement de vous détailler le montage en lui-même, après tout il ne s'agit que d'une horloge décorative avec un microcontrôleur AVR, un module RTC, quelques LED adressables APA106 et le framework Arduino. Non, l'idée est simplement de vous présenter quelques « points de friction » et les approches utilisées pour les « lubrifier »...

– Une horloge Arduino : quelques techniques intéressantes –

À l'heure où sont couchés ces mots sur l'écran, nous sommes à une paire de week-ends des fêtes de fin d'année et donc de Noël. Étant le premier à critiquer l'utilisation abusive de décorations lumineuses, que le ou la propriétaire ne voit presque jamais si elles sont en extérieur, je m'amuse de l'ironie consistant à construire un montage plein de LED, destiné à devenir un présent pour la fête des sapins agonisants. Quoi qu'il en soit, et malgré des préférences personnelles compatibles avec celles du Grinch, me voici donc à concevoir un projet qui semble initialement simple…

Pour allier l'utile au décoratif, et inversement, le projet consiste à faire usage de quelques LED APA106, changeant de couleurs de façon douce et reposante, tout en fournissant une indication de l'heure actuelle, fournie par une RTC de type DS1307. Les APA106 sont des LED RGB adressables se présentant sous la forme de gros composants translucides de 8 mm de diamètre. Ces LED sont utilisables de la même façon que les classiques WS2812b, mais ont un côté « jumbo » et vintage que je trouve fort sympathique. À cela s'ajoutent deux boutons capacitifs TTP223 permettant un réglage, qui m'éviteront de devoir gérer les rebonds des boutons mécaniques et apporteront une touche moderne au résultat. Le tout est géré par un modeste clone d'Arduino Nano équipé d'un ATmega328.



Quel que soit le projet, la platine à essais reste l'étape de prototypage par défaut permettant de mettre au point son code et de se heurter aux différents problèmes que seule une mise en pratique peut faire apparaître.

Côté bibliothèques, la RTC est pilotée par « *Rtc by Makuna* » de Michael C. Miller, tandis que les APA106 sont prises en charge par la fantastique FastLED de Daniel Garcia. Si vous avez l'habitude de travailler avec la « *NeoPixel Library* » d'Adafruit, prenez le temps de vous pencher sur FastLED, vous ne le regretterez pas et ne reviendrez certainement pas en arrière. Pour finir avec l'aspect logiciel, précisons que le tout est développé dans Visual Studio Code avec l'environnement PlatformIO, même si ceci est d'une importance très secondaire puisque le *framework* Arduino/Wiring est identique à celui de l'IDE Arduino.

Voilà qui plante parfaitement le décor : un projet, une poignée de composants qui traînaient, de bonnes bibliothèques et quelques heures devant l'écran et le tour sera joué…

1. DONNER L'HEURE AVEC UN MINIMUM DE LED

Voilà un problème dont la solution est aisée à première vue : il suffit de présenter alternativement l'heure et les minutes sous forme binaire. Des heures entre 0 et 23, des minutes entre 0 et 59 et nous voici à n'utiliser que 6 LED (2^6 = 64). Ceci, bien entendu, à condition que la personne lisant l'heure soit capable de convertir du binaire en décimal de tête, et ce, à la vitesse imposée par l'horloge (ou soit désireuse d'apprendre à le faire). Je ne sais pas pour vous, mais d'expérience, je dirais que les cadeaux accompagnés d'une obligation d'assimiler un concept exotique, et d'y devenir compétent, ne sont pas très populaires. De plus, ce choix technique suppose que les LED aient des positions fixes et/ou qu'une valeur leur soit explicitement associée, ce qui est peu décoratif.

Les humains « standard » préfèrent compter en décimal et ne sont généralement pas naturellement enclins à procéder à des opérations mathématiques complexes de tête et rapidement pour le plaisir. Pour minimiser la quantité de LED sans que cela ne devienne une torture, nous n'avons d'autre choix que de sacrifier de la précision. Exit donc des 24 heures quotidiennes, de la notion de matin et d'après-midi, et de celle de minutes en guise de plus petite unité de mesure des fractions d'heure.

Les LED adressables APA106 sont relativement proches de leurs cousines, les WS2812b. Elles font partie des très nombreux modèles nativement supportés par la bibliothèque FastLED et sont, je trouve, bien plus faciles à intégrer esthétiquement dans une réalisation.

Nous aurons 12 LED présentant les heures de 1 à 12 dans une couleur, puis les minutes dans une autre couleur sous la forme de multiples de 5. Une heure sera donc divisée en 12 paquets de 5 minutes, de 0 à 11, le nombre de LED indiquant que « LED fois 5 minutes se déjà sont écoulées depuis l'heure donnée précédemment ». Ceci est relativement facile à calculer de tête via différentes astuces lorsqu'on se souvient de sa table de 5. Si peu de LED sont allumées, on multiplie simplement par 5. S'il y a plus de LED allumées qu'éteintes, on compte celles qui sont éteintes, on soustrait de 12 et l'on multiplie le résultat par 5.

L'échelle des heures, de 1 à 12, nous évite à la fois de différencier midi et minuit et de nous retrouver dans une situation où l'on peut douter du fonctionnement du montage. Si rien ne se passe et rien ne s'allume, de nuit comme de jour, on saura que quelque chose ne va pas. N'oubliez pas, ceci est une décoration qui donne l'heure et non une horloge décorative à LED.

2. ALLUMAGE ALÉATOIRE DES LED

Vous l'avez compris, la position des LED n'a aucune importance, uniquement leur nombre. En cela, répéter le même motif avec la même alternance durant 5 longues minutes n'est pas intéressant. À 15 h 32 par exemple, nous aurons donc toujours 3 des 12 LED allumées dans la couleur des heures et 6 des 12 LED dans la couleur des minutes, mais leur position peut (et doit) changer à chaque transition.

La question est donc : comment allumer aléatoirement un nombre donné de LED lorsque celles-ci sont naturellement indexées de 0 à 11 ? En effet, les APA106 comme les WS2812b proposent 4 broches, l'alimentation, la masse, « data in » et « data out ». Les composants sont chaînés les uns aux autres, la « data in » de la première LED connectée à une sortie du microcontrôleur et sa « data out » connectée à la « data in » de la LED suivante. Et FastLED représente cela ainsi :

```
#include <FastLED.h>

#define NUM_LEDS        12
#define DATA_PIN         8

CRGB leds[NUM_LEDS];

void setup()
{
  FastLED.addLeds<APA106, DATA_PIN, RGB>(leds, NUM_LEDS);
  FastLED.setBrightness(128);

  for(int i=0; i<NUM_LEDS; i++) {
    leds[i] = CHSV(i*20, 255, 255);
  }
  FastLED.show();
}
```

Pour définir la couleur des 12 LED, nous parcourons le tableau `led[]` et définissons, à chaque emplacement, une couleur via un triplet Teinte/Saturation/Valeur ou TSV (HSV en anglais). L'approche première permettant d'allumer par exemple 6 LED à des positions aléatoires sur les 12 emplacements pourrait être de choisir au hasard les positions une à une avec `random()`. Mais l'aléatoire étant... aléatoire, rien ne vous assure de ne pas piocher deux fois une même position. Il faudrait alors tester l'état et, le cas échéant, par exemple, tester la suivante, puis la suivante, etc., jusqu'à en trouver une de libre.

Une autre solution, bien plus intéressante, consiste à créer un tableau contenant les positions, de 0 à 11, de le mélanger et de prendre les *n* premiers éléments, où *n* est le nombre de LED à allumer. Peu importe alors le numéro « pioché », il sera forcément disponible, exactement comme on bat un jeu de cartes dans lequel on pioche ensuite par le haut du tas.

```
int rled[NUM_LEDS];

[...]

for(int i=0; i<NUM_LEDS; i++)
  rled[i]=i;
```

```
[...]

shuffle(rled, NUM_LEDS);

for(int i=0; i<cf_heure; i++)
  leds[rled[i]] = cf_coulh;
```

`rled[]` est notre tableau d'index et nous utilisons une simple boucle pour l'initialiser avec des valeurs successives de 0 à 1. La fonction `shuffle()` mélange tout cela et nous utilisons `rled[]` pour obtenir la position dans `leds[]` où changer la couleur. `cf_heure` est ici la valeur de l'heure lue depuis la RTC, modulo 12 (la valeur `0` est remplacée systématiquement par `12` pour éviter que minuit et/ou midi n'allument aucune LED). Bien entendu, tout repose sur `shuffle()` :

```
void shuffle(int *array, size_t n)
{
  if(n<2) return;

  for(size_t i=0; i<n-1; i++) {
    size_t j = i+rand()/(RAND_MAX/(n-i)+1);
    int t = array[j];
    array[j] = array[i];
    array[i] = t;
  }
}
```

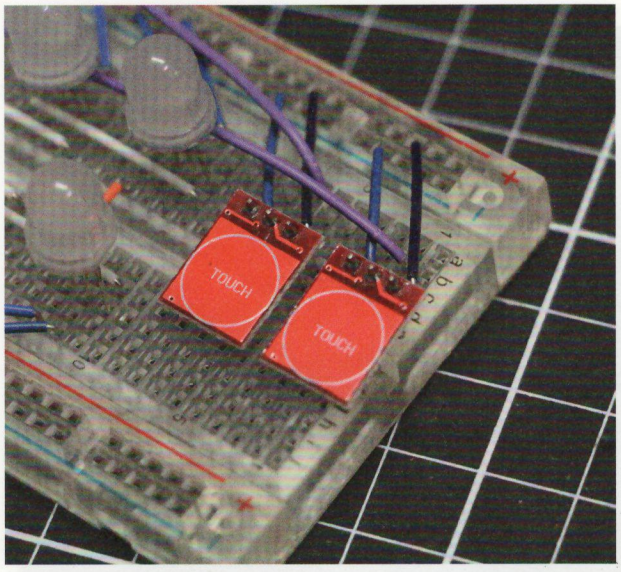
`n` correspond à la taille du tableau dont le pointeur est passé en argument de la fonction et nous bouclons simplement de 0 à `n`-1 pour permuter deux valeurs entre l'emplacement courant (`i`) et un emplacement aléatoire (`j`). Pour que cela fonctionne correctement, il faut que `RAND_MAX` soit très supérieur à `n`, ce qui est le cas ici avec `RAND_MAX` valant 32767. Avec un tableau plus grand, nous aurions dû utiliser autre chose, comme la fonction Arduino `random()` retournant une valeur 32 bits.

3. FADING NON LINÉAIRE AVEC UN (CO)SINUS

Si vous vous êtes déjà amusé à faire varier l'intensité d'une LED, que ce soit avec de la PWM ou en utilisant une WS2812b, et plus exactement à faire pulser une LED, vous avez sans doute remarqué que l'effet obtenu en augmentant, puis en réduisant linéairement son intensité n'est pas très satisfaisant. Pourtant, passer de 0 à 255, en utilisant `FastLED.setBrightness()` par exemple, puis en faisant de même de 255 à 0 est parfaitement logique. Après tout, vous procédez à des variations successives, de valeurs égales, sur une période constante et récurrente. Cela devrait fonctionner...

Le problème n'est pas la PWM, la variation commandée ou votre code, c'est vous. Plus exactement, c'est votre perception de cette variation qui n'est pas satisfaisante. En effet, pour obtenir le rendu souhaité, il faut que le changement soit plus naturel, plus organique. Si nous faisons l'analogie avec un véhicule par exemple, celui-ci n'atteint pas la vitesse souhaitée instantanément, il accélère doucement, s'approche de la vitesse, la maintient, puis décélère de plus en plus rapidement, pour enfin ralentir doucement avant l'arrêt complet. Ceci est naturel et perçu comme tel. Il s'agit de la même chose avec une LED.

Pour obtenir quelque chose de similaire, la technique est relativement simple. Plutôt qu'une progression linéaire, nous devons utiliser une fonction sinusoïdale et il existe



Les boutons capacitifs TTP223 se présentent sous la forme de modules avec au recto, la surface de détection et à l'arrière, le circuit de contrôle. Ce genre d'élément ne coûte pas beaucoup plus cher qu'un bouton mécanique et évite de devoir gérer les rebonds (matériellement ou de manière logicielle).

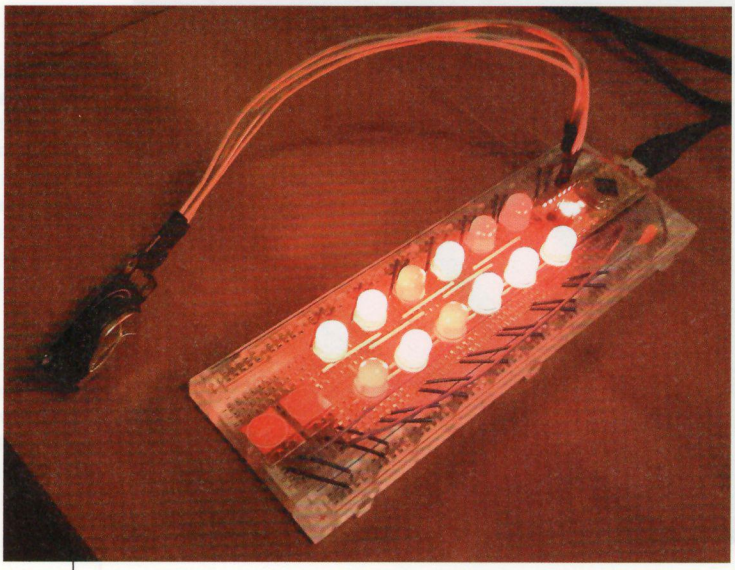
maintes solutions pour arriver à un tel résultat. L'une d'elles consiste à précalculer une liste de valeurs qu'on pourra ensuite appliquer en guise de rapport cyclique (PWM) ou d'intensité lumineuse (FastLED). Ceci peut être fait, par exemple, comme je l'avais évoqué dans mon précédent article sur l'utilisation des PIO et du DMA sur Raspberry Pi Pico **[1]**, avec un simple script GNU bc :

```
define i(x) {
   auto s
   s = scale
   scale = 0
   x /= 1
   scale = s
   return (x)
}

nbr=80
amp=231

for(a=1;a<=nbr;a=a+1) {
  i(((s(3.14159*(0.5+((2/nbr)*a)))+1)/2)*amp)
}

quit
```

En jouant sur `nbr` et `amp`, nous pouvons ainsi obtenir une liste de valeurs (`nbr`) variant entre zéro et `amp` :

```
230 229 227 225 222 218 213 208 203 197 190
183 175 167 159 151 142 133 124 115 106 97
88 79 71 63 55 47 40 33 27 22 17 12 8 5 3 1
0 0 0 1 3 5 8 12 17 22 27 33 40 47 55 63 71
79 88 97 106 115 124 133 142 151 159 167
175 183 190 197 203 208 213 218 222 225 227
229 230 230
```

Il suffirait alors de stocker tout cela dans un tableau et prendre les valeurs une à une pour obtenir l'effet que nous désirons. Cependant, tout ceci reste statique et ne permet pas de changer des paramètres sans avoir à générer un nouveau tableau à placer dans les sources. Dans le cadre du projet qui nous intéresse, le nombre de valeurs



Il est 6 heures... ou 18 heures, peut-être.

est fixe (256), mais la valeur maximum dépendra de l'intensité maximum souhaitée, et celle-ci est configurable par l'utilisateur en fonction de son environnement (12 APA106 peuvent être vraiment très lumineuses et dérangeantes dans l'obscurité d'une chambre ou d'un salon, par exemple).

Calculer 256 échantillons prend du temps et il n'est aucunement nécessaire de le faire de manière récurrente. Ce n'est que lorsque la configuration est changée qu'il faut rafraîchir les valeurs. L'option choisie est donc de créer une table de 256 valeurs appelée `sintable[]`, la remplir au démarrage et aux changements de configuration, puis l'utiliser pour notre effet de *fading*. Commençons par les macros et variables :

```
#define SINSTEPS       256
#define FADEDELAY       10
#define MINBRIGHT        0
#define MAXBRIGHT       32

int cf_maxbright=MAXBRIGHT

int sintable[SINSTEPS];
```

Le générateur de valeurs sera :

```
void gensintable(int min, int max, int size)
{
  int range = max-min;
  int pos=0;
  for(float i = 0; i<2*PI; i+=(2*PI/size)) {
    sintable[pos++] = (int)((((cos(i)+1)/2)*range)+min);
  }
}
```

Notez que la fonction utilise un nombre de valeurs en argument, bien qu'ici ce ne soit pas utile. L'idée est de rendre, bien entendu, le code générique et utilisable par ailleurs. Second point, bien que la table et la fonction semblent concerner un sinus, nous utilisons en réalité la fonction cosinus afin que la première valeur du tableau soit la valeur maximum et non zéro (c'est une simple préférence personnelle). Il ne nous reste plus qu'à appeler cette fonction avec les bons paramètres au moment opportun :

```
gensintable(MINBRIGHT, cf_maxbright, SINSTEPS);
```

À ce stade, vous devez sans doute commencer à comprendre l'objet du préfixe `cf_` de certaines variables. Remarquez également que `sintable[]` est une variable globale, mais que dans certaines situations, il serait préférable d'en faire un argument de `gensintable()` pour générer ou rafraîchir plusieurs tables de valeurs, éventuellement de différentes tailles.

Il ne nous reste plus maintenant qu'à créer notre fonction d'effet qui se content alors d'appliquer successivement la moitié des valeurs de la table en guise d'intensité lumineuse :

```
void sinfade(bool up)
{
  if(up) {
    for(int i=SINSTEPS/2; i<SINSTEPS; i++) {
      FastLED.setBrightness(sintable[i]);
      FastLED.show();
      delay(FADEDELAY);
    }
  } else {
    for(int i=0; i<SINSTEPS/2; i++) {
      FastLED.setBrightness(sintable[i]);
      FastLED.show();
      delay(FADEDELAY);
    }
  }
}
```

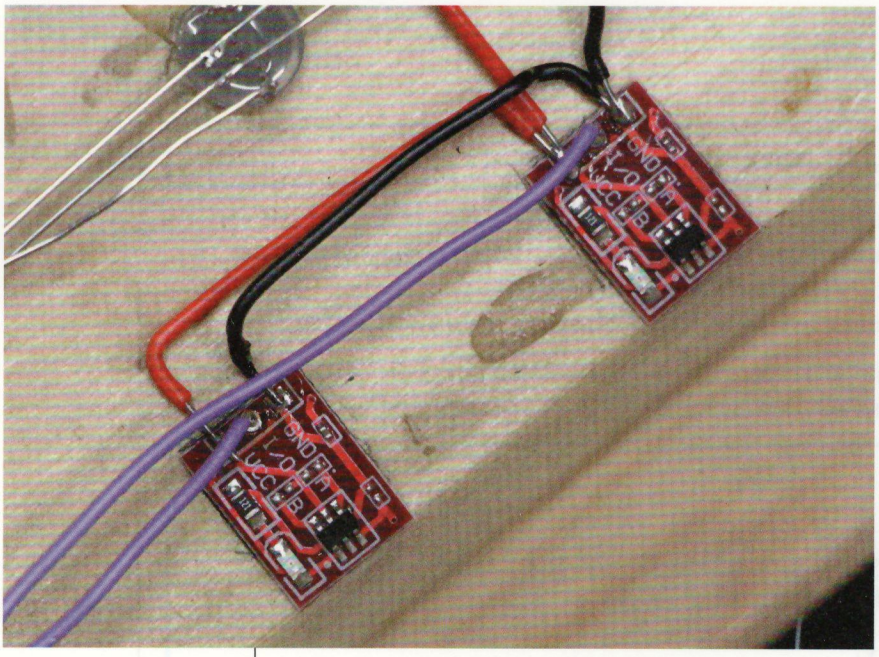
Là encore, c'est une question de préférence personnelle. Nous aurions tout aussi bien pu créer deux fonctions, `fadein()` et `fadeout()`, mais je trouve le passage d'un paramètre plus souple et générique. On pourrait également remplacer le `bool` par un `int`, utiliser la valeur absolue passée en guise d'argument pour `delay()` et le signe comme direction (*in* ou *out*).

4. GÉRER UNE INTERFACE AVEC DEUX BOUTONS

Créer une interface utilisateur n'est pas facile. Il faut se mettre dans la peau d'une personne « normale » utilisant une logique souvent différente de celle d'un développeur, et qui ne comprendrait pas pourquoi telle chose fonctionne de telle manière, même si les raisons techniques sont évidentes. La plupart des horloges digitales ou des réveils modernes utilisent deux ou trois boutons seulement :

- un bouton pour la configuration (« SET ») ;
- un bouton pour changer ce qu'on configure, parfois le même ;
- et un bouton (« UP ») permettant une incrémentation.

Rien de plus énervant avec ce genre de matériel d'entrée de gamme qu'un passage à l'heure d'hiver où il faut reculer d'une heure. On est obligé de faire le « tour du cadran » en l'absence d'un bouton pour décrémenter. En construisant soi-même une horloge ou une décoration comme



Autre avantage de l'utilisation de capteurs capacitifs, les boutons sont faciles à dissimuler puisqu'ils sont parfaitement capables de détecter un contact au travers des quelque 3 millimètres de contreplaqué de la réalisation. Ceci ne fonctionnera cependant qu'avec des matériaux diélectriques.

le projet « excuse » pour le présent article, on comprend mieux pourquoi. Non seulement le bouton en lui-même à un coût de production, mais il faut également lui attacher une fonction et donc du code. C'est bien plus simple et économique d'avoir un bouton pour passer en mode configuration et un unique bouton d'incrémentation.

Mon plan a donc été le suivant, où chaque pression sur le bouton de configuration change de mode :

- 0. Mode affichage avec pulsation des LED.
- 1. Réglage de l'heure : affichage de l'heure courante, puis incrémentation de 1 à 12.
- 2. Réglage des minutes : affichage des groupes de minutes courantes, puis incrémentation par « paquet » de 5 minutes.
- 3. Réglage de la couleur des heures : incrémentation de 0 à 7 correspondant à un index sur une palette de 8 de couleurs prédéfinies.
- 4. Réglage de la couleur des minutes : idem avec la même palette.
- 5. Réglage de l'intensité maximale des LED : affichage d'un arc-en-ciel sur les 12 LED à l'intensité maximale courante, puis incrémentation jusqu'à 248 par multiple de 8 (on boucle ensuite sur 8).
- Retour au mode 0.

Il n'est pas possible de quitter les modes de configuration une fois en 1 et il est donc nécessaire de suivre le processus jusqu'à revenir au mode 0 d'affichage. Il n'est pas prévu non plus que les préférences de l'utilisateur soient enregistrées et survivent à une coupure d'alimentation. L'heure est enregistrée dans la RTC qui dispose d'une pile bouton type CR2032, c'est suffisant (pour l'instant).

La technique choisie pour implémenter cela consiste à utiliser les interruptions. Les modules capacitifs TTP223 utilisés, dans leur configuration par défaut, fonctionnent comme de simples boutons temporaires, passant leur unique sortie à l'état haut lorsqu'un contact est détecté et le laissant à bas dans le cas contraire. En plus de cette sortie, les TTP223 ne nécessitent qu'une alimentation et une masse et évitent de devoir gérer le phénomène de rebond, tel qu'on en rencontre avec des boutons mécaniques.

Sur une carte Arduino Nano, équipée d'un ATmega328, seules deux entrées sont capables de générer une interruption lors d'un changement d'état. Ce sont les broches 2 et 3. Les interruptions sont gérées ainsi :

```
#define BPLUS             2
#define BCONF             3

[...]
volatile int confmode = 0;
volatile int val = 0;
[...]

void isr_conf()
{
    confmode++;
    if(confmode > 5)
        confmode =0;
}
void isr_plus()
{
    val++;
}

void setup()
{
    pinMode(BCONF, INPUT);
    pinMode(BPLUS, INPUT);
[...]
    attachInterrupt(
      digitalPinToInterrupt(BCONF),
      isr_conf, RISING);
    attachInterrupt(
      digitalPinToInterrupt(BPLUS),
      isr_plus, RISING);
}
```

Comme il se doit, les routines d'interruption `isr_conf()` et `isr_plus()` sont réduites à leur plus simple expression et ne font qu'incrémenter `confmode` et `val`, correspondant respectivement aux modes de configuration (1 à 5) et à la valeur incrémentée pour chaque mode. Ainsi, dans la fonction `loop()` nous appellerons à intervalles réguliers `configure()` qui sera chargé de tester et de dispatcher en fonction de `confmode` :

```
void configure()
{
  while(confmode) {
    switch(confmode){
      case 1: // mode heures
        conf_heure();
        break;
      case 2: // mode minutes
        conf_minute();
        break;
```

```
      case 3: // mode couleur heure
        conf_coulh();
        break;
      case 4: // mode couleur minute
        conf_coulm();
        break;
      case 5: // mode maxbright
        conf_bright();
    }
  }
  return;
}
```

Chaque fonction appelée sera chargée de gérer son propre mode et ne retournera que si `confmode` n'est plus égal à la valeur lui correspondant. Voici la fonction du mode de réglage de l'heure, par exemple :

```
void conf_heure()
{
  Serial.println("> mode heure");

  // initialisation
  int pval=val=cf_heure;

  // max luminosité
  FastLED.setBrightness(cf_maxbright);
  fill_solid(leds, NUM_LEDS, CRGB(0,0,0));
  for(int i=0; i<val; i++) {
    leds[i] = cf_coulh;
  }
  FastLED.show();

  // choix
  while(confmode==1) {
    // max ?
    if(val > 12) val=1;
    if(val != pval) {
      // présentation de l'heure
      fill_solid(leds, NUM_LEDS, CRGB(0,0,0));
      for(int i=0; i<val; i++) {
        leds[i] = cf_coulh;
      }
      FastLED.show();
      pval=val;

      // lecture RTC
      now = Rtc.GetDateTime();
      // M.A.J.
      RtcDateTime newtime = RtcDateTime(
        now.Year(), now.Month(),
```

```
            now.Day(), val,
            now.Minute(), now.Second());
          // enregistrement RTC
          Rtc.SetDateTime(newtime);
        }
      }
      return;
}
```

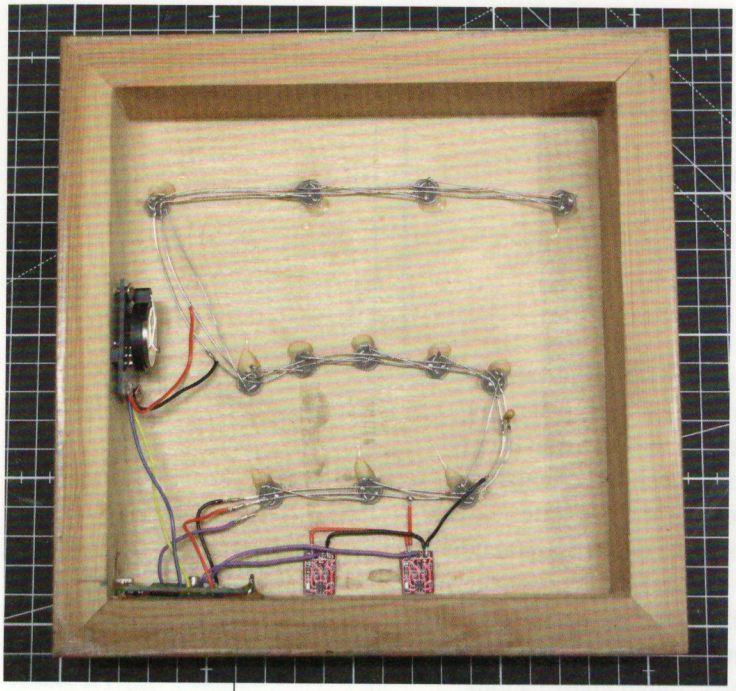
`cf_heure` est une variable globale mise à jour dans `loop()` servant également à l'affichage. Celle-ci sert à initialiser `val` et `pval` afin de présenter la valeur actuelle sur les LED et à suivre un éventuel changement par incrémentation de `val` via la routine d'interruption. Lorsque `val` est différent de `pval`, c'est que l'utilisateur a incrémenté en utilisant le bouton « BPLUS » et nous répercutons ce changement sur les LED, mais également dans la RTC. D'autres approches seraient possibles ici, comme enregistrer l'heure dans la RTC en quittant le mode (et donc la boucle `while()`), mais cela ne présente pas grand intérêt et l'approche actuelle mettra à jour l'horloge, même si l'alimentation est coupée alors qu'on est dans ce mode.

5. SAVOIR S'ARRÊTER QUAND LE MIEUX DEVIENT L'ENNEMI DU BIEN

Ces quelques éléments de code sont perfectibles, le code l'est toujours, telle est sa nature. Cependant, il est important de garder à l'esprit qu'à force d'ajustements et de modifications, il est facile de se retrouver dans une situation où l'on perd de vue l'objectif premier consistant à finir le projet dans les temps et avec le niveau d'utilisabilité suffisant qu'on s'était fixé.

La gestion des boutons, couplée au rafraîchissement de l'affichage et à la gestion des délais est une problématique pénible, mais également récurrente pour de nombreuses applications. Ici, mon objectif était de réaliser quelque chose rapidement et avec un minimum de contraintes. Le fait, par exemple, d'introduire un petit délai entre le passage du

Voici la réalisation finale avec un simple motif pyrogravé à l'aide d'un fer à souder équipé d'une vieille panne en fin de vie. Les LED APA106 s'intègrent relativement bien dans ce mélange très « nature » qui semble être une simple décoration, mais donne l'heure à qui saura observer attentivement.



Comme souvent, l'envers du décor est un peu moins attrayant, avec en bas à gauche la carte Arduino, à gauche la RTC, en bas les deux TTP223 et au centre les 12 LED sécurisées avec de la résine UV.

mode d'affichage au mode de configuration n'était pas un problème grave, après tout ce n'est pas un produit fini, mais une simple décoration.

Gérer l'ensemble de façon optimale tient en peu de choses : il suffit de supprimer toute occurrence d'un appel à la fonction `delay()`. Plus facile à dire qu'à faire cependant, car cela reviendrait à revoir entièrement le code sous la forme d'une unique boucle gérant l'ensemble des modes et des temporisations en utilisant `millis()`. Ceci permettrait d'interrompre sans problème, et instantanément, une pause ou un *fading* pour changer de mode.

Mais cette éternelle perfectibilité est tout aussi valable concernant les fonctionnalités, car une fois l'ensemble prenant forme, les idées ne manquent pas :

- créer une animation comme un scintillement en faisant passer rapidement les LED de la couleur des heures et minutes au blanc de façon aléatoire ;
- ajouter un mode de configuration pour choisir les temporisations et en particulier le temps d'affichage fixe de l'heure et des minutes ;
- afficher également la date, jour puis mois, avec deux autres couleurs (et intégrer cela à la configuration, bien entendu) ;
- afficher plus précisément les informations horaires en présentant les minutes écoulées après 5, 10, 15, 20, etc. ;
- offrir un mode d'affichage « *nerd* », en binaire avec 5 bits pour l'heure et 6 bits pour les minutes, laissant une LED comme séparateur à faire pulser ;
- compléter le tout avec une notification sonore et pourquoi pas une fonction réveil (avec toute la configuration qui s'y rattache) ;
- basculer de l'ATmega328 à quelque chose de plus performant, comme un ESP8266 ou ESP32 pour une connectivité Wi-Fi et une configuration via une application mobile ou une page web embarquée ;
- etc.

En vérité, chaque projet est un projet sans fin. Ce n'est que le contexte qui détermine à quel moment il devient juste d'estimer l'objectif comme atteint et de passer à autre chose... **DB**

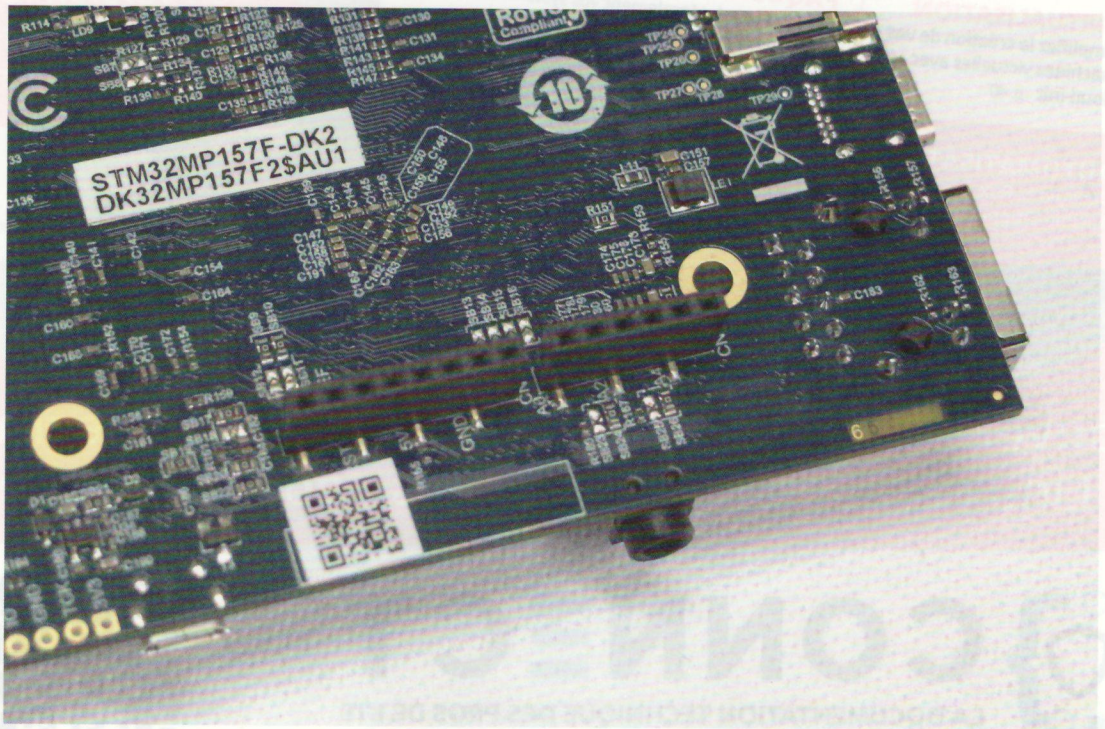
RÉFÉRENCE

[1] **https://connect.ed-diamond.com/contenu-premium/raspberry-pi-pico-pio-dma-et-memoire-flash**

UTILISER VOTRE DEVKIT STM32MP157 AVEC BUILDROOT

Denis Bodor

Dans le précédent article, nous avons vu comment prendre en main le STM32MP157F-DK2, mettre à jour le système, utiliser le SDK et reconstruire l'ensemble avec OpenEmbedded sur la base d'OpenSTLinux, mais aussi de Poky. Le système de build du projet Yocto n'est cependant pas le seul utilisable et, grâce au travail de Bootlin en partenariat avec ST, il est également possible d'utiliser Buildroot. Voyons cela ensemble...



La philosophie de Buildroot est sensiblement différente de celle d'OpenEmbedded. Il n'y a pas ici de *layers* (couches) s'ajoutant les uns aux autres sur une base minimale pour composer une distribution qu'on enrichira, ensuite, avec un ou des *layers* personnalisés. Buildroot est un ensemble complet destiné à produire un système en fonction des éléments que le développeur activera ou non, via une interface similaire à celle utilisée pour configurer le noyau Linux (outil kconfig). L'objectif reste identique, que ce soit avec OpenEmbedded ou Buildroot, nous voulons un noyau, une chaîne de compilation, un *bootloader* et un système de fichiers racine, mais la façon de construire cet ensemble sera radicalement différente. Autre différence importante, il n'y a pas de notion de paquet avec Buildroot, l'objectif étant de produire un système complet en une fois.

Mais ce qui distingue réellement les deux systèmes de *build* sera avant tout leur complexité relative en termes de configuration. OpenEmbedded est très modulaire, ce qui implique que la configuration se trouve dispersée dans une collection de fichiers, dont les interactions et les dépendances peuvent être difficiles à appréhender. Buildroot, en revanche, centralise la configuration en un seul fichier et un seul projet, rendant l'ensemble plus simple à embrasser globalement. Comme nous le verrons plus loin dans l'article, ceci n'empêche pas l'intégration d'éléments externes maintenus en parallèle (`BR2_EXTERNAL`), mais ce n'est pas la base de l'architecture, contrairement à ce que propose le projet Yocto.

La préférence envers l'un ou l'autre système est une affaire de goût personnel, mais aussi, et surtout, de disponibilité de prise en charge de la cible (SoC ou *devkit*) choisie. Fort heureusement, dans le cas du STM32MP157, les deux environnements de construction sont parfaitement supportés, ce qui en fait une plateforme idéale d'un point de vue pédagogique. En effet, quel que soit votre sentiment sur ces systèmes de *build*, il est important de ne pas privilégier l'un au détriment de l'autre, car les deux sont activement maintenus et utilisés. Il faut donc savoir utiliser à la fois OpenEmbedded et Buildroot.

Dans le précédent article **[1]**, nous avons fait connaissance avec la distribution OpenSTLinux et ce choix n'était pas totalement innocent. Non seulement il s'agit de la distribution officielle initialement supportée, mais sa relation avec OpenEmbedded implique davantage d'efforts de compréhension. Face à cela, un des objectifs de Buildroot est de rester concis et simple, le rendant facile à comprendre, en particulier pour un utilisateur GNU/Linux connaissant son sujet. Par rapport à Yocto, c'en est presque un soulagement...

1. CONSTRUIRE UN SYSTÈME AVEC BUILDROOT

Contrairement à OpenEmbedded, nous n'avons ici qu'une seule source pour récupérer Buildroot, ou presque. En effet, à l'heure où est composé cet article, les deux *devkits* STM32MP157 ne sont encore que partiellement supportés par le projet officiel (mais l'intégration est en cours). Il est donc nécessaire de compléter l'environnement avec une branche externe (`BR2_EXTERNAL`) ajoutant des éléments de configuration qui ne font pas partie de l'arborescence officielle. Notez que même si ce mécanisme ressemble aux *layers* d'OpenEmbedded, avec Buildroot il n'est possible de

l'utiliser qu'une fois et ceci uniquement pour ajouter des éléments et non écraser ceux qui sont existants. C'est une solution pour ajouter le support d'une plateforme, mais de préférence comme phase préliminaire d'intégration au projet officiel.

Étant donné la nature « temporaire » du support des cartes DK, il sera nécessaire d'utiliser un *fork* Bootlin de Buildroot **[2]** en version LTS (2021.02.*) en compagnie de l'arborescence externe, mais tout ceci est susceptible d'évoluer dans le futur. Quoi qu'il en soit, Buildroot étant très facile à « comprendre », adapter ces explications par la suite ne devrait pas être un problème.

Nous commençons donc par récupérer l'arborescence « officielle » depuis GitHub, ainsi que l'arborescence externe :

```
$ cd quelquepart
$ git clone -b st/2021.02 \
https://github.com/bootlin/buildroot.git

$ git clone -b st/2021.02 \
https://github.com/bootlin/buildroot-external-st.git
```

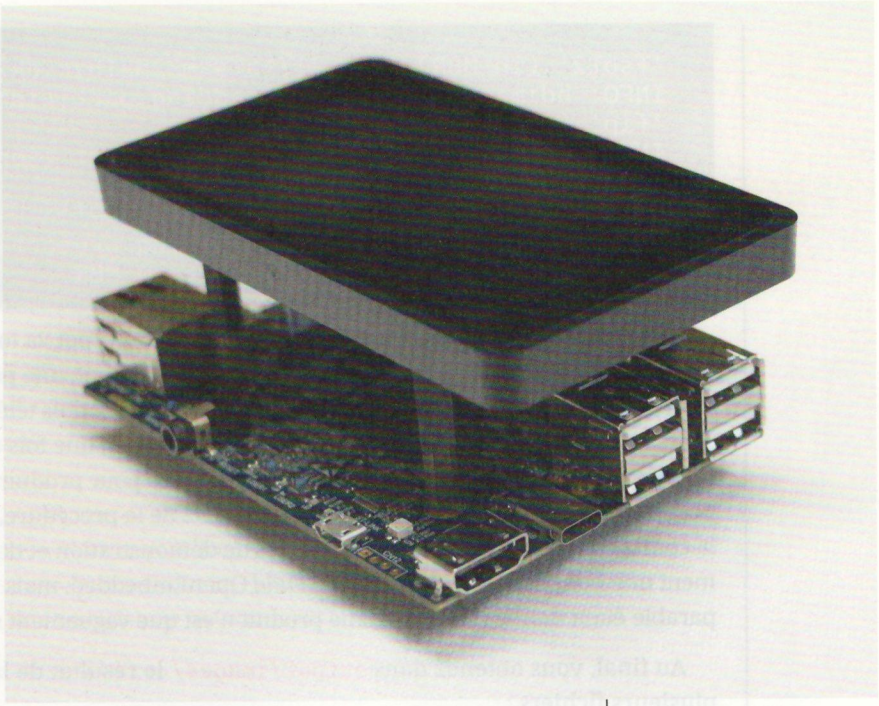
Notez l'utilisation de branches Git identiques pour les deux dépôts afin de disposer de configurations parfaitement synchronisées. Nous avons maintenant deux répertoires, `buildroot` pour l'environnement de construction standard et `buildroot-external-st` constituant l'arborescence externe. La prochaine étape consiste donc à lier les deux :

```
$ cd buildroot
$ make BR2_EXTERNAL=../buildroot-external-st \
st_stm32mp157c_dk2_demo_defconfig
[...]
# configuration written to /mnt/SSD2T/BR/buildroot/.config
#
```

Nous utilisons ici l'une des configurations par défaut (`defconfigs`) fournies par l'arborescence externe, destinée au SoC STM32MP157CACx et incluant un certain nombre de codes de démonstration. La variable d'environnement `BR2_EXTERNAL` nous permet de spécifier l'emplacement de l'arborescence externe durant cette phase de configuration, mais il ne sera pas nécessaire de l'utiliser par la suite, ceci est inclus dans la configuration locale (`BR2_EXTERNAL_ST_PATH` dans le `.config`). Vous pouvez lister toutes les configurations par défaut, de la branche principale et de l'arborescence externe avec un simple `make list-defconfigs`. Remarquez que nous utilisons une carte STM32MP157**F**-DK2, équipée d'un SoC STM32MP157FACx, sensiblement différent du STM32MP157CACx équipant le STM32MP157**C**-DK2 mais majoritairement compatible (ce qui expliquera deux messages d'erreur concernant l'OPP, pour *Operating Performance Points*, au moment du *boot*, le *devicetree* pour le STM32MP157CACxdu ne spécifiant pas de table OPP).

Nous avons ici un mécanisme assez similaire à une configuration de noyau Linux, puisque nous appliquons une configuration par défaut qui sera copiée dans un `.config` local. Nous pourrons ensuite modifier ce fichier à l'aide d'un `make menuconfig` pour éventuellement ajuster les options. Ceci est généralement une bonne idée, en particulier sur une machine où vous travaillez en parallèle. Dans le menu présenté avec un `make menuconfig`, visitez « *Build options* » et vous trouverez l'entrée « *Number of jobs to run simultaneously* » ajustant la valeur

de BR2_JLEVEL, correspondant à l'option -j de make. La configuration par défaut règle cette valeur automatique sur le nombre de CPU (ou cœurs/*threads*) plus 1. En réduisant celle-ci, à la moitié par exemple, vous permettez à votre système de rester réactif pendant la construction. Vous pouvez également activer « *Enable compiler cache* » (BR2_CCACHE) permettant d'utiliser ccache afin d'accélérer grandement les compilations consécutives (le cache sera créé dans ~/.buildroot-ccache/ par défaut, mais peut être ajusté également). Enfin, l'entrée « *gcc optimization level* » correspond à l'option -O de GCC et donc au niveau d'optimisation du compilateur. Le réglage par défaut sur -Os (BR2_OPTIMIZE_S) optimise pour la taille, mais vous pouvez vouloir optimiser pour les performances (-O3 / BR2_OPTIMIZE_3) ou ne pas optimiser du tout (-O0 / BR2_OPTIMIZE_0).



Le STM32MP157F-DK2 est la déclinaison la plus complète permettant d'évaluer la plateforme STM32MP157 puisqu'en plus de proposer un touch screen, le Wi-Fi et le Bluetooth, c'est également celle vous permettant d'expérimenter les fonctionnalités Trust Zone OP-TEE, contrairement au devkit STM32MP157D-DK1.

> Dans la suite de cet article, je préciserai systématiquement, entre parenthèses, le nom de l'élément de configuration tel qu'il apparaît dans le fichier de configuration (.config). L'interface accessible via make menuconfig, comme celle du noyau Linux permet de procéder à une recherche sur ces éléments via le raccourci /, vous présentant alors le ou les résultats, accompagnés d'un descriptif, des dépendances liées et aussi de l'emplacement de l'option dans l'arborescence de menus pour y accéder facilement.

Une fois satisfait de vos réglages, quittez l'interface de configuration et lancez la construction d'un simple make :

```
$ make
[...]
INFO: hdimage(sdcard.img): adding partition
'fsbl1' (in MBR) from 'tf-a-stm32mp157c-dk2-mx.stm32' ...
INFO: hdimage(sdcard.img): adding partition
```

```
'fsbl2' (in MBR) from 'tf-a-stm32mp157c-dk2-mx.stm32' ...
INFO: hdimage(sdcard.img): adding partition
'fip' (in MBR) from 'fip.bin' ...
INFO: hdimage(sdcard.img): adding partition
'rootfs' (in MBR) from 'rootfs.ext4' ...
INFO: hdimage(sdcard.img): writing GPT
INFO: hdimage(sdcard.img): writing protective MBR
INFO: hdimage(sdcard.img): writing MBR
```

Comme avec d'autres systèmes de construction, Buildroot va tout d'abord produire une chaîne de compilation adaptée à la cible (deux dans le cas présent, une pour le Cortex-A7 et la seconde pour le M4 afin de compiler les codes de démonstration), puis télécharger les sources des composants du système pour les extraire et les compiler. Enfin, une fois cet ensemble d'étapes accomplies, le ou les systèmes de fichiers seront assemblés pour produire une image qu'il vous sera possible de flasher sur la plateforme. L'ensemble de la procédure, sur mon bi-Xeon E5520, avec la configuration par défaut (avec les codes de démonstration et donc Qt) et 10 *jobs* prend initialement une bonne heure. Moins que le *build* OpenEmbedded, mais ceci n'est pas réellement comparable étant donné que le système produit n'est que vaguement similaire.

Au final, vous obtenez dans `output/images/` le résultat de la construction sous la forme de plusieurs fichiers :

- `fip.bin` : le FIP ou *Firmware Image Package* regroupant de façon structurée et binaire les *bootloaders*, un *devicetree* et un certificat pour le *boot* TF-A (*Trusted Firmware-A* **[3]**).
- `rootfs.ext2` : une image du système de fichiers racine au format EXT4 (l'extension `ext2` étant présente pour des raisons de compatibilité).
- `rootfs.ext4` : un lien symbolique vers `rootfs.ext2` avec une extension plus adaptée.
- `sdcard.img` : une image du support amovible contenant plusieurs partitions GPT (FSBL1, FSBL2, FIP et système de fichiers racine).
- `stm32mp157c-dk2.dtb` : un *devicetree* binaire utilisé par le noyau Linux pour la prise en charge des périphériques non détectables.
- `tf-a-stm32mp157c-dk2-mx.stm32` : un *bootloader* de premier niveau (ou FSBL pour *First Stage BootLoader*) utilisé comme FSBL1 et FSBL2.
- `tee.bin`, `tee-header_v2.bin`, `tee-pageable_v2.bin` et `tee-pager_v2.bin` : les éléments composant l'environnement d'exécution OP-TEE, la partie sécurisée du système Trust Zone.
- `u-boot.dtb` : le *devicetree* binaire utilisé par le bootloader U-Boot.
- `u-boot-nodtb.bin` : le binaire U-Boot lui-même.
- `zImage` : l'image du noyau Linux.

Comme avec OpenSTLinux, ces éléments peuvent être utilisés pour écrire la microSD ou être utilisés avec STM32CubeProgammer. Pour ce faire, la branche externe développée par Bootlin intègre un fichier TSV que vous trouverez dans `buildroot-external-st/board/stmicroelectronics/stm32mp157` sous le nom `flash.tsv`. Pour mettre à jour votre *devkit*, vous pouvez placer les micro-interrupteurs BOOT0 et BOOT2 sur OFF, connecter la carte en USB-C, l'alimenter puis utiliser :

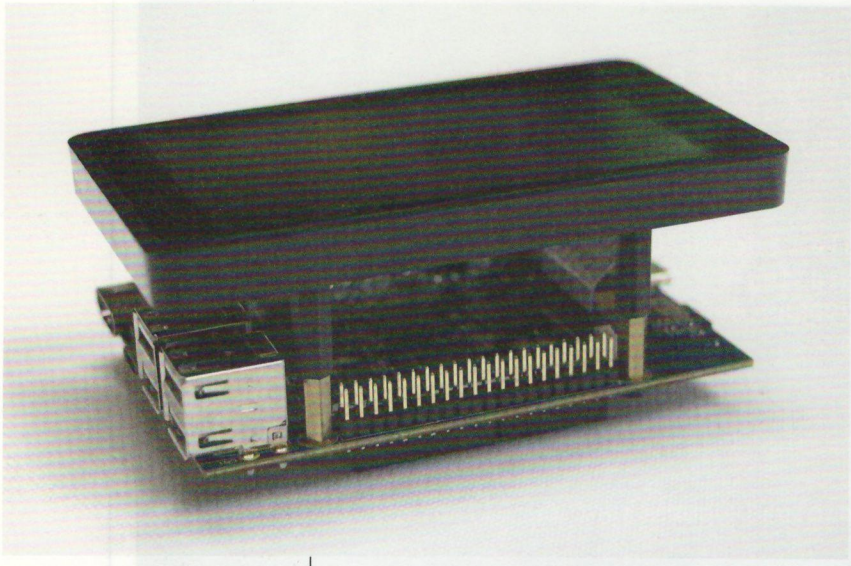
```
$ cd output/images
$ STM32CubeProgrammer/bin/STM32_Programmer_CLI \
-c port=usb1 -w ../../../buildroot-external-st/\
board/stmicroelectronics/stm32mp157/flash.tsv
[...]
Memory Programming ...
Opening and parsing file: sdcard.img
  File          : sdcard.img
  Size          : 212290048 Bytes
  Partition ID  : 0x10

Download in Progress:
[==================================] 100%
File download complete
Time elapsed during download operation: 00:01:05.783

RUNNING Program ...
  PartID:       :0x10
Start operation done successfully at partition 0x10
Flashing service completed successfully
```

Une fois l'opération terminée, repositionnez les micro-interrupteurs sur ON et procédez à un *reset* via le bouton dédié. En utilisant la console série mise à votre disposition via le connecteur ST-LINK/V2-1 (ttyACM et 115200 8N1), vous devrez alors constater le démarrage du système et arriver à l'invite de connexion. Vous pouvez alors vous identifier comme `root` et obtenir une invite de shell sans saisir de mot de passe. Bravo, vous avez construit, flashé et démarré votre premier système GNU/Linux sur STM32MP157F-DK2 avec Buildroot.

2. PROCÉDONS À QUELQUES AJUSTEMENTS

Si vous faites le tour du propriétaire, vous vous rendrez rapidement compte que tout ceci est assez spartiate. Vous n'avez, par exemple, ni console sur l'écran LCD ni services intéressants (SSH, Avahi, etc.) ou même de connectivité réseau. Vous trouverez cependant quelques éléments intéressants découlant de la configuration utilisée par défaut, comme le contenu de `/usr/lib/qt/examples` réunissant quelques exemples amusants de ce qu'il est possible de faire avec Qt, ou encore dans `/usr/lib/Cube-M4-examples`, un jeu de *firmwares* à destination du coprocesseur Cortex-M4 intégré au STM32MP157.

Même si la plateforme n'est en rien destinée à être une sorte de mini-ordinateur générique, un minimum de confort est nécessaire, ne serait-ce que pour sereinement développer sa ou ses applications pour ensuite les faire s'exécuter sur la cible. Notre objectif est de changer la configuration de Buildroot pour disposer d'un système toujours aussi *light*, mais capable de supporter un minimum de manipulations (oui, c'est une excuse pour découvrir les spécificités de Buildroot).



Les devkits STM32MP157 proposent un connecteur 40 broches similaire à celui d'une Raspberry Pi avec une disposition majoritairement compatible (3v3, GND, port série, i2c, etc., aux mêmes emplacements).

Une partie des modifications que nous souhaitons apporter impliquent l'utilisation de fichiers qui vont venir s'ajouter en complément de ceux déjà utilisés ou remplacer ceux existants. Il nous faut donc un emplacement pour stocker ces fichiers. Un parfait exemple concerne l'ajout d'un utilisateur standard, qui passe généralement par la création d'un fichier (la *users table*) stockant les informations utilisées par Buildroot. L'emplacement recommandé pour un tel fichier, selon la documentation officielle, est `board/<company>/<boardname>/` et donc, dans le `board/stmicroelectronics/stm32mp157` de `buildroot-external-st`. Nous ne voulons cependant pas modifier notre copie du dépôt de Bootlin.

Nous pourrions *forker* ce dépôt sur GitHub, mais dans ce cas, il ne nous sera pas possible d'en réduire la visibilité pour le rendre privé. La solution consiste donc à créer un dépôt vide quelque part (GitHub ou ailleurs) avec `git init --bare` et de tout simplement ajouter le dépôt distant avec `git remote add`, puis de faire un `git push -u` suivi du nouveau nom. Nous pouvons même basculer sur la nouvelle branche avec `git checkout -b` et donc suivre nos modifications avec Git, sans pour autant risquer de nous mélanger les pinceaux.

2.1 Réseau, SSH, etc.

La première chose qu'on peut souhaiter activer est le support réseau ou plus exactement sa configuration. En effet, les interfaces sont bien présentes, comme en témoigne la sortie d'un simple `ifconfig -a` ou `ip addr`. Nous retrouvons là l'interface *loopback* (`lo`), Ethernet (`eth0`), Wi-Fi (`wlan0`) et l'interface virtuelle IPv6/IPv4 (`sit0`). Ce qui nous manque en revanche, c'est la configuration de l'interface `eth0`, comme en témoigne le contenu du fichier `/etc/network/interfaces` :

```
# interface file auto-generated by buildroot

auto lo
iface lo inet loopback
```

Le client DHCP est également déjà présent puisqu'il s'agit de celui fourni par BusyBox **[4]**, `/sbin/udhcpc`. Nous pouvons d'ailleurs nous assurer du bon fonctionnement de l'ensemble en l'invoquant manuellement :

```
# udhcpc -i eth0
udhcpc: started, v1.33.0
udhcpc: sending discover
udhcpc: sending discover
udhcpc: sending discover
udhcpc: sending select for 192.168.0.92
udhcpc: lease of 192.168.0.92 obtained, lease time 600
deleting routers
adding dns 81.253.149.13
adding dns 80.10.246.5

# ping connect.ed-diamond.com
PING connect.ed-diamond.com (185.169.94.231): 56 data bytes
64 bytes from 185.169.94.231: seq=0 ttl=45 time=22.206 ms
64 bytes from 185.169.94.231: seq=1 ttl=45 time=22.132 ms
64 bytes from 185.169.94.231: seq=2 ttl=45 time=22.487 ms
^C
--- connect.ed-diamond.com ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 22.132/22.275/22.487 ms
```

Pour modifier la configuration en place, nous pouvons utiliser la notion d'*overlay* de Buildroot. Cette approche nous permet de modifier le système de fichiers racine avant la création de l'image et un certain nombre de choses sont d'ores et déjà ajustées de cette façon. Un coup d'œil au répertoire `buildroot-external-st/board/stmicroelectronics/stm32mp157/dk2-overlay` nous montre, entre autres choses, une configuration ALSA via `etc/asound.conf`. Pour ajouter le support DHCP, il nous suffit donc de créer un fichier `interfaces` dans un sous-répertoire `network/` de `etc/` contenant :

```
auto lo eth0

iface lo inet loopback

iface eth0 inet dhcp
```

Nous pouvons également choisir de ne pas changer l'arborescence existante et dupliquer `dk2-overlay` sous un autre nom avant d'y faire notre ajout. Il suffira alors de faire la modification, via `make menuconfig`, « *System configuration* » puis « *Root filesystem overlay directories* » (`BR2_ROOTFS_OVERLAY`) pour adapter le chemin correspondant (relatif à `BR2_EXTERNAL_ST_PATH`). Ceci d'autant que, puisque nous sommes dans l'interface, nous pouvons également en profiter pour :

- changer le nom d'hôte de « buildroot » en quelque chose de plus reconnaissable (« stm32mp1br » par exemple) : « *System configuration* », « *System hostname* » (`BR2_TARGET_GENERIC_HOSTNAME`) ;

- activer le serveur SSH : « *Target packages* », « *Networking applications* », « *dropbear* » (`BR2_PACKAGE_DROPBEAR`) ;
- activer Avahi pour la résolution mDNS : « *Target packages* », « *Networking applications* », « *avahi* » (`BR2_PACKAGE_AVAHI` et `BR2_PACKAGE_AVAHI_DAEMON`).

En quittant l'interface, sauvegardez les changements (dans `.config`), puis relancer la construction avec un simple `make`. De nouvelles archives sources seront automatiquement téléchargées, désarchivées, compilées et intégrées au système de fichiers racine pour produire une nouvelle image que vous pourrez immédiatement flasher, comme précédemment. Toutes les dépendances liées aux deux ajouts auront bien entendu été traitées dans le même temps.

En surveillant sur la console série ce premier redémarrage, on pourra constater le lancement du serveur SSH, ainsi que la résolution DHCP et l'exécution du démon `avahi-daemon`. La résolution fonctionnera d'ailleurs sans problème en utilisant le nom d'hôte défini depuis une autre machine du réseau :

```
$ ping stm32mp1br.local
PING stm32mp1br.local (192.168.0.92) 56(84) bytes of data.
64 bytes from 192.168.0.92 (192.168.0.92): icmp_seq=1 ttl=64 time=0.870 ms
64 bytes from 192.168.0.92 (192.168.0.92): icmp_seq=2 ttl=64 time=0.464 ms
64 bytes from 192.168.0.92 (192.168.0.92): icmp_seq=3 ttl=64 time=0.445 ms
^C
--- stm32mp1br.local ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2025ms
rtt min/avg/max/mdev = 0.445/0.593/0.870/0.196 ms
```

OpenSSH nous pause par contre un problème sensiblement différent :

```
$ ssh root@stm32mp1br.local
Host key fingerprint is SHA256:
Xwgn9vIHvhB7Fkrwb7YaOoqh0Bt7eezDZNowVxN6lo8
root@stm32mp1br.local's password:
```

En effet, non seulement notre `root` n'a pas de mot de passe, mais en plus, il n'est pas question de procéder à une connexion SSH sous cette identité. La configuration par défaut du serveur SSH (que ce soit Dropbear ou OpenSSH) ne le permet pas et c'est très bien ainsi. Pour nous connecter depuis une autre machine, nous devons disposer d'un autre utilisateur.

2.2 Le root de tous les maux

N'avoir que l'utilisateur `root` et/ou l'utiliser pour des tâches courantes, même de développement et de mise au point, est une mauvaise idée. Mieux vaut disposer d'un utilisateur standard qui, le moment venu, obtiendra temporairement les privilèges adéquats pour des actions spécifiques. Nous devons donc intégrer la création de nouveaux utilisateurs dans notre configuration Buildroot. Pour cela, nous avons à notre disposition un mécanisme adapté passant par la création d'un fichier faisant office de *users table*.

Ce fichier sera référencé via l'interface accessible obtenue avec `make menuconfig`, sous « *System configuration* » puis « *Path to the users tables* » (`BR2_ROOTFS_USERS_TABLES`). Nous le placerons dans l'arborescence externe, en compagnie des *overlays* et le nommerons `mkusers.table`. Il sera donc désigné par `$(BR2_EXTERNAL_ST_PATH)/board/stmicroelectronics/stm32mp157/mkusers.table` dans la configuration.

Ce fichier respecte une syntaxe relativement simple, voici le nôtre (sur une ligne !) :

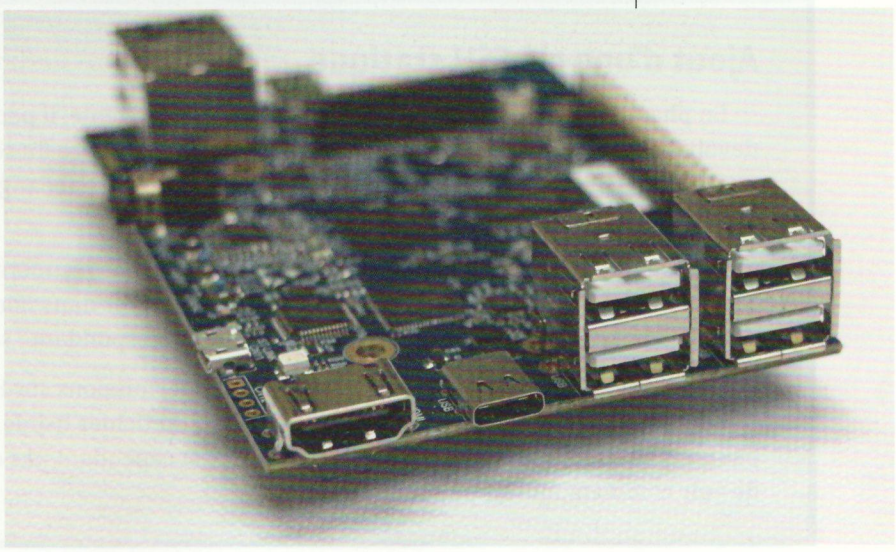
```
denis -1 denis -1 =coucou /home/denis /bin/sh
wheel,plugdev,dialout,sudo Utilisateur de base
```

Nous avons une liste de champs séparés par des espaces avec, de gauche à droite :

- le nom d'utilisateur (`denis`) ;
- son ID, avec `-1` pour laisser Buildroot le déterminer à la construction ;
- le nom du groupe de l'utilisateur (`denis`) ;
- l'ID du groupe (GID) également calculé par Buildroot (`-1`) ;
- le mot de passe en clair, précédé de `=`, ceci sera chiffré (`crypt`) avant d'être intégré au système. Notez qu'en faisant précéder le tout d'un `!`, nous pouvons interdire le *login* (pour un service fonctionnant sous une identité particulière, par exemple) ;
- le répertoire personnel de l'utilisateur (`$HOME`) ;
- le shell par défaut, ici `/bin/sh` (fourni par BusyBox) ;
- une liste de groupes supplémentaires séparés par des virgules ;
- et enfin, un commentaire qui sera intégré dans (`/etc/passwd`), ce champ étant le dernier, l'utilisation d'espaces est possible.

Nous avons ajouté ce nouvel utilisateur dans le groupe `sudo`, car la configuration par défaut de cette commande dans Buildroot ajoute automatiquement un `/etc/sudoers` incluant une ligne `%sudo ALL=(ALL) ALL`, signifiant que tous les utilisateurs du groupe `sudo` peuvent passer super-utilisateur pour n'importe quelle commande, en s'authentifiant. Ce réglage est fait par le script `package/sudo/sudo.mk`, et donc par Buildroot.

Tout ce que nous avons à faire est de nous plier d'un `make menuconfig` et d'activer « *Target packages* », « *Shell and utilities* » et « *sudo* » (`BR2_PACKAGE_SUDO`). Nous en profitons au passage pour



Le connecteur USB-C situé entre le hub USB et l'HDMI est un port USB OTG permettant au bootloader de fournir une interface DFU pour le flashage de la microSD. Ce port peut, bien entendu, être utilisé par Linux via l'API USB Gadget, pour émuler toutes sortes de périphériques USB.

faire de même, quelques lignes plus haut, avec le multiplexeur de terminal GNU Screen (BR2_PACKAGE_SCREEN) (ou Tmux (BR2_PACKAGE_TMUX) si vous préférez), qui peut également s'avérer très utile, que ce soit en SSH ou via la console série. Il peut être également intéressant d'activer « linux-pam » (BR2_PACKAGE_LINUX_PAM) pour *Pluggable Authentication Modules* permettant d'ajuster finement les paramètres d'authentification, de validation de compte et de session des utilisateurs.

À ce stade (peut-être après avoir préalablement testé les changements), nous pouvons terminer en faisant un tour dans « *System configuration* » puis désactiver « *Enable root login with password* » (BR2_TARGET_ENABLE_ROOT_LOGIN). Nous interdisons alors l'utilisation du compte `root`, ne laissant plus que `sudo` être là pour obtenir les privilèges en question :

```
Welcome to Buildroot
stm32mp1br login: root
Password:
Login incorrect

stm32mp1br login: denis
Password:

$ sudo -s
Password:

$ id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
```

2.3 Quelque chose à l'écran

Notez tout d'abord que ce qui va suivre n'est ni nécessaire, ni forcément souhaitable. L'objectif ici sera simplement de prendre en main le système de *build* en obtenant un résultat visuel amusant, tout en apprenant quelque chose. Raisonnablement, la configuration par défaut, n'affichant strictement rien à l'écran, si ce n'est en appelant les codes de démonstration Qt, est l'approche à adopter dans un projet destiné à présenter une IHM à l'utilisateur.

Cependant, comme nous aimons fouiller dans les coins et faire pleinement le tour du propriétaire, nous nous fixons comme objectif de présenter sur l'écran

Ajout d'une clé SSH statique

En phase de développement et de composition du système, il peut rapidement devenir pénible de devoir sans cesse supprimer et valider à nouveau la signature du serveur SSH du système embarqué. En effet, à chaque premier démarrage du système, Dropbear génère une nouvelle clé d'hôte qui sera forcément différente de la précédente.

Pour nous simplifier la vie, nous pouvons parfaitement récupérer cet élément (via `scp`) depuis `/etc/dropbear/dropbear_ecdsa_host_key` et l'intégrer dans notre *overlay*. Ainsi, la clé d'hôte sera déjà présente dans le système et l'empreinte correspondante sera donc toujours la même.

Ce faisant, vous remarquerez peut-être que les permissions sur le fichier seront sensiblement différentes, passant de 600 à 644, rendant le contenu du fichier lisible par tout le monde. Cela ne pose pas un problème particulier à Dropbear, mais n'est cependant pas correct, et nous donne l'occasion de voir comment ajuster ce genre de choses.

LCD une console telle que celle dont nous disposons sur la liaison série. Ceci va nécessiter une modification de la configuration du noyau Linux, sa recompilation et naturellement, la génération d'une nouvelle image.

L'ensemble nous est grandement simplifié par Buildroot, puisque les composants majeurs d'un système embarqué utilisent généralement le même système de configuration que Buildroot. Ainsi, nous connaissons déjà `make menuconfig`, mais avons également à notre disposition :

- `make busybox-menuconfig` pour BusyBox ;
- `make uclibc-menuconfig` pour la bibliothèque C standard uClibc ;
- `make uboot-menuconfig` pour le bootloader U-Boot ;
- `make barebox-menuconfig` pour Barebox, un *bootloader* alternatif à U-Boot ;
- et `make linux-menuconfig` pour le noyau Linux.

Ces différentes interfaces de configuration peuvent être utilisées, sous réserve que l'élément concerné soit activé dans la configuration principale, pour procéder à une configuration de ces composants. Il ne s'agit cependant pas d'une intégration complète à Buildroot, mais d'interfaces distinctes. Ainsi, `make linux-menuconfig` nous renvoie sur l'interface classique de configuration du noyau Linux, où nous pouvons ajuster des éléments sur la base d'un fichier de configuration préalablement chargé et ceci fait, nous enregistrons ces changements dans le même fichier ou sous un autre nom. Le fichier de configuration en question est ensuite spécifié dans la configuration Buildroot, exactement comme nous l'avons fait pour les tables des utilisateurs ou des permissions.

L'arborescence externe proposée par Bootlin intègre une configuration Linux stockée dans `board/stmicroelectronics/stm32mp157/linux.config`. Nous devons donc charger ce fichier via « < *Load* > » (au bas de l'écran) pour ensuite modifier cette configuration. Mais avant de faire cela, nous avons besoin de parler de la gestion de l'affichage avec Linux.

Les plus anciens utilisateurs du système se souviennent très certainement du périphérique *framebuffer*, alias *fbdev*, généralement accessible via un pseudofichier `/dev/fb0`. La notion de *framebuffer* est relativement simple, il s'agit d'une zone mémoire (*buffer*) représentant littéralement la valeur de chaque pixel à l'écran. Ceci explique pourquoi un `cat /dev/urandom > /dev/fb0` permet de remplir l'écran de pixels aux couleurs aléatoires, et il était donc très facile de dessiner à l'écran de cette manière en considérant le *buffer* comme un canevas. Facile, certes, mais totalement inefficace, car *fbdev* était très limité (*buffer* de taille statique préallouée, pas de pipeline, problème de synchronisation, etc.)

De la même manière que nous avons utilisé une table pour créer l'utilisateur supplémentaire, il en existe une pour fixer les permissions, c'est `system/device_table.txt` depuis la racine de notre Buildroot. Nous pouvons donc copier ce fichier au côté de `mkusers.table` (en le renommant éventuellement) et y ajouter une entrée pour notre fichier de clé. Là, nous ajoutons : `/etc/dropbear/dropbear_ecdsa_host_key f 600 0 0 - - - - -`, puis ajustons le chemin depuis l'interface `menuconfig`, « *System configuration* », « *Path to the permission tables* » (`BR2_ROOTFS_DEVICE_TABLE`), exactement comme nous l'avons fait pour `mkusers.table`.

L'espace entre l'écran LCD et la carte étant relativement réduit, l'accès aux GPIO n'est pas aisé en phase de développement. Il peut donc être intéressant de rehausser l'ensemble à l'aide d'entretoises pour gagner en souplesse.

et obligeait donc des serveurs d'affichage comme Xorg à accéder directement aux registres matériels pour reconfigurer l'affichage. Un autre système a donc été créé pour corriger ces imperfections et permettre au noyau de prendre en charge totalement l'infrastructure d'affichage incluant la gestion du *framebuffer*, mais également la configuration du matériel et de l'affichage.

Pour comprendre ce système, il est tout d'abord important de faire la distinction entre un contrôleur d'affichage (*display controller*) et un GPU (*Graphics Processing Unit*). Le travail du contrôleur d'affichage se limite à une tâche et une seule : copier le contenu d'un ou plusieurs *framebuffers* à l'écran, qu'il soit interfacé en HDMI, DisplayPort ou MIPI DSI. Il peut utiliser et combiner plusieurs *framebuffers* pour composer cet affichage, mais ne fait absolument aucun travail de *rendering* 3D ou autre opération impliquant des choses comme OpenGL.

Le GPU, en revanche, est chargé du *rendering*. C'est un processeur graphique programmable, généralement avec OpenGL, dont le travail est de décharger le processeur généraliste (CPU) des opérations graphiques. Mais le GPU, par lui-même, n'affiche rien et, au contraire, s'en remet au contrôleur d'affichage pour cela. Voilà pourquoi, il est parfaitement possible d'utiliser OpenGL et des *shaders* (microprogramme fonctionnant purement sur le GPU) sans afficher quoi que ce soit à l'écran (voir **[5]** pour un exemple). Dans le monde de l'embarqué, certains SoC disposent d'un GPU et d'autres non, mais si un écran ou une sortie vidéo est disponible, il y a toujours un contrôleur d'affichage. Le STM32MP157 dispose des deux.

Avec le noyau Linux moderne, ces deux éléments sont à présent contrôlés par un sous-système appelé DRM pour *Direct Rendering Manager*, parfois également appelé DRM KMS (pour *Kernel Mode-Setting*), avec KMS étant un sous-élément de l'API DRM, chargé de la configuration du mode (résolution, nombre de couleurs, vitesse de rafraîchissement) d'affichage. KMS règle précisément le problème évoqué plus haut, évitant un accès privilégié au matériel depuis l'espace utilisateur (pilotes X). D'autre part, l'architecture DRM permet un niveau abstraction supplémentaire puisque n'importe quel programme de l'espace utilisateur faisant usage de l'API DRM fonctionnera forcément avec n'importe quel pilote DRM, quel que soit le matériel contrôlé.

Aujourd'hui, l'ancien *fbdev* est considéré comme totalement obsolète et n'est presque plus utilisé, au bénéfice de DRM KMS. Le seul élément du noyau en faisant encore usage est... *fbcon*, la console permettant un affichage des messages

de démarrage et la gestion d'un terminal associé (TTY). Fort heureusement, DRM met à disposition une émulation *fbdev* et donc une solution pour disposer d'un `/dev/fb0`, mais aussi, et surtout, d'une console. Précisément ce que nous désirons obtenir.

Nous disposons déjà d'un support DRM pour le contrôleur d'affichage et le GPU (Vivante) intégré au SoC dans la configuration par défaut du noyau, comme le prouve la parfaite exécution des démonstrations QT (alors même que rien ne s'affiche au démarrage du *devkit*). Vous pouvez d'ailleurs très simplement tester cette fonctionnalité (via la console série ou l'accès SSH) en utilisant la commande `modetest`. `sudo modetest -M stm` permettra d'afficher des informations (connecteurs, encodeurs, modes disponibles, etc.) et de procéder à un test avec `sudo modetest -M stm -s` suivi d'un `33:1280×720` pour la sortie DVI, ou d'un `35:480×800` pour l'écran LCD (à ajuster en fonction de la sortie de `sudo modetest -M stm -c`).

Ce qui nous manque, en premier lieu, c'est une console pour afficher les messages du noyau et éventuellement nous fournir une interface textuelle pour le *login*. En d'autres termes, il nous manque *fbcon*. Tout ce que nous avons donc à faire est d'activer quelques fonctionnalités.

Commençons par l'émulation *fbdev* en passant par :

- « *Device Drivers* » ;
- « *Graphics support* » ;
- « *Direct Rendering Manager* » ;
- « *Enable legacy fbdev support for your modesetting driver* » (`CONFIG_DRM_FBDEV_EMULATION`).

Comme l'émulation est active, nous pouvons avoir accès à l'activation de *fbcon* :

- « *Device Drivers* » ;
- « *Graphics support* » ;
- « *Console display driver support* » ;
- « *Framebuffer Console support* » (`CONFIG_FRAMEBUFFER_CONSOLE`).

Et puisqu'un simple défilement de texte sur un écran serait bien triste sans pingouins (et pas assez nostalgique), nous pouvons également activer l'affichage du logo :

- « *Device Drivers* » ;
- « *Graphics support* » ;
- « *Bootup logo* » (`CONFIG_LOGO_LINUX_MONO`, `CONFIG_LOGO_LINUX_VGA16` et `CONFIG_LOGO_LINUX_CLUT224`).

C'est tout. Nous enregistrons les modifications avant de quitter l'interface de configuration et nous stockons cette configuration, au même emplacement que `linux.config`, mais en nommant le fichier `linux-fbcon.config`. Il nous suffit ensuite d'un petit `make menuconfig` pour ajuster cet élément via « *Kernel* » et « *Configuration file path* » (`BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE`).

Il faudra ensuite nettoyer ce composant du système et reconstruire le noyau avec :

```
$ make linux-dirclean
$ make linux-rebuild
```

Mais avant de reconstruire l'image et de la flasher sur la plateforme, nous avons un dernier point à configurer. Dans l'état, le noyau configurera effectivement *fbcon* et nous verrons les messages de *boot* s'afficher en compagnie des deux

petits pingouins, mais nous n'avons pas pour autant de terminal sur cette sortie. Pour cela, nous devons nous pencher sur la configuration de `getty`, fourni par BusyBox, ainsi que sur le système d'*init*, également pris en charge par cet outil.

Fort heureusement, la configuration par défaut proposée par Bootlin n'utilise pas systemd (contrairement au *build* Yocto/OpenSTLinux) et tout cela reste donc extrêmement simple, efficace et cohérent. Comme l'option « *Run a getty (login prompt) after boot* » (`BR2_TARGET_GENERIC_GETTY`) est activée, ceci signifie qu'un squelette de base (`package/busybox/inittab`) est utilisé, puis modifié par le système de *build* pour décommenter la ligne concernant la console série. Celle-ci est ajustée en fonction des paramètres choisis dans l'interface (port, vitesse, type de terminal).

Nous ne pouvons donc pas simplement et brutalement remplacer `inittab` avec l'*overlay*, mais Buildroot propose d'autres solutions. Dans `board/stmicroelectronics/stm32mp157/` vous trouverez, par exemple, le fichier `post-image.sh`, un script permettant de générer le fichier `genimage.cfg` utilisé pour produire l'image finale. Ce script est exécuté à un moment précis du *build*, juste après que Buildroot ait composé le système de fichiers racine. Vous retrouvez cette configuration dans « *System configuration* » et « *Custom scripts to run after creating filesystem images* » (`BR2_ROOTFS_POST_IMAGE_SCRIPT`).

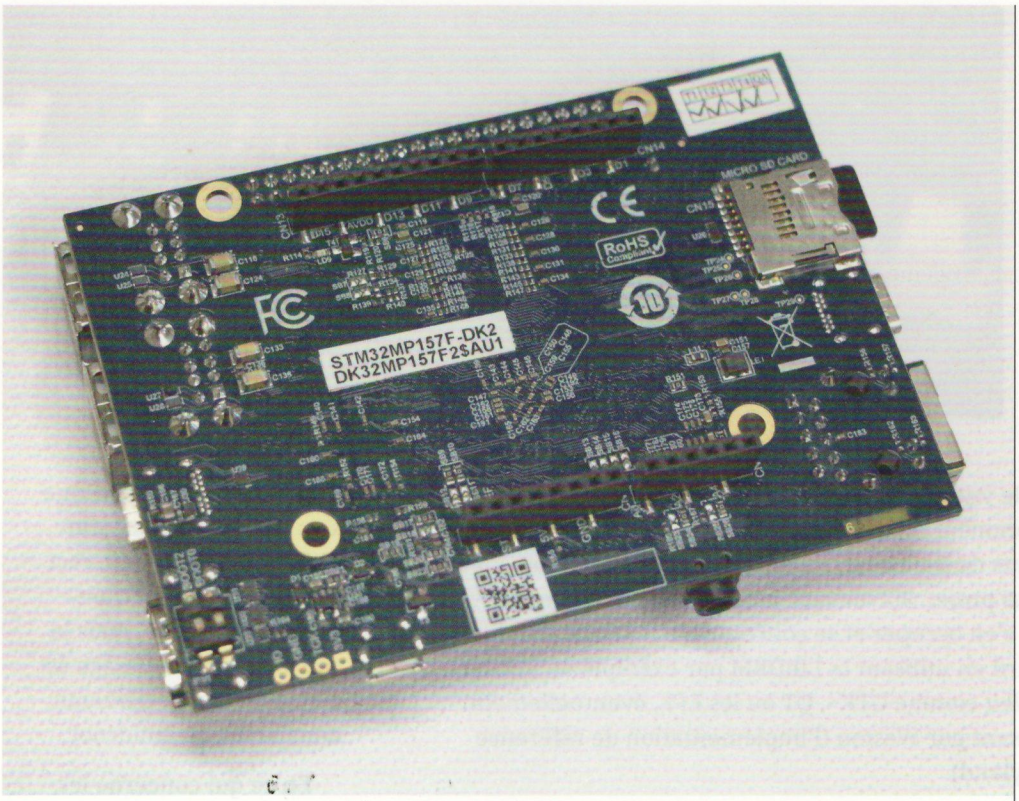
Ce *hook* n'est pas le seul utilisable, il en existe un autre, désigné par « *Custom scripts to run before creating filesystem images* » (`BR2_ROOTFS_POST_BUILD_SCRIPT`) exécutable juste après le *build* et avant la composition de l'image du système de fichiers. C'est précisément ce qu'il nous faut pour exécuter un script, que nous appelons `post-build.sh`, pour modifier à la voler le contenu du fichier `inittab` :

```
#!/bin/sh

set -u
set -e

# Add a console on tty1
if [ -e ${TARGET_DIR}/etc/inittab ]; then
    grep -qE '^tty1::' ${TARGET_DIR}/etc/inittab || \
        sed -i '/GENERIC_SERIAL/a\
tty1::respawn:/sbin/getty -L  tty1 0 vt100 # LCD console' ${TARGET_DIR}/
etc/inittab
fi
```

Ce script n'est pas de moi, il est utilisé dans plusieurs entrées dans `buildroot/board/`, `raspberrypi/` ou n'importe quelle autre carte proposant une sortie HDMI. Il se contente de trouver la ligne concernant la console série (via la chaîne `GENERIC_SERIAL` en commentaire) et d'en ajouter une, juste après, pour lancer un nouveau `getty`.

Nous plaçons ce script à côté de `post-image.sh` et pouvons alors lancer un nouveau *build*, puis flasher la carte. Celle-ci va alors présenter les messages de démarrage et se terminer sur :

```
Welcome to Buildroot
stm32mp1br login:
```

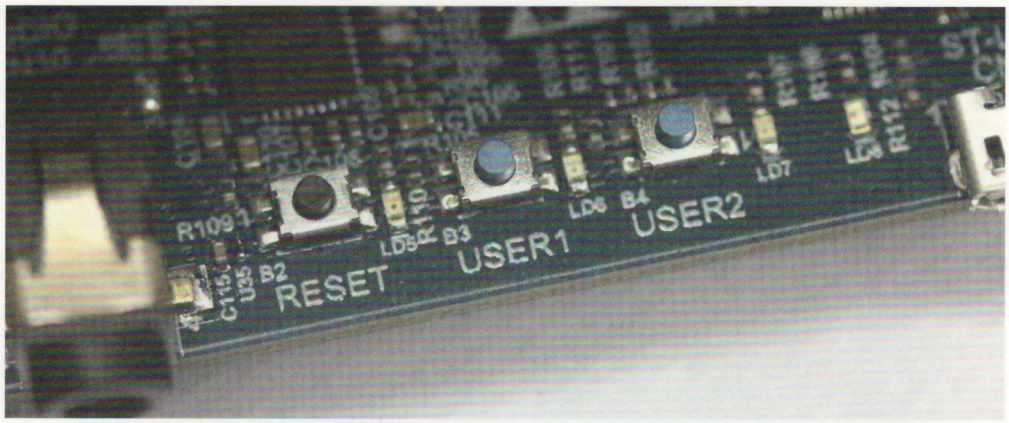


Le dessous du devkit propose des « connecteurs Arduino » permettant d'utiliser des shields compatibles, qu'il s'agisse des « expansion boards Nucleo » du constructeur, ou d'autres modèles.

Si un clavier USB est branché, nous pouvons alors entrer notre *login* et son mot de passe pour obtenir un shell. À ce stade, si vous êtes satisfait du résultat, vous pouvez choisir d'enregistrer votre configuration Buildroot courante (présentement dans `.config`), sous un nom plus convenable et à un emplacement adapté, comme `buildroot-external-st/configs/`. Vous pouvez également utiliser `make savedefconfig` qui produira un fichier plus concis en supprimant les redondances et les lignes indiquant les éléments non activés, mais mettra à jour le fichier initialement utilisé (`st_stm32mp157c_dk2_demo_defconfig`). Vous pouvez cependant forcer l'emplacement et le nom du fichier en ajoutant `BR2_DEFCONFIG=` à la commande, suivie du chemin complet. Une commande équivalente existe pour la configuration du noyau Linux (`make linux-savedefconfig` ou `make linux-update-defconfig BR2_LINUX_KERNEL_CUSTOM_CONFIG_FILE=` suivi du chemin vers le fichier), pour U-Boot (`make uboot-savedefconfig` ou `uboot-update-defconfig`) et Barebox (`make barebox-savedefconfig`).

Attention cependant, si vous comptez partager cette configuration, à ne pas oublier d'ajuster l'ensemble en conséquence en supprimant les éléments qui ne doivent pas être statiques, je pense en particulier à la clé d'hôte de Dropbear.

Enfin, je le précise une nouvelle fois, ajouter la compatibilité *fbdev* et donc *fbcon* est amusant d'un



Sur la tranche de la carte, un jeu de LED et de boutons est à la disposition de l'utilisateur. Leurs connexions ainsi que leur configuration par défaut sont décrites dans le premier document que vous devrez lire en recevant le matériel : l'user manual UM2534 [7].

point de vue pédagogique, puisque cela nous a permis de voir comment modifier la configuration du noyau et d'autres éléments de Buildroot, mais ce n'est pas une bonne idée pour un projet. Si l'on veut proposer une IHM, mieux vaut ne pas s'en occuper et se concentrer sur la pile graphique standard en utilisant la LibDRM par exemple ou utiliser un *toolkit* comme GTK+, QT ou les EFL, éventuellement en passant par Weston (l'implémentation de référence de Wayland).

CONCLUSION

Je ne vous cacherai pas une certaine préférence personnelle pour Buildroot, que je trouve bien plus « cohérent » qu'OpenEmbbeded d'un point de vue de l'architecture. Certes, cette approche monolithique est plus rigide que ce qu'il est possible d'obtenir avec un mécanisme de *layers*, mais en contre-partie, la composition et la configuration « locale » d'un système s'en trouvent grandement simplifiée. Bien entendu, la comparaison est totalement dépendante du support effectivement offert par les constructeurs, car même si l'approche consistant à simplement produire un *layer* fournissant un BSP pour une carte est fort séduisante, en pratique, la quantité de *layers* non maintenus est relativement importante **[6]**. Inversement, Buildroot incite les constructeurs à proposer un support s'intégrant pleinement au système de *build*, les options comme `BR2_EXTERNAL` n'étant souvent que les prémices à une intégration *upstream*. Au final, Buildroot est plus « maîtrisable » de la part de l'utilisateur final et l'adaptation à un projet spécifique s'en trouve d'autant simplifiée. Bien entendu, ceci est également une question d'expérience, mais la courbe d'apprentissage est clairement plus clémente dans le cas de Buildroot.

En ce qui concerne les manipulations que nous venons de voir, un certain nombre d'améliorations restent à faire. Le fait que ce soit les anciennes versions du *devkit* (STM32MP157**A**-DK1 et STM32MP157**C**-DK2), maintenant considérées comme obsolètes, car remplacées par les STM32MP157**D**-DK1 et STM32MP157**F**-DK2, ne pose pas de grands problèmes, même si cela mériterait d'être complété. Vous avez sans doute remarqué un message du noyau au démarrage vous faisant remarquer que « *OPP table can't be empty* ». Ceci provient d'éléments manquants dans le *device-tree*, en particulier concernant les horloges du SoC et

le support OPP gérant des paires fréquence/tension supportées par le SoC. Il est possible de régler ce problème en générant un nouveau source du *devicetree* avec STM32CubeMX et en l'adaptant (voir le fichier `doc/stm32cubemx.md` de `buildroot-external-st` pour plus d'informations).

Un autre élément à régler est le fait que le *devkit*, démarré avec le système de démonstration, perturbe l'hôte auquel il est connecté en USB, remplissant par la même occasion les logs de messages comme :

```
new high-speed USB device number 19 using ehci-pci
device descriptor read/64, error -110
evice descriptor read/64, error -110
attempt power cycle
new high-speed USB device number 20 using ehci-pci
unable to enumerate USB device
```

Le port USB-C par lequel nous flashons la microSD est, en effet, USB OTG, et donc en mesure de se comporter comme un périphérique. Mais rien n'est configuré par défaut et l'hôte ne cesse de tenter d'énumérer ce qu'il pense être un périphérique. L'une des conséquences est une perturbation complète du sous-système USB de l'hôte qui devient incapable, par exemple, d'utiliser correctement une YubiKey 5 (ce qui est relativement pénible lorsqu'on tente de s'authentifier sur GitHub pour proposer un *patch* justement). Pour régler ce problème, la solution simple est de débrancher le câble USB, mais il serait bien plus intéressant de configurer l'USB OTG via l'API USB Gadget au démarrage (pour fournir un port série ou une interface Ethernet).

Il est fort probable que nous parlions encore à l'avenir de cette sympathique plateforme, peut-être pour traiter de ces points, peut-être pour nous pencher sur le Cortex-M4 intégré au SoC, ou tout simplement pour faire plus ample connaissance avec Buildroot... **DB**

RÉFÉRENCES

[1] **https://connect.ed-diamond.com/hackable/hk-041/stm32mp1-le-soc-qui-etend-l-ecosysteme-stm32-vers-linux-embarque**

[2] **https://github.com/bootlin/buildroot-external-st/blob/st/2021.02/docs/internals.md**

[3] **https://wiki.st.com/stm32mpu/wiki/TF-A_overview#FIP**

[4] **https://busybox.net/**

[5] **https://github.com/matusnovak/rpi-opengl-without-x**

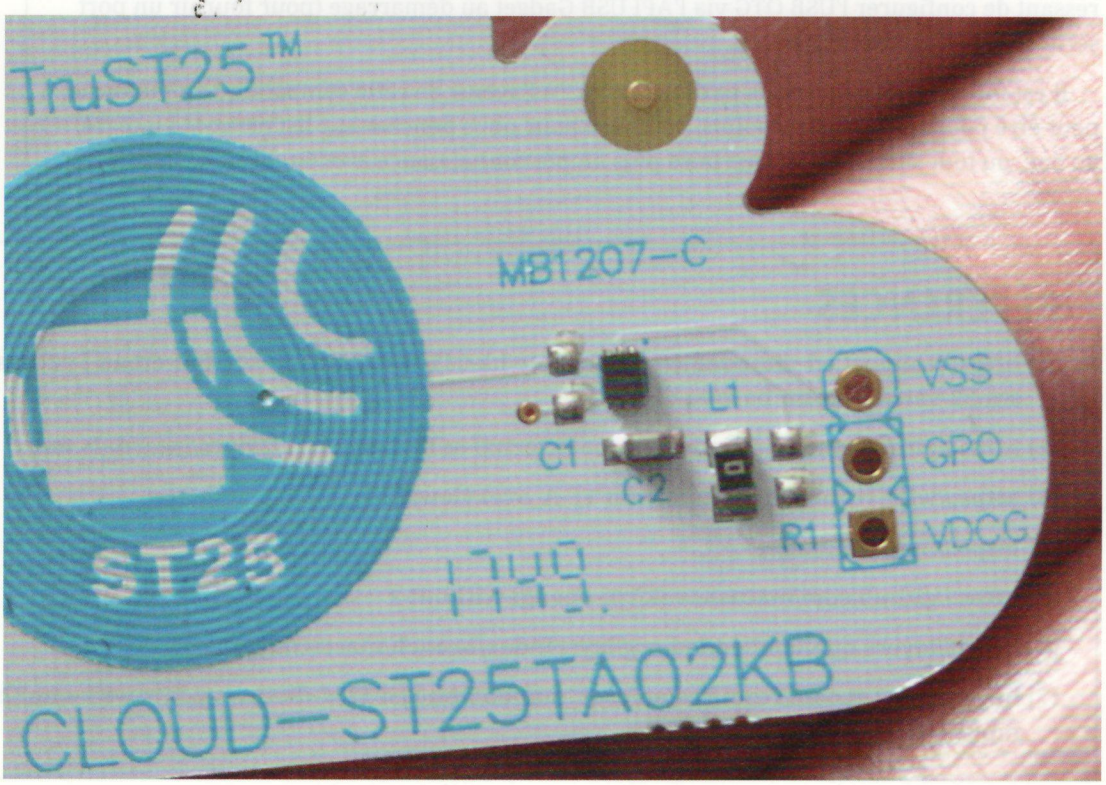
[6] **https://layers.openembedded.org/layerindex/branch/master/layers/**

[7] **https://www.st.com/resource/en/user_manual/dm00591354-discovery-kits-with-stm32mp157-mpus-stmicroelectronics.pdf**

MANIPULER LES TAGS ST25 AVEC LA LIBNFC

Denis Bodor

Lorsqu'on parle de technologies NFC, on pense généralement aux tags comme les NTAG, les MIFARE Classic ou plus raisonnablement, les DESFire EV1. Mais il existe tout un monde en dehors des produits NXP souvent parfaitement pris en charge par des bibliothèques de haut niveau, et c'est là l'occasion parfaite d'explorer, plus en profondeur, les protocoles et fonctionnalités offertes par la libNFC. Penchons-nous donc sur les tags ST25TA de chez STMicroelectronics...



– Manipuler les tags ST25 avec la libNFC –

Nous avons, par le passé, fait connaissance avec l'univers merveilleux des technologies RFID et NFC (voir Hackable 10 **[1]**), avons manipulé des tags et leur contenu et même testé leur sécurité avec l'outil par excellence dans ce domaine qu'est le Proxmark3 (dans le numéro 30 **[2]**). Dans nos expérimentations, cependant, nous sommes restés à un niveau d'abstraction assez élevé en utilisant principalement l'excellente libfreefare **[3]** reposant sur libNFC et prenant en charge une collection sympathique de tags : FeliCa Lite, MIFARE Classic, MIFARE DESFire, MIFARE DESFire EV1, MIFARE Ultralight, MIFARE Ultralight C et NTAG21x.

Remarquez que dans cette liste, seul FeliCa n'est pas un produit NXP (mais Sony) et que nous n'avons aucune trace de ST. Comment faire alors pour communiquer avec ces autres tags lorsqu'un projet le demande ? La réponse est relativement simple, « *il faut descendre d'un niveau* », ce qui signifie utiliser la libNFC directement, assimiler la documentation et comprendre l'utilisation des protocoles et standards d'usage. Choses que nous allons précisément faire ici en nous penchant sur les tags ST25TA de STMicroelectronics.

1. RAPPEL SUR LES TYPES DE TAGS NFC

Le NFC Forum est l'association ou consortium international, initialement créé par Sony, Nokia et Philips (maintenant NXP) en 2004 afin de faire la promotion du NFC et d'établir les standards qui dictent l'implémentation et l'utilisation de cette technologie. Le NFC Forum définit cinq types de tags tels que détaillés dans le tableau suivant :

	Type 1	Type 2	Type 3	Type 4	Type 5
UID	4 ou 7 octets	4 ou 7 octets	8 octets	7 octets	8 octets
Protocole de transmission	ISO 14443A	ISO 14443A	ISO 18092 / JIS X 6319	ISO 14443A+B	ISO 15693
Taille mémoire (octets)	96 à 2K	64 à 2K	jusqu'à 1M	jusqu'à 64K	jusqu'à 64K
Bits OTP	48	32	-	-	-
Anticollision	non	oui	oui	oui	oui
Vitesse de transmission	106 kbit/s	106 kbit/s	212 kbit/s, 424 kbit/s	106 kbit/s, 212 kbit/s, 424 kbit/s	26,48 kbit/s
Exemple de tag	Topaz	MIFARE Ultralight	FeliCa Lite	MIFARE DESFire EV1	ICODE SLIx
Remarque	Usage spécifique	Peu cher, très courant	Marché asiatique	Courant pour les applications nécessitant plus de mémoire	Détection d'effraction, grande distance de lecture

Notez l'absence de MIFARE Classic qui est une catégorie à part, car ne correspondant à aucun des cinq types standardisés. Techniquement, un MIFARE Classic n'est donc pas un tag NFC, même s'il s'agit encore aujourd'hui du tag que l'on rencontre malheureusement le plus couramment (malgré ses problèmes de sécurité).

Pour nos développements nous aurons besoin d'un périphérique RFID/NFC comme l'un de ceux présentés ici. De gauche à droite, un ASK/LoGO, un SCL3711 très compact et un très populaire ACR122U.

En plus de ces spécifications, et de bien d'autres, le NFC Forum a également standardisé le format d'échange de données, ou NDEF (pour *NFC Data Exchange Format*) structurant la manière dont sont organisées et présentées les données dans un tag, ou dans les échanges avec un tag. Avec NDEF, les données sont stockées sous forme d'enregistrements, comportant un type (URI, texte, type MIME, etc.) et une charge utile (l'information elle-même). Ici encore, le standard est clair, un tag ne contenant pas des données NDEF n'est techniquement pas un tag NFC.

2. ST25

L'écosystème NFC/RFID chez STMicroelectronics est assez riche et se divise en quatre gammes ou séries de tags :

- Série ST25TV : tags NFC type 5 généralement utilisés pour les applications commerciales ou de consommation (livres, tickets, produits médicaux, etc.), dispositif anti-contrefaçon, authentification de produit, etc. Ces tags proposent également des mécanismes « *tamper detect* » fonctionnant comme des étiquettes de garantie, ainsi que l'Augmented NDEF permettant de dynamiquement actualiser les données NDEF.
- Série ST25TN : tags NFC type 2, économiques, proposant jusqu'à 208 octets de stockage, principalement destinés à l'étiquetage de produits (vêtements, équipement de sport, alcool, etc.). Ces tags proposent également l'Augmented NDEF pour générer dynamiquement du contenu NDEF.
- Série ST25TB : tags RFID ISO14443-2 type B avec de 512 à 4096 bits d'EEPROM pour les titres de transport ou la billetterie (spectacles, salon, etc.).
- Série ST25TA : tags NFC type 4 utilisant ISO 14443A et permettant un stockage NDEF jusqu'à 8 Ko pour des usages génériques allant de l'étiquetage aux cartes de visite NFC en passant par le *pairing* Blutooth, etc. Cette gamme est anciennement connue sous le nom SRTAG. À noter que certains de ces tags/puces supportent également une sortie programmable (GPO) permettant de les interfacer avec un microcontrôleur.

Nos expérimentations ici concerneront les ST25TA qui sont adaptés à un usage versatile tout en offrant des capacités de stockage intéressantes, et donc se rapprochant le plus des produits pris en charge par la *libfreefare*. Pour procéder à nos développements et nos essais, rien de plus simple puisque non seulement ces tags utilisent un protocole de transmission ISO 14443A parfaitement pris en charge par les lecteurs les plus courants (ACR122U, ASK/LoGO et SCL3711), mais sont disponibles à la vente sous la forme de lots d'échantillons chez des détaillants comme Mouser par exemple.

Pour cet article, j'ai donc jeté mon dévolu sur trois produits :

- ST25-TAG-BAG-UB : un lot de 21 tags à différents formats (autocollant, carte, étiquette de garantie, etc.) couvrant les séries ST25TA et ST25TV (référence « 511-ST25-TAG-BAG-UB » chez Mouser pour ~20 €).
- ST25-TAG-BAG-A : un lot de 6 tags ST25TA de différentes formes collés sur un support cartonné (référence « 511-ST25-TAG-BAG-A » pour ~3,50 €).
- CLOUD-ST25TA02KB : un tag ST25TA02KB-P de démonstration prenant la forme d'un amusant circuit imprimé à l'apparence d'un nuage, permettant d'évaluer la fonctionnalité GPO (référence « 511-CLOUD-ST25TA02KB » pour ~4,50 €). On remarquera que, pour des gens du marketing, une sortie à connecter à un microcontrôleur suffit à nous basculer dans le monde du « cloud ». On vit une époque formidable ! Messieurs les ingénieurs de ST, sachez que je compatis...

La série ST25TA se compose de 10 produits, dont 6 sont « *non recommandés pour de nouveaux designs* », ce qui ne laisse que 4 modèles :

- ST25TA02KB-D : EEPROM 2 kbit avec GPO (*General Purpose digital Output*) à collecteur ouvert nécessitant donc une résistance de rappel.
- ST25TA02KB-P : EEPROM 2 kbit avec GPO avec *buffer/driver* CMOS (nécessite de fournir une tension de référence, mais sans courant consommé). C'est le composant équipant le PCB « nuage ».
- ST25TA16K : EEPROM 16 kbit (2 Ko).
- ST25TA64K : EEPROM 64 kbit (8 Ko).

Les quatre modèles proposent, bien entendu, un stockage NDEF ainsi qu'une protection d'accès aux données (lecture et/ou écriture) par mot de passe 128 bits. Un mécanisme de signature électronique est également disponible sous la désignation « TruST25 », mais la note d'application (AN5101) n'est pas disponible publiquement et nécessite la signature d'un accord de non-divulgation (NDA).

Je ne détaillerai pas dans cette présentation l'architecture et l'organisation mémoire des ST25TA. Ceci est parfaitement documenté en détail dans les *datasheets* (**[4] [5] [6]**), qui sont précisément faites pour cela. Nous verrons plutôt ceci au fur et à mesure de notre prise en main de la libNFC dans la suite de l'article.

3. PHASE 1 : ÉTABLIR LE CONTACT

Nous allons utiliser la libNFC en version 1.8.0 (la plus récente stable à cette date), normalement disponible pour toutes les distributions GNU/Linux. Étant donné que notre cadre rédactionnel est celui des systèmes embarqués et du développement bas niveau, nous ne traiterons que de ce système d'exploitation, même si la libNFC est également utilisable sous Windows et/ou macOS.

Si vous utilisez une distribution basée sur Debian (Ubuntu, Raspberry Pi OS, Raspbian, Armbian, etc.), vous aurez besoin du minimum pour développer (Make, GCC, etc.), des paquets de développement pour la libUSB (`libusb-dev`) et bien entendu, de ceux pour la libNFC (`libnfc-dev`) elle-même. Je pars ici du principe que votre lecteur RFID/NFC est correctement configuré et pris en charge, et qu'il fonctionne sans problème. Ainsi, un simple `nfc-poll` (paquet `libnfc-examples`) doit vous afficher effectivement le minimum d'informations sur le tag présenté (ATQA, UID, ATS, etc.) avant d'entamer vos expérimentations.

Le paquet `libnfc-dev` intégrant un profil `pkg-config`, la compilation de notre code pourra se faire très simplement avec quelque chose comme `gcc -O2 -Wall 'pkg-config --cflags --libs libnfc' -o st25 source.c` ou, plus raisonnablement, en écrivant rapidement un `Makefile` sur cette base.

La première étape pour manipuler nos tags ST25TA consiste à initialiser le *framework*, accéder au périphérique, sélectionner le tag et obtenir une réponse de sa part. La libNFC maintient une structure interne de fonctionnement formant un contexte qu'il faut initialiser avant toute opération. De plus, une autre structure sert de représentation pour le (ou les) périphérique(s) NFC que nous comptons utiliser. Ces deux éléments sont déclarés en variable globale :

```
nfc_context *context;
nfc_device *pnd;
```

Cette approche nous permet d'implémenter une fonction utilitaire chargée de faire le ménage en cas d'erreur :

```
void failquit()
{
  if(pnd) nfc_close(pnd);
  if(context) nfc_exit(context);
  exit(EXIT_SUCCESS);
}
```

Également pour nous faciliter la vie et rendre le code plus facile à lire, nous ajoutons une fonction permettant d'afficher une série de `uint8_t` (`unsigned char`) en hexadécimal :

```
void print_hex(uint8_t *pbtData, size_t szBytes)
{
  size_t  szPos;
  for(szPos = 0; szPos < szBytes; szPos++) {
    printf("%02X", pbtData[szPos]);
  }
}
```

Ceci fait, nous pouvons enfin nous attaquer au sujet qui nous intéresse, et donc au `main()` où notre première tâche sera d'initialiser le contexte de la libNFC (oui, mes sorties seront en anglais, et elles le sont toujours, dans l'éventualité que le code devienne quelque chose de plus sérieux et diffusable) :

```
int main(int argc, const char *argv[])
{
  nfc_init(&context);
  if(context == NULL) {
    printf("Unable to init libnfc (malloc)\n");
    exit(EXIT_FAILURE);
  }
```

Nous pouvons maintenant utiliser la bibliothèque et commencer par nous connecter au lecteur NFC/RFID avec :

```
  pnd = nfc_open(context, NULL);

  if(pnd == NULL) {
    fprintf(stderr, "Unable to open NFC device!\n");
    exit(EXIT_FAILURE);
  }
```

`nfc_open()`, utilisé de cette façon, ouvrira le premier périphérique disponible, qui peut être celui spécifié par la variable d'environnement `LIBNFC_DEFAULT_DEVICE`, le premier listé dans `/etc/nfc/libnfc.conf` ou `/etc/nfc/devices.d/*`, ou encore tout simplement le premier automatiquement détecté. Une fois `pnd` nous permettant d'accéder au périphérique, nous l'ouvrons en tant qu'initiateur (par opposition à « cible » pour l'émulation de tags) :

```
  if(nfc_initiator_init(pnd) < 0) {
    nfc_perror(pnd, "nfc_initiator_init");
    exit(EXIT_FAILURE);
  }
  printf("NFC reader: %s opened\n", nfc_device_get_name(pnd));
```

Nous voici fin prêts pour nous occuper de ce qui se trouve potentiellement posé sur le lecteur, mais devons auparavant spécifier le type de modulation et de débit que nous souhaitons utiliser, ainsi que déclarer une nouvelle variable qui représentera notre tag dans la suite du code :

```
  const nfc_modulation mod = {
    .nmt = NMT_ISO14443A,
    .nbr = NBR_106
  };
  nfc_target nt;
```

`NMT_ISO14443A` et `NBR_106` sont spécifiés dans `nfc-types.h` et correspondent respectivement à la modulation et au débit correspondants au tag que nous cherchons, nous pouvons ensuite sélectionner le premier tag présent (normalement le seul), avec :

```
if(nfc_initiator_select_passive_target(pnd, mod, NULL, 0, &nt) > 0) {
  printf("ISO14443A tag found. UID: ");
  print_hex(nt.nti.nai.abtUid, nt.nti.nai.szUidLen);
  printf("\n");
} else {
  fprintf(stderr, "No ISO14443A tag found!\n");
  failquit();
}
```

`nfc_initiator_select_passive_target()` retournera le nombre de cibles sélectionnées ou une valeur négative en cas d'échec, mais surtout peuplera notre structure `nt` que nous pourrons ensuite analyser pour, comme ici, obtenir l'UID du tag. Notre cible restera sélectionnée jusqu'à utilisation de `nfc_initiator_deselect_target()` ou la fin de notre programme qui, précisément, se résume à :

```
  // Close NFC device
  nfc_close(pnd);
  // Release the context
  nfc_exit(context);
  exit(EXIT_SUCCESS);
}
```

L'exécution du présent exemple, après compilation, devrait vous afficher quelque chose comme :

```
$ ./st25ta_read
NFC reader: ASK / LoGO opened
  ISO14443A tag found. UID: 02C4004E3771A4
```

Nous savons maintenant nous adresser à notre tag, via la libNFC et le lecteur USB (ici ASK/LoGO), et le sélectionner pour ouvrir un canal de communication pour de futurs échanges. Il est temps de passer aux choses sérieuses et de gentiment (mais fermement) le questionner.

4. PHASE 2 : APPRENDRE À PARLER APDU

Oublions un instant qu'il s'agit là de RFID ou même d'une communication sans contact. Oublions même la libNFC puisque celle-ci n'est qu'un intermédiaire entre nous et le tag, au même titre que le lecteur ou la liaison USB. Tout ceci fonctionne exactement comme un protocole réseau, par encapsulation, au travers de couches successives. Ce qui nous intéresse ici cependant, c'est l'échange de données entre nous (notre code) et le tag. Le protocole utilisé à ce niveau est celui décrit par le standard ISO/IEC 7816-4 et repose sur l'utilisation d'APDU.

L'APDU ou *Application Protocol Data Unit* est l'unité de communication utilisée dans le domaine des *smartcards* (et non uniquement des tags RFID/NFC), exactement comme les trames, datagrammes et paquets le sont dans celui du réseau. Il existe deux types d'APDU, ceux de commande

ST met à disposition des lots de tags permettant d'évaluer les solutions et de procéder aux tests. Voici le ST25-TAG-BAG-UB regroupant plusieurs modèles de la gamme ST25 sous différentes formes (21 pièces).

ou C-APDU, et ceux de réponse ou R-APDU. Un C-APDU se compose ainsi :

- *CLA* (1 octet) : la classe ou type d'instructions utilisé (standard, propriétaire, etc.) ;
- *INS* (1 octet) : le code d'instruction (sélection de fichiers, d'application, lecture, etc.) ;
- *P1 P2* (2 octets) : les paramètres de l'instruction ;
- *Lc* (0, 1 ou 3 octets) : encode la longueur en octets des données envoyées. Ce champ n'est **pas** la taille en octets, mais un encodage de cette valeur qui est nommée *Nc* ;
- *DATA* (variable) : les données à envoyer (s'il y en a) ;
- *Le* (0, 1, 2 ou 3 octets) : encode la longueur maximum en octets des données attendues en retour. À quelques nuances près, le même mécanisme que pour *Lc* est utilisé et la taille est désignée par *Ne*.

Un R-APDU est sensiblement plus simple :

- *DATA* (variable) : les données de la réponse (s'il y en a) ;
- *SW1 SW2* (2 octets) : le code d'état (ou de réponse) de la commande (`0x90 0x00` pour « commande exécutée » par exemple).

Pour utiliser ce mécanisme afin de procéder à des échanges avec le tag, il nous suffit donc de lui envoyer des C-APDU véhiculant des ordres et d'attendre des R-APDU, contenant soit une réponse (code d'état), soit les données attendues + le code d'état. Ces APDU sont majoritairement standardisés et, dans le cas des ST25TA, se divisent en trois catégories :

- le jeu de commandes NFC Forum Type 4 Tag permettant la sélection, la lecture et l'écriture (classe `0x00`) ;
- celui du standard ISO/IEC 7816-4 gérant les commandes dépendantes du contrôle d'accès aux données, dont l'authentification (classe `0x00` également) ;
- et celui propriétaire de ST (classe `0xa2`) fournissant entre autres une commande de lecture alternative, ainsi que d'autres, sortant du cadre du standard NFC.

Nous allons utiliser ces commandes, décrites relativement clairement dans la *datasheet* des ST25TA, dans un instant. Mais avant cela, il est important de comprendre l'organisation de la mémoire et la structure même d'un tag NFC. Le terme « *smartcard* » n'est pas utilisé par hasard et un tag n'est pas une simple EEPROM sans contact. Il se compose d'une ou plusieurs applications, fournissant l'accès à un ou plusieurs fichiers. Dans le cas des ST25TA, il n'y a qu'une application, fournissant un unique fichier NDEF (cf. note), mais les MIFARE DESFire EV1, par exemple, peuvent contenir jusqu'à 28 applications gérant chacune 32 fichiers.

Il semble y avoir une omission dans la *datasheet* du ST25TA64K, puisque l'application Android « *ST25 NFC Tap* » **[7]** permet de reconfigurer le tag de façon à diviser sa mémoire jusqu'à 8 zones (*area*) de tailles égales, et donc supporter jusqu'à 8 fichiers NDEF (en mettant à jour le fichier CC en conséquence).

La *datasheet* (« DocID027764 Rev 4 » - 21/02/2017) n'en fait étrangement aucunement mention, si ce n'est en précisant que l'octet à l'offset 7 du fichier système indique le « *NDEF File number* » (*number* dans le sens « nombre » et non « numéro ») en spécifiant « RFU » pour « *Reserved for Future Use* » et un accès en écriture à « *none* », alors que l'application Android change effectivement cette valeur (avec un simple `0x00 0xd6`, *UpdateBinary*).

La *datasheet* serait-elle périmée ou nous cache-t-on des choses ?

Ainsi pour accéder aux données, il est nécessaire de sélectionner l'application qui gère le fichier, puis le fichier dans lequel nous souhaitons lire ou écrire. Chacune de ces étapes utilise des commandes et réponses transmises sous forme d'APDU. Applications et fichiers sont désignés par un identifiant (ID). L'application « *NDEF Tag* » d'un ST25TA possède l'ID `0xD2760000850101` et gère trois fichiers :

- `0xE101` : le fichier système ST contenant diverses informations dont l'UID du tag, le volume de mémoire ou le code identifiant le modèle. Dans les cas des tags ST25TA02KB-P et D, c'est également là que se trouvent la configuration GPO ainsi qu'un compteur 20 bits configurable.
- `0xE103` : *Capability Container* (CC) qui est un fichier standardisé fonctionnant comme un *README* pour les applicatifs NFC en fournissant la taille maximum du fichier NDEF, son ID, la taille maximum des données dans les APDU, ou encore les permissions sur le fichier.
- `0x0001` : le fichier NDEF lui-même, composé de deux octets spécifiant sa taille suivie d'un message NDEF composé d'un ou plusieurs enregistrements, eux-mêmes composés d'un en-tête et d'une charge utile. Le format NDEF sort du cadre du présent article, mais nous évoquerons brièvement une solution pratique en conclusion. Notez simplement qu'il y a une distinction entre le **fichier** NDEF débutant par deux octets spécifiant sa taille et le **message** NDEF qui se trouve juste passé ces octets, et qui est constitué d'**enregistrements**

NDEF. Comprenez bien que tout ce petit monde n'est rien d'autre qu'une série d'octets les uns à la suite des autres en mémoire et que c'est à la charge du développeur d'interpréter ces informations. Ce n'est pas le problème du tag si vous lisez au-delà de l'enregistrement que vous visez et dans bien des cas, vous n'aurez aucune erreur en retour.

La *datasheet* ST précise clairement la manière « officielle » de lire le message NDEF :

- sélectionner l'application « *NDEF Tag* » ;
- sélectionner le fichier CC ;
- lire le fichier CC ;
- vérifier les permissions ;
- le cas échéant, fournir le mot de passe de lecture ;
- sélectionner le fichier NDEF dont l'ID a été obtenu via le fichier CC ;
- lire le contenu du fichier NDEF.

Oui, nous connaissons déjà l'ID du fichier NDEF puisque celui-ci est spécifié à maintes reprises dans la *datasheet*, mais la norme impose qu'il soit nécessaire d'utiliser le *Capability Container* pour l'obtenir. Ceci est une procédure standard qui assure la compatibilité et fait que votre smartphone est capable d'afficher le contenu NDEF d'un tag ST25TA, mais également de n'importe quel autre produit compatible NFC.

Le lot ST25-TAG-BAG-A est plus modeste et ne regroupe que 6 tags, mais il s'agit exclusivement de ST25TA dont il est question dans cet article. Très économique pour rapidement expérimenter, ce « bag » présente les tags déjà collés à un support cartonné, ce qui est relativement peu pratique.

5. PHASE 3 : POUVOIR COMMUNIQUER AVEC LE TAG

Je pense que nous avons à présent assez parlé de théorie et il est grand temps de passer à la pratique. Notre code, pour l'heure, ne fait pas grand-chose, mais nous avons engagé les pourparlers et sommes donc prêts pour échanger des APDU. La libNFC met à notre disposition une fonction `nfc_initiator_transceive_bytes()` qui, comme son nom l'indique, permet de « *transceiver* » des octets ou, en d'autres termes, envoyer des données à notre cible et en recevoir en retour (comme *transmit* et *receive*).

La fonction prend en argument un pointeur vers le périphérique (`pnd`), un pointeur vers untableau de `uint8_t` (`unsigned char`), un entier (`size_t`) en précisant la taille, un pointeur vers un tableau de `uint8_t` destiné à recevoir la réponse, un entier (`size_t`) spécifiant la taille maximum des données à mettre dans ce tableau et enfin, un entier

spécifiant un délai d'expiration en millisecondes (`-1` pour utiliser la valeur par défaut). Cette fonction retourne le nombre d'octets reçus en réponse en cas de réussite ou une valeur négative en cas d'échec (correspondant à un code d'erreur libNFC).

Plutôt que d'utiliser directement `nfc_initiator_transceive_bytes()`, nous pouvons créer une fonction utilitaire prenant les mêmes arguments, mais affichant, en prime, les APDU à l'écran pour nous faciliter la mise au point du code :

```
int CardTransmit(nfc_device *pnd,
                 uint8_t *capdu,
                 size_t capdulen,
                 uint8_t *rapdu,
                 size_t *rapdulen)
{
  int res;
  size_t  szPos;

  printf("=> ");

  for(szPos=0; szPos<capdulen; szPos++) {
    printf("%02x ", capdu[szPos]);
  }
  printf("\n");

  if((res = nfc_initiator_transceive_bytes(
        pnd, capdu, capdulen, rapdu, *rapdulen, -1)) < 0) {
    fprintf(stderr, "transceive error! %s\n", nfc_strerror(pnd));
    return(-1);
  }

  printf("<= ");
  for(szPos = 0; szPos < res; szPos++) {
    printf("%02x ", rapdu[szPos]);
  }
  printf("\n");

  *rapdulen = (size_t)res;

  return(0);
}
```

La petite subtilité ici réside dans le fait d'utiliser la valeur de retour (`res`) pour mettre à jour l'un des paramètres passés en argument. Ainsi, nous passons à notre fonction un pointeur vers le tableau destiné à accueillir le R-APDU ainsi qu'un **pointeur** vers la valeur en décrivant la taille (`rapdulen`), et nous obtenons en retour, dans la même variable (`*rapdulen`), la quantité d'octets constituant effectivement la réponse.

Cette fonction impose de préparer des tableaux pour procéder à l'échange alors que, dans bien des cas, et en particulier en phase d'expérimentation, il peut être bien plus intéressant de passer des C-APDU arbitraires de façon plus souple. Nous implémentons donc une seconde déclinaison de cette fonction ainsi :

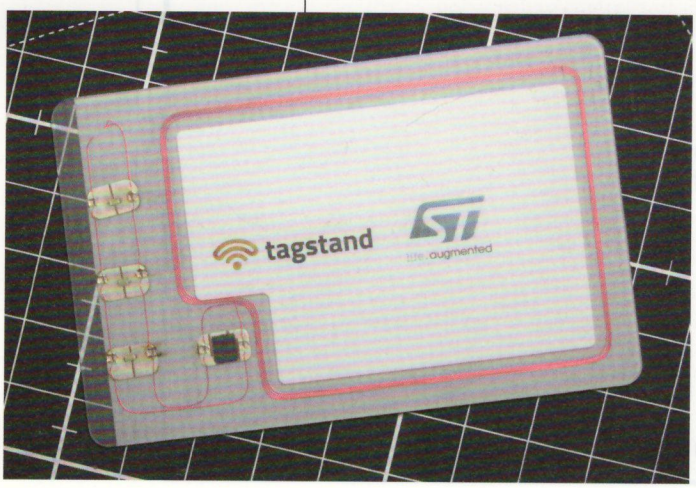
```
#define RAPDUMAXSZ 512
#define CAPDUMAXSZ 512

int strCardTransmit(nfc_device *pnd,
                    const char *line,
                    uint8_t *rapdu,
                    size_t *rapdulen)

{
  int res;
  size_t szPos;
  uint8_t *capdu = NULL;
  size_t capdulen = 0;
  *rapdulen = RAPDUMAXSZ;

  uint32_t temp;
  int indx = 0;
  char buf[5] = {0};

  // vérifications
  if(!strlen(line) || strlen(line) % 2 ||
     strlen(line) > CAPDUMAXSZ*2)
    return(-1);

  // allocation buffer
  if(!(capdu = malloc(strlen(line)/2))) {
    fprintf(stderr, "malloc error: %s\n", strerror(errno));
    nfc_close(pnd);
    nfc_exit(context);
    exit(EXIT_FAILURE);
  }

  // conversion
  while (line[indx]) {
    if(line[indx] == '\t' || line[indx] == ' ') {
      indx++;
      continue;
    }

    if(isxdigit(line[indx])) {
      buf[strlen(buf) + 1] = 0x00;
      buf[strlen(buf)] = line[indx];
    } else {
      // pas de l'hexa
      free(capdu);
      return(-1);
    }

    if(strlen(buf) >= 2) {
      sscanf(buf, "%x", &temp);
      capdu[capdulen] = (uint8_t)(temp & 0xff);
```

```
        *buf = 0;
        capdulen++;
      }
      indx++;
    }

    // erreur si octet hex incomplet
    if(strlen(buf) > 0) {
      free(capdu);
      return(-1);
    }

  [...]
```

Le reste sera identique à `CardTransmit()`, mais nous intégrons directement la conversion d'une chaîne de caractères (`*line`) représentant une suite de valeurs hexa en un tableau de `uint8_t`. Ceci nous permet de très facilement envoyer des C-APDU, spécifiés sous une forme humainement intelligible, et ce, sans nous soucier de leur taille ou encore de devoir à chaque fois préciser la taille du tableau pour le R-APDU (qui sera déclaré directement en utilisant la valeur `RAPDUMAXSZ`). Nous verrons plus loin que la taille des données dans les APDU est bornée et qu'il nous faudra prendre en compte cette information par ailleurs, mais ces deux fonctions sont prévues pour être génériques et réutilisables avec d'autres tags, voire d'autres applications de type RFID/NFC (voir l'article sur l'exploration de l'écran e-*paper* NFC de WaveShare **[8]**).

6. PHASE 4 : LIRE LE TAG

Nous voici maintenant équipés pour « parler APDU » avec notre tag et nous pouvons, juste après l'affichage de l'UID, commencer à nous adresser à lui. Les APDU sont directement spécifiés dans la *datasheet*, à commencer par la sélection de l'application « NDEF Tag ». Il s'agit d'une commande de la classe `0x00`, identifiée par l'instruction `0xa4` et ayant pour paramètres `0x04 0x00`. La taille des données transmises est de 7 octets et celles-ci se résument tout simplement à l'ID de l'application : `0xD2760000850101`. Enfin, la taille maximum des données attendues est à `0x00` et donc 256 (voir le standard ISO 7816-4, Section 5 « *Basic Organizations* », si vous en avez le courage et quelque 200 € à dépenser **[9]**).



Le lot de tags ST25-TAG-BAG-UB inclut un détecteur de champ RFID (« NFC Field Detector ») qui s'illumine à proximité d'un lecteur actif. C'est également un tag ST25TV02K qui n'est malheureusement pas compatible avec un lecteur comme l'ACR122U ne supportant pas ISO 15693.

Pour sélectionner l'application, nous devons donc utiliser :

```
// Select App 0xD2760000850101
if(strCardTransmit(pnd,
                   "00a4 0400 07 d2760000850101 00",
                   resp, &respsz) < 0)
  fprintf(stderr, "CardTransmit error!\n");

if(respsz < 2 || resp[respsz-2] != 0x90 ||
   resp[respsz-1] != 0x00) {
  fprintf(stderr, "Application select. Bad response !\n");
  failquit();
}
```

Notre C-APDU est `"00a4 0400 07 d2760000850101 00"` et l'utilisation de notre fonction `strCardTransmit()` nous permet de garder un semblant de lisibilité. Pour nous assurer que la réponse est valide, nous vérifions sa taille puis le fait qu'elle soit effectivement `0x9000`, « *Command completed* ». Ceci sera fait après chaque utilisation de `strCardTransmit()` ou `CardTransmit()`, mais ne sera pas repris dans le code qui suit. Nous pouvons ensuite enchaîner sur la sélection du fichier CC et sa lecture :

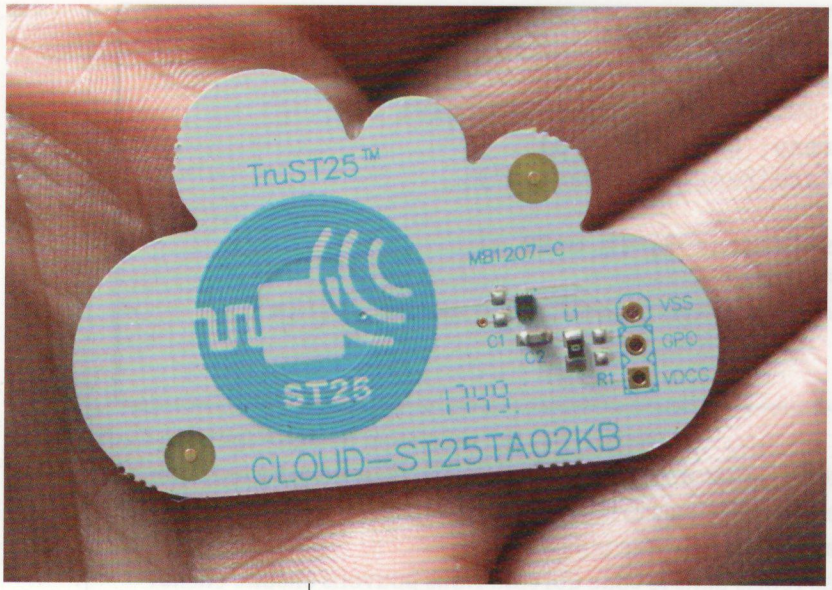
```
if(strCardTransmit(pnd,
                   "00a4 000c 02 e103",
                   resp, &respsz) < 0)
  fprintf(stderr, "CardTransmit error!\n");

if(strCardTransmit(pnd,
                   "00b0 0000 0f",
                   resp, &respsz) < 0)
  fprintf(stderr, "CardTransmit error!\n");
```

Remarquez que la première commande utilise le même couple classe/instruction puisqu'il s'agit également d'une sélection, mais les paramètres sont différents puisque cela concerne un fichier et ce dernier est désigné par son ID, `0xe103`, en guise de données.

La lecture utilise l'instruction `0xb0` de la même classe où les paramètres P1 et P2 précisent l'offset à partir duquel la lecture doit commencer par rapport au début du fichier (ici 0) et où `0x0f` correspond à *Le*, le nombre d'octets à lire et à nous retourner.

L'exécution de ce code nous affiche à l'écran :

```
=> 00 a4 04 00 07 d2 76 00 00 85 01 01 00
<= 90 00
=> 00 a4 00 0c 02 e1 03
<= 90 00
=> 00 b0 00 00 0f
<= 00 0f 20 00 f6 00 f6 04 06 00 01 20 00
00 00 90 00
```



*Discussion hypothétique marketing/ingénieur chez ST : « C'est un ST25TA02KB-P ? », « Oui », « Et ça a une sortie qu'on branche à un microcontrôleur », « Oui », « Les microcontrôleurs ont des interfaces réseau, hein ? », « Oui, parfois. Pourquoi ? », « Et on peut les connecter au Web », « Hmmm oui, par exemple, mais… », « Donc, c'est du cloud ! », « Attends, quoi ?! », « Hey Roger ! Les ST25TA02KB-P, c'est du cloud ! On va faire un circuit en forme de nuage ! », *facepalm*…*

Cette dernière réponse, moins « 90 00 », constitue le contenu du fichier CC qui, selon la *datasheet*, nous apprend que nous pouvons lire et écrire des données par paquets d'un maximum de 246 octets (0x00f6), que le fichier NDEF possède l'ID 0001 et que sa taille maximum est de 8192 octets (0x2000). On pourra éventuellement copier ce contenu dans un autre tableau qu'on *castera* sur une structure comme :

```
struct st25taCC_t {
      uint8_t size[2];
      uint8_t vmapping;
      uint8_t nbread[2];
      uint8_t nbwrite[2];
      uint8_t tfield;
      uint8_t vfield;
      uint8_t id[2];
      uint8_t maxsize[2];
      uint8_t readaccess;
      uint8_t writeaccess;
};
```

Gardons cependant cet article le plus concis possible afin d'arriver à notre objectif de lecture rapidement. À l'offset 0x09, nous trouvons l'ID du fichier, et pouvons l'utiliser pour une nouvelle sélection, en composant un C-APDU de toutes pièces. Mais avant cela, nous devons nous assurer que le fichier est effectivement lisible. Les deux derniers octets des données de la réponse que nous venons de recevoir déterminent les permissions actives sur le fichier NDEF, respectivement en lecture ou en écriture. 0x00 indique que le fichier est accessible sans aucune sécurité. 0x80 fixe l'état à « verrouillé » (*locked*) signifiant qu'une authentification est nécessaire avant une tentative de lecture ou d'écriture (mais après la sélection).

Comprenez bien que les deux opérations sont totalement distinctes. Nous avons un octet pour la lecture et un autre pour l'écriture, et de la même manière un mot de passe de 128 bits pour la lecture et un autre pour l'écriture. Les deux mots de passe sont par défaut 128 bits (16 octets) à zéro.

Et, enfin, 0xFE pour la lecture et 0xFF pour l'écriture indiquent que le fichier est définitivement verrouillé et de ce fait que les opérations de lecture et/ou d'écriture ne sont pas possibles. Ceci permet, par

exemple, de configurer le tag en lecture seule de façon définitive. Cette configuration ne peut être changée que dans un sens et via l'utilisation d'un C-APDU spécifique à ST (classe/instruction `0xA2 0x28`). Le verrouillage permanent n'est, de plus, possible que si le fichier est déjà verrouillé (0x80).

Dans notre cas d'exemple, nous nous contenterons de nous assurer que l'octet à l'offset 13 (`0x0D`) du CC est à bien à `0x00` et, ce faisant, pourrons entamer la lecture du tag. Nous aurons également besoin de faire attention à une autre valeur fournie par le CC à l'offset `0x03`, nous indiquant la taille maximum de données pouvant être lues en une fois. Celle-ci est stockée sur deux octets, mais la *datasheet* nous indique qu'elle est fixée à `0x00F6` et donc vaut 246. Ceci signifie que, quelle que soit la taille du fichier NDEF, nous devrons, dans la majorité des cas, lire le fichier en plusieurs fois.

Avec tous ces éléments, nous pouvons maintenant sélectionner le fichier NDEF :

```
// taille de lecture
uint16_t readsz = (resp[3] << 8) | resp[4];

// NDEF lisible ?
if(resp[13] != 0) {
  fprintf(stderr, "NDEF file locked!\n");
  failquit();
}

// C-APDU sélection
uint8_t selndefapdu[7] =
   { 0x00, 0xa4, 0x00, 0x0c, 0x02, 0x00, 0x00 };
// ajustement ID
selndefapdu[5] = resp[9];
selndefapdu[6] = resp[10];

// sélection
respsz = RAPDUMAXSZ;
if(CardTransmit(pnd, selndefapdu, 7, resp, &respsz) < 0)
  fprintf(stderr, "CardTransmit error!\n");
```

Pour effectivement lire le fichier, nous devons commencer par en récupérer les deux premiers octets afin d'en connaître la taille. Ceci est impératif et va au-delà de la simple économie en termes d'opérations de lecture. Si nous tentons de lire au-delà du fichier, nous obtiendrons une erreur avec l'APDU de lecture standardisée NFC Forum (c'est la seule vérification existante, la lecture d'enregistrements dans le message NDEF n'est pas encadrée par le tag). Si nous souhaitons lire la mémoire directement et arbitrairement, il nous faudrait utiliser le jeu de commandes propriétaires ST (`0xa2 0xb0` pour *ExtendedReadBinary*).

Commençons par récupérer ces deux octets et les convertir en un entier 16 bits que nous utilisons directement pour allouer la mémoire où nous copierons le message NDEF :

```
if(strCardTransmit(pnd, "00b0 0000 02", resp, &respsz) < 0)
  fprintf(stderr, "CardTransmit error!\n");

unsigned int bytestoread = (resp[0] << 8) | resp[1];

uint8_t *ndef;
if(!(ndef = malloc(bytestoread))) {
  fprintf(stderr, "malloc error: %s\n", strerror(errno));
  failquit();
}
```

Et nous procédons ensuite à la lecture dans une simple boucle `while` :

```
uint16_t pos = 2;
uint16_t end = bytestoread+pos;
uint8_t rndefapdu[5] = { 0x00, 0xb0, 0x00, 0x00, 0x00 };

while(pos < end-1) {
  uint8_t nread = pos+readsz < end ? readsz : end-pos;

  // ajustement C-APDU
  rndefapdu[2] = (pos >> 8);
  rndefapdu[3] = pos & 255;
  rndefapdu[4] = nread;

  // Lecture
  respsz = RAPDUMAXSZ;
  if(CardTransmit(pnd, rndefapdu, 5, resp, &respsz) < 0)
    fprintf(stderr, "CardTransmit error!\n");

  // Vérification
  if(respsz < 2 || resp[respsz-2] != 0x90 ||
     resp[respsz-1] != 0x00) {
    fprintf(stderr, "NDEF file read. Bad response !\n");
    failquit();
  }

  // Copie en buffer
  memcpy(ndef+pos-2, resp, nread);
  pos=pos+nread;
}
```

Et affichons le contenu légèrement formaté :

```
for(int i=0; i < bytestoread; i++) {
  printf("%02x ", ndef[i]);
  if(!((i+1)%40))
    printf("\n");
}
printf("\n");

if(ndef)
  free(ndef);
```

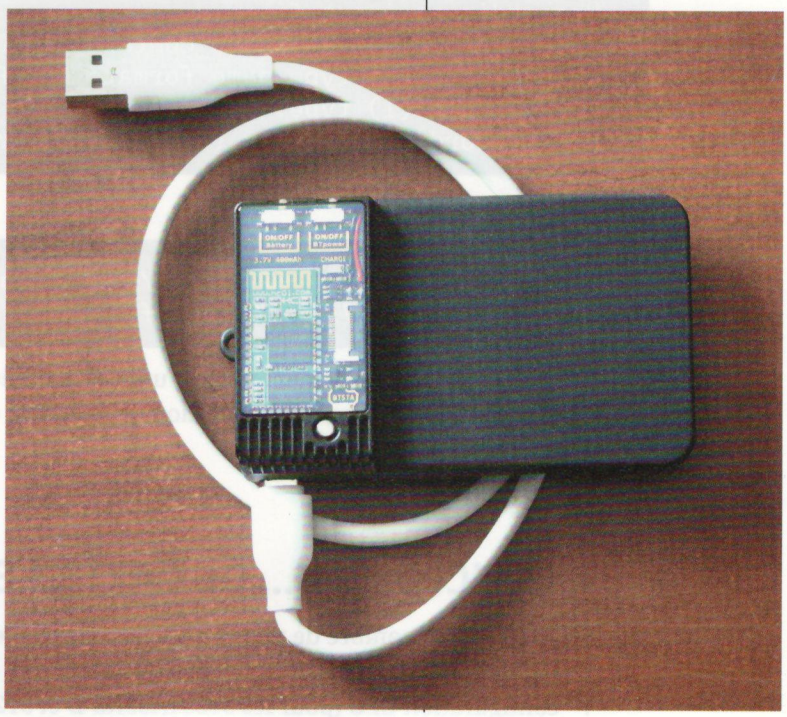
Bien entendu, tout ceci n'est qu'expérimental, mais nous pouvons désormais, en supprimant les autres messages, rediriger la sortir standard de notre code pour créer un fichier, ou simplement copier coller depuis le terminal :

```
$ ./st25ta_read
91 01 07 54 02 66 72 70
6c 6f 70 11 01 09 54 02
66 72 74 69 74 69 74 69
51 01 17 55 04 63 6f 6e
6e 65 63 74 2e 65 64 2d
64 69 61 6d 6f 6e 64 2e
63 6f 6d
```

7. BONUS NDEF

Le décodage et l'encodage des données au format NDEF sortent totalement du cadre du présent article et ne sont pas quelque chose qu'il est possible de traiter de manière concise (cf. cette page W3C pour comprendre **[10]**). Ce qu'il est possible de faire, en revanche, est de tout simplement s'en remettre à un outil tiers comme la libNDEF qui, même si elle n'est pas toute jeune (dernier *commit* il y a 7 ans), fournit deux outils fort pratiques : `ndef-decode` et `ndef-encode`.

La libNDEF est écrite en C++ et possède une dépendance vers Qt Core, ce qui n'est pas forcément



Le Proxmark 3 RDV4 est un outil incontournable dès lors que l'on souhaite explorer sérieusement les technologies RFID/NFC. C'est à la fois un outil de développement, une plateforme d'expérimentation, un analyseur de protocole et un équipement de pentest. Le tout avec une communauté de développeurs très active et passionnée.

très amusant. En revanche, le dépôt GitHub **[11]** contient les éléments permettant de créer proprement un paquet Debian/Ubuntu/Raspbian relativement simplement. Pour cela, récupérez les sources avec `git clone` puis pliez-vous simplement d'un `dpkg-buildpackage -b` pour créer les paquets que vous installerez directement avec `dpkg -i`. En cas de problème, vérifiez simplement que vous ayez bien installé toutes les dépendances de construction : `debhelper` (>= 7), `libqt4-dev`, et `qt4-qmake`.

Si vous voulez bien faire et étant donné que ces paquets n'installent pas par défaut les deux utilitaires, utilisez `dch -i` pour ajouter une entrée à `debian/changelog`, puis éditer le fichier `debian/libndef.install` pour ajouter une ligne et changer le contenu en :

```
usr/lib/lib*.so.*
usr/bin/ndef*
```

Reconstruisez les paquets (`libndef` et `libndef-dev`), installez, et vous voici avec le duo `ndef-decode`/`ndef-encode` parfaitement et proprement intégrés à votre système et en mesure d'encoder et décoder du NDEF très simplement. Ainsi, en couplant cela avec `xxd`, il est possible de traiter directement le contenu NDEF sous forme de valeurs hexadécimales :

```
$ echo "d1010c550173742e636f6d2f73743235" \
| xxd -r -p - | ndef-decode

Use stdin as input file
NDEF message is valid and contains 1 NDEF record(s).
NDEF record (1) type name format: NFC Forum well-known type
NDEF record (1) type: U
NDEF record (1) payload (uri): http://www.st.com/st25
```

et inversement :

```
$ ndef-encode -u http://www.st.com/st25 | xxd -p -
d1010c550173742e636f6d2f73743235
```

Vous pouvez également vouloir utiliser la bibliothèque elle-même pour vos développements (`libndef-dev`) à condition de tolérer la dépendance à Qt et, bien entendu le mélange C/C++...

CONCLUSION... DE L'ARTICLE

Un grand nombre de points n'ont pas été couverts dans cet article et en particulier l'authentification, la configuration de l'accès aux données, les procédures d'écriture, la configuration GPO (pour les ST25TA02KB-D et ST25TA02KB-P), etc. L'idée ici n'était pas

de détailler tout le code de ce qui va très certainement se transformer en un utilitaire de lecture (et/ou d'écriture) de tag ST25TA, mais d'aborder les concepts NFC au plus bas niveau et des APDU en particulier.

Nous avons vu qu'implémenter une telle solution, une fois les principes assimilés, tient en peu de choses et que la très grande majorité des informations est présente dans les *datasheets*. Prendre en charge un tag ne présente pas de difficulté particulière et ce que nous venons de voir avec les ST25TA est parfaitement transposable à n'importe quel autre produit, pour peu que le constructeur daigne diffuser les informations sans trop de restrictions.

Bien entendu, de la même manière que les caractéristiques spécifiques des MIFARE DESFire EV2 ne sont pas supportées par la libfreefare (uniquement ce qui est compatible avec EV1), il est peu probable qu'une solution pleinement *open source* puisse voir le jour pour les fonctionnalités spécifiques comme TruST25 ou Augmented NDEF, puisque les informations ne sont distribuées que sous NDA. La concurrence acharnée entre les fabricants explique peut-être cette pratique courante dans le domaine, sans parler du vol de technologie pure et simple de la part d'acteurs peu scrupuleux souvent hors de portée de la législation occidentale. Mais cela reste toutefois extrêmement frustrant d'un point de vue technique, sachant que ces intéressantes fonctionnalités sont à portée de main, mais pourtant inaccessibles... **DB**

RÉFÉRENCES

[1] **https://connect.ed-diamond.com/Hackable/hk-010**

[2] **https://connect.ed-diamond.com/Hackable/hk-030/proxmark-l-incontournable-materiel-pour-tester-la-securite-rfid-et-nfc**

[3] **https://github.com/nfc-tools/libfreefare**

[4] **https://www.st.com/resource/en/datasheet/st25ta64k.pdf**

[5] **https://www.st.com/resource/en/datasheet/st25ta16k.pdf**

[6] **https://www.st.com/resource/en/datasheet/st25ta02kb-d.pdf**

[7] **https://play.google.com/store/apps/details?id=com.st.st25nfc**

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

[9] **https://www.iso.org/standard/77180.html**

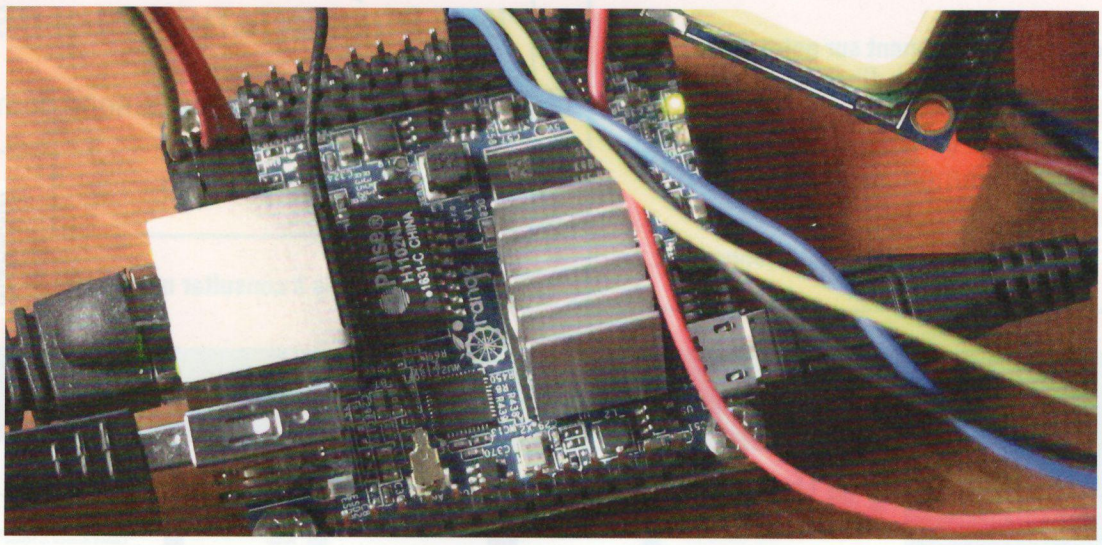
[10] **https://w3c.github.io/web-nfc/#the-ndef-record-and-fields**

[11] **https://github.com/nfc-tools/libndef**

MANIPULATION DE TRAMES WI-FI : EXPLORER LE MONDE DES BEACON FRAMES AVEC UN ESP8266

Denis Bodor

Pour faire connaissance avec une technologie, qu'elle soit nouvelle ou non, il y a généralement deux méthodes envisageables. Potasser sagement la documentation et étudier ce qui existe, ou avoir une approche plus empirique consistant à jouer et expérimenter au risque d'éventuellement « casser » quelque chose. Lorsqu'il est question de matériel, on y réfléchira à deux fois, mais s'il s'agit d'un protocole réseau, l'approche la moins ennuyeuse est évidente. Voyons cela avec les trames de gestion 802.11.



– Manipulation de trames Wi-Fi : explorer le monde des beacon frames avec un ESP8266 –

L'idée n'est pas nouvelle et il existe plusieurs implémentations directement réutilisables : envoyer des trames de gestion Wi-Fi pour annoncer la présence de points d'accès qui n'existent pas. L'intérêt pratique est relativement réduit, si ce n'est pour s'amuser à afficher une myriade de « réseaux » aux yeux éberlués de quiconque jetterait un coup d'œil. En vérité, l'objectif est purement pédagogique, puisqu'il permet de faire connaissance avec les mécanismes des protocoles Wi-Fi, les techniques corollaires, et les fonctionnalités peu connues de certains SDK.

Notre objectif pour cet article sera de créer un montage sur base ESP8266 (WeMos D1 Mini), envoyant des *beacon frames* forgées de toutes pièces et annonçant la présence d'une liste arbitraire de points d'accès aux SSID farfelus. Aucun composant particulier autre que la carte ESP8266 ne sera nécessaire, si ce n'est une Raspberry Pi, ou toute autre machine disposant d'un adaptateur Wi-Fi de qualité, nous permettant de vérifier effectivement ce qui est émis et de décortiquer les données.

On pourrait se contenter d'un smartphone équipé d'une application comme WiFi Analyzer **[1] [2]**, une application de diagnostic dont les sources sont disponibles sur GitHub **[3]**, mais le but n'est pas tant le fonctionnement effectif du projet que l'exploration du protocole lui-même. Nous avons donc besoin d'un outil permettant d'effectivement « voir » ce qui circule, d'analyser les données en question et de confirmer l'impact des changements du code.

1. UN MOT SUR ESP8266 BEACON SPAM ET NOTRE APPROCHE

Un projet similaire bien connu existe, reposant exactement sur la même plateforme, il s'agit de *ESP8266 Beacon Spam* de Stefan Kremser (alias spacehuhn **[4]**), également auteur de *ESP8266 Deauther* **[5]**. Comme le code à la base de cet article, *ESP8266 Beacon Spam* n'existe que comme chausse-pied pour qui voudrait se pencher sur le monde du *hacking* (au sens noble du terme) et s'intéresser en détail aux technologies Wi-Fi.

Mon propre code a initialement débuté sur la base de celui de Stefan, mais a rapidement évolué au point de réécrire la totalité, ou presque, de son `esp8266_beaconSpam.ino`. Son approche consiste à stocker les noms des SSID à annoncer dans un grand tableau de `char` stocké en flash et à composer, à la volée, les trames à envoyer avec la fonction `wifi_send_pkt_freedom()` interne au SDK ESP8266, et ce, en changeant le canal et l'adresse MAC aléatoirement. Deux modes de fonctionnement sont proposés, l'une avec des SSID de taille fixe (32 octets) et un *padding* avec des espaces, et l'autre en gérant les tailles variables des SSID.

Ici, les choses seront sensiblement différentes puisque nous cherchons à avoir davantage de contrôle. Nous créerons donc une trame « modèle » **par SSID** dont nous ne changerons que quelques éléments avant envoi. Ceci consomme davantage de RAM, mais contrairement au code de Stefan, le but n'est pas d'annoncer un nombre maximum de SSID, mais de le faire avec un ensemble plus réduit et en tentant de paraître plus légitime.

Et puis c'est une bonne excuse pour jouer avec des structures, des pointeurs et des allocations mémoires, histoire de se faire la main...

2. CRÉER SON « OUTIL » D'ANALYSE

Avant de nous attaquer au protocole et au code, nous devons créer notre outil de vérification et d'analyse. Une carte comme la Raspberry Pi se prête parfaitement à cet usage, mais n'importe quelle plateforme capable de faire fonctionner une paire d'outils dédiés sous GNU/Linux fera également l'affaire.

Pour surveiller ce qui se passe dans les airs, nous avons besoin d'un adaptateur Wi-Fi capable de fonctionner en **mode moniteur** sans trop de difficultés. Ce n'est pas le cas avec celui intégré aux Raspberry Pi, du moins avec le pilote/*firmware* par défaut **[6]**. La voie du moindre effort, adaptée à n'importe quelle espèce de framboises, ou de SBC en général, consiste donc à faire usage d'un adaptateur USB de qualité, de préférence réputé pour ses capacités de *sniffing*.

Vérifier les modes d'un adaptateur se fera très simplement via la commande `iw list` et en guettant ce qui se trouve à la section « *Supported interface modes* » de la sortie. Ce que vous voulez voir ressemble à ceci (ici un adaptateur Qualcomm Atheros AR9271) :

```
[...]
Supported interface modes:
        * IBSS
        * managed
        * AP
        * AP/VLAN
        * monitor      <------
        * mesh point
        * P2P-client
        * P2P-GO
        * outside context of a BSS
[...]
```

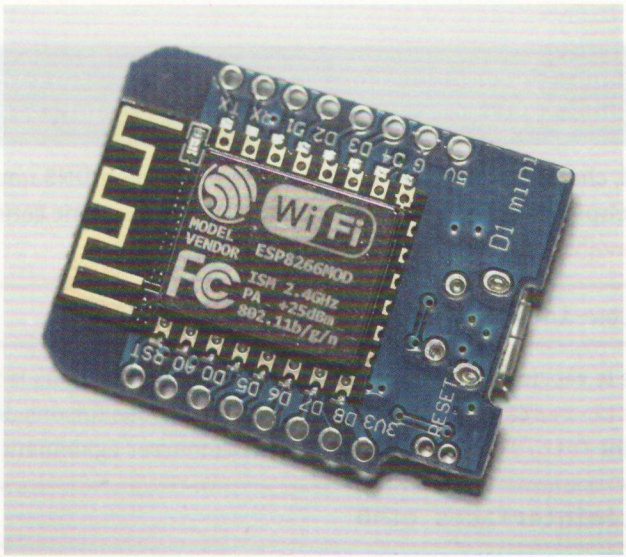
Et ce que vous ne voulez surtout pas voir est (ici l'interface d'une Pi 3B+) :

```
Supported interface modes:
        * IBSS
        * managed
        * AP
        * P2P-client
        * P2P-GO
        * P2P-device
```

Notez que le mode moniteur (*monitor mode*) est souvent confondu avec le *promiscuous mode*, alors que techniquement, il s'agit de deux choses différentes. Le mode moniteur est exclusif au Wi-Fi alors que le *promiscuous mode* fait référence à la capacité d'une interface réseau à accepter/traiter tous les paquets qu'elle reçoit, même s'ils ne lui sont pas destinés. Ce qui signifie que le mode *promiscuous* concerne un réseau auquel on est connecté, ce qui n'est pas le cas lorsqu'on veut recevoir des trames Wi-Fi. Le mode moniteur d'un adaptateur Wi-Fi est l'un des huit modes de fonctionnement définis par le standard 802.11, au même titre que le mode *Master* (AP), *Managed* (client), *Mesh* ou encore *Ad hoc* (IBSS).

Comme vous pouvez le voir, tous les adaptateurs (ou *chipsets*) Wi-Fi ne proposent pas le mode moniteur, mais c'est également une question de *firmware* que le *chipset* exécute. Les *chipsets* les plus connus pour proposer ce mode sont Atheros AR9271, Ralink RT3070, RT3572, RT5572, RT5370N, Realtek RTL8812AU et MediaTek MT7610U. Attention cependant, *chipsets* ne veux pas dire adaptateur et il n'est pas rare que certains constructeurs changent le *chipset* d'un modèle, tout

en continuant à utiliser la même dénomination. Pour choisir un adaptateur, si vous n'en possédez pas encore, les sites traitant du matériel recommandé pour la distribution Kali Linux sont généralement une excellente piste. Pour cet article, le matériel utilisé sera un adaptateur Alfa Network AWUS036NHA utilisant un *chipset* AR9271.



Ce genre de « module » ESP8266, clones de Wemos D1 mini, se trouve pour quelques euros sur les sites comme Banggood, eBay, AliExpress, etc. La qualité n'est pas toujours au rendez-vous, mais à ce prix, on peut facilement tolérer un faible pourcentage de rebut dans un lot.

Pour créer notre « équipement » d'analyse, nous partirons d'une installation fraîche de *Raspberry Pi OS Lite* en version la plus récente. Après transfert de l'image sur le support microSD, nous ajoutons un fichier `ssh` vide sur la partition FAT32 afin d'activer le serveur OpenSSH dès le premier *boot*, et éditons également les fichiers `/etc/hostname` et `/etc/hosts` du système de fichiers ext4 afin de personnaliser le nom d'hôte. Ce faisant, nous pouvons immédiatement nous connecter en SSH en utilisant `pi@<nomhote>.local` (et en profiter pour utiliser `ssh-copy-id` afin de simplifier les connexions ultérieures).

Une fois un shell obtenu, nous utilisons `raspi-config` pour configurer le système de façon tout à fait standard (locales, fuseau horaire, etc.), mettons à jour avec `apt-get upgrade`, installons quelques paquets utiles selon nos préférences personnelles (`bc`, `vim`, `bash-completion`, `mc`, `screen`, etc.), modifions/transférons nos configurations usuelles (`.bashrc`, `.inputrc`, `.vim*`, etc.) et enfin, installons ce dont nous aurons besoin pour la suite : `aircrack-ng` et `tshark`/`wireshark`.

Le choix de la version de l'analyseur, `tshark` ou `wireshark`, dépendra de la façon dont vous utiliserez la plateforme. TShark est la version en ligne de commande de l'outil et Wireshark est la version disposant d'une interface utilisateur en mode graphique, pour une utilisation directe sur la Pi donc. Une autre option consiste à installer Wireshark sur un autre système (Windows, *BSD, macOS, etc.). La capture avec `tshark` générera des fichiers au format PCAP qui pourront facilement être transférés, puis analysés n'importe où.

Ceci fait, il ne nous reste plus, ensuite, qu'à faire un brin de ménage, à commencer par la désactivation des adaptateurs Wi-Fi et Bluetooth embarqués sur la Pi 3 en éditant `/boot/config.txt` et en ajoutant deux petites lignes en fin de fichier :

```
dtoverlay=disable-bt
dtoverlay=disable-wifi
```

La commande `airmon-ng` sera utilisée pour activer le mode moniteur sur l'adaptateur connecté en USB, mais certains services peuvent, en accédant à l'interface, désactiver ce mode. Pour éviter qu'une telle chose se produise, nous devons désactiver ou reconfigurer ces services. *WPA supplicant* sera donc purement et simplement désactivé via les commandes :

```
$ sudo systemctl stop wpa_supplicant.service
$ sudo systemctl disable wpa_supplicant.service
```

Avahi, chargé entre autres choses de la résolution mDNS nous sera utile, mais il ne doit pas s'occuper de l'interface Wi-Fi. Nous lui signifions donc cette exigence en éditant `/etc/avahi/avahi-daemon.conf` et en ajoutant une ligne :

```
allow-interfaces=eth0
```

afin que le service exclue toutes autres interfaces, puis nous le redémarrons avec `sudo systemctl restart avahi-daemon.service`. Nous faisons de même avec le client DHCP, en éditant `/etc/dhcpcd.conf` avant de redémarrer également ce service :

```
allowinterfaces eth0
```

Vous pouvez ensuite éventuellement redémarrer le système pour avoir l'esprit tranquille et vous assurer que tout fonctionne correctement, même si cela n'est pas vraiment nécessaire. Nous pouvons alors vérifier la présence d'une interface Wi-Fi avec :

```
$ ifconfig -a
[...]
wlan0: flags=4098<BROADCAST,MULTICAST>  mtu 1500
  ether 00:c0:ca:92:3c:2d  txqueuelen 1000  (Ethernet)
  RX packets 0  bytes   0  (0.0 B)
  RX errors  0  dropped 0  overruns 0  frame   0
  TX packets 0  bytes   0  (0.0 B)
  TX errors  0  dropped 0  overruns 0  carrier 0  collisions 0
```

Puis utiliser `airmon-ng` pour lister plus clairement les interfaces utilisables et leur état :

```
$ sudo airmon-ng
PHY   Interface  Driver     Chipset
phy0  wlan0      ath9k_htc  Qualcomm Atheros Communications AR9271 802.11n
```

Tout à l'air en ordre et nous pouvons donc passer l'adaptateur en mode moniteur :

```
$ sudo airmon-ng start wlan0
Found 3 processes that could cause trouble.
Kill them using 'airmon-ng check kill' before putting
the card in monitor mode, they will interfere by changing channels
and sometimes putting the interface back in managed mode

  PID Name
  335 avahi-daemon
  349 avahi-daemon
  483 dhcpcd
```

```
PHY    Interface  Driver     Chipset

phy0   wlan0      ath9k_htc  Qualcomm Atheros Communications AR9271 802.11n
       (mac80211 monitor mode vif enabled  for [phy0]wlan0 on [phy0]wlan0mon)
       (mac80211 station mode vif disabled for [phy0]wlan0)
```

Notez que l'outil se plaint toujours de la présence de processus qu'il estime perturbateurs, mais nous avons effectivement déjà traité ce problème. À ce stade, une nouvelle interface à fait son apparition :

```
wlan0mon: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
  unspec 00-C0-CA-92-3C-2D-18-6C-00-00-00-00-00-00-00-00  txqueuelen 1000
  RX packets 5757  bytes 1505319 (1.4 MiB)
  RX errors  0  dropped 0  overruns  0  frame   0
  TX packets 0  bytes   0  (0.0 B)
  TX errors  0  dropped 0  overruns  0  carrier 0  collisions 0
```

`wlan0mon` pourra désormais être utilisée pour « écouter » le trafic Wi-Fi dans le voisinage. Vous pourrez immédiatement vérifier cela avec `sudo airodump-ng wlan0mon` qui devrait vous afficher très rapidement la liste des points d'accès à proximité, de leurs clients, et bien d'autres informations. Vous pouvez également ajuster certains paramètres, comme le canal avec `--channel` (ou `-c`) ou encore utiliser `--write` (ou `-w`) suivi d'une base de nom de fichier pour enregistrer le trafic au format PCAP.

On préférera cependant utiliser directement `tshark -i wlan0mon -c 500 -w essai.pcap`, par exemple, pour capturer 500 paquets et les stocker dans `essai.pcap` pour analyse, `airodump-ng` étant, en premier lieu, destiné à une utilisation avec `aircrack-ng` pour les tests de sécurité. Notez que Kismet (paquet éponyme) pourra également être intéressant dans une certaine mesure, bien qu'il s'agisse d'un outil relativement lourd et peu adapté à la tâche qui nous incombe.

Pour quitter le mode moniteur, vous pourrez utiliser la commande `sudo airmon-ng stop wlan0mon`.

3. ENTRONS DANS LE CODE

3.1 Structure du code

Classiquement, en utilisant le *framework* Arduino, le code se répartit entre les fonctions `setup()` et `loop()`. La première sera celle où nous configurerons l'ESP8266 (GPIO, communication série, Wi-Fi, etc.) et où nous créerons les trames de base propres à chaque point d'accès simulé. `loop()` sera chargé d'envoyer ces trames après en avoir sensiblement modifié le contenu si nécessaire, et ce, de façon répétée toutes les 100 ms par défaut.

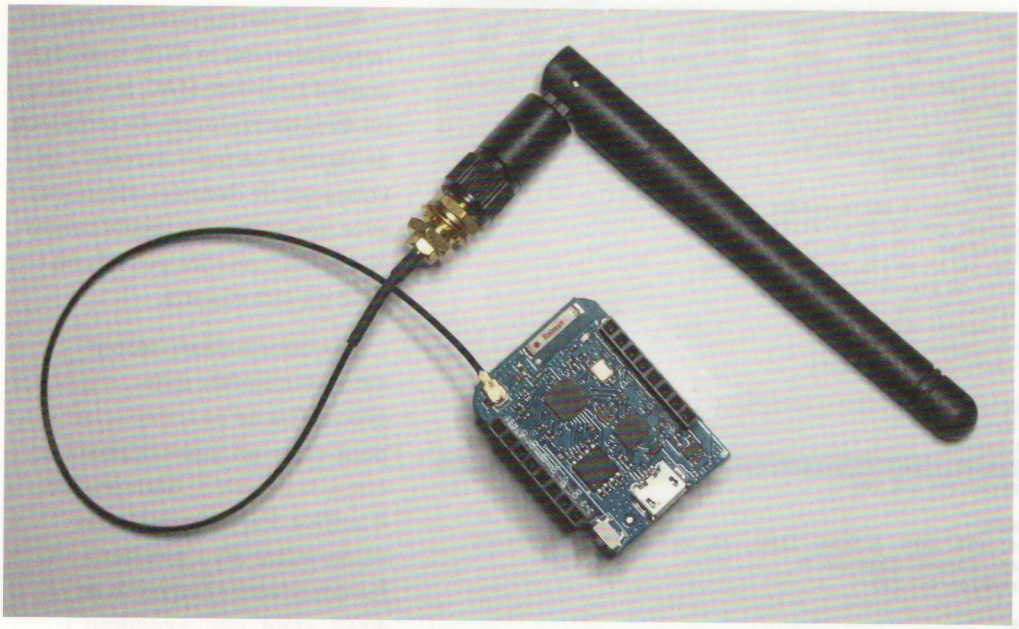
Pour forger les trames, nous utiliserons, comme Stefan Kremser, un tableau de `char` formant une trame par défaut totalement générique, mais contenant tous les éléments fixes. Ce `uint8_t beaconPacket[83]`, réduit à sa plus simple expression, se présentera comme une variable globale, ainsi :

```
uint8_t beaconPacket[83] = {
  // MAC header octets 0 à 23
  0x80, 0x00,                                 // Type/Sous-type: beacon frame
  0x00, 0x00,                                 // Duration
  0xFF, 0xFF, 0xFF, 0xFF, 0xFF, 0xFF,         // Destination (broadcast)
  0x01, 0x02, 0x03, 0x04, 0x05, 0x06,         // Source
  0x01, 0x02, 0x03, 0x04, 0x05, 0x06,         // BSSID
  0x00, 0x00,                                 // Fragment & numéro séquence

  // 24 à 31 horodatage - non utilisé
  0x83, 0x51, 0xf7, 0x8f, 0x0f, 0x00, 0x00, 0x00,
  0x78, 0x00,                                 // 32 a 33 Intervalle
  0x21, 0x00,                                 // 34 à 35 capabilities
                                              // ESS(AP) + short preamble

  // Tags:
  //   36 à 69 SSID
  0x00, 0x20,                                 // Tag SSID, longueur 32
  0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
  0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
  0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,
  0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20, 0x20,

  //   70 à 79 Débits
  0x01, 0x08,                                 // Tag: Rates, longueur 8
  0x82, 0x84, 0x8b, 0x96,                     // 1, 2, 5.5, 11
  0x24, 0x30, 0x48, 0x6c,                     // 18, 24, 36, 54

  //   80 à 82 Canal
  0x03, 0x01,                                 // Tag Channel, longueur 1
  0x01                                        // numéro canal
};
```

Un *beacon frame* respecte un format prédéfini avec certaines parties obligatoires et d'autres optionnelles, et se compose de données de taille fixe et d'autres de taille variable. Nous pouvons diviser sommairement notre trame en deux, avec tout d'abord la partie MAC (*Medium Access Control*) indiquant le type de trame, la destination, la source, l'émetteur et le numéro de séquence. Ici, la destination sera toujours FF:FF:FF:FF:FF:FF afin de spécifier une diffusion *broadcast*, et la source et l'émetteur seront le BSSID du point d'accès que nous simulons. Le numéro de séquence est une valeur incrémentée permettant de donner une chronologie aux trames, nous pouvons gérer cela nous-mêmes ou laisser le *framework* Espressif s'en charger.

Vient ensuite la partie de taille variable composée de tags débutant par une valeur en décrivant leur type, suivi d'une taille, puis des données en question. Nous avons ici trois tags :

- le SSID du point d'accès d'une longueur de 32 octets. C'est la taille maximum d'un SSID et notre trame de base précise un SSID simplement composé de 32 espaces ;
- les débits (*supported rates*) indiquant les différentes vitesses auxquelles notre point d'accès peut travailler ;
- le canal précisant, tout naturellement, le canal ou les canaux Wi-Fi utilisés par le point d'accès entre 1 et 13 (en Europe).



Ce Wemos D1 minipro (soi-disant) présente un intérêt tout particulier en proposant une antenne céramique intégrée ou la connexion d'une antenne externe (avec un meilleur gain) via un connecteur U.FL.

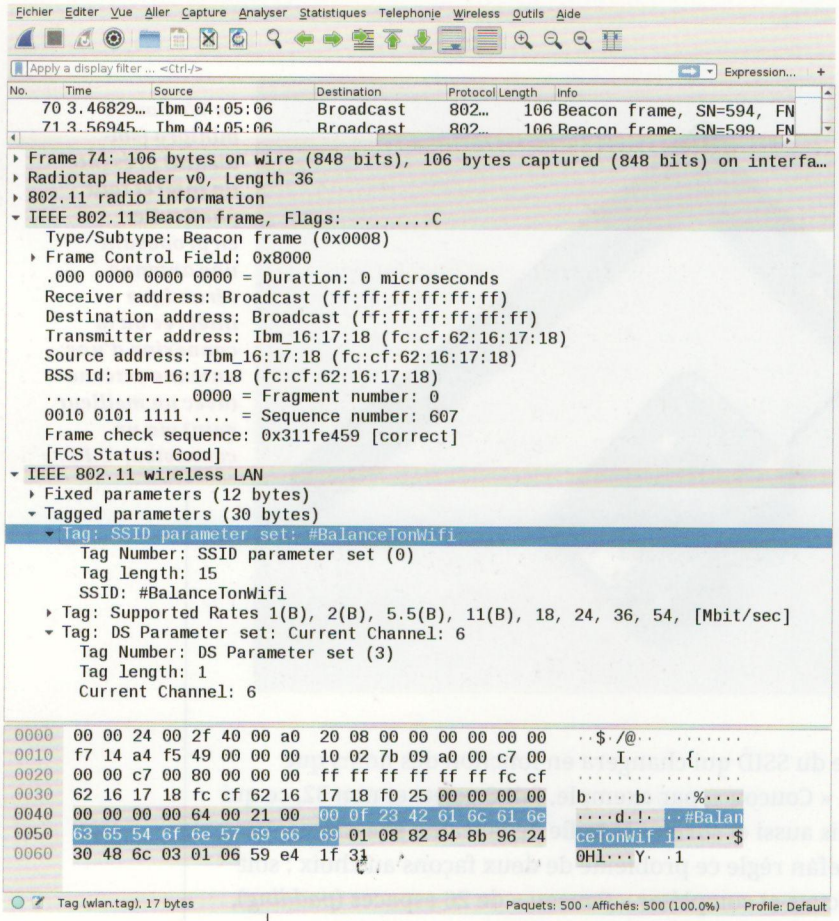
La difficulté ici réside dans la taille du SSID qui changera en fonction des noms que nous donnerons à nos points d'accès. « Coucou », par exemple, fait 6 octets et non 32, ce qui change la position du tag suivant, mais aussi et surtout la taille de la trame elle-même. Le code de l'*ESP8266 Beacon Spam* de Stefan règle ce problème de deux façons au choix : soit conserver un SSID de 32 caractères/octets et compléter « Coucou » de 26 espaces (*padding*), soit créer un *buffer* temporaire dans `loop()` et le remplir à coup de `memcpy()`.

Notre approche sera différente, puisqu'il s'agit de créer une trame pour chaque point d'accès. Pour cela, nous nous baserons sur un tableau initial regroupant les informations structurées de la façon suivante :

```
struct fakeap_t {
    char *ssid;
    uint8_t macAddr[6];
    uint8_t channel;
    uint16_t seqnum;
};
```

Et donc, par exemple :

```
struct fakeap_t fakeap[] = {
  { (char *)"Pizza",    { 0xfc, 0xcf, 0x62, 0x04, 0x05, 0x06},  4, 0 },
  { (char *)"Souriez",  { 0xfc, 0xcf, 0x62, 0x0a, 0x0b, 0x0c},  2, 0 },
  { (char *)"Oups",     { 0xfc, 0xcf, 0x62, 0x10, 0x11, 0x12},  2, 0 },
  { (char *)"#Wifi",    { 0xfc, 0xcf, 0x62, 0x16, 0x17, 0x18},  6, 0 },
  { (char *)"Coucou",   { 0xfc, 0xcf, 0x62, 0x1c, 0x1d, 0x1e}, 11, 0 }
};
```



Wireshark est l'outil par excellence pour inspecter ce qui est effectivement transmis par notre code. Ici, l'une des trames en question où l'on retrouve toutes nos informations décodées dans un format parfaitement lisible.

Nous déterminons et stockons non seulement le SSID, mais également l'adresse MAC, le canal et un éventuel numéro de séquence, de façon à ne pas « arroser » le voisinage de données aléatoires avec des annonces incohérentes où les SSID ne sont pas liés au MAC ou aux canaux.

Ces informations seront alors combinées, dans `setup()`, avec celles de `beaconPacket[]` pour créer un autre tableau de `char`, `fakepkt[]`, dans lequel il suffira d'ajuster quelques octets avant d'y « piocher » les données à envoyer avec `wifi_send_pkt_freedom()`.

Notez que les 4 octets FCS (*Frame Check Sequence*) normalement présents en fin de trames ne sont pas spécifiés, car automatiquement calculés et ajoutés par le SDK.

3.2 Composition des trames par AP

Nous allons ici faire un petit raccourci peu économe en mémoire. En effet, nous devons stocker une trame type pour chaque point d'accès (AP), mais les SSID de ces derniers n'ont pas des tailles identiques. Nous pourrions stocker chaque trame dans une mémoire allouée spécifiquement, puis stocker pointeur et taille dans un autre tableau (voir une liste chaînée), sauf que cela prendrait des proportions sans rapport avec l'objet de la réalisation (mais c'est un bel exercice). Le raccourci du développeur feignant consiste donc, tout simplement, à allouer la mémoire pour *n* points d'accès en se basant sur la taille de `beaconPacket[]` :

```
// globales
uint8_t *fakepkt;
int nbrap;

[...]

// nombre d'AP
nbrap = sizeof(fakeap)/sizeof(struct fakeap_t);
```

```
// allocation
if((fakepkt = (uint8_t *)malloc(sizeof(beaconPacket) * nbrap)) == NULL) {
  Serial.printf("malloc fakepkt error: %s\n", strerror(errno));
} else {
  Serial.printf("%u bytes allocated\n", sizeof(beaconPacket) * nbrap);
}

// zéro
memset(fakepkt, 0x00, sizeof(beaconPacket) * nbrap);
```

Nous sommes ainsi assurés que, quelle que soit la taille du SSID, `fakepkt` pointe sur une quantité de mémoire suffisante. Il ne reste plus, ensuite, qu'à remplir tout cela :

```
for(int i=0; i < nbrap; i++) {
  // base pour l'AP
  int pos = i*sizeof(beaconPacket);

  // copie première partie
  memcpy(fakepkt+pos, beaconPacket, 35);

  // changement MAC source
  memcpy(fakepkt+pos+10, &fakeap[i].macAddr, 6);

  // changement MAC BSSID
  memcpy(fakepkt+pos+16, &fakeap[i].macAddr, 6);

  // ajustement taille SSID
  *(fakepkt+pos+37) = strlen(fakeap[i].ssid);

  // ajout "chaîne" SSID
  memcpy(fakepkt+pos+38, fakeap[i].ssid, strlen(fakeap[i].ssid));

  // copie du reste de la trame
  memcpy(fakepkt+pos+38+strlen(fakeap[i].ssid),
      beaconPacket+70, 13 );

  // ajustement canal
  *(fakepkt+pos+sizeof(beaconPacket)+strlen(fakeap[i].ssid)-32-1)
      = fakeap[i].channel;
}
```

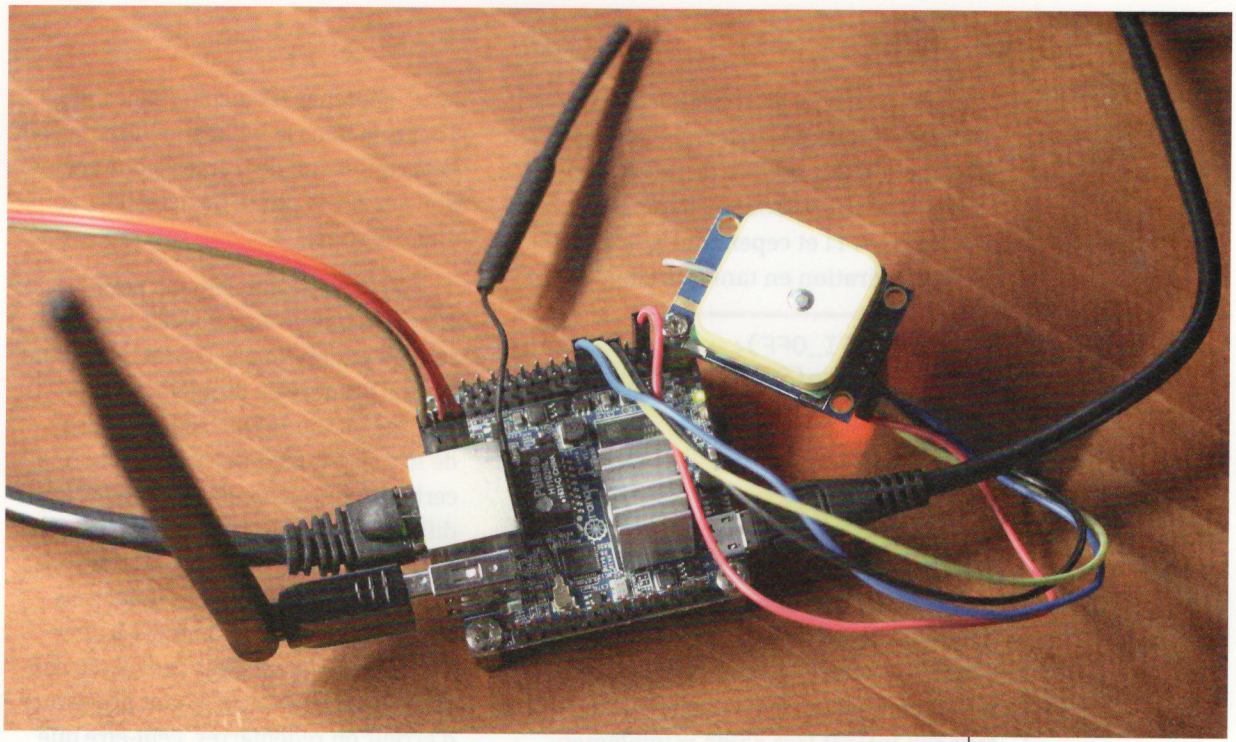
Il s'agit ici avant tout de jonglages avec des pointeurs puisque nous nous promenons dans la mémoire en nous basant sur `pos` et `i`, contenant respectivement la position dans `fakepkt[]` à laquelle nous copions les données et l'index sur le tableau de `struct` contenant les informations des points d'accès. Notez que lors de l'ajustement de la taille de la chaîne de caractères (qui n'en est plus une ensuite) représentant le SSID, nous devons déréférencer le pointeur pour spécifier la valeur, et utiliser `strlen()` qui retourne la bonne taille, **sans le `\0` de fin**. Nous procédons de même pour le numéro du canal, repris directement depuis le tableau `fakeap[]`.

Nous allouons davantage de mémoire que nous n'en avons besoin et notre `fakepkt[]` s'en trouve bourré de `0x00` sans grand intérêt. Mais cela nous facilite le travail lorsqu'il s'agit d'envoyer les trames. Tout ce que nous avons à faire est de savoir où commencer (ce qui est fonction du numéro de l'AP multiplié par la taille de `beaconPacket[]`), et de spécifier la taille des données, qui n'est autre que celle de `beaconPacket[]`, moins 32, plus la taille du SSID (`strlen()`).

3.3 Envoyer les trames

Une fois notre lot de trames composé, il nous suffit de parcourir la masse de données et de boucler pour tout envoyer en lots :

```
#define INTERVAL ((int16_t)100)
#define AUTOSEQ  true

void loop()
{
  currentTime = millis();

  if(currentTime - previousTime > INTERVAL) {
    previousTime = currentTime;

    int i = 0;
    // on boucle sur les AP
    while(i < nbrap) {
      // position dans le buffer
      int pos = i*sizeof(beaconPacket);

      // configuration canal
      wifi_set_channel(fakeap[i].channel);
      delay(2);

      // incrémentation sequence number
      if(!AUTOSEQ) {
        *(fakepkt+pos+22) = (fakeap[i].seqnum << 4);
        *(fakepkt+pos+23) = (fakeap[i].seqnum << 4) >> 8;
      }

      // envoi
      if(wifi_send_pkt_freedom(fakepkt+pos,
          sizeof(beaconPacket)+strlen(fakeap[i].ssid)-32,
          AUTOSEQ) == 0) {
        if(!AUTOSEQ) fakeap[i].seqnum++;
      }
      delay(1);
      i++;
    }
  }
}
```



La Raspberry Pi n'est pas le seul SBC utilisable ni la seule à proposer un chipset difficile à basculer en mode moniteur. Voici une Orange Pi Zero v1.1, très compacte, en phase de transformation en outil d'analyse Wi-Fi portable équipée, en prime, d'un récepteur GPS.

La seule subtilité réside dans `AUTOSEQ` puisque cette macro conditionne l'incrémentation par le SDK ou par notre propre code. Ceci est non seulement le troisième argument de `wifi_send_pkt_freedom()`, mais également la condition du changement de deux octets dans la trame. Notez les décalages nécessaires pour créer ces valeurs, provenant du fait que les octets 22 et 23 ne représentent pas directement une valeur 16 bits.

C'est là précisément tout l'intérêt d'utiliser Wireshark pour analyser effectivement les données et comprendre de quoi il retourne. Un numéro de séquence d'une trame comme 607 (`0x025f` en hexa), par exemple, et sans fragmentation (`0`) est représenté par `0xf0` et `0x25`, soit en binaire `1111 0000` et `0010 0101`, car les 4 bits de poids faible correspondant au fragment doivent être écartés et comme nous sommes en *little endian*, la valeur obtenue réagencée.

3.4 Quelques autres points intéressants

La fonction `wifi_send_pkt_freedom()` n'est pas directement accessible depuis le *framework* Arduino. Pour pouvoir l'utiliser, nous devons inclure à notre code un *header* du SDK écrit en C, et donc spécifier que la fonction est définie par ailleurs et utilise la convention d'appel du langage C :

```
extern "C" {
  #include "user_interface.h"
  int wifi_send_pkt_freedom(uint8* buf, int len, bool sys_seq);
}
```

De plus, ce type d'utilisation peu conventionnelle nécessite que la « radio » Wi-Fi ne soit pas utilisée de façon standard par le SDK/ESP8266. Il faut donc arrêter le Wi-Fi et cependant spécifier un mode d'opération en tant que client :

```
WiFi.mode(WIFI_OFF);
wifi_set_opmode(STATION_MODE);
```

4. POUR FINIR

Ce projet n'a pas réellement d'usage pratique, si ce n'est une blague bon enfant à faire en soirée ou chez des amis. Il ne vaut même pas vraiment la peine d'être utilisé comme tel, mais révèle pourtant tout son intérêt lors d'une réimplémentation. Tout l'attrait de l'exercice consiste précisément à choisir une façon d'architecturer le code (et les données), puis de l'implémenter en observant les changements physiquement dans les trames capturées avec Tshark. Le tout en butant sur les classiques problématiques de gestion mémoire, d'arithmétique des pointeurs et de manipulations de bits.

Le présent code, en version complète, est à disposition sur le GitHub du magazine, mais je vous invite à uniquement vous en inspirer, en parallèle à celui de Stefan Kremser, pour implémenter votre propre version avec vos propres spécificités. Le *timestamp* peut ainsi être amusant à traiter puisqu'il est ici totalement ignoré, mais mieux encore, il peut être très instructif de chercher des solutions de contre-mesure et de détection : vérifier la chronologie des séquences, repérer la récurrence de certaines adresses MAC, chercher les incohérences, etc. Tout cela dans le but d'améliorer le code et la confection de trames, pour ensuite boucler sur la phase de détection, et ainsi de suite...

Je préciserai simplement, pour conclure, que ce type d'usage n'est clairement pas prévu par Espressif et qu'un certain nombre de problèmes apparaissent lorsque l'ESP8266 est utilisé de manière aussi intensive durant une certaine période. Très rapidement, les différentes plateformes testées (Wemos, NodeMCU, ESP8266 Thing, etc.) se sont mises à monter en température, puis après une dizaine de minutes, à devenir très irrégulière dans l'envoi effectif des trames, sans pour autant provoquer d'erreur ou redémarrer. Peut-être que ce problème n'existe pas sur ESP32 et ses dérivatifs (S3, S3, C3), mais je vous laisserai le soin de vous en assurer... **DB**

Voici le résultat de nos expérimentations divertissantes. Ces points d'accès n'existent pas et non, le standard 802.11-2012 (Section 6.3.11.2.2) ne vous interdit absolument pas d'utiliser des caractères UTF-8 pour vos SSID (mais il ne faut pas dépasser les 32 octets).

RÉFÉRENCES

[1] **https://play.google.com/store/apps/details?id=com.vrem.wifianalyzer**

[2] **https://f-droid.org/packages/com.vrem.wifianalyzer/**

[3] **https://github.com/VREMSoftwareDevelopment/WiFiAnalyzer**

[4] **https://github.com/spacehuhn/esp8266_beaconSpam**

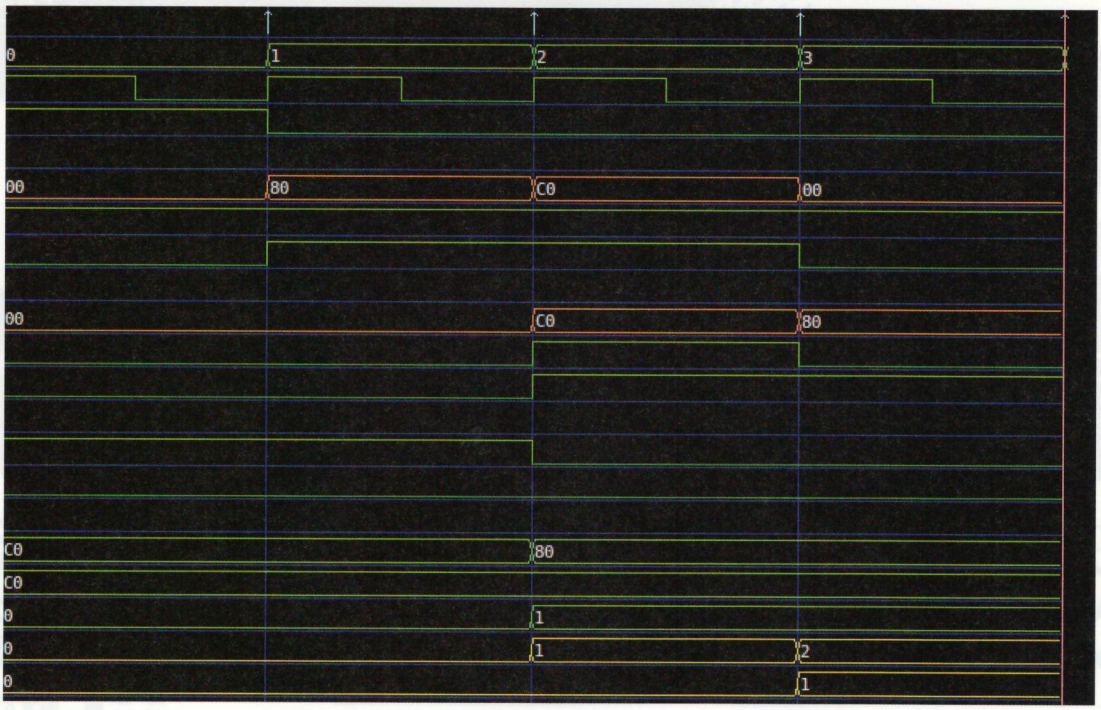
[5] **https://github.com/SpacehuhnTech/esp8266_deauther**

[6] **https://github.com/seemoo-lab/nexmon**

DE LA PREUVE FORMELLE EN VHDL, LIBREMENT

Fabien Marteau - Front de libération des FPGA

Dans cet article, on se propose d'aborder la méthode de vérification formelle pour le VHDL. Cette méthode a récemment été rendue possible avec GHDL et Yosys grâce au projet d'extension ghdl-yosys-plugin qui fait le lien entre les deux logiciels. Nous allons également découvrir le langage PSL (*Properties Specification Langage*) qui permet de décrire efficacement les propriétés utilisées en preuve formelle. Le support du PSL ayant été ajouté dans GHDL, il sera possible de l'utiliser librement en VHDL.



– De la preuve formelle en VHDL, librement –

Les FPGA sont très utilisés dans des domaines industriels critiques comme l'aéronautique, l'espace, le médical ou le ferroviaire. Ces domaines requièrent des méthodes de conception très exigeantes quant à la qualité du code produit. En plus de relectures par des tiers et le respect du cycle en V, il est nécessaire de simuler les systèmes et d'assurer une bonne couverture de tests. On pourra se référer au standard **DO-254 [1]** utilisé dans l'aéronautique, par exemple. La preuve formelle s'inscrit dans ces méthodes permettant de valider profondément le fonctionnement d'un système. Le principe consiste à décrire des propriétés au moyen d'assertions (`assert()`) et de lancer un « moteur de preuve » qui tentera de nous prouver qu'il peut les faire échouer.

Contrairement aux tests classiques, le cheminement qui nous mène à faire échouer une assertion n'est pas écrit par un humain. C'est la machine qui va parcourir l'ensemble des chemins possibles pour prouver que les propriétés que nous avons écrites sont vraies.

C'est une méthode très puissante à utiliser au cours du développement, il n'est généralement pas nécessaire de faire de longues simulations pour arriver aux comportements qui nous intéressent. Et comme la machine teste tous les chemins possibles, elle va souvent passer par des chemins que l'on n'avait pas envisagés et trouver des bugs bien enfouis.

Plutôt que de se laisser surprendre par un bug avec une simulation classique, avec la preuve formelle, on va plutôt « partir à la chasse aux bugs » et tenter de débusquer les plus retors.

Cette méthode de test a longtemps nécessité l'achat de licences très onéreuses réservant les outils aux grosses sociétés. Mais, dans un précédent article de Hackable **[2]**, nous avons vu qu'il était possible de faire de la preuve formelle en **Verilog**. Nous utilisions pour cela le logiciel de synthèse **Yosys** couplé à **Yosys-SMTBMC**. C'est ce même logiciel que nous allons utiliser ici.

Dans l'« industrie du FPGA », le Verilog n'est pas le seul langage disponible, en effet le **VHDL** est très utilisé en Europe. Jusqu'à récemment, il n'existait pas de logiciels libres permettant de faire de la preuve formelle en VHDL, jusqu'à l'apparition de l'extension nommée **ghdl-yosys-plugin [3]**.

Cette extension va nous permettre de profiter des avancées du logiciel libre de simulation **GHDL** avec le support (partiel) du **VHDL-2008** et surtout, le support du langage **PSL** qui simplifie grandement la description des propriétés.

Dans cet article, nous allons commencer par l'installation et la présentation des outils. Puis nous décrirons les principes utilisés et nous terminerons par un exemple concret de conception d'une **FIFO** en VHDL.

1. LES OUTILS

Pour faire de la preuve formelle en VHDL, nous allons avoir besoin des logiciels suivants :

- **Yosys** : le logiciel couteau suisse de la synthèse Verilog ;
- **SymbiYosys** : une interface permettant de scripter Yosys et les différents moteurs de preuve ;
- **Yices2** : le moteur de preuve par défaut de Yosys ;
- **GHDL** : le logiciel libre de référence pour la simulation (et synthèse) VHDL ;

- **ghdl-yosys-plugin** : l'extension Yosys qui fait le lien entre GHDL et Yosys pour faire de la synthèse VHDL.

Et pour visualiser les traces **VCD** générées par le moteur de preuve, nous aurons également besoin de **GTKWave**.

1.1 Yosys, le couteau suisse du gateware

Yosys est un logiciel de synthèse Verilog. Il est préférable de partir des sources pour l'installer :

```
$ git clone https://github.com/YosysHQ/yosys.git yosys
$ cd yosys
$ make -j$(nproc)
$ sudo make install
```

Attention au temps de compilation, qui peut être vraiment long si votre ordinateur prend de l'âge (avec un petit *netbook* Celeron, cela m'a pris presque deux heures).

1.2 SymbiYosys

SymbiYosys est un système de scripts pour piloter les différents logiciels pour faire de la preuve formelle. La configuration du projet se fait au moyen d'un script au format `.sby`.

L'installation à partir des sources est très simple et rapide :

```
$ git clone https://github.com/YosysHQ/SymbiYosys.git SymbiYosys
$ cd SymbiYosys
$ sudo make install
```

1.3 Yices2

Yices2 est le moteur de preuve utilisé par défaut par SymbiYosys, pour l'installer à partir des sources, nous ferons :

```
git clone https://github.com/SRI-CSL/yices2.git yices2
cd yices2
autoconf
./configure
make -j$(nproc)
sudo make install
```

1.4 GHDL, la référence en simulation VHDL libre

Pour le moment, nous n'avons installé que des logiciels « orientés Verilog ». Mais comme nous voulons travailler sur du VHDL, nous allons avoir besoin d'un logiciel capable de lire le langage. GHDL est le logiciel libre le plus abouti en matière de simulation VHDL, il supporte les standards VHDL 1987, 1993, 2002 et même une partie du VHDL-2008. C'est ce dernier standard avec le support PSL qui va d'ailleurs nous intéresser pour la preuve formelle.

Le support du standard VHDL-2008 étant en cours de développement, il est fortement conseillé d'installer et de compiler GHDL à partir du dépôt à jour. La procédure d'installation depuis les sources est donnée dans la documentation officielle **[4]** et requière l'installation du compilateur **Ada GNAT**, en plus du compilateur **GCC** habituel. Car oui, puisque le VHDL ressemble fortement à l'Ada, autant écrire le compilateur dans ce langage :

```
$ sudo apt install gnat
$ git clone https://github.com/ghdl/ghdl.git
$ cd ghdl
$ ./configure
$ make
$ sudo make install
```

Les sources VHDL seront analysées grâce à GHDL puis nous passerons à Yosys pour la « synthèse » du *gateware* qui alimentera ensuite le moteur de preuve formelle.

1.5 GTKWave, des vagues en libre accès

GTKWave est le logiciel libre de visualisation des chronogrammes au format `VCD`. GTKWave supporte d'autres formats plus spécifiques à un HDL comme le `FST`, `LXT` ou `GHW`. C'est un logiciel de référence qui est aujourd'hui considéré comme stable. Il n'est pas nécessaire de l'installer depuis les sources, le paquet de la distribution fera parfaitement l'affaire.

GTKWave va nous servir à visualiser les traces générées par le moteur de preuve quand il trouve des failles dans nos assertions. Nous pourrons également visualiser les traces de « couvertures » générées par le mode `cover`.

1.6 ghdl-yosys-plugin

ghdl-yosys-plugin est l'extension de Yosys qui fait le lien avec GHDL. Elle consiste en une bibliothèque `.so` que l'on va charger en lançant Yosys, comme nous le verrons plus tard. C'est un projet en cours de développement, mais qui est suffisamment avancé pour être parfaitement utilisable :

```
$ git clone https://github.com/ghdl/ghdl-yosys-plugin
$ cd ghdl-yosys-plugin
$ make
```

La compilation génère un fichier nommé `ghdl.so` qu'il suffira de charger sous forme de module avec l'option `-m` de Yosys :

```
$ yosys -m ghdl.so
yosys> ghdl
1. Executing GHDL.
error: no top unit found
ERROR: vhdl import failed.

yosys>
```

Maintenant que tous les outils sont installés, entrons dans le vif du sujet de la preuve formelle.

2. QU'EST-CE QUE LA MÉTHODE FORMELLE ET LE PSL

L'objectif de la méthode formelle est de s'assurer que le module fonctionne. Pour cela, le développeur ou la développeuse décrit le comportement attendu avec des propriétés.

Ces propriétés sont données au moyen des fonctions `assert` et `assume`. `assert` affirme qu'une propriété doit être vraie (valeur booléenne). `assume` restreint l'espace de validité de la simulation, elle « assume » que la propriété est vraie.

Ces propriétés peuvent tout à fait être utilisées dans le cadre d'une simulation classique. Dans ce cas, les deux fonctions `assert` et `assume` auront le même comportement : elles lèveront une erreur si la propriété est fausse à un moment de la simulation. Si l'on souhaite tout de même restreindre sans générer d'erreurs en simulation, on utilisera `restrict` à la place de `assume`.

Mais le grand intérêt de ces fonctions est de faire tourner un moteur de preuve formelle pour que la machine « cherche d'elle-même les bugs », sans que l'on ait à écrire de fastidieux stimuli.

Il existe plusieurs moteurs de preuve libre, dont beaucoup sont utilisables avec Yosys-SMTBMC. Pour cet article, nous utiliserons **Yices**, qui en est le moteur par défaut.

Ces moteurs ont deux modes de fonctionnement pour prouver le fonctionnement, et un troisième utilisé pour atteindre (couvrir) un état, qui n'est pas à proprement parler un mode de « preuve » :

- le mode **BMC** (*Bounded Model Checking*) : c'est le mode que l'on utilisera généralement en premier. Dans ce mode, on part de l'état initial du module et l'on affirme que toutes les propriétés sont vraies au bout d'un certain nombre de cycles d'horloge donné en paramètres ;
- le mode **prove** ou *k-Induction* (Preuve par induction) : dans ce mode, le moteur va partir de n'importe quel état valide et parcourir un certain nombre de cycles donné en paramètre pour prouver que l'on reste toujours dans un état valide du système ;
- le mode **cover** : ce mode est très pratique pour visualiser le comportement de son module. Le moteur de preuve va tenter d'atteindre la propriété en partant de l'état initial. Il s'arrêtera à la première « trace » qui validera la propriété décrite avec `cover`.

Yosys-SMTBMC est un logiciel conçu initialement pour le langage Verilog. En Verilog, nous utilisions les fonctions `assert()`, `assume()`, `restrict()` et `cover()` directement dans le code Verilog. Comme ces fonctions font partie du langage **SystemVerilog**, elles sont généralement encadrées par des macros `ifdef` qui permettent de désactiver le code pour la synthèse. La plupart des logiciels de synthèse ne supportent pas bien le SystemVerilog, d'où la nécessité de le désactiver quand il est inutilisé.

Pour plus de précisions sur les modes `BMC`/`prove`/`cover`, le lecteur pourra aller visionner la conférence de W.Clifford **[5]**, qui n'est autre que l'auteur de Yosys et Yosys-SMTBMC.

En VHDL nous allons écrire les propriétés dans un fichier à part au format `.psl`. Le PSL (pour *Property Specification Language*) est un langage défini par **Accellera [6]**. Le PSL n'est pas spécifique au VHDL, il est possible de l'utiliser en Verilog également. On prendra bien garde cependant quand nous définirons des signaux intermédiaires, ils seront eux définis dans le langage cible (donc, du VHDL dans notre cas).

ACCELLERA

Accellera est un consortium de grosses entreprises opérant dans le domaine de la microélectronique comme ARM, Cadence, Freescale, Mentor, Siemens, Synopsis, STMicroelectonic... Ce consortium développe et discute des standards et formats utilisés pour la conception numérique. Certains de ces standards sont ensuite ratifiés par l'IEEE (*Institute of Electrical and Electronic Engineers*) comme le Verilog ou le VHDL, mais également le PSL, le SystemVerilog ou le SystemC. Tous ces standards sont accessibles librement et ont pour but d'être supportés par la plupart des outils. Mais en réalité, rares sont les formats vraiment bien supportés par l'ensemble des outils utilisés en microélectronique. Il n'est jamais facile de passer un projet d'un outil constructeur à l'autre.

Les propriétés PSL sont généralement décrites dans un fichier à part avec l'extension `.psl` et contenant un bloc `vunit` :

```
vunit NOM_INSTANCE(NOM_ENTITÉ(NOM_ARCHITECTURE)) {
    -- les propriétés
}
```

Les directives ont la forme suivante :

```
f_label : assert always {empty_o = '0'} @rising_edge(clk);
```

Que l'on décompose comme suit :

- `f_label` : une chaîne de caractères qui nomme la directive. Très utile pour s'y retrouver dans les messages d'erreurs ;
- `assert` : la directive qui peut être `assert`, `assume`, `restrict` ou `cover` ;
- `always` : le mot clef `always` est un « opérateur d'occurrence », il signifie que la propriété doit être vraie sur tous les cycles d'horloge. Si l'on souhaite vérifier une propriété à l'étape initiale, il suffira de l'omettre ;
- `{empty_o = '0'}` : la condition booléenne à vérifier. Notez que la propriété à vérifier est ici un peu stupide, puisqu'on demande au signal `empty_o` d'être toujours à « 0 » ;
- `@rising_edge(clk)` : l'horloge utilisée pour les cycles.

Généralement, le module testé ne possède qu'une seule horloge. Pour éviter d'avoir à la rappeler dans chaque directive avec `@rising_edge(clk)`, on ajoutera l'horloge par défaut en début de code :

```
default clock is rising_edge(clk);
```

Cette directive valide un état pendant un cycle d'horloge, il est possible d'utiliser un opérateur d'implication pour indiquer le déclencheur de la condition. C'est une autre manière d'utiliser un `if`. Il existe deux opérateurs d'implication :

- **trigger** |-> **condition** : la condition doit-être vérifiée sur le même cycle d'horloge que celui qui l'a déclenché ;
- **trigger** |=> **condition** : la condition doit-être vérifiée un cycle après le déclenchement.

Voici une directive qui illustre l'opérateur d'implication :

```
f_reset : assert {rst = '1'; rst = '0'} |-> {empty_o = '1'};
```

Nous n'avons pas mis l'horloge ici, car on suppose qu'elle a été définie en début de bloc. Cette directive affirme que le signal `empty_o` doit-être à « 1 » au passage à « 0 » du signal `rst`, comme illustré sur la figure 1. Ici, la propriété sera vérifiée sur le même cycle que celui où `rst` est à « 0 ».

Si nous avions voulu laisser un cycle d'horloge après la remise à zéro, nous aurions écrit :

```
f_reset : assert {rst = '1'; rst = '0'} |=> {empty_o = '1'};
```

Le fonctionnement est illustré par la figure 2.

Le point-virgule `;` utilisé ici décrit une **séquence**. La séquence qui déclenche la vérification de la propriété est un front descendant de `rst`. Au premier cycle, `rst` est à « 1 » et au second cycle, le déclencheur est à « 0 ».

Il est possible de décrire les séquences en utilisant des expressions régulières. Si nous souhaitons que le signal *reset* soit à « 1 » pendant deux cycles avant de passer à « 0 », plutôt que d'écrire :

```
{rst = '1'; rst = '1'; rst = '0']
```

Nous pouvons utiliser la notation `[*N]` pour simplifier :

```
{rst = '1'[*2]; rst='0'}
```

Il est également possible de définir un intervalle, par exemple entre 2 et 10 cycles :

```
{rst = '1'[*2:10]; rst='0'}
```

Le symbole `*` seul signifie « zéro ou plus » :

```
{rst = '1'[*]; rst='0'}
```

Le `+` permet quant à lui de spécifier « un cycle ou plus » :

```
{rst = '1'[+]; rst='0'}
```

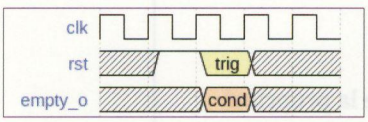
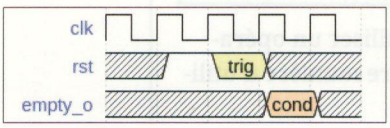


Fig. 1 : Avec l'opérateur `|->`, l'assertion est vérifiée sur le dernier cycle de la trame de déclenchement.

Fig. 2 : Avec l'opérateur `|=>`, l'assertion est vérifiée sur le cycle suivant de la trame de déclenchement.



Les opérateurs `*` et `+` sont utilisés pour des cycles qui se suivent. Si on les remplace par `=` on spécifiera des cycles non consécutifs.

La syntaxe des expressions régulières possède beaucoup de vocabulaire qui dépasse le cadre de cet article, le lecteur pourra se référer à l'article de Wikipédia en anglais [7] pour plus d'informations.

Certaines tournures sont utilisées si souvent que des fonctions ont été intégrées au langage comme :

- `rose(arg)` : fonction « front montant » qui retourne vrai si `arg` passe de « 0 » à « 1 » ;
- `fell(arg)` : fonction « front descendant » qui retourne vrai si `arg` passe de « 1 » à « 0 ». Nous aurions pu l'utiliser dans notre exemple précédent pour détecter le front descendant de `rst` par exemple :

```
f_empty_o: assert always {fell(rst)} |-> {empty_o = '1'};
```

- `prev(arg[, N])` : renvoie la valeur de l'argument N cycles avant le présent. N est fixé à 1 par défaut s'il n'est pas indiqué ;
- `stable(arg)` : retourne vrai si le signal donné en argument n'a pas changé depuis le cycle précédent.

Nous avons les bases du langage PSL pour écrire nos premières propriétés, voyons maintenant comment s'en servir avec un exemple concret de développement d'une FIFO.

3. CONCEPTION D'UNE FIFO

Une FIFO pour *First In First Out* est une pile de valeurs que l'on remplit d'un côté (on écrit des valeurs qui sont « empilées ») et que l'on vide de l'autre. La première valeur entrée dans la pile sera la première valeur à sortir à la lecture.

Les FIFO sont très utilisées dans les systèmes numériques. Elles sont très utiles lorsque l'on doit communiquer des informations entre deux systèmes n'ayant pas la même cadence. Le protocole série UART, par exemple, utilise généralement une FIFO pour faire attendre les données reçues le temps que le programme s'interrompe et les traite.

Une FIFO est également utilisée pour la transmission UART. La génération de la trame de données sur le bus UART étant généralement beaucoup plus lente que le programme, on utilisera la FIFO pour charger les données à envoyer. Ce qui permettra de laisser le composant faire la transmission pendant que le programme fera autre chose.

On trouve également des FIFO dans les convertisseurs analogiques numériques pour faire des saisies rapides d'échantillons dans le temps et les traiter ensuite.

La manière la plus classique de faire une FIFO dans un FPGA (mais la méthode fonctionne également en programmation « normale » sur un CPU) est d'utiliser un tableau circulaire de taille 2^N. Ce tableau est muni de deux pointeurs :

- un pointeur de lecture (`rd_fifo`), pour la sortie de la FIFO ;
- un pointeur d'écriture (`wr_fifo`), pour écrire les valeurs dans la FIFO.

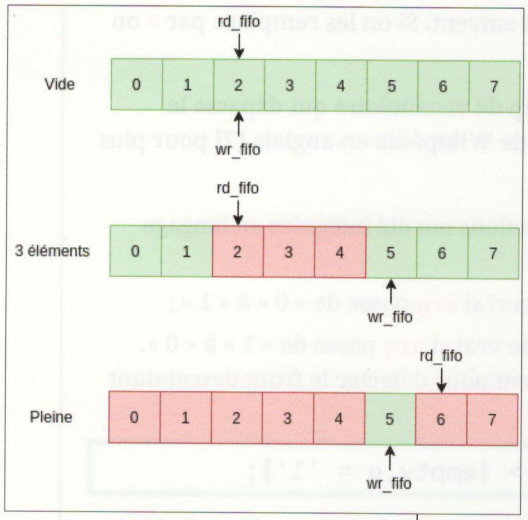


Fig. 3 : Schéma de principe de notre FIFO.

La figure 3 montre une FIFO vide de taille 8 (2^3). Lorsque l'on écrit dans la FIFO, le pointeur d'écriture est incrémenté. Lorsqu'on la lit, c'est le pointeur de lecture qui est incrémenté. Si l'adresse du pointeur d'écriture est égale à celle du pointeur de lecture, alors la FIFO est vide. Si le pointeur d'écriture est situé une unité avant le pointeur de lecture, alors la FIFO est pleine. Dans la mesure où la taille de la FIFO est égale à une puissance de 2, il n'est pas nécessaire de gérer le retour à 0 des pointeurs (le débordement s'en charge).

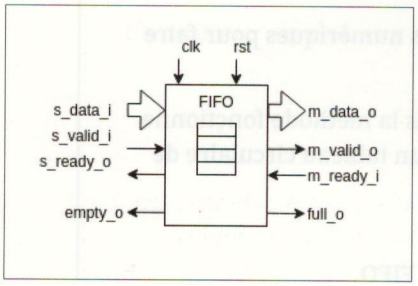
La manière la plus simple de savoir si la FIFO est vide ou pleine est d'utiliser un compteur interne au module. Si le compteur est à 0, la FIFO est vide, si le compteur est égal à la valeur maximale, la FIFO est pleine.

La figure 4 présente les interfaces de la FIFO que nous allons concevoir. On retrouve l'horloge `clk` et le *reset* `rst` indispensables à tout système synchrone. De chaque côté (entrée et sortie), le signal de donnée est accompagné de deux signaux de contrôle `ready` et `valid` utilisés comme poignée de main (*hanshake*) pour valider la transaction.

La présence de la valeur source est donnée par le signal `valid`. Le récepteur signale sa disponibilité à recevoir une nouvelle donnée par le signal `ready`. Le signal `valid` va toujours dans le même sens que la donnée. Lorsque les signaux `valid` et `ready` sont actifs, alors il y a transfert d'une donnée dans le cycle d'horloge.

Dans la suite, nous parlerons d'interfaces esclave ou *slave* pour définir l'ensemble des trois signaux `s_data_i`, `s_valid_i` et `s_ready_o`, et d'interface maître ou *master* pour les trois signaux de sortie `m_data_o`, `m_valid_o` et `m_ready_i`. Le suffixe `_o`/`_i` donne quant à lui la direction de chaque signal par rapport à la FIFO.

Fig. 4 : Schéma bloc des interfaces de la FIFO que nous allons concevoir.



On ajoute enfin un signal pour indiquer que la FIFO est vide `empty_o` et qu'elle est pleine `full_o`.

On dit toujours que les bugs détectés au plus tôt sont les bugs qui coûtent le moins cher. Plutôt que d'écrire les propriétés une fois le VHDL « terminé », commençons par écrire le code VHDL en même temps que les propriétés. Pour cela, nous allons écrire les interfaces du module VHDL comme suit :

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.numeric_std.all;

entity fifo is
generic(FIFO_DEPTH: natural := 3;
        DATA_SIZE: natural := 8);
port(
    clk : in std_logic;
    rst : in std_logic;

    s_data_i : in std_logic_vector(DATA_SIZE-1 downto 0);
    s_valid_i : in std_logic;
    s_ready_o : out std_logic;

    full_o : out std_logic

    m_data_o : out std_logic_vector(DATA_SIZE-1 downto 0);
    m_valid_o : out std_logic;
    m_ready_i : in std_logic;

    empty_o : out std_logic;
);
end entity fifo;

architecture fifo_1 of fifo is
begin

    --XXX debug
    s_ready_o <= '0';
    empty_o <= '0';
    m_data_o <= x"00";
    m_valid_o <= '0';
    full_o <= '0';

end architecture fifo_1;
```

Pour que le code soit « compilable », il faut définir une valeur à tous les signaux de sortie comme visible à la suite du commentaire `XXX debug`. Le commentaire `XXX` est un mot clef que beaucoup d'éditeurs de texte mettent en surbrillance pour rappeler au développeur/à la développeuse qu'il faut corriger/modifier.

Deux fichiers supplémentaires sont nécessaires au bon fonctionnement de la preuve formelle avec Yosys-SMTBMC :

- le script du projet au format `.sby` ;
- le fichier de description des propriétés avec l'extension `.psl`.

Nous rangerons le fichier `fifo.vhd` dans un répertoire `src/` et les deux fichiers « de preuve » dans un répertoire `formal/`. Il est préférable de bien séparer la partie source synthétisable de la partie preuve.

```
├── formal
│   ├── fifo.psl
│   └── fifo.sby
└── src
    └── fifo.vhd
```

3.1 Le script sby

La mise en place de la preuve formelle avec Yosys-SMTBMC monopolise plusieurs logiciels. L'utilisation de scripts SymbiYosys simplifie le processus en rassemblant toute la configuration dans un seul fichier au format `.sby`.

Un script SymbiYosys se lit en commençant par la fin.

```
[options]
mode bmc
depth 2

[engines]
smtbmc

[script]
ghdl --std=08 fifo.vhd fifo.psl -e fifo
prep -top fifo

[files]
../src/fifo.vhd
fifo.psl
```

Les chemins des fichiers sources sont donnés dans la rubrique `[files]`. Une fois configurés dans cette rubrique, on pourra indiquer le nom seul du fichier sans avoir à donner tout le chemin.

La rubrique `[script]` donne les commandes à lancer pour compiler le code HDL. Ici, on compile le module `fifo.vhd` avec le fichier des propriétés `fifo.psl` dans GHDL. On indique ensuite le nom du module **top** avec la commande `prep -top`.

En mettant `smtbmc` dans la rubrique `[engines]` on configure le moteur de preuve par défaut de Yosys-SMTBMC qui est Yices2.

On termine enfin par le mode de preuve que l'on souhaite lancer dans la rubrique `[options]`. Ici, nous souhaitons lancer le moteur de preuve en mode *Bounded Model Checking* sur 2 cycles d'horloge.

Pour exécuter ce script, nous utiliserons la commande `sby` avec l'option de chargement de GHDL en module :

```
$ sby --yosys "yosys -m /opt/ghdl-yosys-plugin/ghdl.so" -f fifo.sby
```

Il est possible de définir des tâches dans le fichier de configuration en ajoutant la rubrique `[tasks]` :

```
[tasks]
bmc
prove
```

On s'y référera de cette manière dans la rubrique `[options]` pour configurer chaque tâche :

```
bmc: mode bmc
bmc: depth 2
prove: mode prove
prove: depth 50
```

Pour lancer une tâche en particulier, il suffira de lui donner son nom dans la ligne de commande :

```
$ sby --yosys "yosys -m /opt/ghdl-yosys-plugin/ghdl.so" -f fifo.sby bmc
```

Ce système de tâches devient vite indispensable pendant le développement, si l'on souhaite commuter les différents modes sans avoir à gérer plusieurs fichiers de configuration très similaires.

Si tout se passe bien, l'exécution de la commande se terminera par la ligne :

```
SBY  8:01:59 [fifo] DONE (PASS, rc=0)
```

Un fichier `fifo/PASS` est également créé pour indiquer que tout s'est bien passé.

3.2 Les propriétés PSL

Dans notre exemple, le fichier de propriétés se nomme `fifo.psl`. Il est constitué de la « fonction » suivante :

```
vunit i_fifo(fifo(fifo_1))
{
    -- code VHDL et propriétés PSL
}
```

Le nom de la fonction, qui retourne `vunit`, est arbitraire. On lui passera le nom de l'entité (architecture) à tester. Ici, l'entité est `fifo` et l'architecture que nous souhaitons vérifier est `fifo_1`.

3.3 Développement conjoint VHDL/PSL

Pour le moment, notre fichier de propriétés ne contient qu'un commentaire en VHDL. On peut donc le considérer comme vide. Dans ces conditions, la vérification BMC passe bien. Mais c'était inutile !

Dans un premier temps, nous allons commencer par donner l'horloge du module, elle définira les cycles de simulation utilisés pour la preuve. Nous éviterons ainsi d'avoir à ajouter des `@rising_edge(clk)` dans chacune des directives que nous allons écrire.

```
vunit i_fifo(fifo(fifo_1))
{
    default clock is rising_edge(clk);
}
```

La structure de base étant écrite, nous allons pouvoir rentrer dans le vif du sujet.

Le principe général qui guide l'écriture des propriétés est d'« assumer » (avec `assume`) les signaux d'entrée et d'« affirmer » (avec `assert`) les signaux de sortie. Ce principe de base n'empêche pas d'étendre les propriétés sur les signaux internes du module et/ou d'ajouter des signaux dans le fichier `.psl`.

On va commencer par s'assurer qu'à l'étape initiale, le module est bien en *reset* avec la directive suivante :

```
    f_reset: assume {rst};
```

Il est conseillé de faire précéder chaque directive d'un label pour s'y retrouver ensuite dans les messages de simulation. Cette directive que l'on nomme `f_reset` « assume » que le signal de *reset* soit vrai (à « 1 ») pendant un cycle d'horloge. L'absence de `always` indique que cette propriété doit être vraie à l'étape initiale.

La notation `{rst}` est une **séquence** constituée ici d'un seul cycle d'horloge. Nous pouvons également décrire une suite de cycles en séparant les états par des « ; ». Par exemple, si nous voulons nous assurer que le *reset* passe bien à « 0 » au démarrage :

```
    f_reset: assume {rst; not rst};
```

Pour le moment, le moteur de preuve nous passe les tests BMC sans problèmes. Et pour cause : nous n'avons encore rien affirmé. Le code « bouchon » que nous avons écrit dans le module `fifo` ne pose donc pas vraiment de problèmes.

Pour notre première assertion, on s'assurera que le signal `empty_o` est bien à « 1 » à la sortie du *reset* :

```
    f_empty_o: assert always {not rst} |-> {empty_o = '1'};
```

Plusieurs choses ont été ajoutées ici.

Tout d'abord, le mot clef `always` qui signifie que la propriété doit être toujours vraie, quel que soit le cycle d'horloge concerné.

Ensuite, l'opérateur d'implication `|->` indique que la séquence `{not rst}` implique l'affirmation donnée dans la deuxième partie de la directive `{empty_o = '1'}`.

La propriété va échouer puisque nous avions forcé le signal `empty_o` à `'0'`.

```
$ sby --yosys "yosys -m /opt/ghdl-yosys-plugin/ghdl.so" -f fifo.sby

[messages de compilation]

engine_0: ##   0:00:00  Solver: yices
engine_0: ##   0:00:00  Checking assumptions in step 0..
engine_0: ##   0:00:00  Checking assertions in step 0..
engine_0: ##   0:00:00  Checking assumptions in step 1..
engine_0: ##   0:00:00  Checking assertions in step 1..
engine_0: ##   0:00:00  BMC failed!
engine_0: ##   0:00:00  Assert failed in fifo: i_fifo.f_empty_o
engine_0: ##   0:00:00  Writing trace to VCD file: engine_0/trace.vcd
engine_0: ##   0:00:00  Writing trace to Verilog testbench: engine_0/trace_tb.v
engine_0: ##   0:00:00  Writing trace to constraints file: engine_0/trace.smtc
engine_0: ##   0:00:00  Status: failed
engine_0: finished (returncode=1)
engine_0: Status returned by engine: FAIL
summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:00 (0)
summary: Elapsed process time [H:MM:SS (secs)]: 0:00:00 (0)
summary: engine_0 (smtbmc) returned FAIL
summary: counterexample trace: fifo/engine_0/trace.vcd
DONE (FAIL, rc=2)
```

Le test échoue sur la propriété `i_fifo.f_empty_o`. On comprend ici l'intérêt d'avoir nommé chaque propriété. Une trace est générée dans le répertoire `fifo/engine_0` que l'on peut visualiser avec `gtkwave` comme visible en figure 5 :

```
$ gtkwave fifo/engine_0/trace.vcd
```

Les traces sont également générées au format Verilog si l'on souhaite les « rejouer » dans un simulateur Verilog.

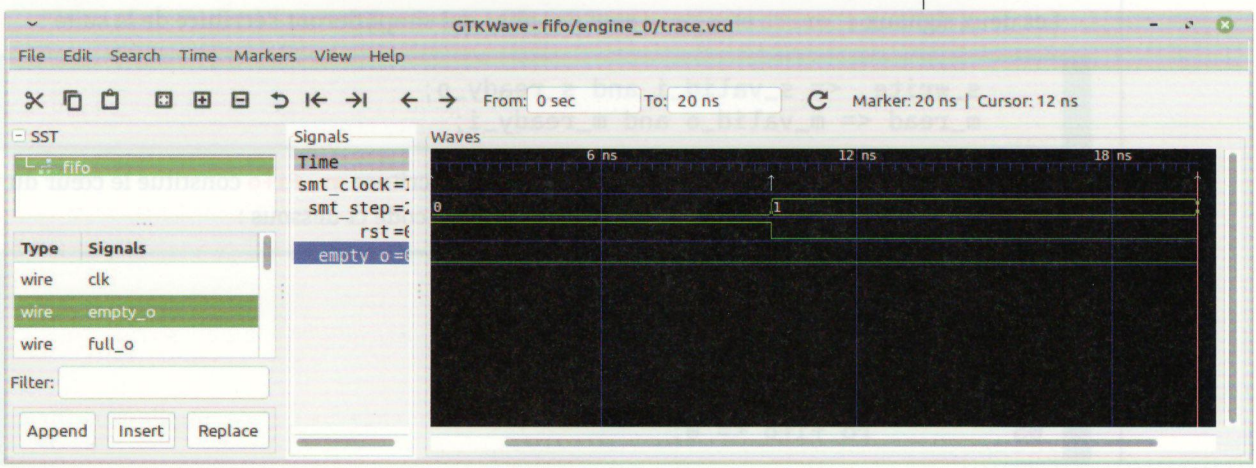


Fig. 5 : Comme on s'y attendait, le test échoue, car le signal empty_o est à '0' lorsque la FIFO sort du reset.

Il est normal que la directive échoue dans notre cas, puisque `empty_o` est fixé en dur à « 0 ». Cependant, l'écriture est erronée, puisque l'on affirme que le signal `empty_o` est toujours à 1 lorsque `rst` est à 0. Si la FIFO était vide tout le temps, elle ne nous serait pas très utile. Nous cherchions plutôt à écrire une directive qui affirme que la FIFO est vide **au front descendant** du signal `rst`. Nous écrirons donc plutôt la condition comme cela :

```
    f_empty_o: assert always {fell(rst)} |-> {empty_o = '1'};
```

Assertion qui va persister à échouer tant que nous n'aurons pas écrit un minimum de code VHDL dans le module `fifo`.

Commençons donc à écrire le contenu de notre FIFO avec la déclaration du *buffer* circulaire.

On déclare un type tableau nommé `t_circbuf_array` de taille `2^FIFO_DEPTH` constitué de vecteurs `std_logic_vector` de taille `DATA_SIZE`. Ce type est utilisé ensuite pour déclarer le tableau `circbuf` :

```
architecture fifo_1 of fifo is
    type t_circbuf_array is array(natural range 0 to (2**FIFO_DEPTH)-1)
        of std_logic_vector(DATA_SIZE-1 downto 0);
    signal circbuf : t_circbuf_array;

    signal wr_fifo : natural range 0 to (2**FIFO_DEPTH)-1;
    signal rd_fifo : natural range 0 to (2**FIFO_DEPTH)-1;

    signal fifocount : integer range 0 to (2**FIFO_DEPTH) - 1;

    signal s_write : std_logic;
    signal m_read  : std_logic;
```

Les deux signaux `wr_fifo` et `rd_fifo` sont des entiers naturels respectivement pointeur d'écriture et pointeur de lecture.

On ajoute le compteur `fifocount` pour simplifier la gestion du remplissage/vidage de la FIFO.

Les deux signaux `s_write` et `m_read` sont des alias qui simplifieront l'écriture de la suite :

```
    s_write  <= s_valid_i and s_ready_o;
    m_read <= m_valid_o and m_ready_i;
```

La gestion des deux pointeurs de lecture `rd_fifo` et d'écriture `wr_fifo` constitue le cœur du fonctionnement de la FIFO, comme on peut le voir dans le code ci-dessous :

```
00 process(clk, rst)
01 begin
02     if(rst) then
03         fifocount <= 0;
04         wr_fifo <= 0;
05         rd_fifo <= 0;
```

```
06          m_data_o <= (others => '0');
07      elsif(rising_edge(clk)) then
08          -- nombre de données dans la FIFO
09          if(s_write = '1' and m_read = '0') then
10              fifocount <= fifocount + 1;
11          elsif(s_write = '0' and m_read = '1') then
12              fifocount <= fifocount - 1;
13          end if;
14
15          -- Incrémentation de l'index d'écriture
16          if(s_write = '1' and full_o = '0') then
17              if(wr_fifo = 2**FIFO_DEPTH - 1) then
18                  wr_fifo <= 0;
19              else
20                  wr_fifo <= wr_fifo + 1;
21              end if;
22          end if;
23
24          -- Incrémentation de l'index de lecture
25          if(m_read = '1' and empty_o = '0') then
26              if(rd_fifo = 2**FIFO_DEPTH - 1) then
27                  rd_fifo <= 0;
28              else
29                  rd_fifo <= rd_fifo + 1;
30              end if;
31          end if;
32
33          -- écriture dans la FIFO
34          if(s_write = '1') then
35              circbuf(wr_fifo) <= s_data_i;
36          end if;
37
38          -- lecture dans la fifo
39          m_data_o <= circbuf(rd_fifo);
40
41      end if;
42  end process;
```

Lorsqu'une donnée se présente sur l'entrée (`s_valid_i` et `s_ready_o` à « 1 »), on écrit la valeur dans la mémoire (l. 33). Et l'on incrémente le pointeur `wr_fifo` (l. 16).

S'il n'y a pas de lecture au même moment, on incrémente le compteur `fifocount` (l. 09). De même, lorsqu'une donnée est lue sur la sortie (`m_valid_o` et `m_ready_i` à « 1 »), on incrémente le pointeur `rd_fifo` (l. 25).

Le signal `full_o` est à « 1 » lorsque le compteur indique la valeur maximum possible :

```
full_o <= '1' when fifocount = 2**FIFO_DEPTH - 1 else '0';
```

De la même manière, le signal `empty_o` passe à « 1 » lorsque le compteur est à 0 :

```
empty_o <= '1' when fifocount = 0 else '0';
```

Les signaux `s_ready_o` et `m_valid_o` ne sont que des variations des pointeurs `empty_o` et `full_o` :

```
s_ready_o <= not full_o;
m_valid_o <= not empty_o;
```

L'intégralité du code se trouve sur le dépôt de l'auteur **[8]** avec un *makefile* pour reproduire les différents modes.

Nous avions déjà vu comment s'assurer que la FIFO soit vide à la sortie du *reset*. Nous pouvons également affirmer que la FIFO n'est pas vide lorsqu'elle est pleine et inversement :

```
f_fullempty: assert always {empty_o} |-> {not full_o};
f_emptyfull: assert always {full_o} |-> {not empty_o};
```

La propriété principale de notre FIFO est qu'une donnée entrée en premier doit être sortie en premier également. Nous avons donc besoin de vérifier l'ordre des données en sortie, en fonction des entrées.

Nous allons pour cela définir deux signaux constants de données `f_data1` et `f_data2` qui feront office de valeurs entrées dans la FIFO.

```
signal f_data1 : std_logic_vector(DATA_SIZE-1 downto 0);
signal f_data2 : std_logic_vector(DATA_SIZE-1 downto 0);
```

Nous avons besoin de tester toutes les combinaisons possibles de valeurs de ces deux données. Nous ne pouvons donc pas les initialiser à une valeur fixe.

Le langage nous fournit l'attribut `anyconst` pour ça :

```
attribute anyconst : boolean;
attribute anyconst of f_data1 : signal is true;
attribute anyconst of f_data2 : signal is true;
```

Cet attribut indique que la variable est une constante qui peut prendre n'importe quelle valeur. Le moteur de preuve va parcourir tout l'espace des valeurs possibles pour ces deux signaux, comme il le fait pour les signaux d'entrée.

Maintenant que nous avons deux valeurs à surveiller, nous allons pouvoir créer des signaux indiquant que les données ont été écrites et lues dans la FIFO :

```
signal f_in_valid1, f_in_valid2 : std_logic := '0';
signal f_out_valid1, f_out_valid2 : std_logic := '0';
```

Il est possible d'écrire du VHDL tout à fait standard dans le fichier PSL. Cela va nous être utile pour décrire le comportement des quatre signaux que nous venons de déclarer. Dans un `process` VHDL, nous décrivons leur comportement :

- `f_in_validx` passe à « 1 » lorsque la valeur x est écrite dans la FIFO ;
- `f_out_validx` passe à « 1 » lorsque la valeur x est lue en sortie de FIFO.

```
p_ordo : process(clk, rst)
begin
    if(rst) then
        f_in_valid1 <= '0';
        f_in_valid2 <= '0';
        f_out_valid1 <= '0';
        f_out_valid2 <= '0';
    elsif(rising_edge(clk)) then
        if s_valid_i and s_ready_o then
            if s_data_i = f_data1 then
                f_in_valid1 <= '1';
            end if;
            if s_data_i = f_data2 then
                f_in_valid2 <= '1';
            end if;
        end if;
        if m_valid_o and m_ready_i then
            if m_data_o = f_data1 then
                f_out_valid1 <= '1';
            end if;
            if m_data_o = f_data2 then
                f_out_valid2 <= '1';
            end if;
        end if;
    end if;
end process p_ordo;
```

Si l'on assume que `f_in_valid1` passe à « 1 » avant `f_in_valid2` :

```
f_indata_order : assume always {f_in_valid2} |-> {f_in_valid1};
```

On peut ensuite affirmer que `f_out_valid2` passera à « 1 » après `f_out_valid1` :

```
f_outdata_order : assert always {f_out_valid2} |-> {f_out_valid1};
```

Avec toutes ces propriétés, on peut lancer une première passe de preuve en mode BMC sur une dizaine de cycles d'horloge pour voir comment le module se comporte. On ajoutera pour cela une tâche `bmc` dans le script de configuration `sby`.

```
[tasks]
bmc
...
[options]
bmc: mode bmc
bmc: depth 10
...
```

Mais le test échoue :

```
$ sby --yosys "yosys -m /opt/ghdl-yosys-plugin/ghdl.so" -f fifo.sby bmc
...
##   0:00:00  Solver: yices
##   0:00:00  Checking assumptions in step 0..
##   0:00:00  Checking assertions in step 0..
##   0:00:00  Checking assumptions in step 1..
##   0:00:00  Checking assertions in step 1..
##   0:00:00  Checking assumptions in step 2..
##   0:00:00  Checking assertions in step 2..
##   0:00:00  Checking assumptions in step 3..
##   0:00:00  Checking assertions in step 3..
##   0:00:00  BMC failed!
##   0:00:00  Value for anyconst in fifo (/367): 192
##   0:00:00  Value for anyconst in fifo (/368): 128
##   0:00:00  Assert failed in fifo: i_fifo.f_outdata_order
##   0:00:00  Writing trace to VCD file: engine_0/trace.vcd
##   0:00:00  Writing trace to Verilog testbench: engine_0/trace_tb.v
##   0:00:00  Writing trace to constraints file: engine_0/trace.smtc
##   0:00:00  Status: failed
finished (returncode=1)
Status returned by engine: FAIL
SBY  DONE (FAIL, rc=2)
```

L'assertion nommée `f_outdata_order` échoue bien avant la fin des 10 cycles d'horloge demandés.

Deux choses l'une, soit nous avons un bug dans la FIFO, soit nous avons mal écrit nos propriétés. Pour le savoir, le plus simple est d'aller voir la trace générée avec GTKWave :

```
$ gtkwave fifo_bmc/engine_0/trace.vcd
```

Ce qui donne le chronogramme de la figure 6.

On voit ici que le signal `f_out_valid2` passe à « 1 » avant `f_out_valid1`, contrairement à ce que nous avions affirmé dans la directive `f_outdata_order`. Le moteur de preuve a tout naturellement échoué.

En inspectant le chronogramme, on se rend compte qu'il y a un bug. En effet, la première valeur 0x80 entrée à l'étape 1 (`smt_step`) dans la FIFO n'est pas la première valeur lue à l'étape 2.

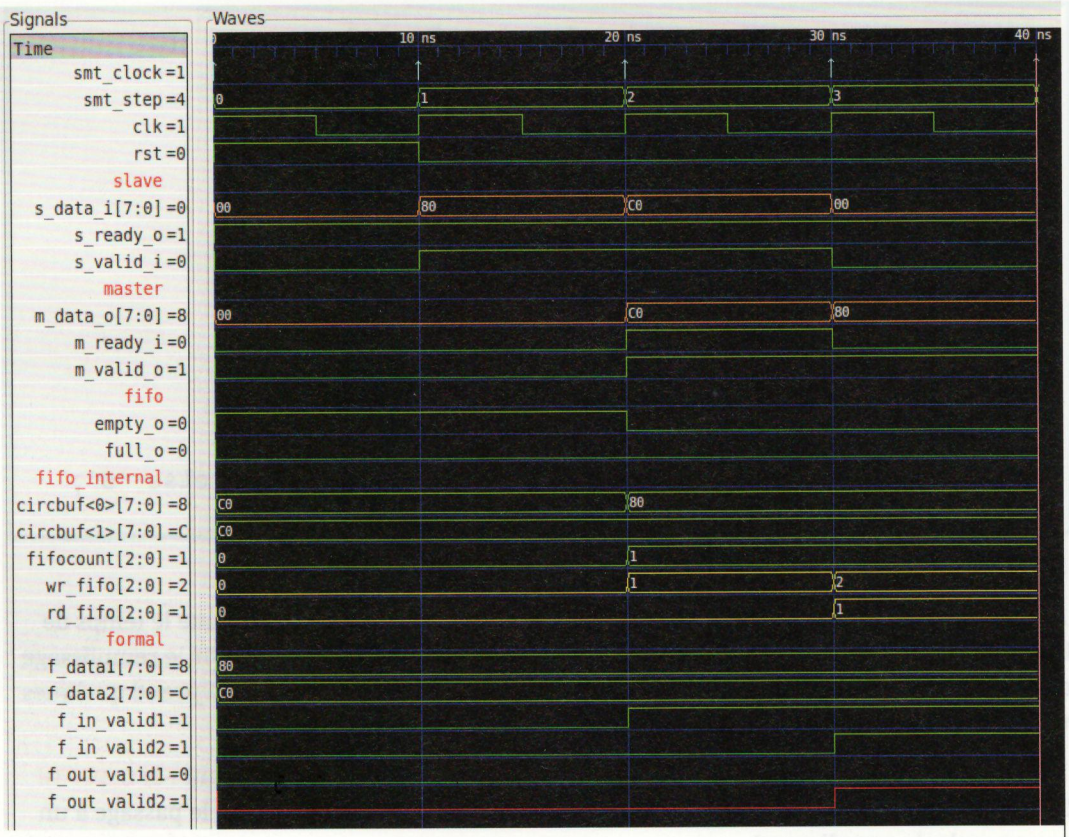


Fig. 6 : Le signal f_out_valid2 passe à '1' alors que f_out_valid1 est encore à '0'.

Nous avons donc bien détecté un bug dans la FIFO, ça n'est pas un problème de propriétés manquantes ou mal écrites.

Le problème vient ici de l'enchaînement d'une écriture, puis d'une lecture le cycle suivant. La lecture étant synchrone de l'horloge, la donnée lue dans la mémoire est celle du cycle précédent.

Ce bug peut se résoudre simplement en passant à une lecture « asynchrone » de la mémoire, tout en conservant l'écriture synchrone.

Pour avoir une lecture asynchrone, on sortira la ligne de lecture du `process` tout en supprimant l'initialisation au *reset* de `m_data_o` comme ceci :

```
--- fifo.vhd    2021-12-23 08:27:21.189733419 +0100
+++ fifo_new.vhd    2021-12-23 08:27:24.925619361 +0100
@@ -48,7 +48,6 @@
             fifocount <= 0;
             wr_fifo <= 0;
             rd_fifo <= 0;
-            m_data_o <= (others => '0');
         elsif(rising_edge(clk)) then
```

```
                -- nombre de données dans la fifo
                if(s_write = '1' and m_read = '0') then
@@ -80,11 +79,12 @@
                    circbuf(wr_fifo) <= s_data_i;
                end if;

-               -- lecture synchrone dans la fifo
-               m_data_o <= circbuf(rd_fifo);

            end if;
        end process;
+
+       -- lecture asynchrone dans la fifo
+       m_data_o <= circbuf(rd_fifo);

        full_o <= '1' when fifocount = 2**FIFO_DEPTH - 1 else '0';
        empty_o <= '1' when fifocount = 0 else '0';
```

Si l'on relance la preuve en mode BMC, tout se passe bien cette fois.

Au fur et à mesure qu'il incrémente les pas de simulation, le moteur augmente le temps de calcul de manière exponentiel. Si l'on veut tester la FIFO en étant sûr de couvrir le remplissage complet puis le vidage, il faudrait que l'on fasse un minimum de 16 pas. Ce qui prend quelques heures !

Dans le cas de cette FIFO, on peut simplement en diminuer la taille en passant `FIFO_DEPTH` à 2 pour avoir une profondeur de 4 et simuler 10 pas (pour compter le *reset* et le passage à un nouveau cycle de remplissage).

Avec tous ces nouveaux réglages, le test BMC passe rapidement :

```
sby --yosys "yosys -m /opt/ghdl-yosys-plugin/ghdl.so" -f FIFO.sby bmc
...
engine_0: ##   0:00:01  Checking assumptions in step 8..
engine_0: ##   0:00:01  Checking assertions in step 8..
engine_0: ##   0:00:06  Checking assumptions in step 9..
engine_0: ##   0:00:06  Checking assertions in step 9..
engine_0: ##   0:00:16  Status: passed
engine_0: finished (returncode=0)
engine_0: Status returned by engine: pass
summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:16 (16)
summary: Elapsed process time [H:MM:SS (secs)]: 0:00:16 (16)
summary: engine_0 (smtbmc) returned pass
DONE (PASS, rc=0)
```

3.4 Mode preuve par induction

Tous les tests précédents ont été réalisés avec le mode BMC. Ce qui signifie que l'on partait d'un état initial connu pour parcourir toutes les combinaisons possibles jusqu'à un certain nombre de pas.

Avec le mode `prove`, le moteur va se détacher de cette condition initiale pour parcourir toutes les combinaisons possibles d'une longueur donnée en configuration.

Sans changer les propriétés PSL que nous avons écrites pour le mode BMC, nous pouvons lancer le moteur de preuve en mode `prove` avec une profondeur de 10 en ajoutant une tâche dans le fichier `fifo.psl` :

```
[tasks]
...
prove

[options]
...
prove: mode prove
prove: depth 10
```

En partant d'un état valide, qui n'est pas nécessairement l'état initial, le moteur de preuve va tenter de trouver un chemin menant à un état invalide. Ce qu'il trouve assez rapidement dans notre cas :

```
sby --yosys "yosys -m /opt/ghdl-yosys-plugin/ghdl.so" -f fifo.sby prove
...
engine_0.induction: ##   0:00:00  Solver: yices
engine_0.basecase:  ##   0:00:00  Checking assumptions in step 0..
engine_0.basecase:  ##   0:00:00  Checking assertions in step 0..
engine_0.induction: ##   0:00:00  Trying induction in step 10..
engine_0.induction: ##   0:00:00  Trying induction in step 9..
...
engine_0.basecase:  ##   0:00:00  Checking assumptions in step 5..
engine_0.basecase:  ##   0:00:00  Checking assertions in step 5..
engine_0.induction: ##   0:00:00  Trying induction in step 1..
engine_0.induction: ##   0:00:00  Trying induction in step 0..
engine_0.induction: ##   0:00:00  Temporal induction failed!
engine_0.induction: ##   0:00:00  Value for anyconst in fifo (/363): 254
engine_0.induction: ##   0:00:00  Value for anyconst in fifo (/364): 255
engine_0.induction: ##   0:00:00  Assert failed in fifo: i_fifo.f_outdata_order
engine_0.induction: ##   0:00:00  Writing trace to VCD file: engine_0/trace_
induct.vcd
engine_0.induction: ##   0:00:00  Writing trace to Verilog testbench: engine_0/
trace_induct_tb.v
engine_0.induction: ##   0:00:00  Writing trace to constraints file: engine_0/
trace_induct.smtc
engine_0.induction: ##   0:00:00  Status: failed
engine_0.induction: finished (returncode=1)
engine_0: Status returned by engine for induction: FAIL
engine_0.basecase:  ##   0:00:00  Checking assumptions in step 6..
```

```
engine_0.basecase: ##   0:00:00  Checking assertions in step 6..
engine_0.basecase: ##   0:00:00  Checking assumptions in step 7..
engine_0.basecase: ##   0:00:00  Checking assertions in step 7..
engine_0.basecase: ##   0:00:01  Checking assumptions in step 8..
engine_0.basecase: ##   0:00:01  Checking assertions in step 8..
engine_0.basecase: ##   0:00:06  Checking assumptions in step 9..
engine_0.basecase: ##   0:00:06  Checking assertions in step 9..
engine_0.basecase: ##   0:00:16  Status: passed
engine_0.basecase: finished (returncode=0)
engine_0: Status returned by engine for basecase: pass
summary: Elapsed clock time [H:MM:SS (secs)]: 0:00:16 (16)
summary: Elapsed process time [H:MM:SS (secs)]: 0:00:17 (17)
summary: engine_0 (smtbmc) returned FAIL for induction
summary: engine_0 (smtbmc) returned pass for basecase
DONE (UNKNOWN, rc=4)
```

Le moteur de preuve a encore trouvé un cas où le signal `f_out_valid2` passe à « 1 » avant `f_out_valid1`, comme on peut le voir figure 7.

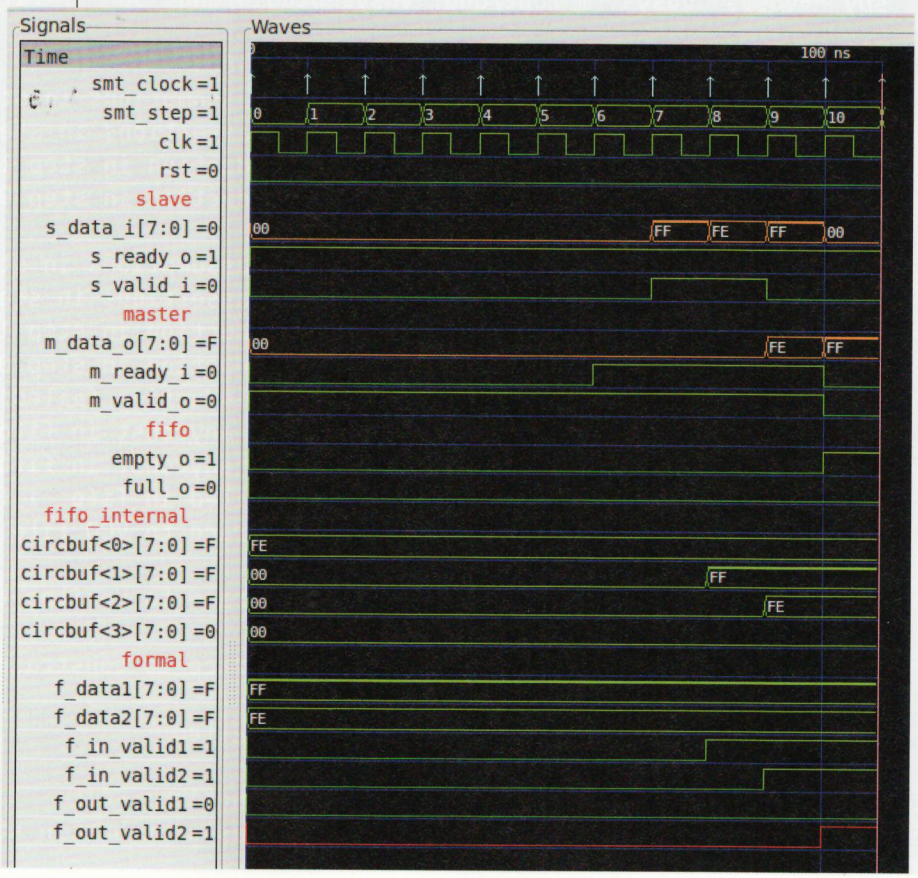


Fig. 7 : Le signal f_out_valid2 passe à '1' alors que f_out_valid1 est encore à '0'.

Le test n'étant pas parti de l'état initial, la FIFO a été remplie d'un certain nombre de valeurs et la valeur de `fifocount` est donnée « arbitrairement » (de manière à faire échouer les tests bien sûr), etc.

Pour passer les tests d'induction, il faut ajouter beaucoup de propriétés pour donner une description complète des états du système. On ne peut plus se contenter d'écrire des propriétés pour les états atteignables depuis l'état initial.

La preuve par induction mériterait un article à part entière. Nous n'avons fait qu'effleurer le sujet ici. Mais il était nécessaire de montrer que c'était possible avec Yosys-SMTBMC.

3.5 Mode couverture de code

À partir du moment où l'on a un code fonctionnel comme on vient de l'écrire, il est intéressant de le voir fonctionner avec un cas d'exemple.

Traditionnellement, en simulation, on écrirait toutes les étapes nécessaires au remplissage de la FIFO pour valider qu'on arrive jusqu'au signal `full_o` à « 1 ». On ajouterait ensuite des étapes pour la vider entièrement et s'assurer que les pointeurs de lecture et d'écriture « débordent » correctement pour revenir à une FIFO vidée et le signal `empty_o` à « 1 ».

Écrire ce cas d'école est fastidieux, prend beaucoup de lignes et de temps. Avec le mode `cover`, il est possible de faire ça beaucoup plus rapidement et de laisser le moteur écrire les stimuli par lui-même.

Résumons, notre objectif est d'avoir une trace qui parte de l'étape initiale quand la FIFO est vide, qui se remplit et lève le signal `full_o`, puis qui se vide jusqu'à lever le signal `empty_o` à nouveau. Pour voir les pointeurs de lecture et d'écriture déborder, on remplira à nouveau la FIFO afin d'obtenir une nouvelle levée du signal `full_o`.

Toute cette séquence est décrite par la directive `cover` suivante :

```
fc_full : cover {
    full_o and not rst;
    not rst [+];
    empty_o and not rst;
    not rst [+];
    full_o and not rst
    };
```

Si pendant l'exécution du mode `BMC` ou `prove`, cette séquence est couverte, Yosys-SMTBMC générera une trace `vcd`. Mais si l'on veut s'assurer que le moteur « couvre » bien cette directive, il faudra exécuter `yosys-smtbmc` en mode `cover`. Pour cela, on peut modifier le script `fifo.sby` et y ajouter les paramètres de couverture :

```
[task]
cover

[options]
cover: mode cover
cover: depth 40
```

La profondeur `depth` doit-être choisie avec soin pour être certain d'atteindre la condition.

La trace, visible en figure 8, se génère au moyen de la commande suivante :

```
$ sby --yosys "yosys -m /opt/ghdl-yosys-plugin/ghdl.so" -f fifo.sby cover
```

La génération de la trace est indiquée par les lignes suivantes :

```
0:00:00  Reached cover statement at i_fifo.fc_full in step 22.
0:00:00  Writing trace to VCD file: engine_0/trace3.vcd
0:00:00  Writing trace to Verilog testbench: engine_0/trace3_tb.v
0:00:00  Writing trace to constraints file: engine_0/trace3.smtc
[...]
8:32:27 [fifo_cover] DONE (PASS, rc=0)
```

Si l'ordinateur ne parvient pas à couvrir la directive, l'exécution se terminera par un FAIL. Dans le cas contraire, il s'arrêtera dès qu'il rencontrera la condition de couverture, sans finir les « pas » indiqués dans la configuration.

4. POUR CONCLURE

La lectrice ou le lecteur a désormais toutes les bases pour faire de la preuve formelle en VHDL. Cette méthode est très utilisée pour la conception d'ASIC. En effet, une fois les masques fabriqués, en conception ASIC, il n'est plus possible de revenir sur le code pour corriger un bug détecté en fonctionnement. La méthode formelle de vérification des propriétés est une bonne solution pour couvrir l'ensemble du fonctionnement du système final.

Pour autant, comme on l'a vu, c'est également une bonne méthode à utiliser dès le début du développement de son composant. On peut écrire les propriétés en même temps que le code synthétisable et lancer les tests au fur et à mesure du développement. On évite comme cela les erreurs de conception à la source qui nous aurait coûté beaucoup d'efforts en reconception de l'architecture.

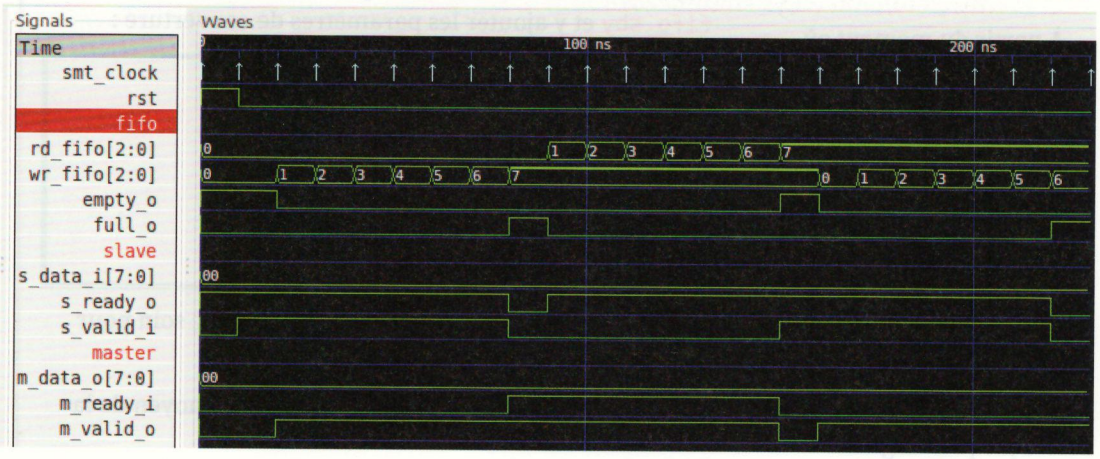


Fig. 8 : Avec le mode `cover`, nul besoin de se coltiner tous les stimuli « à la main », l'ordinateur s'en charge.

Yosys effectue la synthèse du composant pour générer une description compatible avec les moteurs de preuve *open source* « du marché », comme Yices2 utilisé pour cet article. Il est possible également d'utiliser le moteur **Z3** de Microsoft **[9]** (s'il est *open source*) ou **Boolector [10]** de l'université de Linz en Autriche.

Cette méthode donne une relative confiance dans notre code une fois que l'on passe à la synthèse FPGA, on sait déjà qu'il est « synthétisable ».

Grâce à l'utilisation de GHDL, nous pouvons utiliser le langage PSL. Cela simplifie l'écriture des propriétés et les rend beaucoup plus lisibles. Le langage étant un standard, rien n'interdit d'utiliser ensuite les propriétés avec un logiciel commercial.

Les outils libres étant maintenant disponibles pour le VHDL, il n'y a plus d'excuses pour l'utiliser et améliorer ses composants dans ce langage. **FM**

RÉFÉRENCES

[1] RTCA DO-254 / EUROCAE ED-80, Design Assurance Guidance for Airborne Electronic Hardware,
https://en.wikipedia.org/wiki/DO-254

[2] Fabien Marteau, *« De la preuve formelle en Verilog, librement »*, *Hackable 37*,
https://connected-diamond.com/Hackable/hk-037/de-la-preuve-formelle-en-verilog-librement

[3] ghdl-yosys-plugin, extension à Yosys permettant de synthétiser du VHDL avec GHDL,
https://github.com/ghdl/ghdl-yosys-plugin

[4] La documentation officielle de GHDL,
https://ghdl.github.io/ghdl/

[5] Wolf Clifford, vidéo de la conférence 33C3 sur Yosys-SMTBMC,
http://fabienm.eu/ys/smtbmc2016

[6] Accellera, consortium de standardisation pour les FPGA,
https://www.accellera.org/

[7] Property Specification Language, Wikipédia,
https://en.wikipedia.org/wiki/Property_Specification_Language

[8] Fabien Marteau, Diamond_HK_GLMF_OS,
https://github.com/Martoni/Diamond_HK_GLMF_OS

[9] Z3 theorem prover, Microsoft Research,
https://github.com/Z3Prover/z3

[10] Boolector SMT solver, JKU, **http://fmv.jku.at/boolector**

DÉVELOPPER POUR PDA DE PLUS DE 20 ANS EN 2022, C'EST POSSIBLE !

Denis Bodor

Cela n'est plus à démontrer, il existe actuellement un certain engouement pour les technologies des années 80 à 2000. Qu'il s'agisse de consoles, d'ordinateurs familiaux ou même d'équipements haut de gamme professionnels, les artefacts du monde d'avant les smartphones, de Twitter, de Facebook et d'internet partout et tout le temps ont le vent en poupe. Mais, au-delà de la simple collection visant à faire prendre la poussière à un objet à un endroit plutôt qu'un autre, nous avons là une caractéristique absolument fascinante : il est toujours possible de les utiliser et même de développer de nouveaux programmes pour ces reliques. Et c'est le cas, bien entendu, du plus mémorable des assistants personnels numériques, le Palm Pilot.

– Développer pour PDA de plus de 20 ans en 2022, c'est possible ! –

J'entends d'ici les amateurs d'organiseurs Psion hurler au blasphème et à l'hérésie, peut-être à juste titre. Le Psion Organiser II est sans doute le premier PDA ayant fait son apparition et les évolutions qui ont suivi, parmi lesquelles quelques modèles mythiques, sont tout à fait notables, mais... moi j'avais un Palm IIIxe et non pas un Psion série 3. Donc, le Palm est bien plus vivace dans ma mémoire et de ce fait, mémorable.

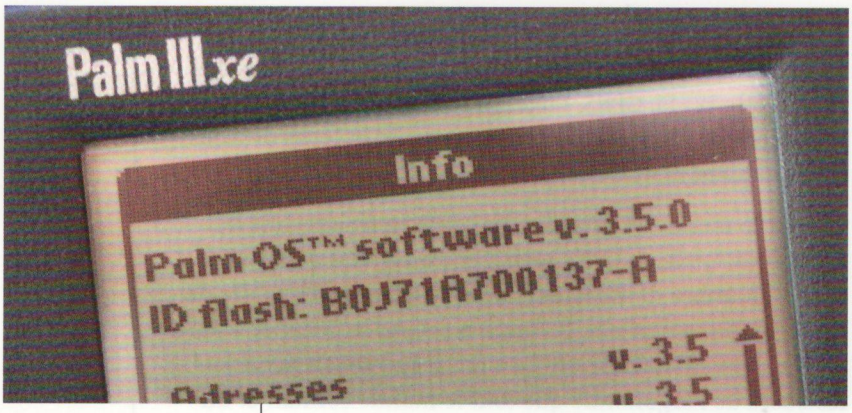
Petit rappel historique pour les plus jeunes d'entre vous, et en particulier ceux ayant fait connaissance avec l'informatique et/ou la programmation alors que Google était déjà là : bien avant que tout ne monde ne se promène avec des supercalculateurs ultra-connectés en poche existaient les PDA ou en bon français, les assistants personnels numériques (ou organiseurs électroniques). Il s'agissait de petits appareils permettant de réaliser des tâches simples comme la gestion de rendez-vous, le suivi des dépenses, la prise de notes, l'organisation des tâches, etc. Les premiers PDA avaient une connectivité se limitant à la synchronisation avec un ordinateur via une liaison série ou infrarouge, et après quelques années, les connexions sans fil ont commencé à être prises en charge (Bluetooth, Wi-Fi, GPRS, Edge, etc.). Mais c'était trop tard, les « mobiles » avaient absorbé le concept et leurs descendants allaient mettre fin au règne du PDA.

Tout le monde se souvient de l'arrivée fracassante de l'iPhone d'Apple en 2007, mais contrairement à ce qu'un certain nombre d'utilisateurs peu versés dans l'histoire de l'informatique pensent, l'iPhone n'est pas le premier smartphone, loin de là. C'est simplement le premier à avoir jeté les bases de l'ergonomie que tout le monde connaît et utilise aujourd'hui (écran capacitif, interface fluide, connectivité permanente et prix déraisonnable). Il fut l'événement qui précipita toute l'industrie vers la débauche de smartphones qu'on connaît aujourd'hui.

1. PALM, PALM OS, 3COM ET PALMPILOT

Durant l'ère des PDA, dans la décennie 1993-2003, la société californienne *Palm Inc* fut à l'origine d'une gamme complète de PDA très reconnaissables, débutant en 1997 avec le Palm Pilot. La société fut ensuite rachetée par U.S. Robotics, elle-même rachetée peu après par 3Com. Un an après, les fondateurs de *Palm Inc* quittèrent 3Com pour crée Handspring qui, par la suite, développa et commercialisa des PDA compatibles avec les appareils 3Com, les Visor, puis les Treo.

Mais les Visor n'étaient pas les seuls PDA compatibles reposant sur le système d'exploitation initialement développé pour le Palm Pilot, appelé Palm OS. D'autres constructeurs se lancèrent sur ce marché qui semblait avoir de longues et radieuses années devant lui. IBM (WorkPad), Sony (CLIÉ), Samsung (SPH-*) ou encore Garmin (iQue) proposèrent des périphériques Palm OS reprenant généralement des caractéristiques qui peuvent paraître bien modestes aujourd'hui : écran LDC monochrome ou couleur de 160×160 ou 320×320 pixels, interface tactile résistive, connectivité PC/Mac série, stylet, parfois clavier mécanique ou encore rétroéclairage électroluminescent. Le tout, le plus souvent mû par un processeur 16 bits dérivé du Motorola 68000, le Motorola DragonBall et, dans les



Palm OS, alias Garnet OS, est le système initialement conçu par Palm Inc pour ses PDA. Ici, ce Palm IIIxe fait fonctionner la version 3.5, l'une des plus populaires, introduite avec l'arrivée des Palm IIIc en 2000. Il est théoriquement possible de mettre à jour le firmware de ce modèle vers Palm OS 4.1 [11], mais personnellement, je ne m'y risquerai pas.

PDA de la fin de période par des SoC ARM (Ti OMAP, Samsung S3C2410 ou Intel XScale (oui, Intel a fabriqué des processeurs ARM avant de revendre la technologie à Marvell)).

Nous avons aujourd'hui tout un écosystème de PDA vintages **[1]** allant du PDA à la montre en passant par le baladeur MP3 ou le récepteur GPS. Il y a quelques années encore, ces équipements se bradaient pour une poignée d'euros sur les sites d'enchères en ligne ou aux marchés aux puces. Mais ce qui était hier des déchets électroniques pour certains est maintenant devenu des pièces de collection pour d'autres. Le prix d'un Palm III avoisine la centaine d'euros, bien plus pour un modèle U.S. Robotics ou un Handspring Visor Neo (boîtier translucide) fonctionnel à quelque 200 €, et parfois plus de 300 € avec sa boîte, ses manuels et tous ses accessoires. Il est fort probable que cette tendance continue, exactement comme on peut le voir avec d'autres matériels de la même époque.

Celui en ma possession est un Palm IIIxe, équipé d'un CPU Motorola 68328EZ Dragonball à 16 MHz, 8 Mo de RAM, 2 Mo de flash, un écran tactile LCD monochrome (4 bits - 16 niveaux de gris) de 160×160 pixels et faisant fonctionner Palm OS 3.5. Le tout avec son « cradle » permettant la connexion à un PC via une liaison série RS-232 (connecteur DB9) et alimenté par deux piles de type AAA. Notez que cette paire de piles permet de faire fonctionner le PDA durant un bon mois, voir plus, pour une utilisation régulière courante. Nous sommes loin des 48 h moyennes de nos smartphones, mais ceci date d'une époque où l'on ne changeait la pile de sa montre qu'au bout d'une paire d'années...

2. CONNEXION AU PC ET OUTILS DE GESTION

Comme dit précédemment, les Palm et appareils compatibles communiquent avec le PC pour se synchroniser via une connexion série. Le classique connecteur DB9 (et DB25) RS-232 a disparu depuis fort longtemps de nos machines de bureau, remplacé par l'USB pour le même type d'usage. Connecter le *cradle* d'un Palm nécessite donc l'utilisation d'un adaptateur série/USB. Mais attention, il ne s'agit pas du type de matériel que l'on utilise généralement avec des montages électroniques du type Arduino. Nous parlons ici d'une norme faisant usage de tensions positives *et* négatives, généralement +12 v/-12 v. L'adaptateur doit donc respecter cette norme et prend généralement la forme d'un convertisseur disposant d'un connecteur DB9 mâle d'un côté et USB (A, B, mini ou micro) de l'autre.

Vous pourrez trouver ce type de périphériques (RS-232 vers USB-A, le plus souvent) pour une dizaine d'euros sur les sites habituels (eBay, Amazon, etc.) et ils seront parfaitement pris en charge par une machine GNU/Linux, qu'il s'agisse d'un PC ou d'un SBC (Raspberry Pi ou autre).

Côté logiciel, comme on peut s'en douter, les applicatifs historiques, le plus souvent pour Windows ou macOS (Mac OS X, OS X voir Mac OS 9), n'ont aucune chance de fonctionner sur un système moderne. Fort heureusement cependant, un outil *open source* du nom de Pilot-Link est encore vaguement disponible (même si le site **pilot-link.org** est aujourd'hui inaccessible), y compris sous la forme d'un paquet si votre distribution est assez ancienne (Stretch ou Buster). Avec Raspberry Pi OS cependant, nous n'avons aucune trace de l'outil (tout comme dans une Debian/Ubuntu récente) et c'est l'occasion rêvée de rapidement voir comment « backporter » un tel paquet.

Normalement, on évitera de compiler sur un système embarqué, mais comme un PC, une Raspberry Pi que l'on choisit d'utiliser pour jouer avec un Palm Pilot peut être vue comme un ordinateur générique et l'on se permettra alors un petit écart à la règle. Pour construire un paquet, nous avons besoin d'outils et nous commencerons donc par installer `dpkg-dev`, nous fournissant les commandes `dpkg-source` et `dpkg-buildpackage`.

Nous pourrons ensuite pointer notre navigateur sur la page dédiée au paquet `pilot-link` sur le site Debian **[2]**. Là, nous trouvons les dernières versions binaires pour les différentes plateformes supportées, mais ce qui nous intéresse se trouve sur le côté. À droite de la page se trouve une liste de liens permettant de télécharger les éléments du paquet source : `pilot-link_0.12.5-dfsg-2.dsc`, `pilot-link_0.12.5-dfsg.orig.tar.gz` et `pilot-link_0.12.5-dfsg-2.debian.tar.xz`.

Nous récupérons donc ces fichiers et les plaçons à un endroit quelconque du système de fichiers (`~/DEB/` par exemple). Ceux-ci permettent de créer une arborescence composée des sources *upstream*, patchées par Debian et intégrant les éléments de configuration du paquet. Pour créer cette arborescence, nous utilisons la commande :

```
$ dpkg-source -x pilot-link_0.12.5-dfsg-2.dsc
gpgv: unknown type of key resource 'trustedkeys.kbx'
[...]
dpkg-source: info: mise en place de 10_clie_sj22.patch
dpkg-source: info: mise en place de 31_pilot-addresses.1
```

Nous obtenons un répertoire `pilot-link-0.12.5-dfsg/` dans lequel nous entrerons dans un instant. Mais avant cela, nous devons installer ce dont a besoin le paquet binaire pour être créé (les dépendances de construction). `apt-get` peut être utilisé pour cela :

```
$ sudo apt-get build-dep ./pilot-link-0.12.5-dfsg
Note, utilisation du répertoire
  ./pilot-link-0.12.5-dfsg pour obtenir
  les dépendances de construction
Lecture des listes de paquets... Fait
Construction de l'arbre des dépendances... Fait
[...]
```

Notez que `apt-get build-dep` utilise normalement un nom de paquet, mais ici `pilot-link` n'existe pas dans les dépôts et nous devons donc nous rabattre sur le contenu du répertoire. Nous pouvons maintenant nous placer dans ce dernier et sommes presque prêts pour la construction, mais nous ne voulons pas faire cela comme des sauvages. Il convient donc d'éditer `debian/changelog` pour garder une trace de nos manipulations (et éviter de brutaux avertissements concernant la signature du paquet). Pour cela, vous pouvez utiliser la commande `dch -i` du paquet `devscripts`, mais ce dernier installera une myriade de dépendances. Une approche plus économe est de simplement éditer le fichier à la main et d'ajouter soigneusement une entrée dans le log :

```
pilot-link (0.12.5-dfsg-3) unstable; urgency=medium

  * backport

-- Nom <user@dom.tld>  Mon, 07 Feb 2022 15:53:00 +0200

pilot-link (0.12.5-dfsg-2) unstable; urgency=medium

  * Fix "deprecation of python-support" use dh-python instead of
    python-support (Closes: #786254)
[...]
```

Ceci fait, nous pouvons lancer la construction avec `dpkg-buildpackage -b` qui prendra un certain temps. Au terme de l'opération, ce n'est pas un, mais neuf paquets qui auront été créés. En effet, Pilot-link n'est pas qu'un ensemble d'outils, mais également un jeu de bibliothèques permettant de créer d'autres outils capables de dialoguer et de synchroniser des données avec un équipement Palm OS. Nous n'avons cependant pas besoin de tout et pouvons nous contenter d'installer uniquement deux de ces paquets :

```
$ sudo dpkg -i \
pilot-link_0.12.5-dfsg-3_armhf.deb \
libpisock9_0.12.5-dfsg-3_armhf.deb
```

Il ne nous reste plus qu'à faire un petit essai en utilisant `pilot-xfer` pour lister les applications installées :

```
$ pilot-xfer -p /dev/ttyUSB0 -l
   Listening for incoming connection on
     /dev/ttyUSB1... connected!
   Reading list of databases in RAM...
   AddressDB
   ExpenseDB
   MailDB
   MemoDB
[...]
   AddressStatesDB
```

```
AddressTitlesDB
VendorsDB

List complete. 24 files found.

Thank you for using pilot-link.
```

`-p` permet de spécifier le port série à utiliser (ici, l'adaptateur USB) et `-l` liste les applications. Une fois la commande validée et le Palm sur son *cradle*, une simple pression sur le bouton de synchronisation de ce dernier et... « pililiiii »... « pililuuuu »... deux petites mélodies caractéristiques qui rappellent de lointains souvenirs (notez que Pilot-link vous remercie sympathiquement de l'utiliser, chose assez rare de nos jours).

Vous voici maintenant en mesure d'installer (option `-i`) des fichiers et donc des applications sur votre appareil. Il ne reste plus qu'à trouver un moyen d'en développer...

3. PAS DE MATÉRIEL VINTAGE ? ÉMULATION ?

La réponse est « non ». Du moins pas sous GNU/Linux et à partir des sources officielles **[3]**. L'émulateur le plus populaire est sans le moindre doute POSE (*Palm OS Emulator*) dont la dernière version 3.5-2 date d'avril 2002. Cet émulateur, écrit en C++, utilise le *toolkit* FLTK et plus exactement, comme le précise le fichier `_Building.txt`, FLTK 1.0.7 avec un support minimal pour la version 1.1.0 qui était encore en bêta lors de la dernière mise à jour de POSE.

Le « cradle » était livré avec le Palm et permettait la connexion, via un port série, à un PC ou un Mac. Le petit bouton à l'avant du socle permet de déclencher la synchronisation et/ou toute opération constituant un échange de données. Remarquez le câble de quelque 2,3 mètres. Il y a 20 ans, il semblerait qu'on était un peu moins radin sur la connectique...

Aujourd'hui, Debian propose FLTK en version 1.1.10 (`libfltk1.1-dev`) ou 1.3.4 (`libfltk1.3-dev`). Pire encore, POSE utilise `fluid`, le générateur d'interface FLTK, mais le paquet éponyme dans Debian possède une dépendance vers `libfltk1.3`. Ceci s'ajoute à bien d'autres problèmes liés à la compilation du code C++ qui était prévu pour GCC 2.95 / EGCS 1.1.2 (nous en sommes à GCC 11.2 !) et une bibliothèque standard C++ d'un autre âge. La quantité de modifications à apporter dans le code pour supporter un système récent est phénoménale et même si, comme moi, vous passez des heures à péniblement les traquer et les corriger, vous buterez sur un problème d'édition de liens, car FLTK n'est tout simplement pas dans la bonne version.

C'est triste, mais force est de constater que POSE n'a pas survécu au temps qui passe inéluctablement. Le projet a été abandonné trop longtemps et reposait sur un *toolkit* qui n'est plus qu'un vague souvenir pour bon nombre de développeurs, contrairement à GTK+ ou Qt, qui sont cités dans la documentation (tout comme « RedHat Linux 6.0 », « XFree86 3.3.3.1 », ou encore le « kernel 2.2.5 »... diantre, que ça passe vite).

Les ports série RS-232 ont disparu depuis bien longtemps de nos machines de bureau et portables, mais il est toujours possible d'en faire usage en utilisant un adaptateur USB comme ceux-ci.

Il existe vraisemblablement une solution pour utiliser le binaire de l'époque sur un système actuel (Ubuntu 18.04 LTS 64-Bit **[4]**), mais la simple lecture des manipulations fait froid dans le dos tant la concession sur les usages « propres » d'un système est énorme. Installer manuellement un binaire 32 bits issu d'un RPM accompagné d'une dépendance à une bibliothèque X tiré d'une vieille Debian, le tout à grands coups de `sudo rsync`, n'est pas quelque chose que je suis près à faire pour avoir un émulateur fonctionnel. Vous ferez peut-être un autre choix... mais je préfère me passer de POSE.

Remarquez cependant que ne pas disposer d'un émulateur fonctionnel va bien au-delà du simple fait de pouvoir tester un système sans pour autant avoir le précieux matériel à disposition. Un émulateur est également un outil de développement dans le sens où il rend la mise au point de programmes bien plus aisée en permettant d'inspecter la mémoire, connecter un GDB ou agir sur les registres du processeur. La vétusté de POSE, permettant historiquement ce type d'utilisation, est donc un handicap plus important qu'il n'y paraît.

– Développer pour PDA de plus de 20 ans en 2022, c'est possible ! –

Peut-être qu'un développeur courageux daignera, à un moment, ressusciter POSE. Ceci demandera de bonnes compétences en C++, car le projet repose massivement sur les exceptions, les *templates* et autres concepts propres au langage, totalement étrangers à votre humble serviteur...

La source d'alimentation du IIIxe consiste en deux piles AAA permettant un fonctionnement d'un mois ou deux (selon usage). Sur les modèles plus récents, les piles ont été remplacées par un accu lithium-ion (Palm IIIc, par exemple).

4. INSTALLER L'ENVIRONNEMENT DE DÉVELOPPEMENT

Pour développer des applications Palm OS en 2022 sous GNU/Linux, cela se passe exactement comme il y a presque 25 ans. Nous avons besoin d'une chaîne de compilation m68k, d'un SDK pour Palm OS et de différents outils permettant de gérer les ressources de la GUI, de créer un exécutable dans un format dédié ou encore de transférer l'application sur le périphérique (heureusement réglé grâce à Pilot-Link).

La première étape pour assembler tout cela débute avec la chaîne de compilation et les outils permettant de produire des programmes au format PRC (*Palm Resource Code*). Un projet né dès 1997 regroupe ces éléments, c'est *prc-tools*, mais celui-ci n'a plus évolué depuis la version 2.3 datant de 2003. Heureusement, il existe toujours des développeurs passionnés par les Palm et une version compatible avec des distributions GNU/Linux (et macOS) récentes est maintenue par Chuan Ji et disponible sur GitHub, c'est *prc-tools-remix* **[5]**.

Les scripts de compilation nécessitent de menus ajustements, puisqu'il est parfaitement hors de question qu'ils exécutent des commandes via `sudo` et installent tout et n'importe quoi dans `/usr`. Mais fort heureusement, il est possible de créer un paquet Debian pour une installation « réglementaire ». Ceci fonctionnera tout aussi bien sur PC avec n'importe quelle distribution compatible Debian (type Ubuntu ou Kali), mais également sur Raspberry Pi.

Pour créer ce paquet, nous commençons par installer les dépendances de construction avec `sudo apt-get install build-essential autoconf automake libncurses-dev flex bison texinfo gperf`, puis dans un emplacement de notre préférence, nous clonons le dépôt Git :

```
$ git clone https://github.com/jichu4n/prc-tools-remix.git
$ cd prc-tools-remix
```

Nous trouvons, dans le sous-répertoire `tools/`, tout le nécessaire pour compiler, construire et créer notre paquet. Nous éditons simplement `build-deb.sh` pour commenter la ligne `install_build_deps` en fin de script et donc éviter de faire appel inutilement à `sudo`. Nous pouvons éventuellement également éditer `build.sh` pour ajouter un paramètre `-j` à `make` suivi d'une valeur représentant le nombre de tâches à exécuter en parallèle, mais ceci n'est pas forcément nécessaire.

Pour créer le paquet, il nous suffit d'invoquer `build-deb.sh` en spécifiant en argument deux paramètres :

- la distribution utilisée (le nom sera simplement ajouté au fichier `.deb` obtenu) ;
- l'architecture pour laquelle nous créons le paquet. Dans bien des cas, ce sera `amd64`, mais sur Pi, il faudra opter pour `armhf` ou `arm64`. Vous pouvez obtenir l'architecture de paquets courante avec `dpkg --print-architecture` et utiliser la valeur rapportée.

Au terme de la construction qui devrait prendre plus ou moins de temps selon votre configuration, vous devez obtenir quelque chose comme ceci :

```
[...]
+ package
+ cd /home/denis/tmp/artPALM/prc-tools-remix/dist
+ mkdir -p DEBIAN
+ cat
+ cd /home/denis/tmp/artPALM/prc-tools-remix
+ dpkg-deb --build /chemin/prc-tools-remix/dist
  prc-tools-remix_2.3.5~stretch_amd64.deb
dpkg-deb: building package 'prc-tools-remix'
  in 'prc-tools-remix_2.3.5~stretch_amd64.deb'.
$
```

Le fichier résultant, ici `prc-tools-remix_2.3.5~stretch_amd64.deb` pourra alors être installé avec `sudo dpkg -i` et vous devez alors disposer de deux compilateurs fonctionnels. Un pour les Palm et Handspring Visor à base de Motorola 68000 (Motorola DragonBall) :

```
$ m68k-palmos-gcc -v
Reading specs from /usr/lib/gcc-lib/m68k-palmos/2.95.3-kgpd/specs
Reading specs from /usr/lib/gcc-lib/m68k-palmos/specs
gcc version 2.95.3-kgpd 20010315 (release)
```

Et un autre pour ceux reposant sur l'architecture ARM (Palm Tungsten, Zire 21, Handspring Treo et suivants) :

```
$ arm-palmos-gcc -v
Reading specs from /usr/lib/gcc-lib/arm-palmos/3.3.1/specs
Reading specs from /usr/lib/gcc-lib/arm-palmos/specs
[...]
Thread model: single
gcc version 3.3.1 (prc-tools)
```

GCC 2.95.3 et 3.3.1 ne sont pas tout jeunes, mais largement suffisants pour créer des applications pour une plateforme d'un tel âge. Notez que ceci est relativement « bricolo », puisque la configuration de la construction du paquet n'utilise pas `fakeroot` et que les fichiers sont installés dans le système sous l'identité de l'utilisateur ayant procédé à la compilation. `/usr/share/prc-tools`, par exemple, ou encore `/usr/lib/gcc-lib/m68k-palmos` n'appartiennent pas à `root` et l'utilisateur standard peut, de ce fait, écrire dans ces répertoires puisqu'ils lui appartiennent. C'est quelque chose qui mériterait d'être corrigé, en créant, par exemple de vrais fichiers de configuration dans `debian/`.

Une chaîne de compilation, même accompagnée de ses outils complémentaires (`build-prc`, `palmdev-prep`, etc.) ne sera pas en mesure de produire des binaires viables pour Palm OS. Il sera donc nécessaire d'installer les SDK qui, de longue date, ne sont plus proposés par le fabricant. Mais, là encore, une solution existe puisque l'auteur de *prc-tools-remix* met également à disposition ces éléments sur GitHub **[6]**.

Une fois encore, on évitera bien soigneusement d'utiliser des scripts faisant appel à `sudo` et l'on préférera simplement cloner le dépôt pour copier manuellement son contenu dans `/opt/palmdev`, le chemin de recherche par défaut. Ceci fait, il sera nécessaire d'invoquer `palmdev-prep` ainsi :

```
$ palmdev-prep -d sdk-3.5 -v
Checking SDKs in /opt/palmdev
  sdk-3.5       headers in 'include', libraries in 'lib'
  sdk-4         headers in 'include', libraries in 'lib'
  sdk-5r3       headers in 'include', libraries in 'lib'
  sdk-3.1       headers in 'include', no libraries
  sdk-5r4       headers in 'include', libraries in 'lib'
  sdk-1         headers in 'include', no libraries
  sdk-2         headers in 'include', no libraries

Writing SDK details to configuration files...
Wrote arm-palmos specs to '/usr/lib/gcc-lib/arm-palmos/specs'
Wrote m68k-palmos specs to '/usr/lib/gcc-lib/m68k-palmos/specs'
Parsed trap numbers in '/opt/palmdev/sdk-3.5/include/Core/CoreTraps.h'
  and wrote them to '/usr/share/prc-tools/trapnumbers'
...done
```

Cet outil, fourni par *prc-tools-remix*, scanne automatiquement le système à la recherche des différentes versions du SDK et met à jour les fichiers `specs` pour chaque chaîne de compilation. Ceux-ci contiennent les informations permettant au compilateur de retrouver les *headers* et bibliothèques pour chaque version de Palm OS. L'option `-d` nous permet de spécifier un SDK par défaut, avec ici `sdk-3.5` correspondant au Palm OS présent sur mon Palm IIIxe. Notez qu'il nous est possible de lancer cette commande sans problème avec l'utilisateur standard, uniquement parce que les répertoires en question, dans `/usr/lib` et `/usr/share`, lui appartiennent.

– Développer pour PDA de plus de 20 ans en 2022, c'est possible ! –

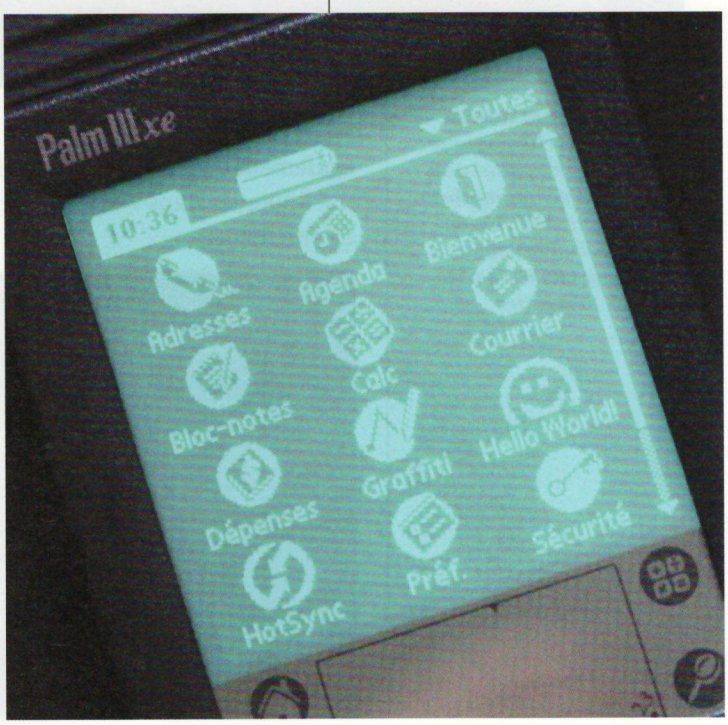
À ce stade, il nous est déjà possible de produire un binaire pour Palm OS pour tester notre environnement. Dans le sous-répertoire `tools/hello-world-app` des sources de *prc-tools-remix*, vous trouverez un « Hello World » basique composé d'un simple fichier source C d'une trentaine de lignes. Nous pouvons le compiler et créer un fichier PRC avec :

```
$ m68k-palmos-gcc hello.c -o hello
$ m68k-palmos-obj-res hello
$ build-prc hello.prc "Hello, World" WRLD *.hello.grc
```

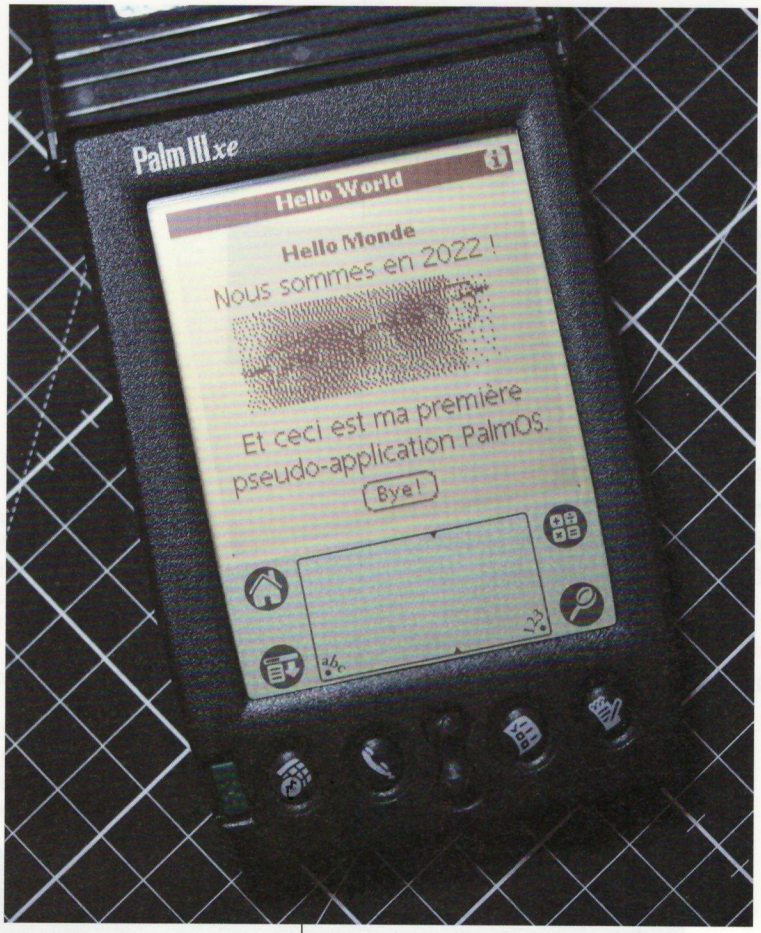
La première ligne nous permet de produit un binaire `hello` pour architecture « mc68k » au format COFF. La seconde découpe le binaire en différents fichiers en fonction des sections dans lequel se trouve les objets et enfin, la troisième réutilise ces éléments pour créer le PRC [7]), contenant à la fois le binaire, ses ressources et les méta-informations, tels son nom et son identifiant (`WRLD`). Ce `hello.prc` peut ensuite être installé sur la machine en utilisant `pilot-xfer -p /dev/ttyUSB0 -i hello.prc`. Notez que le passage par `m68k-palmos-obj-res` n'est ici donné qu'à titre d'exemple, il est également possible d'utiliser directement le binaire `hello` avec `build-prc` ainsi : `build-prc hello.prc "Hello, World" WRLD hello`.

On pourrait croire que notre installation est terminée, du fait de la possibilité de compiler, construire, d'installer et d'exécuter cet exemple, mais en réalité, il manque une pièce importante à notre puzzle. Ce « Hello World » très basique n'est absolument pas typique d'une application Palm OS comprenant généralement une interface utilisateur plus étoffée. À la manière d'autres processus de développement d'applications graphiques comme ceux utilisant Glade pour GTK+ ou Qt Designer pour Qt, une application Palm OS se divise entre une partie « interface » et une partie « code » (correspondant peu ou prou à *Vue* et *Contrôleur* du modèle MVC).

Notre « Hello World » n'a pas d'interface graphique, ce n'est qu'un code qui place un texte à l'écran et attend l'événement `appStopEvent` pour rendre



L'une des grandes nouveautés du Palm IIIxe était son écran tactile « Super-Twisted Nematic » (STN) d'une résolution phénoménale de 160×160 pixels. Celui-ci était rétroéclairé à l'aide d'un panneau électroluminescent, une technologie excessivement moderne à l'époque.



Voici le résultat de tout ce travail archéologique : une modeste application Palm OS créée, compilée, installée et exécutée en 2022 sur un PDA apparu sur le marché il y a plus de deux décennies.

la main. Si nous voulons créer quelque chose de plus étoffé, nous devons écrire une description de l'interface, la transformer en ressources binaires et combiner ces dernières avec notre binaire compilé pour créer le PRC. L'outil pour créer ces ressources à partir d'un script est PilRC **[8]** dont la dernière version 3.2.90 date de 2007. Là encore, Chuan Ji nous simplifie le travail, en mettant à disposition une version sensiblement mise à jour pouvant fonctionner sur un OS moderne.

Comme précédemment, après avoir récupéré les sources via `git`, nous pouvons nous placer dans `tools/`, éditer `build-deb.sh` pour supprimer la nécessité d'utiliser `sudo` (et donc installer manuellement les paquets `autoconf` et `automake`), puis invoquer le script avec les mêmes arguments que précédemment. Plus simplement, nous pouvons également éviter une nouvelle installation de binaire sous une identité douteuse dans `/usr` et procéder à la configuration et compilation manuellement :

```
$ cd pilrc/unix
$ autoreconf -vis

$ ./configure
autoreconf: Entering directory `.'
autoreconf: configure.ac: not using Gettext
autoreconf: running: aclocal
[...]
configure: creating ./config.status
config.status: creating Makefile
config.status: executing depfiles commands

$ make
```

Nous obtenons alors un binaire `pilrc` que nous pouvons placer à un endroit quelconque de notre système de fichiers (`$HOME/bin` par exemple) pour ensuite invoquer la commande avec son chemin complet.

Ceci termine l'installation de notre environnement de développement et nous sommes maintenant prêts à développer !

4.1 Juste une petite application

La création d'applications pour Palm OS est un sujet à part entière qui mériterait (ou pas) un article à lui seul. Nous n'allons donc pas nous lancer dans une telle aventure didactique ici, mais il serait tout de même dommage d'avoir installé tout cela pour rien. Je vous propose donc de rapidement créer une application graphique résumant la base d'un développement Palm OS, sous la forme d'un « Hello World » un peu plus évolué que le précédent, mais sans m'étendre outre mesure sur les détails.

Ceci commence par le fichier `hello.rcp` formant le script décrivant l'interface qui sera transformée en ressources binaires par `pilrc`. Vous pouvez voir cela un peu comme un fichier HTML décrivant une page web :

```
#include "hello.h"

FORM ID helloForm AT (0 0 160 160)
USABLE
MODAL
HELPID helloHelp
BEGIN
  TITLE "Hello World"
  LABEL "Hello Monde" AUTOID
    AT (CENTER 24) FONT 1
  LABEL "Nous sommes en 2022 !"
    AUTOID AT (CENTER PREVBOTTOM+1) FONT 2
  FORMBITMAP AT (20 PREVBOTTOM+1) BITMAP 2005
  LABEL "Et ceci est ma première" AUTOID
    AT (CENTER PREVBOTTOM+40) FONT 2
  LABEL "pseudo-application Palm OS."
    AUTOID AT (CENTER PREVBOTTOM+1) FONT 2

  BUTTON "Bye !" ID Ok AT (CENTER 140 AUTO AUTO)
END

STRING helloHelp "Juste un \"Hello World\".\nEnjoy !"

ICON "hello.bmp"
SMALLICON "hellosmall.bmp"

BITMAP ID 2005 "meeeee.bmp"

VERSION 1 "0.0.1"
```

Cette description comprend des identifiants numériques permettant de référencer les objets qui la composent. L'inclusion du fichier `hello.h` nous permet d'utiliser des noms à la place des valeurs, mais également de faire la liaison avec le code C qui va accéder à certains de ces objets :

```
#define helloForm     1000
#define Ok            1001
#define helloHelp     1002
```

Le reste se passe dans `hello.c` dont le travail consiste principalement à réagir à des événements (lancement, affichage, interaction, etc.) :

```
#include "hello.h"
#include <System/SystemPublic.h>
#include <UI/UIPublic.h>

UInt32 PilotMain(UInt16 cmd, MemPtr cmdPBP, UInt16 launchFlags)
{
  short err;
  EventType e;
  FormType *pfrm;

  // On ne gère que NormalLaunch (pas Reset, Find, GoTo, etc.)
  if(cmd == sysAppLaunchCmdNormalLaunch) {
    FrmGotoForm(helloForm);
    while(1) {
      EvtGetEvent(&e, 100);
      if(SysHandleEvent(&e))
        continue;
      // pas de callback pour le menu
      if(MenuHandleEvent((void *)0, &e, &err))
        continue;
      // gestion événement
      switch (e.eType) {
        // clic sur "Ok"
        case ctlSelectEvent:
          if(e.data.ctlSelect.controlID == Ok)
            goto _quit;
          goto _default;
          break;
        // Formulaire chargé
        case frmLoadEvent:
          FrmSetActiveForm(FrmInitForm(e.data.frmLoad.formID));
          break;
        // formulaire ouvert
        case frmOpenEvent:
          pfrm=FrmGetActiveForm();
          FrmDrawForm(pfrm);
          break;
        // Menu non géré
```

```
        case menuEvent:
          break;
        // Arrêt
        case appStopEvent:
          goto _quit;
        default:
_default:
        // dispatch événement
        if(FrmGetActiveForm())
          FrmHandleEvent(FrmGetActiveForm(), &e);
      }
    }
_quit:
    // nettoyage
    FrmCloseAllForms();
  }
  return 0;
}
```

Vous serez peut-être surpris de la présence de labels et de `goto` dans ce code C, mais c'est là quelque chose de tout à fait typique d'un développement Palm OS. Notre code se résume donc à une simple boucle `while` réagissant sur ce que retourne `EvtGetEvent()` dans `e` et que nous traitons ensuite via un `switch`/`case`.

Ces fichiers, plus les images BMP seront utilisés via les différents outils installés pour construire notre application PRC. Pour ce faire, classiquement, nous composons un `Makefile` relativement simple :

```
CC = m68k-palmos-gcc
CFLAGS = -O2 -palmos3.5
PILRC = "/home/denis/bin/pilrc"

all: hello.prc

hello.prc: hello bin.stamp
        build-prc hello.prc "Hello World!" WRLD hello *.bin

hello: hello.o
        $(CC) $(CFLAGS) -o hello hello.o

hello.o: hello.c hello.h
        $(CC) $(CFLAGS) -c hello.c

bin.stamp: hello.rcp hello.h hello.bmp
        $(PILRC) hello.rcp

clean:
        rm -f *.o *.bin *.prc hello
```

Je vous laisse le soin de lire la documentation de `build-prc` et de parcourir les nombreux exemples qu'il est encore possible de trouver sur le Web, pour comprendre les tenants et aboutissants d'un tel développement. Mais méfiez-vous, on se prend rapidement au jeu, car en dehors de points très spécifiques, l'ensemble est relativement facile à prendre en main et la satisfaction de voir le résultat effectivement s'exécuter sur le périphérique est tout simplement addictive. Nous sommes très loin des environnements de développement et des frameworks complexes et difficiles à appréhender comme Android Studio ou Xcode/iOS, car tout ceci date d'une époque où tout était réellement bien plus simple...

CONCLUSION

Personnellement, je trouve excessivement agréable de constater que certaines choses d'un lointain passé (à une échelle informatique du terme) sont encore suffisamment vivantes pour pouvoir toujours en faire l'expérience. Ceci est vrai pour le Palm, comme nous venons de le voir, mais également pour toute une collection de machines, ordinateurs vintages, consoles et systèmes qui sinon risqueraient de n'être que de simples reliques obsolètes et perdraient alors toute réelle signification.

En guise donc de conclusion, je voudrais remercier ici tous ceux qui, d'une manière ou d'une autre, gardent vivante notre histoire et l'héritage de ce qui fait de nous des développeurs. Et plus particulièrement Robbie Nesmith pour son *tweet* annonçant la disponibilité récente de son application PalmWordle **[9]** m'ayant ainsi poussé à fouiller et retrouver mon Palm IIIxe, Chuan Ji pour *prc-tools-remix* et les différents éléments de l'environnement de développement qu'il met à disposition sur GitHub et enfin, Éric Poncet pour maintenir encore aujourd'hui en ligne ses tutoriels **[10]** d'il y a 20 ans sur le développement d'application Palm OS et qui sont une précieuse source de documentation, pour qui veut se lancer dans l'aventure. **DB**

RÉFÉRENCES

[1] https://en.wikipedia.org/wiki/List_of_Palm_OS_devices

[2] https://packages.debian.org/buster/pilot-link

[3] https://sourceforge.net/projects/pose/files/pose/

[4] https://palm2000.com/projects/installingPOSEonUbuntu1804LTS.php

[5] https://github.com/jichu4n/prc-tools-remix

[6] https://github.com/jichu4n/palm-os-sdk

[7] https://en.wikipedia.org/wiki/PRC_(Palm_OS)

[8] http://pilrc.sourceforge.net/

[9] https://github.com/RobbieNesmith/PalmWordle

[10] http://mobile.eric-poncet.com/palm/tutorial/

[11] https://palmdb.net/app/upgrade-os4