ÉLECTRONIQUE | EMBARQUÉ | RADIO | IOT

HACKABLE

L'EMBARQUÉ À SA SOURCE

N° 41
MARS / AVRIL 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 - 41 - F: 14,90 € - RD

CPPAP : K92470

L'électronique pérenne et innovante

Bureau d'études en électronique et informatique embarquée, Agilack vous accompagne dans la conception, le prototypage et l'industrialisation de vos produits.

AGILACK

Cowlab

Des outils plus simples, des développements plus rapides avec Cowlab

OpenHardware

Apprenez à contribuer tout en protégeant votre propriété intellectuelle

CAO

Des logiciels Libres et des formats ouverts pour une pérénité maximale

Prototype

Fabrication des premières unités et des petites séries

Software

Développement de firmwares, drivers, BSP, ...

www.agilack.fr contact@agilack.fr

ÉDITO

La profusion de cartes, SBC et plateformes embarquées de toutes sortes, depuis quelques années déjà, a tendance à nous faire facilement oublier que même si Internet se souvient (théoriquement) de tout, rien n'est éternel.

Aujourd'hui, vous achetez une carte pour une poignée d'euros en vous disant que, même si vous n'en faites rien de suite, car occupé par un nouveau projet apparu entre l'achat et la livraison, ce n'est pas bien grave, vous verrez cela plus tard. Malheureusement, telle une pizza industrielle en promotion, cette carte a bel et bien une date d'expiration. « Bien sûr, un SoC d'il y a quelques années n'est plus dans la course, mais pour un projet générique, cela fera bien l'affaire. », vous dites-vous. Et complétez ce rassurant raisonnement avec « Même avec un BSP un peu ancien et un noyau 4.x, cela restera utilisable. Non ? ». Et la réponse n'est tristement pas toujours positive...

Prenons un exemple. Vous souvenez-vous du C.H.I.P. ? Il y a quelques années, la startup californienne Next Thing Co avait annoncé, produit et distribué un SBC à prix défiant toute concurrence (9 $), vu par certains comme un « Raspberry Pi killer » et basé sur le SoC Allwinner R8. Matériellement, un ARM Cortex-A8 à 1 GHz avec 512 Mo de SDRAM est parfaitement viable aujourd'hui, mais tout l'écosystème a littéralement disparu. Next Thing Co n'existe plus, pas plus que ses dépôts GitHub, sites web, forums officiels et il ne reste que des bribes d'information disséminées à travers le Net...

Sortir un C.H.I.P. d'un tiroir aujourd'hui et vouloir ne serait-ce que reconstruire le système initial (et périmé) tient de l'archéologie. Un effort communautaire avait vu le jour après la faillite en 2018, sous la forme d'un wiki, mais, en 2021, ce site a également cessé de fonctionner et n'est plus désormais accessible que via WayBack Machine. Ce SBC, malgré ses caractéristiques acceptables actuellement, est devenu un presse-papier, parce que plus personne n'y porte d'intérêt. C'est aussi simple que cela.

La leçon à retenir est évidente : un engouement ponctuel pour un SBC, même aux caractéristiques fabuleuses, ne rend pas la solution pérenne et donc viable, seule une communauté de développeurs active et durable permet cela. Choisir une plateforme, comme acheter une maison, c'est aussi acheter ses voisins. Dans les deux cas, c'est quelque chose à posément évaluer...

Denis Bodor

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 €

Suivez-nous sur Twitter
@hackablemag

Alsace

SOMMAIRE

À PROPOS DE HACKABLE...

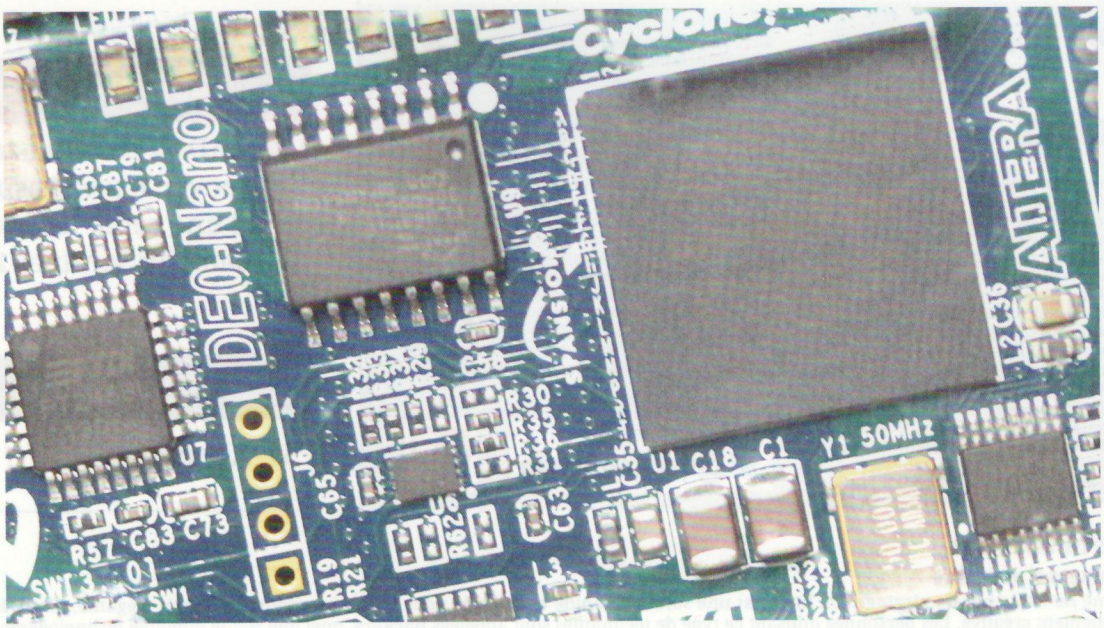
HACKS, HACKERS & HACKABLE

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.

UTILISEZ DOCKER POUR VOS ENVIRONNEMENTS DE DÉVELOPPEMENT

Denis BODOR

Docker et les conteneurs logiciels font maintenant partie intégrante de la vie du sysadmin. Rappelons que les conteneurs permettent généralement d'isoler les processus, souvent des services réseau, afin de les détacher du système et de les faire s'exécuter dans des environnements distincts. Ainsi la ou les applications en question « vivent » dans leur petit monde, ignorant tout de la réalité du système faisant fonctionner le conteneur, mais aussi des autres applications dans leur propre environnement contenu. Cette mécanique peut être également très intéressante et avantageuse pour le développeur embarqué…



– Utilisez Docker pour vos environnements de développement –

Avant toute chose, précisons ici que nous parlons de GNU/Linux, et bien que les mécanismes de conteneurs et Docker existent dans d'autres systèmes, nous nous limiterons à cet environnement. La distribution n'a pas réellement d'importance, et ceci est valable également pour WSL, du moment que vous disposez d'une installation fonctionnelle de Docker (comprendre « que vous avez installé le paquet qui va bien »).

Je ne m'étendrai pas non plus sur les principes de fonctionnement, les différentes déclinaisons des architectures, l'orchestration ou tout autre contexte qui nous écarterait de notre sujet principal. L'objet de cet article est simplement de voir ce que Docker peut nous permettre de simplifier dans nos activités de développement et de construction de *firmware*.

À ce propos justement. Si vous êtes comme moi, à intervalles réguliers, vous êtes coincé entre deux options lorsque votre distribution (Debian en particulier, mais pas seulement) commence à prendre des rides. De plus en plus de paquets affichent des numéros de version qui paraissent venir d'un autre âge et de plus en plus d'outils, compilés localement, demandent des bibliothèques dont vous ne disposez pas ou plus matures que celles actuellement installées. Là, deux choix s'offrent à vous, un `dist-upgrade` à vos risques et périls, soit une réinstallation bien propre assortie d'une sympathique période de reconfiguration et de copies de fichiers. Ne nous voilons pas la face, j'utilise GNU/Linux depuis 25 ans et l'expérience m'a prouvé que, peu importe le choix qu'on fera, ceci n'est jamais exempt de problèmes, de frayeurs, de larmes et d'énervement en tous genres.

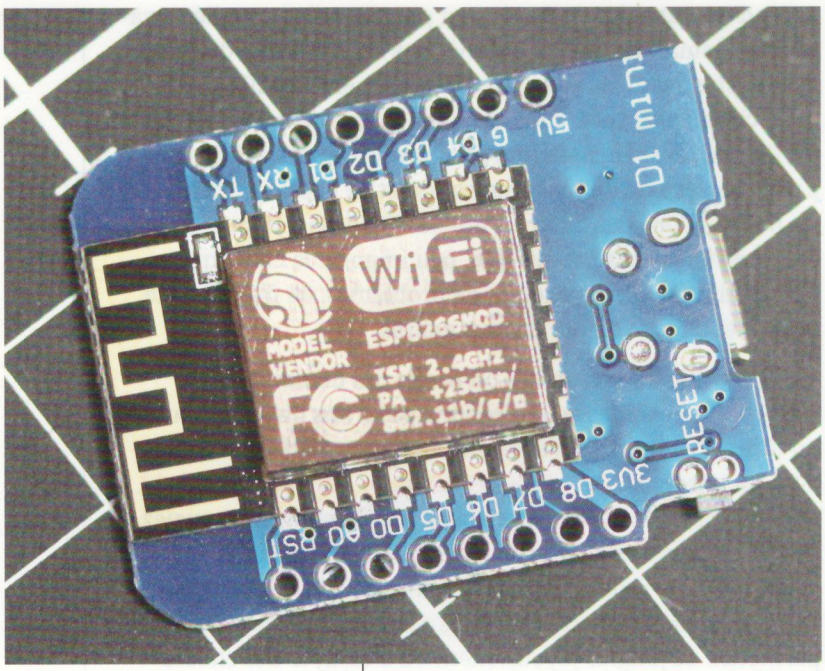
Mais voilà, il y a toujours un projet qui demande un CMake plus récent, une libc qui n'a pas 5 ans de retard ou un utilitaire que vous n'avez vraiment pas envie d'installer avec un horrible `./configure && make && make install` qui ne ferait qu'empirer les choses. Idéalement, vous auriez besoin d'un système dans le système, facile à installer et à utiliser, qui ne repose pas sur une machine virtuelle pour des raisons de performance, mais assez souple pour vous permettre de jongler joyeusement ou d'essayer des choses sans rien casser. Ça tombe bien, parce que Docker permet exactement de faire cela !

Note : L'utilisation de conteneurs ne doit pas être confondue avec l'émulation ou la virtualisation. L'émulation consiste à simuler un matériel (comme une Pi sur un PC avec QEMU par exemple) pour y faire fonctionner tout un système. La virtualisation repose sur un partage de ressources matérielles entre plusieurs systèmes, mais le ou les systèmes virtualisés utilisent leur noyau et leurs pilotes. Avec un conteneur Docker, l'environnement exécuté n'a pas de noyau, mais uniquement un système de fichiers propre, séparé de celui de l'hôte faisant fonctionner Docker.

1. CONSTRUIRE SON OU SES IMAGES

La logique derrière Docker est importante et la comprendre le plus rapidement possible vous permettra de ne pas perdre votre latin. Tout est une question de terminologie :

- un *Dockerfile* est une description permettant de créer une image du système qui sera utilisée dans un conteneur ;
- une image est une représentation du système tel qu'il prendra place dans un conteneur. Voyez cela comme un modèle, un *template* ou une classe qui sera instanciée ;



Les devkits ESP8266 se programment en démarrant le microcontrôleur sur son bootloader qui utilise alors une liaison série pour dialoguer avec l'outil de programmation (esptool.py). L'accès au port série depuis l'environnement en conteneur est donc indispensable pour pouvoir développer correctement.

- et un conteneur est une instance d'une image qui peut être créée, démarrée et stoppée.

En d'autres termes, vous utilisez un *Dockerfile* pour créer une image, mais chassez de votre esprit que cette image est le système, comme le serait une image d'un système de fichiers. C'est un simple point de départ et vous n'utiliserez pas directement l'image en vous servant du système qui s'exécutera dans le conteneur. Pourquoi faire aussi compliqué ? C'est simple, imaginez que vous construisiez le système qui vous convient, avec tous les éléments qui vous intéressent et même les configurations dont vous avez l'habitude. Mais vous avez la ferme intention d'utiliser cette base pour plus d'une expérimentation. Vous allez donc créer plusieurs conteneurs, utilisant tous cette même image comme point de départ, mais vivant leurs vies bien à eux. Un peu comme utiliser une bonne recette de *cupcake*, bien peaufinée et au point, et décliner le *toping* à loisir ensuite.

Exactement comme pour des *cupcakes*, vous allez réaliser votre appareil (le terme qui fait « pro » en pâtisserie pour désigner un mélange homogène de plusieurs ingrédients) à partir d'une recette. Avec Docker, cette recette, c'est le *Dockerfile* et comme en cuisine, ceci s'accompagne d'un contexte, en l'occurrence ce qui est mis à disposition sur le plan de travail, à savoir ce qu'on trouve dans le répertoire courant (c'est important, nous allons le voir dans un instant avec la copie de fichiers).

Ce *Dockerfile* (par défaut `Dockerfile`) se présente comme un simple fichier texte, mi-description, mi-script. Il doit commencer par l'instruction `FROM` permettant de préciser une image de base. Celle-ci peut être n'importe quelle image valide, mais sera généralement tirée du dépôt public (**https://hub.docker.com/search?type=image**). Pour cet exemple, nous utiliserons une image de base Debian Bullseye (Debian 11.1) :

```
FROM debian:bullseye
```

N'hésitez pas à visiter la page web du dépôt, on y trouve de tout, et rien ne vous empêchera d'utiliser une image OpenSuse dans un conteneur sur une Debian ou encore Ubuntu sur une Fedora. Il existe des images très basiques comme celle-ci avec un système de fichiers minimal, mais également d'autres orientées langages de programmation (Python, Go, Ruby, etc.), bases de données (MySQL, PostgreSQL, InfluxDB, etc.) ou encore frameworks applicatifs (Drupal, Gradle, Couchbase, .NET, etc.).

L'image Debian Bullseye nous fournit un environnement simple que nous pouvons personnaliser en exécutant une série de commandes, à commencer par une installation de paquets :

```
RUN apt-get update \
&& DEBIAN_FRONTEND=noninteractive apt-get install -y \
    build-essential file g++ git libelf-dev \
    libncurses5-dev libncursesw5-dev libssl-dev \
    python python2.7-dev python3 python3-distutils \
    python3-setuptools python3-dev rsync time \
    unzip wget zlib1g-dev sudo mc bc vim screen \
    exuberant-ctags bash-completion
```

`RUN` est l'instruction permettant d'exécuter les commandes dans l'environnement et donc dans le « système » qui constituera l'image. Ceci n'est pas très différent de ce que vous feriez vous-même juste après l'installation d'une distribution, mais sera forcément spécifique à l'image de base. Ici, `DEBIAN_FRONTEND` nous permet de préciser à `apt-get` qu'il ne s'agit pas d'une session interactive et que tout devra se dérouler automatiquement. Ainsi après un petit `update`, nous spécifions simplement les paquets selon nos envies et préférences.

D'autres commandes peuvent (doivent) également être spécifiées. Pour créer un nouvel utilisateur par exemple :

```
RUN useradd denis -m -k /dev/null -d /home/denis \
    && echo "denis:mot2passe" | chpasswd \
    && adduser denis sudo \
    && adduser denis dialout \
    && adduser denis plugdev
```

Ceci créera `denis`, définira son mot de passe et le placera dans les groupes `sudo` (pour passer `root`) et `dialout` (pour l'accès aux `/dev/TTYUSB*` et `/dev/ttyACM*`) ainsi que `plugdev` pour accéder aux périphériques *plug'n'play* (mon Dieu, que ce terme fait vieux). Nous pouvons également ajuster la localisation :

```
RUN apt-get update && apt-get install -y locales \
    && rm -rf /var/lib/apt/lists/* \
    && localedef -i fr_FR -c -f UTF-8 -A \
    /usr/share/locale/locale.alias fr_FR.UTF-8 \
    && apt-get update
ENV LANG fr_FR.utf8
```

Nous aurons ainsi directement un système localisé avec des dates et un jeu de caractères adéquats (comprendre « français »). Notez l'utilisation de `ENV` permettant de définir une variable d'environnement (`LANG`) valable pour toutes les instructions suivantes dans le *Dockerfile*.

Nous basculons ensuite sur l'utilisateur défini précédemment pour toutes les autres instructions du fichier :

```
USER denis
```

Puis nous personnalisons son compte en copiant quelques fichiers avec `COPY` :

```
COPY home/denis/.vimrc /home/denis
COPY home/denis/.vim/ /home/denis/.vim/
COPY home/denis/.inputrc /home/denis
COPY home/denis/.bashrc /home/denis
COPY home/denis/.screenrc /home/denis
```

Attention ! L'instruction `COPY` ne peut agir **QUE** sur le répertoire courant, qui est le fameux contexte de construction. Ceci signifie que `home/denis/.vimrc` n'est pas le `.vimrc` présent dans le répertoire personnel de l'utilisateur qui exécutera la construction, mais dans un sous-répertoire `home/denis/` du répertoire courant. En d'autres termes, il convient de copier, dans le répertoire où vous créerez le *Dockerfile*, tous les éléments que vous désirerez placer dans l'image. Ceci est non seulement plus propre, mais permet également de ne pas se mélanger les pinceaux entre ses fichiers `*rc` de l'hôte qui exécute Docker et ceux à destination de l'image.

Enfin, une autre instruction pourra être utile :

```
WORKDIR /home/denis
```

Celle-ci a pour effet de définir, pour toutes les instructions qui suivent, le répertoire courant. J'ai pour habitude de laisser cette ligne dans mes *Dockerfiles*, même s'il n'y a rien ensuite. L'intérêt est de pouvoir ajouter des `RUN` pour ajouter des commandes qui devront être exécutées dans ce répertoire comme, typiquement, un `git clone` ou un téléchargement de fichiers.

Vous pouvez stocker tout cela dans un fichier `Dockerfile` dans le répertoire courant (idéalement prévu à cet effet), mais personnellement, je préfère nommer mes *Dockerfiles* en fonction de ce à quoi ils servent, ici simplement `mydebian.dock` puisque c'est une configuration très générique.

On pourra ensuite créer l'image avec :

```
$ docker build -t articledocker -f mydebian.dock .
Sending build context to Docker daemon  15.15MB
Step 1/12 : FROM debian:bullseye
bullseye: Pulling from library/debian
[...]
```

```
Step 11/12 : COPY home/denis/.screenrc /home/denis
---> 9b0482742f88
Step 12/12 : WORKDIR /home/denis
---> Running in c18182bc487d
Removing intermediate container c18182bc487d
---> 8d4751d0c8ce
Successfully built 8d4751d0c8ce
Successfully tagged articledocker:latest
```

Note : si `docker` vous refuse l'exécution sous prétexte que vous n'avez pas les permissions adéquates, placez simplement l'utilisateur dans le groupe `docker`, déconnectez-vous et reconnectez-vous. N'utilisez pas `sudo`, ce n'est ni nécessaire ni souhaitable.

Cette opération dure un certain temps, car l'image de base doit être téléchargée, les paquets installés et la configuration effectuée. Mais construire une autre image identique (ce qui présente peu d'intérêt), avec un autre nom (`-t`) ira beaucoup plus vite. `docker build -t autre_image -f mydebian.dock .` sera quasiment instantané, puisque le travail a déjà été fait. La commande `docker build` prend en argument un nom d'image, un nom de *Dockerfile* (`-f`) et un répertoire de construction, ici le répertoire courant (`.`).

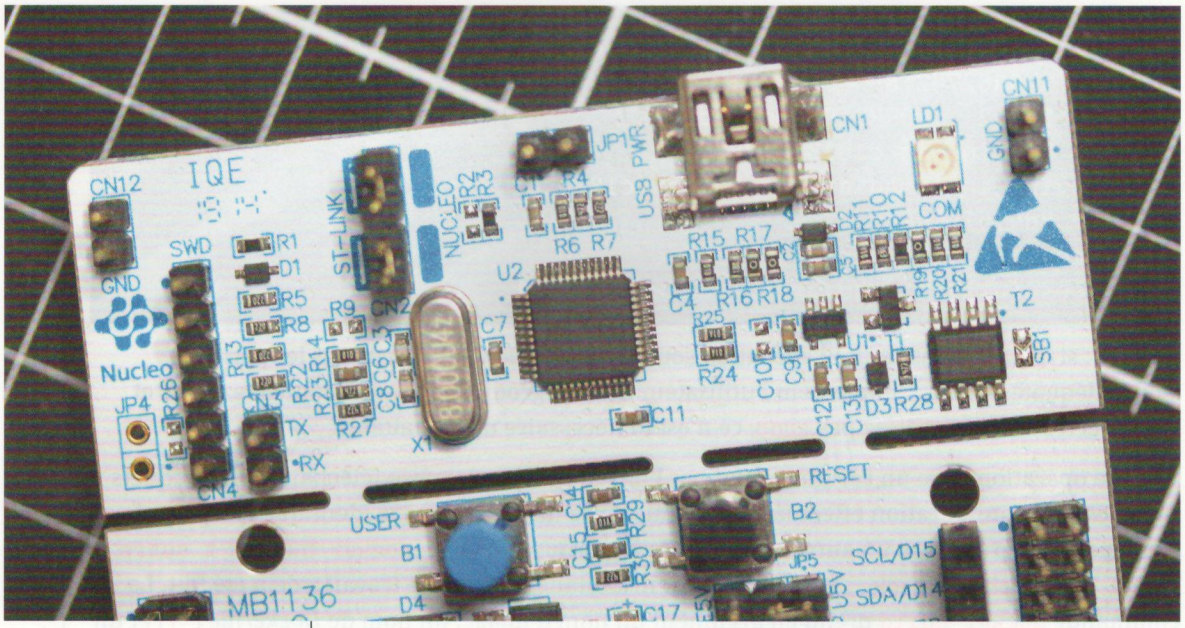
Vous pouvez lister toutes les images construites sur votre système avec :

```
$ docker image ls
REPOSITORY     TAG       IMAGE ID       CREATED         SIZE
articledocker  latest    8d4751d0c8ce   4 minutes ago   791MB
autre_image    latest    8d4751d0c8ce   4 minutes ago   791MB
debian         bullseye  05d2318291e3   7 days ago      124MB
```

Remarquez que nous n'avons créé ici que deux images, `articledocker` et `autre_image`, mais que `debian` figure également dans la liste. De plus, nos deux images ont la même ID. Ceci vient du fait que Docker travaille avec des couches ou *layers*. Il n'est pas nécessaire de dupliquer l'ensemble des éléments des images. Les tailles données ne correspondent pas non plus à l'espace disque effectivement occupé, et c'est précisément là l'intérêt du système. `debian` a été téléchargée comme image de base et pourra être utilisée plus rapidement avec une déclinaison de notre *Dockerfile*. Ou supprimée avec `docker rmi` ou `docker image rm` suivie du nom de l'image, si vous le souhaitez.

2. CRÉATION, EXÉCUTION, CONNEXION...

À ce stade, nous avons deux images parfaitement identiques, basées sur le même *Dockerfile*, mais nous n'avons pas de conteneur « vivant ». Il est donc temps d'utiliser ces images pour profiter des bienfaits de Docker. Il est important avant cela de comprendre qu'un conteneur doit être vu comme l'instanciation d'une image et que celui-ci peut être créé/exécuté (`run`), arrêté



Les interfaces ST-LINK équipant la totalité des cartes d'évaluation STMicroelectronics, comme cette Nucleo-F411RE, nécessitent un accès direct au périphérique USB. Cependant, avec Docker, il ne s'agit pas simplement de jouer avec les règles udev...

(`stop`) et (re)démarré (`start`). La distinction est importante, car la première exécution va créer le conteneur, mais quand vous le quitterez, celui-ci sera arrêté et non détruit. Vous n'allez donc pas l'exécuter à nouveau, mais le redémarrer pour l'utiliser une nouvelle fois.

Mais commençons par le début, en créant le premier conteneur :

```
$ docker run -ti --name deb1 articledocker
denis@b709d79a9bfe:~$
```

Nous utilisons la commande `run` qui s'appelle ainsi, car elle prévue pour « *exécuter une commande dans un conteneur* ». Mais comme nous demandons un fonctionnement interactif (`-i`) et l'utilisation d'un pseudo-TTY (`-t`) sans préciser de commande, nous nous retrouvons avec un shell. `--name` nous permet de donner un nom au conteneur et `articledocker` est tout simplement le nom de l'image à utiliser.

Nous nous retrouvons donc avec une ligne de commandes d'un Bash exécuté dans le conteneur, sous Debian Bullseye. Mais ce n'est pas un « vrai » système, `/boot` est vide, pas de `/var/log/kern.log` et presque rien dans `/dev`.

Plions-nous d'un `touch TOTO` pour créer rapidement un fichier dans le répertoire courant puis quittons le shell avec `exit` ou Ctrl+D, nous revenons à notre ligne de commandes précédente de la machine locale. Là, nous pouvons exécuter à nouveau `docker run` en spécifiant un nouveau nom de conteneur :

```
$ docker run -ti --name deb2 articledocker
denis@4855de3275c3:~$ ls TOTO
ls: impossible d'accéder à 'TOTO':
Aucun fichier ou dossier de ce type
```

Remarquez le nom d'hôte différent dans le *prompt*, et pour cause, ce n'est pas le même « système » et notre fichier `TOTO` n'est plus là. Nous venons de créer un second conteneur, à partir de la même image et, après avoir quitté avec `exit`, nous pouvons lister les conteneurs avec :

```
$ docker container ls -a
CONTAINER ID  IMAGE          COMMAND  CREATED         STATUS                          NAMES
4855de3275c3  articledocker  "bash"   32 minutes ago  Exited (2) About a minute ago  deb2
b709d79a9bfe  articledocker  "bash"   40 minutes ago  Exited (0) 34 minutes ago      deb1
```

Nous avons deux conteneurs, `deb1` et `deb2`, utilisant l'image `articledocker` et dans un état « quitté ». Notez bien qu'ils utilisent `articledocker` comme `base`, mais que cette image n'est **pas** un support de stockage pour le « système ». J'insiste, car c'est une source de confusion, ceci n'a rien à voir avec une image de système de fichiers comme le `.img` d'une Raspberry Pi qu'on pourrait monter « en loopback » (`losetup`) pour y faire des modifications.

Pour reaccéder à un conteneur, nous pourrions être tentés de faire :

```
$ docker run -ti --name deb1 articledocker
docker: Error response from daemon: Conflict.
The container name "/deb1" is already in use by container
"b709d79a9bfec4cf973f37e532bdd31bb46aef21c4219b265ccab97459f62a31".
You have to remove (or rename) that container to be able to reuse that name.
```

Mais `run` ne peut être réutilisé, c'est l'autre point de confusion avec Docker. Le conteneur existe et est simplement arrêté. Pour le redémarrer, il faut utiliser :

```
$ docker start -ai deb1
denis@b709d79a9bfe:~ ls -l TOTO
-rw-r--r-- 1 denis denis 0  9 déc.  18:10 TOTO
```

Nous retrouvons, bien entendu, notre `TOTO` et le conteneur est à nouveau en marche. Les options `-a` et `-i` (combinées en `-ai`) correspondent respectivement au fait de s'attacher automatiquement et de travailler de façon interactive. Autrement dit, les sorties STDOUT/STDERR du conteneur sont envoyées sur notre terminal et notre STDIN devient également celui du conteneur.

Pour laisser fonctionner le conteneur tout en retournant à notre shell d'origine, vous pouvez utiliser la séquence Ctrl+P puis Ctrl+Q. Un `docker container ls` (l'option `-a` pour *all* permet d'afficher aussi les conteneurs à l'arrêt) vous montrera votre conteneur actif.

Pour y retourner, ou plutôt vous y rattacher, utilisez simplement `docker attach deb1`. Vous pouvez également stopper un conteneur détaché avec `docker stop deb1`. Remarquez que l'opération peut paraître lente, mais en réalité, c'est un délai de 10 secondes qui est automatiquement appliqué. Vous pouvez préciser un autre délai avec l'option `-t`, dont `0`.

Enfin, il peut vous venir l'envie de faire un brin de ménage. Vous pouvez donc supprimer un conteneur avec :

```
$ docker container rm deb1
deb1
```

Et une image avec :

```
$ docker image rm artdock:latest
Untagged: artdock:latest
```

Mais si vous essayez de supprimer une image utilisée par un conteneur, Docker vous rappellera à l'ordre sans ménagement :

```
$ docker image rm articledocker
Error response from daemon: conflict: unable to
remove repository reference "articledocker" (must force)
- container 4855de3275c3 is using its referenced
image ee3e2adaf1b9
```

Voici qui conclut la partie sur l'utilisation de base de Docker, pour rapidement créer des conteneurs pour des manipulations sans risques avec un environnement sur mesure, par projet. Ici, on peut se permettre des `sudo make install` sans remords et autres légèretés honteuses du même genre. Notez au passage qu'il est même possible de créer des images à partir d'un conteneur (`docker commit`) et donc de se servir de cette mécanique pour créer des « points de sauvegarde ».

3. ACCÈS AU MATÉRIEL

Pouvoir disposer d'un environnement de développement personnalisé à souhait et ne plus être dépendant de la distribution ou de la version de la distribution qu'on utilise est une bonne chose, tout comme la possibilité de temporairement et impunément saccager tout ce qui nous chante. Mais le développement sur microcontrôleur ou à destination d'un système embarqué n'est pas qu'une affaire de SDK, de chaînes de compilation et de BSP. Il faut pouvoir, à un moment, accéder au matériel pour charger un *firmware*, déboguer ou même tout simplement communiquer avec la plateforme via une application ou un outil spécifique.

Les conteneurs existent principalement pour répondre aux problématiques d'isolation d'applications. Il n'est donc guère étonnant qu'accéder au matériel ne soit pas possible par défaut, et passer outre n'est pas aussi souple d'utilisation que le reste des fonctionnalités. De base, si vous

essayer d'accéder à un périphérique, comme une carte STM32 Nucleo avec la suite *open source* des *ST-Link tools*, on vous remettra promptement à votre place ainsi :

```
denis@ed1df6bb77df:~$ st-info --descr
libusb: error [get_usbfs_fd] File doesn't exist,
  wait 10 ms and try again
libusb: error [get_usbfs_fd] libusb couldn't open
  USB device /dev/bus/usb/002/009, errno=2
```

Il existe plusieurs solutions vous permettant de contourner le problème et donc d'utiliser un périphérique USB, comme une carte de développement accessible avec des outils spécifiques (`st-util`, OpenOCD, PicoTool, etc.) par exemple, ou un port série comme celui proposé par une carte Arduino ou ESP8266/ESP32. La plus simple et brutale d'entre elles consiste à tout simplement oublier une partie de la sécurité **[1]** et de l'isolation fournies par les conteneurs, en utilisant l'option `--privileged` de `docker run`. Celle-ci permet de donner à un conteneur la permission d'accéder à n'importe quel périphérique de la machine local. Et le résultat est, bien entendu, celui espéré :

```
denis@a7c903647f46:~$ st-info --descr
stm32f411re
```

Depuis notre conteneur, l'outil `st-info` des *ST-Link tools* voit effectivement la carte Nucleo et son STM32F411 sans le moindre problème. Bien que nous ne soyons pas dans une logique d'isolation de service ou d'application, mais dans une simple démarche pratique, ceci pourrait être toléré. Mais ce n'en est pas moins une mauvaise idée et une très mauvaise habitude. Mieux vaut ne pas se comporter comme un homme de Cro-Magnon et ne permettre que l'accès au périphérique que nous utilisons.

Ceci est possible en utilisant l'option `--device=` suivie d'un chemin vers le périphérique en question. Sur l'hôte, et toujours avec notre exemple de STM32 Nucleo, nous commençons par identifier le périphérique avec `lsusb` :

```
$ lsusb
[...]
Bus 002 Device 009: ID 0483:374b
   STMicroelectronics ST-LINK/V2.1
[...]
```

Notre carte est sur le bus USB `002` et est le périphérique `009`. Nous utilisons ces éléments avec cette nouvelle option :

```
$ docker run -ti --device=/dev/bus/usb/002/009 \
  --name deb_DEV articledocker

denis@04e70e2cebbe:~$ st-info --descr
stm32f411re
```

Magnifique, nous avons accès à la Nucleo sans compromettre notre sécurité. Ceci fonctionne même avec OpenOCD qui ne trouve rien à redire :

```
denis@04e70e2cebbe:~$ openocd -f board/st_nucleo_f4.cfg
Open On-Chip Debugger 0.11.0-rc2
Licensed under GNU GPL v2
[...]
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
Info : clock speed 2000 kHz
Info : STLINK V2J22M5 (API v2) VID:PID 0483:374B
Info : Target voltage: 3.261782
Info : stm32f4x.cpu: hardware has 6 breakpoints, 4 watchpoints
Info : starting gdb server for stm32f4x.cpu on 3333
Info : Listening on port 3333 for gdb connections
```

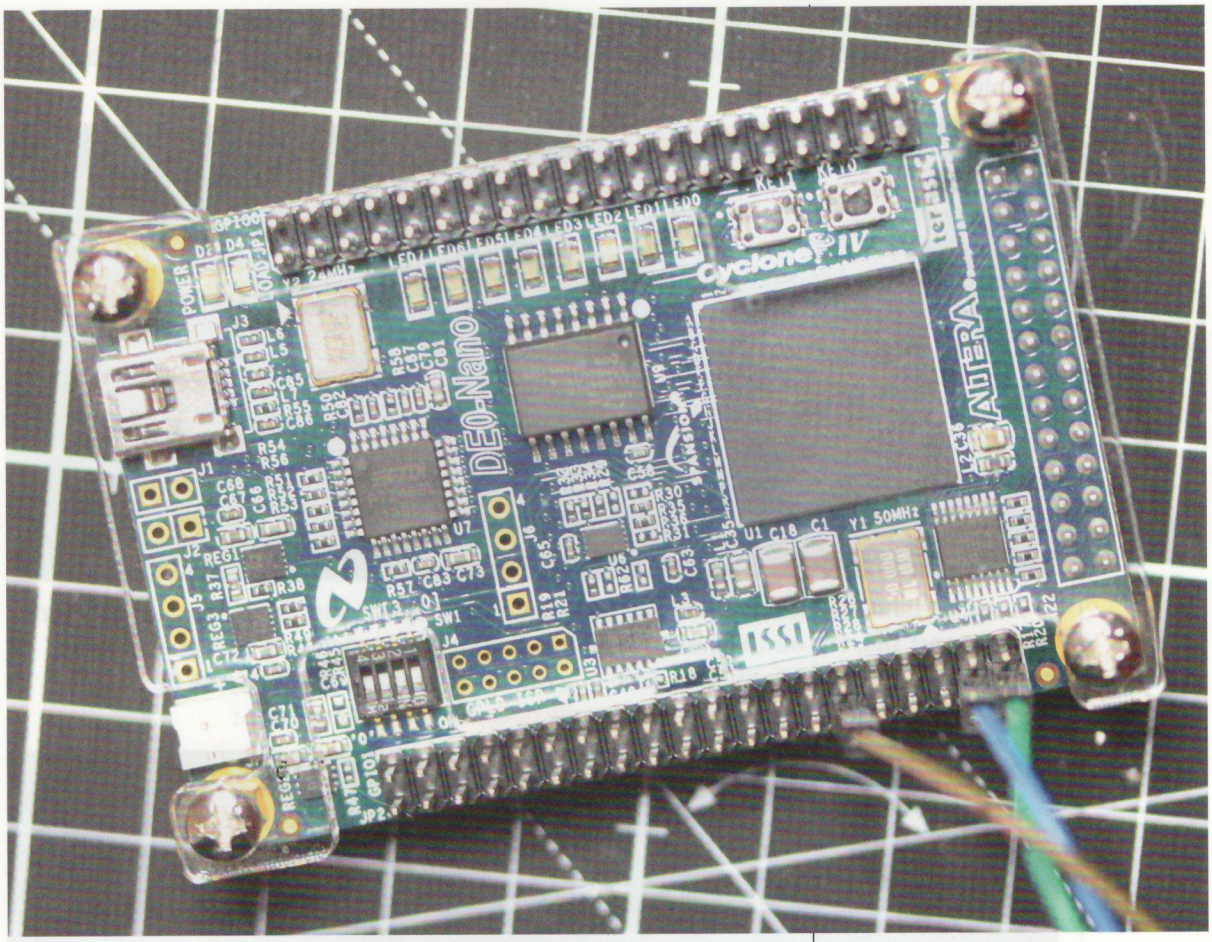
Ceci s'appliquera également aux cartes communiquant via un port série comme les Arduino, apparaissant sous GNU/Linux via des entrées comme `/dev/ttyUSB0` et `dev/ttyACM0`. Il suffira d'utiliser `--device=/dev/ttyUSB0` et le tour sera joué. Seul petit problème, si nous débranchons la carte et la rebranchons, elle apparaîtra probablement sous un autre numéro (et possiblement sur un autre bus selon où elle sera reconnectée). Pire encore, si c'est une Raspberry Pi Pico, elle apparaît et disparaît en fonction du démarrage en mode BOOTSEL ou non. Cette technique fonctionne en jonglant avec les suppressions et exécutions de conteneurs, mais elle est relativement rigide et ne se prête que très mal à un cycle habituel de développement sur microcontrôleur ou SBC. Nous pouvons faire mieux !

Une autre option envisageable est `-v` permettant de lier (*bind*) un volume ou, en d'autres termes, faire apparaître une partie du système de fichiers « réel » (local) dans le conteneur. On pourra donc utiliser quelque chose comme `-v /dev/bus/usb:/dev/bus/usb`, ou plus généralement `-v /dev:/dev`, pour se retrouver avec un `/dev` dans le conteneur, identique à celui de la machine elle-même. Sauf que :

```
libusb: error [get_usbfs_fd] libusb couldn't
open USB device /dev/bus/usb/002/009, errno=1
```

Avoir accès aux pseudofichiers de `/dev` est une chose, mais avoir la permission d'y lire et écrire en est une autre. Le fait que l'utilisateur, dans et hors du conteneur, fasse partie de `dialout` ou `plugdev` importe peu, la restriction vient de *cgroups* (pour *control groups*), une fonctionnalité du noyau Linux, utilisée par Docker afin de limiter, gérer et isoler l'utilisation des ressources. Il nous faut donc spécifier une option `--device-cgroup-rule`, en plus de `-v`, pour changer les permissions accordées. Pour les périphériques USB, nous pouvons utiliser :

```
$ docker run -ti \
  --device-cgroup-rule='c 189:* rmw' \
  -v /dev:/dev --name deb_DEV1 articledocker
```



Cette adorable carte de développement pour le FPGA Altera/Intel Cyclone IV (obsolète, mais toujours intéressante) intègre un programmeur USB-Blaster qui nécessite, lui aussi, un accès direct au périphérique USB. Avec ce type de plateforme et l'installation d'applicatifs tiers propriétaires, on comprend aisément l'intérêt de mettre en conteneur ce type d'environnement de développement.

La règle *cgroups* est « c 189:* rmw » où :

- `c` désigne un périphérique de type « caractère » ;
- `189` est le numéro majeur du périphérique Linux ;
- `:*` désigne tous les numéros mineurs ;
- et `rmw` indique des permissions de lecture (`r`), création de nœuds (`m` pour `mknod`) et d'écriture (`w`).

La notion de numéro de périphérique majeur et mineur est inhérente au noyau Linux et à la façon dont les périphériques sont associés aux pilotes. Ces numéros sont fixes et définis dans les sources du noyau. Mais vous pouvez également les trouver en listant les informations complètes des entrées (ou nœuds) dans `/dev` :

```
$ ls -l /dev/ttyUSB0
crw-rw---- 1 root dialout 188, 0 déc. 10 11:07 /dev/ttyUSB0
```

`c` indique un périphérique en mode caractère, `188` le numéro majeur et `0` le mineur. Nous pouvons également consulter le contenu de `/proc/devices` pour obtenir une liste complète :

```
$ cat /proc/devices
Character devices:
  1 mem
  4 /dev/vc/0
[...]
136 pts
166 ttyACM
180 usb
188 ttyUSB
189 usb_device
226 drm
[...]
```

Ainsi, en liant `/dev` et en autorisant les accès aux périphériques en mode caractère dont le majeur est 166, 188 ou 189, nous obtenons un conteneur capable de dialoguer avec les périphériques USB (via la libUSB), ainsi que ceux apparaissant sous la forme de `/dev/ttyUSB*` et `/dev/ttyACM*`. Notre commande `docker run` devient donc :

```
$ docker run -ti \
  --device-cgroup-rule='c 189:* rmw' \
  --device-cgroup-rule='c 166:* rmw' \
  --device-cgroup-rule='c 188:* rmw' \
  -v /dev:/dev --name deb_DEV2 articledocker
```

Ceci règle notre problème de permissions et nous permet donc de développer, programmer et déboguer pour n'importe quelle cible depuis nos conteneurs.

Pour conclure sur cette partie liée au matériel et au développement, sans doute souhaiterez-vous également transférer des fichiers (sources) vers et depuis vos conteneurs. Ceci n'est largement pas aussi compliqué que l'accès au matériel USB et sera l'affaire d'une simple commande.

Pour copier depuis le système de fichiers local vers le conteneur :

```
$ docker cp ~/SRC/C/base deb2:/home/denis
```

Où `deb2` est le nom du conteneur. Et dans le sens opposé :

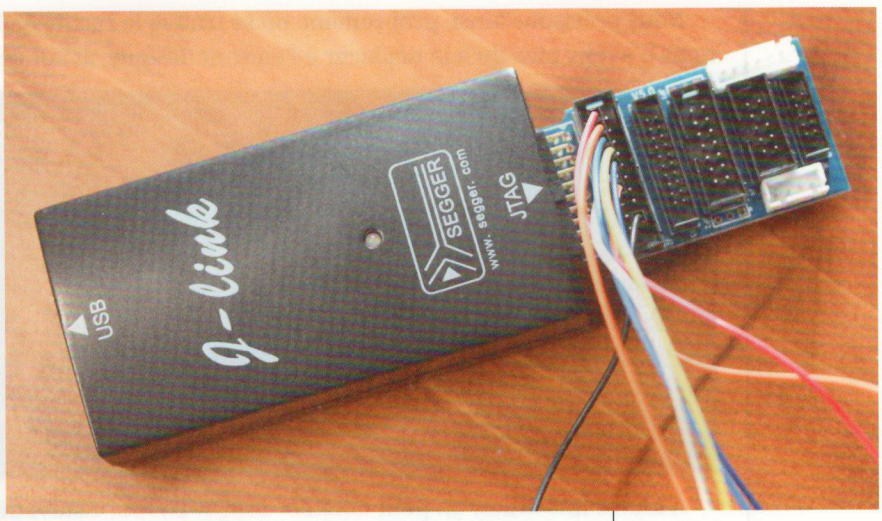
```
$ docker cp deb2:/home/denis ~/SRC/C/truc
```

Notez qu'il n'y a pas d'options permettant une copie récursive de répertoire puisque c'est automatiquement le cas.

4. M'SIEUR ? J'AI PLUS DE PLACE !

Jouer avec Docker et se créer tout un tas d'environnements et de systèmes à torturer dans des conteneurs c'est bien, c'est pratique et c'est amusant. Ce qui l'est moins, en revanche, c'est l'espace occupé par toutes ces données. Arrivera donc le moment fatidique où l'espace disque viendra à faire défaut. Vous serez alors tenté de déplacer tous ces conteneurs dans un autre emplacement que `/var/lib/docker`, sur un SSD supplémentaire monté, par exemple dans `/mnt/SSD2T`.

Déplacer ses « petites affaires » n'est pas très difficile, si l'on s'en réfère à la documentation officielle et non à des posts de blogs faits par des utilisateurs qui sont, passez-moi l'expression, rien de moins que de gros gorets. L'utilisateur Debian bien rôdé aura tendance à aller voir dans `/etc/default/docker`, mais là, une triste surprise l'y attend : « *THIS FILE DOES NOT APPLY TO SYSTEMD* ». C'est amusant comme certaines choses qui vous tapent sur les nerfs ont tendance à revenir vous taquiner à la moindre occasion... Merci Lennart de nous éloigner chaque jour un peu plus de la philosophie UNIX ! Vraiment merci (sarcasmes) !



Notre approche consistant à donner accès en écriture à tous les périphériques ayant un majeur 189 (usb_device) nous permet également d'utiliser des outils de mise au point comme cette sonde J-Link BASE Classic.

Mais ce dont je parle, ce sont plutôt les documentations vous invitant joyeusement à éditer `/lib/systemd/system/docker.service`, histoire d'y glisser une option `-g` obsolète suivi d'un chemin. Non ! Non, re-non, et encore **NON** ! On ne touche pas à un fichier de configuration du système, géré par le gestionnaire de paquets, et qui se fera écraser à la moindre mise à jour. Et on n'entre pas non plus de chemin en dur dans un tel script. Systemd est déjà suffisamment irritant sans qu'on se sente obligé d'empirer le supplice. Je ne parle même pas de l'aspect assez présomptueux consistant à se dire que les développeurs de Docker n'ont pas pensé au problème...

Il y a une façon de faire cela **proprement** et cela commence par stopper le/les service(s) Docker :

```
$ sudo systemctl stop docker
$ sudo systemctl stop docker.socket
$ sudo systemctl stop containerd
```

On peut ensuite créer le fichier `/etc/docker/daemon.json`, contenant :

```
{
    "data-root": "/mnt/SSD2T/DOCKER"
}
```

Nous précisons ainsi, proprement, où se trouve la racine de l'arborescence de données et nous n'avons plus qu'à la déplacer à l'endroit désigné, avant de redémarrer le service :

```
$ sudo mv /var/lib/docker /mnt/SSD2T/DOCKER

$ sudo systemctl start docker

$ docker image ls
REPOSITORY    TAG       IMAGE ID     CREATED         SIZE
mydebian      latest    fd1681b3f43a 45 minutes ago 791MB
openwrt_test  latest    b0d9230ed355 7 hours ago    823MB
debian        bullseye  05d2318291e3 7 days ago     124MB
ubuntu        20.04     ba6acccedd29 7 weeks ago    72.8MB
```

À toutes fins utiles, un petit `docker image inspect` suivi du nom d'une image vous permettra de vous assurer que l'emplacement est bien référencé (`WorkDir`), ou en utilisant simplement la commande `docker info -f '{{ .DockerRootDir}}'`.

5. POUR CONCLURE

Pour être tout à fait honnête avec vous, nous n'avons fait qu'effleurer le sujet ici. Docker permet de faire énormément de choses, comme placer des limitations (CPU, mémoire, etc.) sur les images et les conteneurs, sans compter toutes les options et solutions d'orchestration disponibles pour automatiser vos processus. Cette solution s'est démocratisée auprès des administrateurs système depuis très longtemps, mais il est encore rare de voir cette technologie utilisée dans le milieu du développement embarqué. Pourtant, ceci constituerait une solution très intéressante et pratique pour la diffusion des SDK comme celui du Raspberry Pi Pico, et certains projets l'ont d'ores et déjà compris (comme le *firmware* Proxmark de RFID Research Group **[2]**).

J'ai longtemps ignoré Docker, n'y voyant initialement que peu d'intérêt dans mon domaine de prédilection, mais prendre le temps de se pencher sur le sujet s'est avéré être un investissement très rentable. Je dispose de conteneurs pour mes développements ARM *baremetal*, Raspberry Pi Pico, ESP-IDF, STM32H7 sur Game & Watch, etc., et tout ce petit monde est rangé, ordonné et incroyablement facile à maintenir. Aujourd'hui, je pense que j'aurai bien du mal à m'en passer... **DB**

RÉFÉRENCES

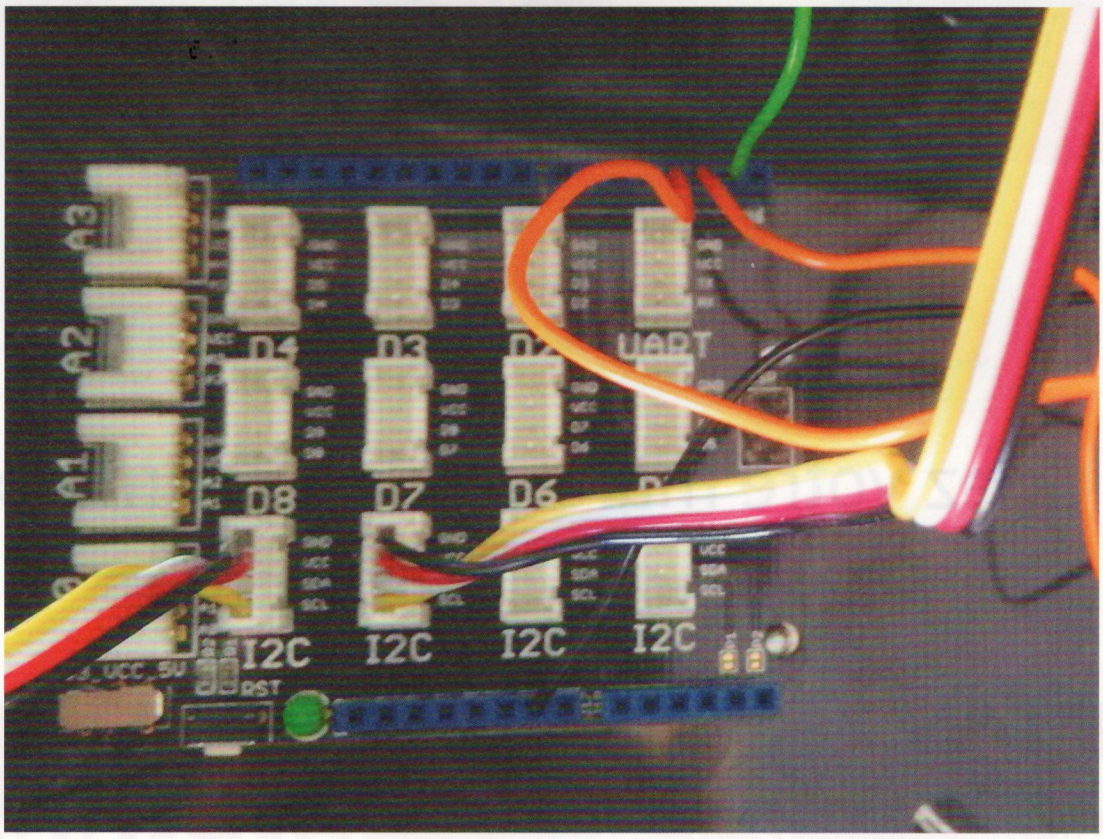
[1] https://blog.trailofbits.com/2019/07/19/understanding-docker-container-escapes/

[2] https://hub.docker.com/r/iceman1001/proxmark3/

CONSTRUISEZ VOTRE MESUREUR DU TAUX DE CO2

Patrice KADIONIK
Maître de conférences HDR à l'ENSEIRB-MATMECA

Cet article présente la réalisation d'un équipement électronique portable de mesure du taux de CO2 dans une pièce. L'idée première est qu'il soit fiable, mais aussi le plus simple possible à construire, sans réalisation d'un circuit imprimé et avec un minimum de soudures.



– Construisez votre mesureur du taux de CO2 –

Depuis 2 ans, nous vivons dans un contexte pandémique qui a bouleversé notre quotidien. Dans les Établissements Recevant du Public (ERP), il est important de maîtriser la transmission du SARS-CoV-2. Le Haut Conseil de la Santé Publique (HCSP) a rendu des avis **[1] [2]** dans ce sens, notamment en indiquant que le taux de dioxyde de carbone (CO2) peut être considéré comme un traceur du renouvellement de l'air d'une pièce. Ainsi, le HCSP recommande pour les commerces et autres ERP de « mettre en œuvre des actions d'aération et d'assurer le bon fonctionnement de la ventilation lorsque la concentration dépasse 800 ppm (parties par million) en CO2 ». Ventilation et aération sont mises en avant.

On entend dans les médias qu'il faut aérer au moins 10 minutes toutes les heures. Mais disposer d'un mesureur du taux de CO2 s'avère plus précis.

C'est dans ce contexte que se place cet article. Il s'agit de réaliser un mesureur du taux de CO2 avec le cahier des charges suivant :

- il doit être le plus simple possible pour qu'il soit réalisable par n'importe quel bricoleur ;
- la réalisation doit se faire avec un minimum de soudures sans fabrication d'un circuit imprimé ;
- l'équipement doit être portable et nomade : on le prend avec soi quand on va dans une pièce qui accueille du public, par exemple un enseignant qui donne un cours à des étudiants ;
- l'équipement doit être le plus autonome possible : alimentation par un port USB ou par une batterie nomade (*tank*) de smartphone ;
- le prix de revient doit être le plus faible possible ;
- les outils de développement sont libres et gratuits ;
- le capteur de CO2 doit être fiable et reconnu comme tel.

Notre mesureur du taux de CO2, outre le capteur de CO2, affichera sur un afficheur LCD la valeur courante du taux de CO2 en ppm (et la température), mais comportera aussi 3 LED :

- LED verte : allumée, le taux de CO2 est inférieur à 700 ppm ;
- LED jaune : allumée, le taux de CO2 est compris entre 700 et 800 ppm. Attention, le taux de CO2 approche la valeur limite préconisée ;
- LED rouge : clignotante, le taux de CO2 est supérieur à 800 ppm. Il faut impérativement aérer/ventiler la pièce comme préconisé par le HCSP.

1. RÉALISATION MATÉRIELLE

La réalisation doit être à la portée de tous. Cela veut dire qu'il n'y a pas de réalisation de circuit imprimé et un minimum de soudures qui se limitera à la soudure des LED/résistances/fils. Tout le reste est de l'assemblage de blocs.

Cela est possible en utilisant des périphériques Grove. La connectique *open source* Grove développée par la société Seeed **[3]** permet un assemblage de périphériques ou modules sans aucune soudure ni *breadboard* ou circuit imprimé supplémentaire. Tout est basé sur un connecteur mâle ou femelle 4 broches et une nappe de 4 fils avec des connecteurs Grove.

Pour s'interfacer à un matériel comme un module Arduino ou bien une carte Raspberry Pi, on utilise un « *shield* » Grove. Afin de limiter le prix de revient et simplifier la partie programmation, nous avons choisi un module Arduino Uno, ce

qui nous permettra de développer l'application avec le langage C. De plus, tous les modules possèdent leur bibliothèque Arduino ainsi que des croquis (*sketchs*) comme exemples d'usage.

Nous avons donc un système bâti autour d'une carte Arduino, un *shield* Grove et des modules Grove. Seules 3 LED classiques avec leur résistance seront à ajouter bien qu'il existe aussi des LED Grove, ce qui induit un petit surcoût pour avoir le zéro soudure.

Nous avons pris le parti de n'avoir qu'une source d'approvisionnement. Il est bien sûr possible d'aller acheter ailleurs.

La liste des matériels chez le revendeur français Lextronic **[4]** est la suivante :

MODULE	QTÉ.	RÉFÉRENCE	PU TTC	LIEN INTERNET
Carte Arduino UNO Rev 3	1	A000066	19,45 €	**https://www.lextronic.fr/carte-arduino-uno-dip-rev3-A000066-2474.html**
Platine Grove Base Shield	1	103030000	4,80 €	**https://www.lextronic.fr/platine-grove-base-shield-v2-103030000-14174.html**
Capteur Grove CO2 température et humidité Sensirion	1	101020634	68,80 €	**https://www.lextronic.fr/capteur-grove-co2-temperature-et-humidite-101020634-40619.html**
Afficheur Grove LCD 2x16 RGB	1	104030001	14,94 €	**https://www.lextronic.fr/module-grove-afficheur-lcd-rgb-2x16-104030001-29434.html**
Cordon USB A mâle - USB B mâle	1	WEN68712	2,30 €	**https://www.lextronic.fr/cordon-usb-a-male-usb-b-male-1-8m-540.html**
Résistance 220 Ω 1/4 W	3	RE14	0,50 €	**https://www.lextronic.fr/10-resistances-carbones-1-4-w-2426.html**
LED verte 3 mm	1	LED3GLN	0,14 €	**https://www.lextronic.fr/led-verte-diffusante-3mm-3551.html**
LED orange 3 mm	1	LED3OLN	0,40 €	**https://www.lextronic.fr/led-standard-3mm-orange-diffusant-8422.html**
LED rouge 3 mm	1	LED3RLN	0,14 €	**https://www.lextronic.fr/led-rouge-diffusant-3mm-3552.html**

Si l'on veut réduire le prix, il est aussi possible de prendre un afficheur LCD noir sur jaune à 6,49 € TTC (**https://www.lextronic.fr/grove-afficheur-lcd-2x16-noir-jaune-104020113-39824.html**). L'afficheur couleur apporte néanmoins un certain confort visuel.

Nous arrivons au prix de revient hors coffret de 111,47 € TTC soit 112 € TTC, dont près de 62 % correspondent au capteur de CO2.

Il faut noter que le capteur de CO2 n'est pas pris au hasard. Le HCSP **[1]** préconise l'usage des détecteurs de CO2 infrarouge de type NDIR (*Non Dispersive InfraRed*) réputés pour leur précision de mesure. C'est le cas du modèle choisi, d'où son prix élevé.

Il faudra au préalable étalonner le capteur de CO2 avec une valeur de référence de taux de CO2. La procédure sera décrite plus tard.

Hormis les 3 LED, l'assemblage consiste simplement à connecter des nappes Grove.

On assemblera d'abord la carte Arduino Uno avec le *shield*. Puis on connectera l'afficheur LCD et le capteur de CO2 qui ont une interface I2C sur l'un des 4 connecteurs Grove I2C, comme indiqué sur la figure 1.

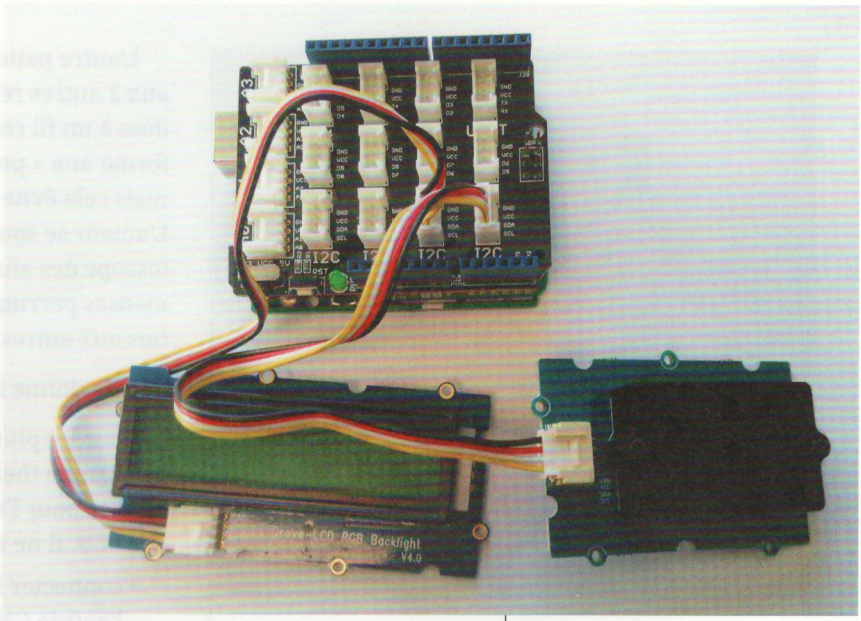


Figure 1 : Assemblage du capteur de CO2 et de l'afficheur LCD.

Enfin, on connectera les 3 LED sur les sorties 2, 3 et 4 du *shield*. Pour cela, on rappelle que sur une LED, la plus grande des pattes correspond au « plus » que l'on connectera ensuite sur les sorties 2, 3 ou 4 de la carte Arduino Uno et la plus courte des pattes soudées à la résistance de 220 Ω comme indiqué sur la figure 2.

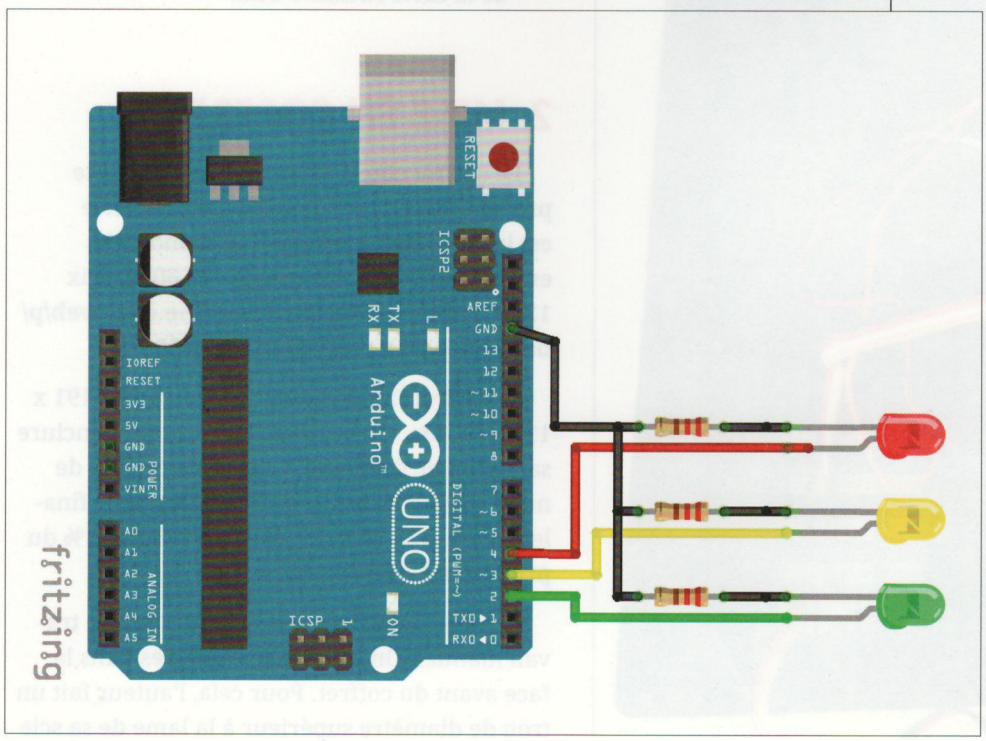


Figure 2 : Câblage des 3 LED.

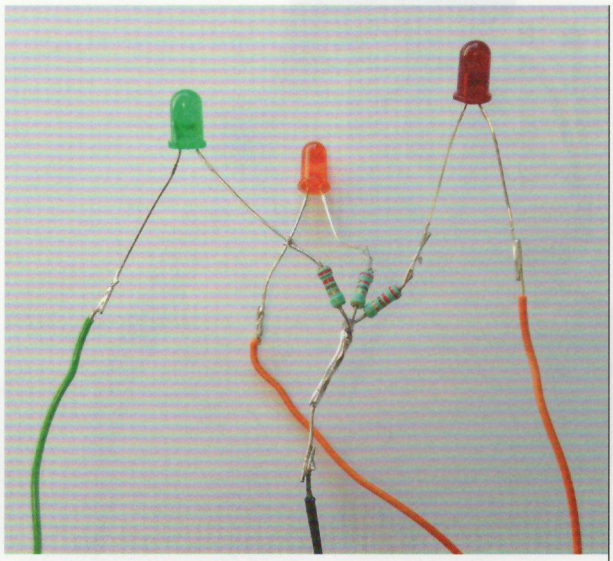
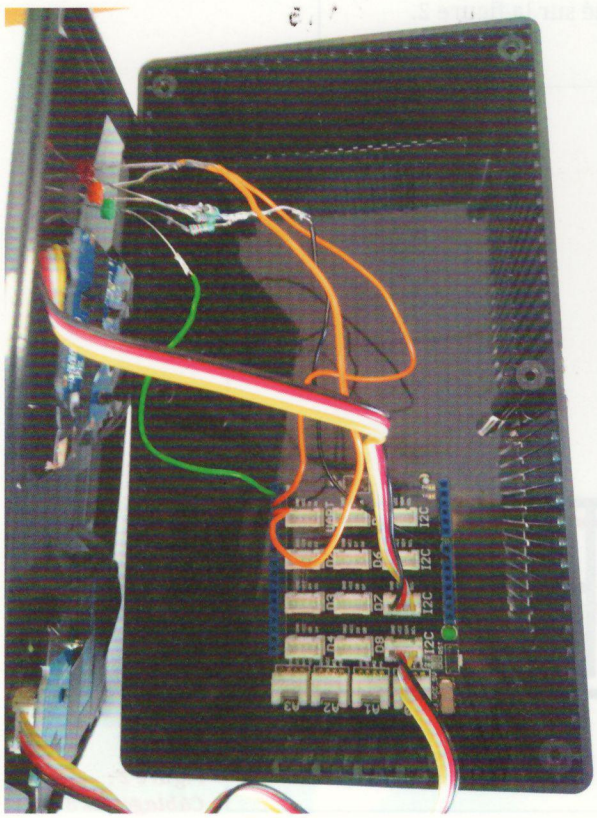


Figure 3 : Soudure des 3 LED.

Figure 4 : Mise en coffret.



L'autre patte de la résistance sera soudée aux 2 autres résistances qui seront ensuite soudées à un fil relié à l'entrée GND du *shield*. Cela forme une « perruque ». Ce n'est pas glamour, mais cela évite de créer un circuit imprimé. L'auteur se souvient avoir démonté un oscilloscope des années (19)60 et découvert d'immenses perruques de composants soudés les uns aux autres sur l'ossature de l'oscilloscope...

Cela donne le résultat de la figure 3.

Pour les plus craintifs, un peu de ruban ou de la gaine thermorétractable peuvent être ajoutés pour l'isolation des parties métalliques des LED. Il ne reste plus qu'à :

- connecter le fil reliant les 3 résistances à l'entrée GND de la carte Arduino Uno ;
- le fil « plus » de la LED verte sur la sortie 2 de la carte Arduino Uno ;
- le fil « plus » de la LED orange sur la sortie 3 de la carte Arduino Uno ;
- le fil « plus » de la LED rouge sur la sortie 4 de la carte Arduino Uno.

2. MISE EN COFFRET

Le coffret choisi est un coffret plastique provenant du revendeur RS. La référence est la suivante : « Boîtier Hammond 1591 en ABS ignifuge », référence 493-6076, prix 13,87 € et URL **https://fr.rs-online.com/web/p/boitiers-pour-usage-general/4936076l**.

Le coffret fait la dimension suivante : 191 x 110 x 61 mm. C'est suffisant pour tout y inclure sans trop de difficulté. Le prix de revient de notre mesureur du taux de CO2 est donc finalement de 125,34 € soit environ 125 € (55 % du prix correspond au capteur).

La mise en coffret demande un peu de travail manuel. On percera les fenêtres dans la face avant du coffret. Pour cela, l'auteur fait un trou de diamètre supérieur à la lame de sa scie

sauteuse, ce qui permet de faire la découpe de la fenêtre rectangulaire. Le perçage des 3 trous pour les 3 LED ne pose aucun problème.

La figure 4 montre le résultat avant fermeture du coffret.

3. OUTIL DE DÉVELOPPEMENT LOGICIEL

On utilisera l'outil Arduino IDE (*Integrated Development Environment*) **[5]** pour le développement logiciel. On pourra l'installer sous Windows ou bien sous GNU/Linux, qui est l'environnement de développement préféré de votre auteur.

Il convient aussi d'installer les bibliothèques « *Adafruit SCD30* » et « *Grove LCD RGB Backlight* » pour les différents modules (capteur de CO2, afficheur LCD). Rien de plus simple sous Arduino IDE. La figure 5 montre les 2 principales bibliothèques installées dans Arduino IDE.

4. ÉTALONNAGE DU CAPTEUR DE CO2

Le capteur de CO2 doit être correctement calibré avant usage. L'annexe 7 du document **[1]** du Haut Conseil de la Santé Publique donne des recommandations générales pour l'étalonnage du capteur de CO2.

On se basera sur la référence du taux de CO2 de l'air extérieur « pur » qui est de l'ordre 400 ppm loin des sources de pollutions comme les zones très urbanisées ou les grands axes routiers.

Dans ces conditions, on fera l'étalonnage à l'air extérieur pour que la procédure soit fiable en absence de vent, entre 14 h et 18 h, moment où l'air est le mieux mélangé et si possible en altitude (depuis la partie supérieure d'un immeuble).

Le capteur de CO2 NDIR utilisé est le modèle SCD30 **[6]** de Sensirion **[7]** **[8]** qui admet 2 procédures de calibrage :

- Procédure ASC (*Automatic Self Calibration*) : procédure de calibrage automatique. Quand le mode ASC est activé, le capteur en mode de mesure continu doit être exposé à l'air extérieur (400 ppm) au moins une heure par jour sur une période minimale de 7 jours pour que l'algorithme trouve sa valeur de référence. Durant cette période, le capteur doit rester sous tension sous peine d'avoir à relancer toute la procédure.

Figure 5 : Bibliothèques Arduino Adafruit SCD30 et Grove LCD RGB Backlight.

- Procédure FRC (*Forced Recalibration*) : on place le capteur dans une enceinte où le taux de CO2 est contrôlé et connu. L'utilisateur programme alors le capteur avec cette valeur de taux de CO2 comme référence.

La procédure ASC prend beaucoup de temps par rapport à la procédure FRC. On préférera alors la procédure FRC. Le taux de CO2 de référence choisi sera celui de l'air extérieur (400 ppm) en respectant les recommandations générales du HCSP pour l'étalonnage. On attendra au minimum 2 minutes **[7]** avant d'appliquer la procédure FRC et de programmer la valeur de 400 ppm comme valeur de référence.

Avec Arduino IDE, on crée alors le croquis suivant :

```
#include "rgb_lcd.h"
#include <Adafruit_SCD30.h>

// Attente de 15 min avant FRC
#define TIME 15L
#define TIME_SECONDE TIME*60L

// Durée de 1 s pour un cycle d'attente
#define TIME_CYCLE 1
#define NBR_CYCLE TIME_SECONDE/TIME_CYCLE

const int colorR = 50;
const int colorG = 0;
const int colorB = 50;
rgb_lcd lcd;

Adafruit_SCD30  scd30;

void setup(void) {

  scd30.begin();

  // Pas de calibrage ASC
  scd30.selfCalibrationEnabled(false);

  lcd.begin(16, 2);
  lcd.setRGB(colorR, colorG, colorB);
}

void loop(void) {
  float co2, temp;
  unsigned int i;
  unsigned long int reste, minute, seconde;

  co2 = 0;
  temp = 0;

  lcd.setCursor(0, 0);
  lcd.print(" CALIBRAGE FRC  ");
```

```
  lcd.setCursor(0, 1);
  lcd.print("v1.0 pk/enseirb");
  delay(2000);
  lcd.clear();

  lcd.setCursor(0, 1);
  lcd.print("Attente de "); lcd.print(TIME, 1); lcd.print(" mn");

  // Attente de 15 min
  while(1) {

    reste = (NBR_CYCLE - i)*TIME_CYCLE;
    minute = reste / 60;
    seconde = reste - (minute * 60);

    lcd.setCursor(0, 1);
    lcd.print("Reste="); lcd.print(minute, 1); lcd.print(":");lcd.
print(seconde, 1);lcd.print("          ");

    delay(TIME_CYCLE*1000); // delay() est en ms
    i++;

    if (i > NBR_CYCLE)
      break;
  }

  // FRC à 400 ppm
  scd30.forceRecalibrationWithReference(400);

  // Lecture du taux de CO2 toutes les 5 secondes
  while(1) {

    scd30.read();
    co2 = scd30.CO2;
    temp = scd30.temperature;

    lcd.setCursor(0, 0);
    lcd.print("CO2="); lcd.print(co2, 0); lcd.print(" ppm        ");

    lcd.setCursor(0, 1);
    lcd.print("Tmp="); lcd.print(temp, 1); lcd.print(" oC        ");

    delay(3*TIME_CYCLE*1000);
  }
}
```

Après avoir placé le capteur de CO2 dans les conditions correctes d'étalonnage, on alimente notre mesureur du taux de CO2.

Le croquis est extrêmement simple. Un compte à rebours de 15 minutes (bien supérieur aux 2 minutes préconisées) s'affiche sur l'afficheur LCD. Puis la procédure FRC est lancée avec la valeur de 400 ppm comme référence. Notre mesureur affiche ensuite toutes les 3 secondes la valeur courante du taux de CO2 mesuré.

Figure 6 : Procédure de calibrage FRC.

Il est à noter que ce n'est pas le croquis final. Lors de la prochaine mise sous tension de notre mesureur du taux de CO2, nous avons au plus 15 minutes pour programmer le croquis final, ce qui est amplement suffisant.

La figure 6 montre la procédure FRC en cours d'exécution.

5. CROQUIS FINAL DU MESUREUR DU TAUX DE CO2

Pour ce croquis final, après calibrage, on affichera sur l'afficheur LCD 2x16 caractères le taux de CO2 mesuré sur la première ligne et la température mesurée sur la deuxième ligne. On fera une lecture du taux de CO2 toutes les 3 secondes.

On gérera en plus les 3 LED :

- si le taux de CO2 est inférieur à 700 ppm, la LED verte est allumée ;
- si le taux de CO2 est compris entre 700 et 800 ppm, la LED orange est allumée. Il va falloir bientôt aérer la pièce ;
- si le taux de CO2 est supérieur à 800 ppm, la LED rouge clignote et change d'état toutes les 200 ms. Il faut aérer la pièce.

Notre mesureur du taux de CO2 sera placé à 1 à 2 m de hauteur **[1]** loin des fenêtres et portes, sources de courant d'air. On ne le placera pas près d'une source de chaleur et l'on ne le mettra pas non plus près d'une source de CO2 comme la bouche d'une personne.

Le croquis Arduino correspondant est alors le suivant :

```
#include "rgb_lcd.h"
#include <Adafruit_SCD30.h>

#define LED_VERTE 2
#define LED_JAUNE 3
#define LED_ROUGE 4

// Durée parcours de la boucle de 200 ms
#define LOOP 200

// Durée du cycle de lecture du capteur de 15 fois LOOP soit 3 s
#define TIME_CYCLE 15

// Seuil jaune 700 ppm CO2
#define SEUIL_JAUNE 700.0
```

```
// Seuil rouge 800 ppm CO2
#define SEUIL_ROUGE 800.0

const int colorR = 50;
const int colorG = 0;
const int colorB = 50;

rgb_lcd lcd;
Adafruit_SCD30  scd30;

void setup(void) {

  pinMode(LED_VERTE, OUTPUT);
  pinMode(LED_JAUNE, OUTPUT);
  pinMode(LED_ROUGE, OUTPUT);
  digitalWrite(LED_VERTE, LOW);
  digitalWrite(LED_JAUNE, LOW);
  digitalWrite(LED_ROUGE, LOW);

  scd30.begin();

  lcd.begin(16, 2);
  lcd.setRGB(colorR, colorG, colorB);
}
void loop(void) {
  float co2, temp;
  int toggle;
  int i;

  co2 = 0;
  temp = 0;
  toggle = 0;
  i = 0;

  lcd.setCursor(0, 0);
  lcd.print("-  Mesure CO2  -");
  lcd.setCursor(0, 1);
  lcd.print("v2.0 pk/enseirb");

  scd30.read();
  delay(3000);
  lcd.clear();

  while(1) {

    if(i == TIME_CYCLE) {
      i = 0;
```

```
      scd30.read();
      co2 = scd30.CO2;
      temp = scd30.temperature;
    }

    if(co2 < SEUIL_JAUNE) {
      digitalWrite(LED_VERTE, HIGH);
      digitalWrite(LED_JAUNE, LOW);
      digitalWrite(LED_ROUGE, LOW);

      lcd.setCursor(0, 0);
      lcd.print("CO2="); lcd.print(co2, 0); lcd.print(" ppm        ");
    }
    if(co2 >= SEUIL_JAUNE && co2 < SEUIL_ROUGE) {
      digitalWrite(LED_VERTE, LOW);
      digitalWrite(LED_JAUNE, HIGH);
      digitalWrite(LED_ROUGE, LOW);

      lcd.setCursor(0, 0);
      lcd.print("CO2="); lcd.print(co2, 0); lcd.print(" ppm        ");
    }
    if(co2 >= SEUIL_ROUGE) {
      lcd.setCursor(0, 0);
      lcd.print("CO2="); lcd.print(co2, 0); lcd.print(" ppm        ");

      digitalWrite(LED_VERTE, LOW);
      digitalWrite(LED_JAUNE, LOW);

        if(toggle) {
          toggle = 0;
          digitalWrite(LED_ROUGE, LOW);
        }
        else {
          toggle = 1;
          digitalWrite(LED_ROUGE, HIGH);
        }
    }

    lcd.setCursor(0, 1);
    lcd.print("Tmp="); lcd.print(temp, 1); lcd.print(" oC        ");

    i++;
    delay(LOOP);
  }
}
```

La figure 7 montre le mesureur du taux de CO2 en cours de fonctionnement.

CONCLUSION

À travers cet article, nous avons pu voir la réalisation complète d'un mesureur du taux de CO2 construit autour d'une carte Arduino Uno.

Il n'y a pas de difficultés majeures et la réalisation est à la portée de tout bon lecteur de Hackable Magazine.

Ce mesureur du taux de CO2 permet de contrôler en temps réel le taux de CO2 d'une pièce et d'agir en conséquence selon les préconisations.

À vous de jouer maintenant !

Les croquis Arduino sont disponibles à l'adresse : **https://kadionik.vvv.enseirb-matmeca.fr/pub/co2/.** PK

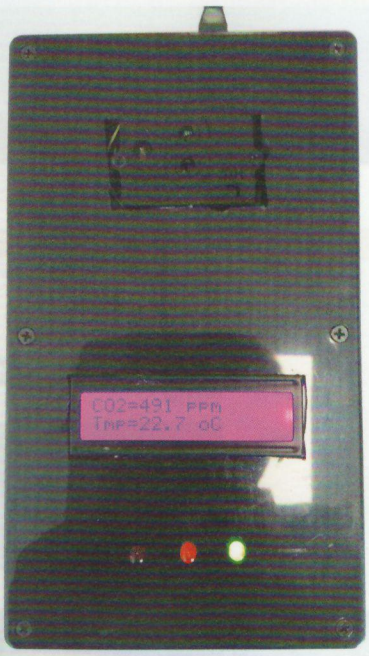


Figure 7 : Notre mesureur du taux de CO2.

BIBLIOGRAPHIE

[1] Avis du HCSP du 28 avril 2021 relatif à l'adaptation des mesures d'aération, de ventilation, et de mesure du dioxyde de carbone (CO2) dans les établissements recevant du public (ERP) pour maîtriser la transmission du SARS-CoV-2 : **https://www.hcsp.fr/explore.cgi/avisrapportsdomaine?clefr=1009**

[2] Avis du HCSP du 14 mai 2021 et du 21 mai 2021 relatif au recours à des unités mobiles de purification de l'air dans le cadre de la maîtrise de la diffusion du SARS-CoV-2 dans les espaces clos : **https://www.hcsp.fr/explore.cgi/avisrapportsdomaine?clefr=1014**

[3] Société Seeed : **https://www.seeedstudio.com/**

[4] Société Lextronic : **https://www.lextronic.fr/**

[5] Arduino IDE : **https://www.arduino.cc/en/software**

[6] Capteur de CO2 Sensirion SCD30 : **https://wiki.seeedstudio.com/Grove-CO2_Temperature_Humidity_Sensor-SCD30/**

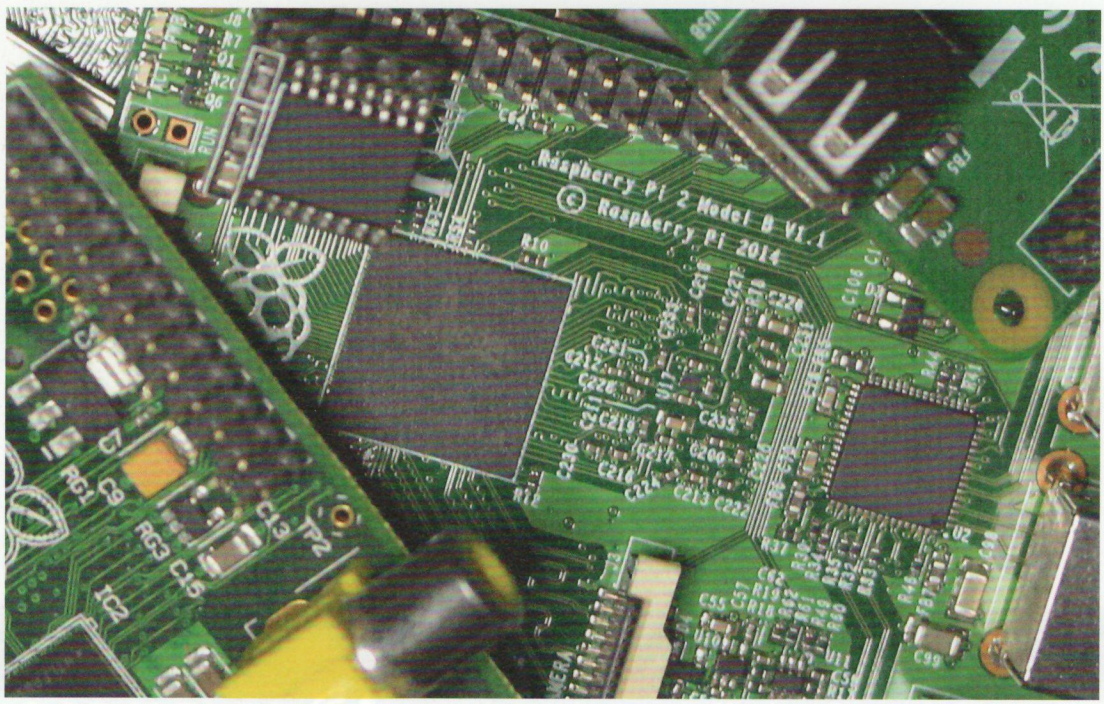
[7] Field calibration for SCD30 : **https://edgecollective.io/img/co2/CD_AN_SCD30_Field_Calibration_D2.pdf**

[8] Interface Description Sensirion SCD30 Sensor Module : **https://files.seeedstudio.com/wiki/Grove-CO2-Temperature-Humidity-Sensor-SCD30/res/Sensirion_CO2_Sensors_SCD30_Interface_Description.pdf**

RASPBERRY PI : ET POURQUOI PAS NETBSD ?

Denis BODOR

Il y a bien des manières d'utiliser un SBC comme une Raspberry Pi et, surtout, bien des systèmes capables d'y fonctionner. Mais qu'il s'agisse du mal nommé Raspberry Pi OS, d'Ubuntu Desktop, de Raspbian, de LibreElec ou encore de RetroPie, nous avons affaire au même système GNU/Linux, ou en d'autres termes plus précis, un système GNU reposant sur un noyau Linux. Pourtant, il existe d'autres options à votre disposition, et NetBSD est l'une d'entre elles.



– Raspberry Pi : et pourquoi pas NetBSD ? –

Au-delà du fait passablement irritant de voir une distribution GNU/Linux être adaptée à la Pi, puis affublée en conséquence du qualificatif de « système », l'omniprésence de Linux a quelque chose de relativement ennuyeux. En effet, une distribution, quelle qu'elle soit, est encore et toujours ce bon vieux système GNU/Linux, une poignée d'outils standard auxquels s'ajoute un système de gestion de paquets spécifique, un *desktop* particulier et une poignée d'applications par défaut, le tout savamment ficelé. À quelques exceptions près, comme RISC OS, Windows IoT Core ou encore MINIX3, on se retrouve donc techniquement toujours avec la même chose. Un peu comme se voir servir sans cesse le même plat, en variant simplement l'assaisonnement... N'avez-vous pas envie de manger autre chose, pour changer ?

1. NETBSD

Je vous ferai grâce ici de la très longue et détaillée leçon d'histoire, mais quelques notions importantes doivent toutefois être établies. La grande aventure des systèmes UNIX débute bien avant l'arrivée de GNU/Linux ou même du PC IBM. En 1969, Ken Thompson et Dennis Ritchie commencent le développement d'un nouveau système qui deviendra rapidement UNIX et fut réécrit par la suite dans un tout nouveau langage inventé par les deux compères : le C.

UNIX prend rapidement de l'ampleur, évolue rapidement et gagne en popularité. En 1976, la version 6 commence à sortir des laboratoires Bell où il a été créé et séduit de nombreux programmeurs, et pour cause : il est excessivement bien structuré, cohérent, logique, propre et surtout conçu pour être portable. À cette époque, peu ou prou tous les constructeurs d'ordinateurs développent leur propre système, incompatible avec les autres. Mais UNIX est différent, il est développé sur PDP-11, mais pensé pour pouvoir être facilement adapté pour n'importe quelle machine. Il est facilement « portable ».

Il est important à ce stade de préciser que la notion de système fait référence à la fois au noyau qui orchestre l'exécution des programmes, et à la collection d'outils permettant d'utiliser l'ensemble. Lorsqu'on parle de portabilité et du fait de produire un système s'exécutant sur une autre architecture ou un autre modèle d'ordinateur, on parle de recompiler le tout, noyau et outils, pour cette cible.

Or justement, en 1977 l'équipe de développement s'étoffe et Bill Joy produit la première distribution appelée *Berkeley Software Distribution* ou BSD. Par « distribution », il faut comprendre ici « lots de modifications dans les sources permettant d'obtenir un système composé différemment ». Ce BSD, et l'année suivante une seconde version appelée 2BSD apportent à UNIX version 6 un compilateur Pascal, l'éditeur ex puis Vi et le C shell (csh), entre autres choses. Cette distribution évolue pour devenir 3BSD (1979) et 4.3BSD (1986). À ce stade, une bonne partie du code appartient à AT&T et les licences concédées sont relativement coûteuses. Une version spécifique, remplaçant le code AT&T, voit le jour sous la forme d'une version Net/1 (*Networking Release 1*) puis Net/2 (*Networking Release 2*), respectivement dans les années 1989 et 1991.

À ce stade, BSD, maintenant un système à part entière, n'est disponible que sur gros systèmes et William et Lynne Jolitz décident de créer une version pour ces machines personnelles ayant gagné fortement en popularité : les PC à processeur Intel 80386. Rapidement, une communauté se crée

autour de ce nouveau 386BSD 0.1 (1992), apportant des modifications et faisant évoluer le code sous forme de *patchs*. La rigidité du projet (et des Jolitz) contrarie grandement certains développeurs qui décident, puisque la licence le permet, de faire bande à part. Deux groupes « dissidents » se forment, le premier donne naissance à FreeBSD et le second à NetBSD, sous l'impulsion de Chris Demetriou, Theo de Raadt (oui, le caustique Theo d'OpenBSD), Adam Glass, et Charles Hannum.

La première version officielle 0.8 de NetBSD voit le jour en avril 1993 sur la base de 386BSD en incluant les modifications (*patchkit* officieux 0.2.2) souhaitées par les développeurs mécontents. Plus tard, en octobre 1994, NetBSD 1.0 arrive, définitivement débarrassé de son héritage litigieux avec 4.3BSD Net/2 et les menaces légales liées aux licences AT&T. En 1994, le comportement de Theo de Raadt envers les utilisateurs et développeurs est jugé intolérable et celui-ci est prestement invité à quitter le projet. Il crée alors OpenBSD en 1995, un système basé sur NetBSD et qui donnera naissance à de nombreux composants massivement utilisés aujourd'hui (OpenSSH, PF, la libc d'Android, etc.).

Comme vous pouvez le voir, la naissance d'UNIX, de BSD et de toutes la collection de systèmes liés est faite de rebondissements, de choix, de litiges et d'évolutions. Mais une chose reste parfaitement constante tout du long : le caractère central et critique de la portabilité, de la standardisation et de la cohérence du système.

S'il fallait faire une illustration de ce qui différencie GNU/Linux d'un héritier de BSD comme NetBSD, je pense qu'un appartement serait l'idéal. GNU/Linux est composite et organique, c'est une chambre d'adolescent, pratique, fonctionnelle et, disons-le honnêtement, un peu bordélique. NetBSD pour sa part, c'est une illustration de catalogue Ikea se résumant à « une place pour chaque chose et chaque chose à sa place ».

Ne le prenez pas mal, je suis utilisateur de GNU/Linux depuis plus de 25 ans et ce système est mon quotidien, car il a ses qualités. Mais à chaque usage son système, les avantages qui l'accompagnent et les défauts qui sont les siens. Si vous voulez jouer ou devez utiliser des applications spécifiques, Windows est un choix. Si vous développez et désirez une station de travail d'usage courant où tout le matériel (ou presque) est supporté sans trop d'encombres, GNU/Linux est une bonne option. Et enfin, si vous aimez l'ordre et la rigueur, un environnement propre, cohérent et maîtrisable, pour une application spécifique, NetBSD méritera toute votre attention.

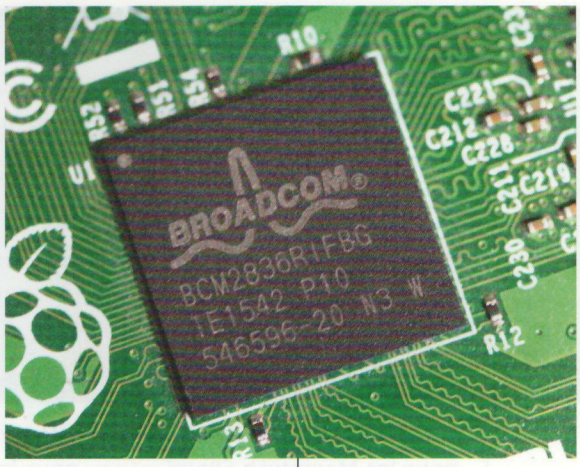
Cet article a pour objectif de vous permettre de faire connaissance avec NetBSD et la philosophie qui dicte sa structure et son fonctionnement. Et ce de façon physique et réelle, et non sur une machine virtuelle. Il y a plus d'une manière d'installer NetBSD, FreeBSD ou OpenBSD sur une Raspberry Pi et nous choisirons ici, non pas la plus simple et la plus rapide, mais la plus pédagogique (et amusante).

2. DANS L'ESPRIT UNIX

Oubliez les « dérives » comme systemd et PulseAudio, la notion de distributions, les installations sauvages à coup de « sudo make install » et les frontières floues entre système et applicatifs. Le monde BSD n'est pas celui de GNU/Linux, de Debian, d'Ubuntu et de Raspberry Pi OS.

NetBSD est un système. Ceci signifie que ce qui est maintenu et développé par le projet consiste en un noyau et un jeu d'outils et bibliothèques formant

un tout utilisable. En ce sens, lorsqu'on parle d'installer NetBSD, l'objectif est d'obtenir quelque chose de complet et d'utilisable. Bien entendu, on pourra compléter l'installation d'éléments supplémentaires sous la forme de paquets (source ou binaires), mais qui ne font pas partie intégrante du système lui-même. Ils ne sont donc pas installés à la racine du système, mais dans une arborescence placée dans `/usr/pkg`. Nous avons alors une parfaite distinction entre ce qu'est le système NetBSD et le reste des outils, applications, utilitaires, configurations, bibliothèques, etc. Ceci est l'une des nombreuses différences entre NetBSD et GNU/Linux, une autre est la façon dont on compilera le système.



Le broadcom BCM2836 n'est qu'un des très nombreux SoC supportés par NetBSD et ceci est valable aussi bien pour les architectures ARM (Allwinner, Apple, NVIDIA, Ti, STM, Amlogic, etc.), MIPS, Risc-V ou PPC.

Oui, il est possible d'installer une distribution binaire de NetBSD, que ce soit sur PC ou l'une des nombreuses plateformes supportées. Mais pour réellement appréhender la philosophie UNIX, et BSD en particulier, quoi de plus normal que de tout simplement compiler le système pour sa Raspberry Pi ?

J'ai choisi ici d'utiliser une Raspberry Pi 2 B 1.1 équipée d'un SoC BCM2836 (32 bits donc). En réalité, la cible importe peu, pas plus que le système hôte, en l'occurrence un Debian GNU/Linux 9.11 (pas de toute première jeunesse). Une Pi est une plateforme embarquée, hors de question donc de compiler sur la cible elle-même, mais ceci n'a pas grande importance, je pourrais parfaitement décider de compiler sur aarch64 (ARMv8 64 bits) et sous NetBSD à destination de amd64 ou x86, les choses se dérouleraient exactement de la même manière. Pourquoi ? Parce que la portabilité est la pierre angulaire du système. Et ceci ne se limite pas à la portabilité du code ou du système, mais bel et bien également de ce qui permet de le construire. En l'occurrence, c'est **le même script** qui sera exécuté et produira **le même système**, quel que soit l'hôte utilisé. Ceci est l'essence même de NetBSD.

Ce script est `build.sh` et se trouve à la racine des sources de NetBSD. Pour construire un système pour la Pi 2, rien de plus simple, il suffit de commencer par récupérer le nécessaire avec CVS (paquet éponyme sous Debian) :

```
$ mkdir -p ~/PI/NetBSD
$ cd ~/PI/NetBSD
$ export CVSROOT="anoncvs@anoncvs.NetBSD.org:/cvsroot"
$ export CVS_RSH="ssh"
$ cvs checkout -r netbsd-9-2-RELEASE -P src
```

Vous obtiendrez un répertoire `src` contenant les sources officielles de NetBSD 9.2. `build.sh` est juste là et vous pourrez immédiatement l'utiliser pour lister les machines et architectures cibles disponibles pour le futur système :

```
$ cd src
$ ./build.sh list-arch
[...]
MACHINE=evbarm  MACHINE_ARCH=earmv6eb    ALIAS=evbearmv6-eb     ALIAS=evbarmv6-eb
MACHINE=evbarm  MACHINE_ARCH=earmv6hfeb  ALIAS=evbearmv6hf-eb   ALIAS=evbarmv6hf-eb
MACHINE=evbarm  MACHINE_ARCH=earmv7      ALIAS=evbearmv7-el     ALIAS=evbarmv7-el
MACHINE=evbarm  MACHINE_ARCH=earmv7eb    ALIAS=evbearmv7-eb     ALIAS=evbarmv7-eb
MACHINE=evbarm  MACHINE_ARCH=earmv7hf    ALIAS=evbearmv7hf-el   ALIAS=evbarmv7hf-el
MACHINE=evbarm  MACHINE_ARCH=earmv7hfeb  ALIAS=evbearmv7hf-eb   ALIAS=evbarmv7hf-eb
MACHINE=evbarm  MACHINE_ARCH=aarch64     ALIAS=evbarm64-el      ALIAS=evbarm64 DEFAULT
[...]
```

Il y en a 106 en tout et dans la liste nous trouvons des choses très amusantes comme `mac68k` pour les Macintosh 68xxx, `riscv` pour les architectures RISC-V, `next68k` pour les NeXT, `i386` pour les PC 32 bits, `amd64` pour les PC récents et toute une collection d'architectures ARM. Parmi celles-ci, nous avons la machine `evbarm` et l'architecture `earmv7hf` correspondants aux cœurs ARM Cortex-A7 du SoC BCM2836 de la Pi 2.

Que vous disposez ou non d'une chaîne de compilation adaptée à la cible est sans importance. L'objectif est de créer un système de façon prédictible et consistante. En ce sens, l'approche souhaitable est d'utiliser une chaîne de compilation de manière déterministe et donc parfaitement maîtrisée. En d'autres termes, il n'est pas raisonnable de reposer sur l'une ou l'autre version d'une chaîne déjà installée, mais on en produira une nouvelle, contenant tous les outils nécessaires. C'est précisément la première chose à faire avec `build.sh` en précisant les options suivantes :

```
./build.sh -U -j10 -m evbarm -a earmv7hf tools
```

Les outils, fonctionnant sur l'hôte, mais destinés à produire des binaires pour la cible choisie seront installés dans `obj/` et, ici, plus précisément `obj/tooldir.Linux-4.16.5-x86_64/bin/`. Là, nous trouvons les bintutils, GCC, gmake, make, etc. Notez que les options permettent respectivement une construction sans utiliser de privilèges particuliers (`-U`), une construction en parallèle de 10 tâches (`-j 10`, à adapter en fonction de votre CPU, ici un bi-Xeon E5520), la « machine » `evbarm` et l'architecture `earmv7hf`. Enfin, ce que nous voulons, c'est produire les outils, `tools`.

Cette étape passée, nous sommes équipés pour créer des binaires ARMv7 et pourrions construire le noyau. Il existe un grand nombre de plateformes basées sur une architecture ARM de ce type et elles ont chacune leurs spécificités. Les configurations du noyau se trouvent dans une arborescence dépendante de l'architecture cible placée dans `sys/arch/`. Dans notre cas, nous trouverons le nécessaire dans `sys/arch/evbarm/conf/` et en particulier le fichier `RPI2`. Si vous jetez un œil au contenu de ce fichier, vous remarquerez qu'il inclut `RPI`, lui-même incluant `std.rpi` et `GENERIC.common`. L'ensemble de ces fichiers détermine les éléments de configuration à appliquer, les options et les fonctionnalités souhaitées. Ici, nous n'avons pas de changement à faire puisque la configuration matérielle est unique et que les options nous conviendraient parfaitement. Nous pourrions donc procéder à la compilation du noyau, toujours avec `build.sh` :

```
./build.sh -u -U -j10 -m evbarm -a earmv7hf kernel=RPI2
===> build.sh command: ./build.sh -u -U -j10 -m evbarm -a earmv7hf kernel=RPI2
===> build.sh started: Mon Nov 29 07:09:08 CET 2021
===> NetBSD version:   9.2
===> MACHINE:          evbarm
===> MACHINE_ARCH:     earmv7hf
===> Build platform:   Linux 4.16.5 x86_64
===> HOST_SH:          /bin/sh
===> No $TOOLDIR/bin/nbmake, needs building.
===> Bootstrapping nbmake
checking for sh... /bin/sh
checking for gcc... cc
[...]
===> Summary of results:
  build.sh command:    ./build.sh -U -j10 -m evbarm -a earmv7hf kernel=RPI2
  build.sh started:    Mon Nov 29 07:09:23 CET 2021
  NetBSD version:      9.2
  MACHINE:             evbarm
  MACHINE_ARCH:        earmv7hf
  Build platform:      Linux 4.16.5 x86_64
  HOST_SH:             /bin/sh
  No $TOOLDIR/bin/nbmake, needs building.
  Bootstrapping nbmake
  MAKECONF file:       /etc/mk.conf (File not found)
  TOOLDIR path:        /home/denis/PI/NetBSD/src/obj/tooldir.Linux-4.16.5-x86_64
  DESTDIR path:        /home/denis/PI/NetBSD/src/obj/destdir.evbarm
  RELEASEDIR path:     /home/denis/PI/NetBSD/src/obj/releasedir
  Created /home/denis/PI/NetBSD/src/obj/tooldir.Linux-4.16.5-x86_64/bin/nbmake
  Updated makewrapper:
  /home/denis/PI/NetBSD/src/obj/tooldir.Linux-4.16.5-x86_64/bin/nbmake-evbarm
  Building kernel without building new tools
  Building kernel:     RPI2
  Build directory:     /home/denis/PI/NetBSD/src/sys/arch/evbarm/compile/obj/RPI2
  Kernels built from RPI2:
  /home/denis/PI/NetBSD/src/sys/arch/evbarm/compile/obj/RPI2/netbsd
  build.sh ended:      Mon Nov 29 07:10:57 CET 2021
===> .
```

Notez que c'est **exactement** la même démarche qui s'appliquera (et le même script) pour recompiler un noyau NetBSD sur le système lui-même, ou un noyau à destination d'une autre cible. Mieux encore, cette compilation du noyau n'est pas réellement nécessaire ici puisque nous pouvons construire tout le système (*userland*) de la même façon, en une fois, avec :

```
./build.sh -U -j10 -m evbarm -a earmv7hf release
[...]
```

Notez que nous nous passons ici de spécifier une configuration particulière pour le noyau. Par défaut, c'est `GENERIC` qui sera utilisé et, de fait, elle conviendra parfaitement pour la Pi. Cette étape sera bien plus longue que la compilation des outils, puisque nous traitons tout le système, mais ce que nous obtiendrons sera directement utilisable sur la Pi. Au risque de me répéter, NetBSD est un système complet que nous construisons à partir de sources uniques. S'il fallait faire le pendant avec GNU/Linux, ceci est comparable à un système de construction pour l'embarqué comme Yocto/OpenEmbedded ou Buildroot, mais avec une origine unique et non des éléments issus d'un ensemble de sources.

Notez que nous nous laissons ici un grand nombre de paramètres et d'options à leurs valeurs par défaut. Je vous recommande de jeter un œil à la sortie d'une commande `./build.sh` sans argument, ainsi qu'au fichier `BUILDING` pour en apprendre davantage et, par exemple, d'indiquer un chemin spécifique pour le résultat de la construction (et non juste stocker dans `obj/releasedir`).

Une fois la construction terminée, nous obtenons, entre autres choses, une image compressée de la carte SD/MMC dans `obj/releasedir/evbarm/binary/gzimg/`. Nous pouvons alors l'enregistrer sur le média :

```
$ cd obj/releasedir/evbarm/binary/gzimg/
$ gunzip armv7.img.gz
$ sudo dd if=armv7.img of=/dev/sdd bs=1M conv=sync
1028+1 enregistrements lus
1029+0 enregistrements écrits
1078984704 bytes (1,1 GB, 1,0 GiB)
  copied, 100,883 s, 10,7 MB/s
$ rm armv7.img
```

Le support est maintenant prêt pour être glissé à l'emplacement dédié de la Pi. Le système est, de base, configuré pour mettre à disposition une console série et vous pourrez donc connecter un adaptateur USB/série pour prendre la main sur le système.

```
[...]
machdep.cpu.frequency.target: 600 -> 900
Updating motd.
Starting ntpd.
Starting sshd.
Starting postfix.
Starting inetd.
Starting cron.
Mon Nov 29 10:00:35 UTC 2021

NetBSD/evbarm (armv7) (constty)

login:
```

3. UTILISATION ET CONFIGURATION

Sachez que le service OpenSSH est également automatiquement actif, tout comme le client DHCP, mais le seul compte utilisateur existant à ce stade est `root` et celui-ci ne peut être utilisé directement via SSH pour des raisons évidentes de sécurité. La première étape consistera donc à se connecter en `root` (pas de mot de passe) via la console série pour créer un utilisateur. Un outil de configuration est disponible, c'est `sysinst` et celui-ci vous permet justement d'ajouter un utilisateur. Personnellement, je préfère le faire manuellement en ligne de commandes pour ensuite utiliser `sysinst` via SSH (plus réactif) :

```
armv7# useradd -m -G wheel denis

armv7# passwd denis
Changing password for denis.
New Password:
Retype New Password:

armv7#
```

Dès lors, les connexions SSH seront possibles avec cet utilisateur. L'adresse IP de la Pi peut être obtenue à l'aide d'un simple `ifconfig`, mais vous pouvez également utiliser la résolution mDNS avec `armv7.local` puisque ce service est également démarré sur le système de base. Le nom d'hôte de la machine pourra être changé via l'édition du fichier `/etc/rc.conf`, le fichier de configuration du démarrage du système.

Notez qu'il est de bon ton d'également de définir un mot de passe (différent) pour `root`, car en l'état, le nouvel utilisateur n'a qu'à utiliser la commande `su` pour passer super utilisateur.

Une fois la connexion établie via SSH, passez `root` puis utiliser `sysinst`. Via l'entrée « *Config menu* », vous pourrez ajuster le fuseau

Rien de plus amusant que de ressortir les vieilles peluches pour l'occasion, puisque ce cher Beastie (alias « Chuck » ou tout simplement « BSD Daemon ») est la mascotte de tous les systèmes de la famille BSD (sauf OpenBSD qui préfère le diodon Puffy).

horaire local (« *Timezone* »), créer un mot de passe pour `root` (« *Change root password* »), choisir les services de base à lancer au démarrage, ajouter un utilisateur, etc. Deux des entrées du menu sont particulièrement utiles :

- « *Fetch and unpack pkgsrc* » : `pkgsrc` a longtemps été la manière standard d'installer des applicatifs tiers dans le système. Ce système consiste à maintenir une arborescence spécifique dans `/usr/pkgsrc` contenant un ensemble de fichiers permettant d'obtenir, de compiler et d'installer des logiciels. Pour ce faire, on se déplace dans l'arborescence à l'endroit où se trouve l'élément de son choix et on se plie d'un simple `make install` en tant que super utilisateur. Contrairement à ce que la commande peut laisser penser, il ne s'agit pas simplement de compiler et d'installer « à la sauvage » un applicatif comme on le ferait avec une commande similaire depuis des sources « standard » de l'applicatif. Ce `make install` n'a rien à voir avec le classique, et horrible, `./configure && make && sudo make install` qu'on voit très malheureusement un peu partout. Jetez simplement un coup d'œil au `Makefile` se trouvant dans un sous-répertoire de `/usr/pkgsrc` et vous comprendrez. L'ensemble de l'arborescence occupe quelques 1,3 Go et procéder de cette manière sous-entend compiler (parfois de très gros éléments) sur une plateforme qui, ici, n'est clairement pas adaptée.
- « *Enable installation of binary packages* » : l'autre solution, similaire au fonctionnement de systèmes comme APT sous Debian, consiste à simplement installer des binaires qui auront été compilés par quelqu'un d'autre, après avoir été téléchargés depuis un dépôt officiel. Ceci vous permet d'utiliser la commande `pkgin` afin de lister, chercher, installer, supprimer, etc., des paquets.

Dans les deux cas, un sous-système de gestion de paquets permet de gérer les dépendances et de maintenir une cohérence dans ce qui est présent sur le système. Utiliser les paquets binaires est la méthode la plus raisonnable sur Raspberry Pi et il vous suffira de valider cette entrée du menu pour disposer de `pkgin` :

```
$ pkgin stats
Local package database:
        Installed packages: 2
        Disk space occupied: 1673K

Remote package database(s):
        Number of repositories: 1
        Packages available: 22784
        Total size of packages: 40G
```

Nous avons quelques 22784 paquets à notre disposition et pouvons alors étoffer l'installation pour la rendre plus confortable d'utilisation. À titre d'exemple, nous allons ajouter la commande `sudo`, Vim (ou `nano` si vous y tenez vraiment) et le shell Bash de GNU avec :

```
$ su
Password:

# pkgin in sudo vim bash
reading local summary...
processing local summary...
calculating dependencies...done.

4 packages to install:
  sudo-1.9.7p1 vim-8.2.3172 bash-5.1.8nb3 vim-share-8.2.3172nb1

0 to refresh, 0 to upgrade, 4 to install
11M to download, 39M to install

proceed ? [Y/n] y
vim-share-8.2.3172nb1.tgz 100% 7565KB 3.7MB/s 00:02
vim-8.2.3172.tgz          100% 1395KB 1.4MB/s 00:00
bash-5.1.8nb3.tgz         100% 2798KB 1.4MB/s 00:02
installing sudo-1.9.7p1...
[...]
installing vim-8.2.3172...
installing bash-5.1.8nb3...
bash-5.1.8nb3: adding /usr/pkg/bin/bash to /etc/shells
installing vim-share-8.2.3172nb1...
pkg_install warnings: 0, errors: 0
reading local summary...
processing local summary...
marking sudo-1.9.7p1 as non auto-removable
marking vim-8.2.3172 as non auto-removable
marking bash-5.1.8nb3 as non auto-removable
```

Il ne nous reste plus ensuite qu'à configurer deux ou trois choses :

- faire de Bash le shell par défaut avec `chpass -s /usr/pkg/bin/bash` ;
- se composer un petit `~/.bashrc` pour se sentir à la maison (ajouter un alias de `vi` vers `vim` par exemple), sans oublier de « sourcer » le fichier depuis `~/.profile` (après avoir testé `$BASH_VERSION` pour être sûr d'exécuter Bash) ;
- ajouter l'utilisateur dans `/usr/pkg/etc/sudoers` (avec quelque chose comme `denis ALL=(ALL:ALL) ALL`) pour pouvoir utiliser `sudo` et non `su` ;
- activer le support de langue native (NLS) en ajoutant `export LANG="fr_FR.UTF-8"` à votre `~/.bashrc`.

Avec cela, vous devriez vous sentir plus où moins comme à la maison et, même s'il ne s'agit pas de Raspbian/Raspberry Pi OS, retrouver assez facilement vos repères. Vous êtes maintenant en mesure d'explorer le système à loisir et de faire connaissance plus avant avec NetBSD. Et ce avec quelque chose qui est plus tangible qu'une simple émulation (VM), sans pour autant bloquer un PC complet, et donc de manière très économique.

4. POUR CONCLURE

Soyons clairs, NetBSD ne remplacera pas une installation générique de GNU/Linux, ni sur une Raspberry Pi ni sur un PC *desktop*. Pour une utilisation quotidienne, même certains des développeurs NetBSD reposent sur GNU/Linux ou FreeBSD (parfois même macOS). Ce que NetBSD a pour lui, en revanche, c'est son architecture et sa cohérence. Ici, nul besoin d'intégrer des choses comme systemd, censés ajouter de l'ordre et de la cohésion (et ne faisant en vérité que déplacer le problème, sinon l'empirer), car ces deux éléments sont d'ores et déjà partie intégrante du système : une place pour chaque chose et chaque chose à sa place.

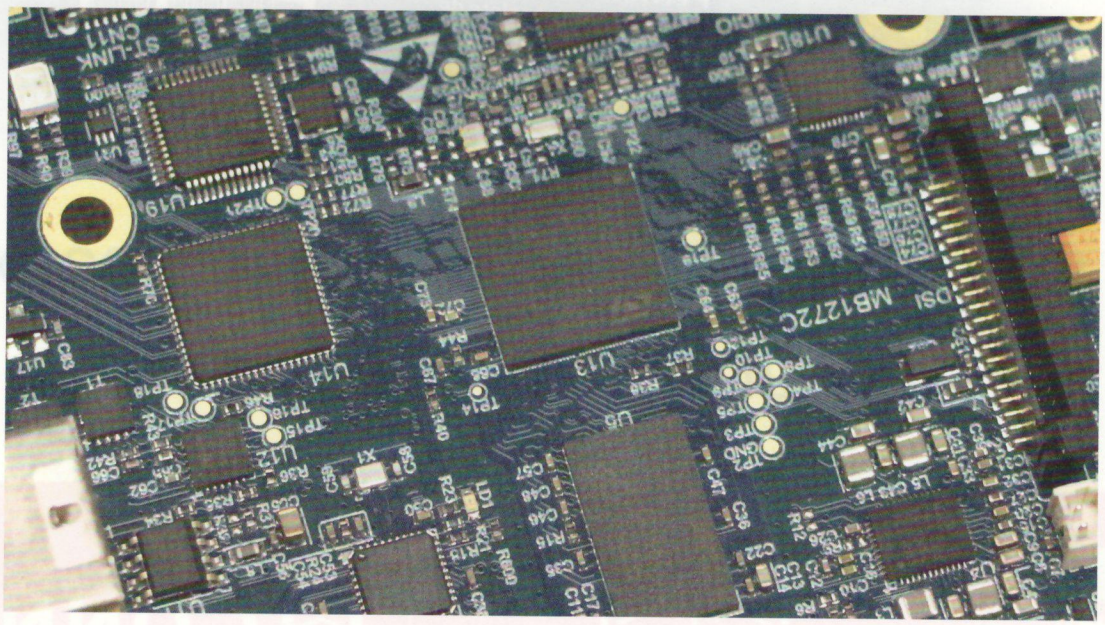
Il est facile de faire le pendant avec les langages de programmation. NetBSD (tout comme OpenBSD, FreeBSD et dans une certaine mesure MINIX) est aux systèmes d'exploitation ce que le C est au développement. La compétence et la rigueur sont les responsabilités de l'opérateur, tout comme l'allocation mémoire, sa libération, les manipulations de pointeurs et autres techniques jugées « dangereuses », sont celles du développeur. Oui, cela signifie implicitement que vous devez faire beaucoup de choses vous-même, mais également que rien ne pourra vous surprendre si vous maîtrisez votre sujet. Vous êtes aux commandes, pas de surprises, pas de choses qui se trament en coulisse et donc pas de comportement « sorti de nulle part »...

En cela, NetBSD est un environnement maîtrisable et déterministe qui se prête admirablement à la mise en œuvre de services réseau. Au-delà donc du caractère pédagogique du présent article, peut-être trouverez-vous cette rigueur et cette cohérence à votre goût et déciderez-vous de mettre en œuvre NetBSD de façon permanente, pour y installer, par exemple, un *broker* MQTT, un proxy, un serveur DHCP, une instance de Grafana... Sachez enfin que cette caractéristique se retrouve également dans les sources du système, tant au niveau noyau que dans l'espace utilisateur (`build.sh` à lui seul est une mine d'or pour un amateur de scripts shell). **DB**

STM32MP1 : LE SOC QUI ÉTEND L'ÉCOSYSTÈME STM32 VERS LINUX EMBARQUÉ

Denis BODOR

Voilà un titre digne d'une publication pour décideur pressé, mais nous parlerons bien de technique (quoi d'autre, sinon ?). Il n'y a pas si longtemps, STMicroelectronics annonçait l'arrivée de la famille de SoC STM32MP1, combinant la puissance de cœurs ARM Cortex-A7 à l'efficacité d'un Cortex-M4, le tout réuni sur une seule puce. La commercialisation récente des « Discovery Kits » et du STM32MP157F-DK2 en particulier est l'occasion rêvée de prendre la bête en main.



Lorsqu'on entend « STM32 », on pense automatiquement « microcontrôleur » ou « MCU », mais cette famille, déjà très importante et diversifiée, s'étend depuis quelque temps déjà bien au-delà de ce que même un STM32H7 (Cortex-M7) est en mesure de fournir. Nous parlons, en effet, ici d'un SoC double-cœur Cortex-A7 bourré de fonctionnalités (HDMI, SPDIF, Ethernet Giga, LCD-TFT, MIPI DSI, GPU Vivante avec OpenGL ES, etc.) avec, en prime, un Cortex-M4 intégré. Là, on ne parle plus de MCU, mais de MPU capable de supporter un OS dit « riche » (en comparaison avec un RTOS comme FreeRTOS, Chibios ou RIOT). Pour prendre en main cette petite merveille, rien de mieux qu'un kit officiel incluant tout le nécessaire : le STM32MP157F-DK2.

Les « *Discovery kits* » se différencient de la gamme Nucleo en intégrant des éléments nécessaires pour évaluer la solution et faire la démonstration des fonctionnalités spécifiques à un MCU ou MPU. Celui qui nous intéresse ici, construit autour du MPU le plus « haut de gamme » de la famille STM32MP1 à ce moment, inclus :

- un STM32MP157F à 800 MHz ;
- 512 Mo de RAM DDR3L ;
- une interface gigabit Ethernet ;
- deux ports USB-C, dont un USB OTG HS (l'autre est pour l'alimentation) ;
- 4 ports USB 2 Type-A ;
- un codec audio ;
- divers LED et boutons ;
- un port HDMI ;
- un connecteur MIPI DSI ;
- un connecteur GPIO 40 broches « compatible Raspberry Pi » ;
- un connecteur « Arduino Uno V3 » ;
- un débogueur/programmeur ST-LINK/V2-1 intégré ;
- un emplacement microSD (avec une carte 16 Go) ;
- Wi-Fi 802.11b/g/n ;
- Bluetooth Low Energy 4.1 ;
- et un écran TFT 4 pouces, 480x800 MIPI, tactile capacitif.

La comparaison avec une Raspberry Pi 3 ou 4 est tentante, mais malgré la présence de connecteurs incitant à le faire, il s'agirait d'une erreur. Pour à peine plus que le prix d'une Pi 4 8 Go, vous avez là un *devkit* et non un SBC généraliste. Ce que j'entends par là concerne bien entendu le matériel, mais également et surtout la documentation (un vrai « *Reference manual* » de quelques 4000 pages), un SDK complet, plusieurs implémentations de référence, des contributions « *mainline* » aux différents projets concernés et, dans le cas du STM32MP157F, un **vrai** support *Trusted Execution Environment* (OP-TEE) et *Trusted Firmware-A* (TF-A) reposant sur TrustZone.

Remarquez que le STM32MP157F-DK2 n'est pas le seul *Discovery Kit* disponible pour ce SoC. Il existe également le STM32MP157D-DK1, moins coûteux, sans écran, Wi-Fi, BLE et accélération matérielle AES 128/192/256. Les deux sont en vente chez les détaillants habituels comme Mouser, Farnel, RS ou Digi-Key, mais au moment de la préparation de cet article, seul le DK2 était disponible chez Mouser et ces problèmes d'approvisionnement ne semblent malheureusement pas près d'être réglés (pas courant 2022, *a priori*).

1. ENVIRONNEMENT OFFICIEL

Contrairement aux solutions économiques chinoises se résumant généralement à une simple carte et éventuellement un BSP (*Board Support Package*) sans aucune évolution planifiée, ou à des SBC aussi populaires que généralistes comme les Raspberry Pi, mais n'offrant que des *datasheets* édulcorées, la famille STM32MP15 dispose d'un écosystème complet, vraiment complet.

ST a en effet créé un environnement complet d'outils et d'éléments logiciels appelés *STM32MPU Embedded Software distribution*, composé d'applicatifs, d'un système de construction et d'outils de développement à destination des MPU STM32. Cet ensemble, en version v3.1.0 actuellement, comprend :

- une distribution GNU/Linux OpenSTLinux basée sur le *framework* de construction OpenEmbedded (LTS Dunfell) et intégrant un BSP supportant une chaîne de *boot* sécurisée (TF-A), un OS sécurisé OP-TEE et un *kernel* Linux standard en mode non sécurisé. Le tout intégrant plusieurs *frameworks* applicatifs modernes comme Wayland/Weston, ALSA, GC Nano ou encore Gstreamer ;
- une distribution Buildroot basée sur le BSP d'OpenSTLinux sous la forme d'une extension (BR2_EXTERNAL) pour Buildroot 2021.02, développée via un partenariat entre ST et Bootlin **[1]** ;
- un paquet STM32CubeMP1 fournissant le support (HAL, CMSIS, pilotes, etc.) pour le coprocesseur ARM Cortex-M4 intégré au SoC. Ainsi qu'une mise à jour de STM32CubeMX permettant à la fois la génération du *device tree* (Cortex-A7) et la configuration des fonctionnalités du Cortex-M4 (pas de génération de `Makefile` pour l'instant) ;
- STM32CubeProgrammer supportant la génération de clés cryptographiques et la signature des *firmwares*, et permettant de programmer le SoC et donc le *Discovery Kit* via USB ou UART, en plus de gérer l'accès à l'interface STLINK/V2-1 intégré au kit ;
- l'environnement de développement intégré STM32CubeIDE basé sur Eclipse (on aime ou l'on n'aime pas).

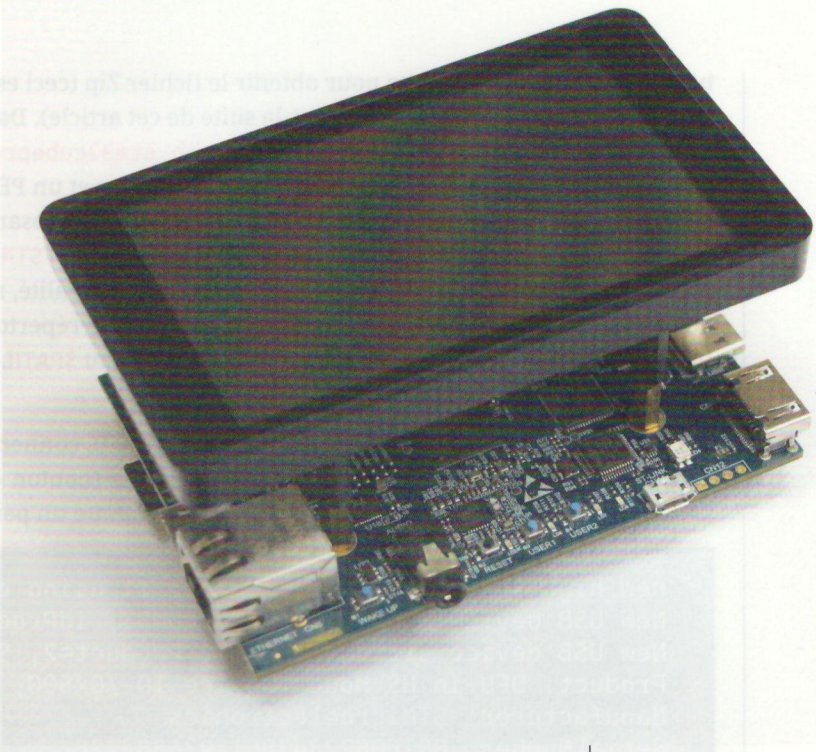
Je ne vous cache pas que cet « écosystème » utilise une terminologie un peu déroutante en mélangeant des notions qui, à mon sens, ne vont pas réellement ensemble et c'est l'une des principales difficultés que l'on rencontre dans la masse importante de documentations en ligne (wiki ST). C'est par exemple le cas pour la notion de « *Software Packages* » qu'on pourrait plus simplement résumer en parlant de « paquets d'éléments pour une tâche donnée » :

- le « *Starter Package* » est simplement un ensemble d'images binaires à enregistrer sur la microSD, mais via une connexion USB-C et STM32CubeProgrammer, pour obtenir un système utilisable. C'est tout bonnement une version installable du système de démonstration, plus récent que celui sans doute déjà présent sur la microSD au moment où vous recevez le produit ;
- le « *Developer Package* » est un SDK contenant les éléments nécessaires pour développer à destination d'OpenSTLinux qu'il s'agisse d'applications, de modules noyau, d'un noyau personnalisé, de modifications sur le *bootloader* U-Boot, etc. Voyez cela comme un simple environnement de cross-développement vous permettant de créer des binaires *userspace* ou

de modifier des composants (*kernel, device tree,* U-Boot, etc.) du système. On comprend déjà ici que « *Starter Package* » et « *Developer Package* » ne se rangent pas vraiment dans la même catégorie ;

- et le « *Distribution Package* » qui, comme le précise le wiki ST, vous permet « *de créer votre propre distribution, votre propre Starter Package et votre propre Developer Package* ». Vous êtes perdu ? Moi aussi, et pour cause, le « *Distribution Package* » n'est rien de plus qu'une installation d'OpenEmbedded-Core avec quelques *layers* supplémentaires, spécifiques à la plateforme, le tout téléchargeable via *repo* **[2]**.

Nous avons donc ici trois choses de nature complètement différentes, présentées comme des « *packages* », alors qu'il s'agit typiquement de trois cas d'usage différents : flasher la carte avec un OS binaire, disposer d'un SDK pour compléter/modifier un système existant et reconstruire tout le système avec le *Build System* OpenEmbedded. De plus, comme nous le verrons en pratique plus loin, deux de ces « *packages* » sont le résultat de l'utilisation du troisième. Il y a donc comme un mélange de choux et de carottes, il me semble...



Le STM32MP157F-DK2 est un ensemble complet comprenant la carte équipée d'un écran LCD tactile de 480x800 pixels. Le modèle STM32MP157D-DK1 est assez similaire, mais n'intègre ni écran ni Wi-Fi et est équipé d'un SoC plus modeste.

1.1 Starter Package : juste pour commencer

Ce package, ou en d'autres termes, l'image de la distribution OpenSTLinux en version binaire, n'a pour but que de vous permettre de flasher le *firmware* sur la carte microSD. Mais cette opération ne se fera pas, comme on peut s'y attendre, en copiant simplement l'image sur le support, avec dd par exemple (même si cela reste parfaitement possible). La mise à jour du système se fera par une connexion USB, en DFU (le protocole standardisé *Device Firmware Upgrade*), via le connecteur USB-C marqué « USB » se trouvant à côté du port HDMI, après avoir passé la carte dans le bon mode à l'aide des micro-interrupteurs BOOT0 et BOOT2 sous PCB (les deux sur OFF).

Bien que DFU soit standard, nous allons utiliser l'outil ST comme recommandé dans la documentation. STM32CubeProgrammer est téléchargeable sur le site officiel **[3]** en version binaire pour GNU/Linux, Win32, Win64 et macOS, gratuitement, mais cela demandera

un enregistrement sur le site pour obtenir le fichier Zip (ceci est également valable pour les autres éléments téléchargeables dans la suite de cet article). Dans le cas d'un environnement GNU/Linux, l'archive, à cette date, se nomme `en.stm32cubeprg-lin_v2-9-0_v2.9.0.zip` et contient étrangement deux exécutables (un EFL Linux et un PE32 pour Windows) ainsi qu'un répertoire `jre`. Il s'agit en réalité d'un installeur vous proposant de déployer l'outil, après acceptation des conditions d'utilisation, par défaut dans `~/STMicroelectronics/STM32Cube/STM32CubeProgrammer`. Peu importe la destination en réalité, mais vous remarquerez rapidement un certain penchant, chez ST, pour les noms de répertoires et de fichiers à rallonge. Inutile non plus d'ajouter le répertoire d'installation au `$PATH`, nous pouvons parfaitement appeler l'outil avec un chemin absolu.

Une fois l'outil installé, la carte passée en mode DFU, connectée en USB-C, alimentée via l'autre connecteur USB-C (« PWR_IN ») et réinitialisée (bouton « RESET » à droite du connecteur jack audio), celle-ci se fera connaître du système comme un périphérique en mode DFU :

```
new high-speed USB device number 11 using ehci-pci
New USB device found, idVendor=0483, idProduct=df11
New USB device strings: Mfr=1, Product=2, SerialNumber=3
Product: DFU in HS Mode @Device ID /0x500, @Revision ID /0x0000
Manufacturer: STMicroelectronics
SerialNumber: 001E00223438511436383238
```

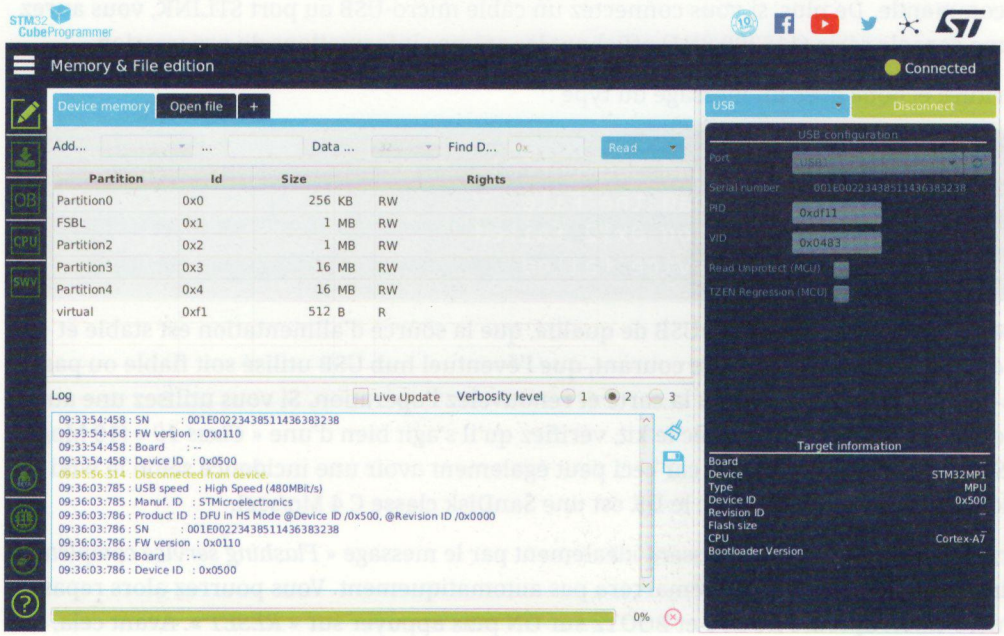
Pour autoriser les utilisateurs non `root` à accéder à ce périphérique (ainsi qu'aux autres interfaces ST), un certain nombre de règles udev sont à votre disposition dans le répertoire d'installation de STM32CubeProgrammer, sous `Drivers/rules`. Vous pouvez les installer en copiant les fichiers dans `/etc/udev/rules.d`, puis en rafraîchissant avec un `udevadm control --reload-rules && udevadm trigger` (en `root`). Dès lors, il vous devient possible d'utiliser STM32CubeProgrammer, que ce soit en version GUI ou en ligne de commandes (préférable) et donc de vérifier que la carte est bien détectée :

```
$ STM32_Programmer_CLI -l
-------------------------------------------
        STM32CubeProgrammer v2.9.0
-------------------------------------------

=====  DFU Interface   =====

Total number of available STM32 device in DFU mode: 1

Device Index       : USB1
USB Bus Number     : 002
USB Address Number : 004
Product ID         : DFU in HS Mode @Device ID /0x500, @Revision ID /0x0000
Serial number      : 001E00223438511436383238
Firmware version   : 0x0110
Device ID          : 0x0500
[...]
```

```
$ STM32_Programmer_CLI -c port=usb1
-------------------------------------------------
        STM32CubeProgrammer v2.9.0
-------------------------------------------------
USB speed   : High Speed (480MBit/s)
Manuf. ID   : STMicroelectronics
Product ID  : DFU in HS Mode @Device ID /0x500, @Revision ID /0x0000
SN          : 001E002234385114363832 38
FW version  : 0x0110
Board       : --
Device ID   : 0x0500
Device name : STM32MP1
Device type : MPU
Revision ID : --
Device CPU  : Cortex-A7
```

Un outil comme `dfu-util` fonctionnera également, mais n'apportera, tout naturellement, pas d'informations spécifiques sur la plateforme. Nous pouvons à présent télécharger le « *Starter Package* » depuis le site ST **[4]** sous la forme d'une archive `en.FLASH-stm32mp1-openstlinux-5-10-dunfell-mp1-21-11-17_tar_v3.1.0.xz` (je vous avais prévenu pour les noms à rallonge).

À ce stade, il est recommandé d'immédiatement faire preuve de rigueur dans votre future arborescence comme le précise le wiki ST **[5]**, pour clairement distinguer les « *packages* ». Ceci peut paraître anodin, mais étant donné les désignations utilisées, cela prend rapidement tout son sens. Nous désarchiverons alors le fichier avec `tar xfJv` pour obtenir un répertoire `stm32mp1-openstlinux-5.10-dunfell-mp1-21-11-17` contenant lui-même le sous-répertoire `images`, puis `stm32mp1`, et trouverons là une arborescence regroupant ce que nous cherchons.



STM32CubeProgrammer existe en deux versions, une en ligne de commandes et une disposant d'une interface graphique. Personnellement, je ne vois pas l'intérêt d'un outil graphique pour ce type d'opérations découlant généralement d'un système de construction (Makefile, OpenEmbedded, Buildroot, etc.) et devant donc être facilement automatisable. Mais tous les goûts sont dans la nature, paraît-il...

Le support microSD est partitionné en fonction des éléments qui y sont placés (FSBL, FIP, U-Boot, rootfs, etc.) et nous avons une image par partition. Plutôt que de flasher individuellement ces images, l'arborescence met à notre disposition des tables de partitions dans `flashlayout_st-image-weston`, sous la forme de fichier `.tsv`. Il nous suffit donc d'utiliser `STM32_Programmer_CLI` en spécifiant le port (`-c`) et l'option `-w` suivie du TSV concerné. Pour flasher la distribution, nous nous plaçons donc dans `stm32mp1-openstlinux-5.10-dunfell-mp1-21-11-17/images/stm32mp1` et utilisons :

```
$ STM32_Programmer_CLI -c port=usb1 -w \
flashlayout_st-image-weston/trusted/\
FlashLayout_sdcard_stm32mp157f-dk2-trusted.tsv
[...]
RUNNING Program ...
  PartID:      :0x06
Start operation done successfully at partition 0x06

Memory Programming ...
Opening and parsing file:
  st-image-bootfs-openstlinux-weston-stm32mp1.ext4
File          : st-image-bootfs-openstlinux-weston-stm32mp1.ext4
Size          : 64 MBytes
Partition ID  : 0x10

Download in Progress:
[============================                    ]  58%
[...]
```

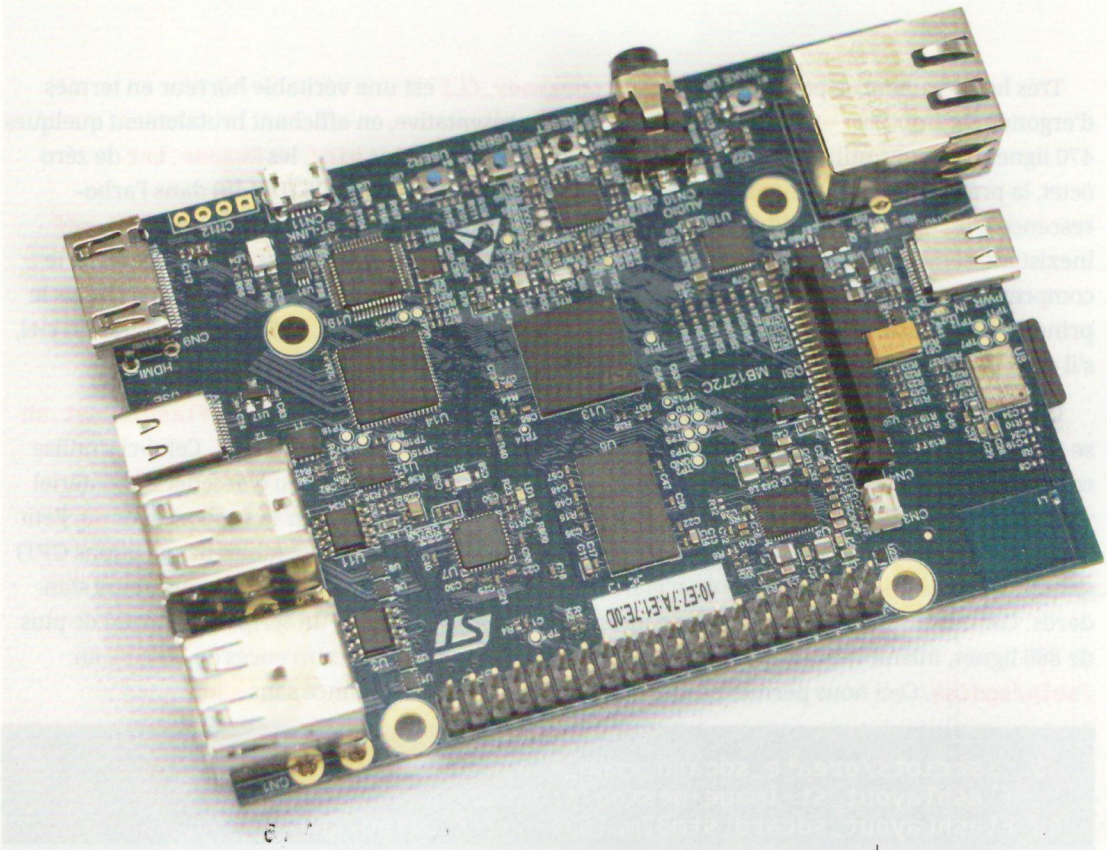
L'opération est relativement longue, étant donné la taille du système et la latence propre à la microSD. L'écran du kit affichera la progression en même temps que votre terminal où est exécutée la commande. De plus, si vous connectez un câble micro-USB au port STLINK, vous aurez accès à une console série (115200 8N1) affichant les mêmes informations de progression.

Si l'écriture échoue avec un message du type :

```
Error: failed to download Segment[0]
Error: failed to download the File
Error: Download partition 0x12 failed
Error: TSV flashing service failed
```

Assurez-vous d'utiliser un câble USB de qualité, que la source d'alimentation est stable et capable de fournir suffisamment de courant, que l'éventuel hub USB utilisé soit fiable ou passez-vous-en. Procédez à un *reset* de la carte et renouvelez l'opération. Si vous utilisez une autre carte microSD que celle fournie avec le kit, vérifiez qu'il s'agit bien d'une « *Class 10* » (10 Mo/s) ou « *UHS Speed Class 1* » (10 Mo/s), car ceci peut également avoir une incidence selon le wiki **[6]** de ST (alors que la carte livrée avec le DK est une SanDisk classe C 4 Mo/sec).

Au terme de la procédure, se finissant idéalement par le message « *Flashing service completed successfully* », la plateforme ne redémarrera pas automatiquement. Vous pourrez alors repasser les micro-interrupteurs BOOT0 et BOOT2 sur ON puis appuyer sur « *RESET* ». Avant cela, si



La carte en elle-même reprend le format Raspberry Pi avec ses 40 connecteurs sur le côté, le hub USB 4 ports, le jack audio, l'interface Ethernet... mais complète l'ensemble avec un agencement sensiblement différent.

ce n'est pas déjà fait, il est recommandé de brancher un câble micro-USB sur le port STLINK et de vous connecter au port série ayant fait son apparition (`/dev/ttyACM0`) avec GNU Screen, Minicom, Picocom ou autre. Vous pourrez alors voir toutes les informations de démarrage, du *bootloader* jusqu'au shell.

Ce premier démarrage prendra un temps plus important que les suivants du fait de la vérification des systèmes de fichiers ainsi que de leur redimensionnement. Soyez patient et au bout de quelques minutes, vous devriez voir apparaître l'application de démonstration sur l'écran LCD et une invite de shell sur la console série :

```
[  OK  ] Started Update UTMP about System Runlevel Changes.
[  OK  ] Started Weston Wayland Compositor (on tty7).

ST OpenSTLinux - Weston - (A Yocto Project Based Distro)
3.1.11-openstlinux-5.10-dunfell-mp1-21-11-17 stm32mp1 ttySTM0

stm32mp1 login: root (automatic login)

Last login: Sun Sep 20 10:44:14 UTC 2020 on tty7

root@stm32mp1:~#
```

Très honnêtement, je pense que `STM32_Programmer_CLI` est une véritable horreur en termes d'ergonomie. L'option `--help` a elle seule est très représentative, en affichant brutalement quelques 470 lignes d'options utilisables, tout comme le `a.out` traînant dans `bin/`, les `Readme.txt` de zéro octet, la présence de bibliothèques dynamique Windows (DLL) et macOS (DYLIB) dans l'arborescence d'un outil ELF GNU/Linux ou encore une FAQ qui parle d'un manuel `doc/UM2336.pdf` inexistant, remplacé par une *Application note* `AN2606.pdf` sur les *bootmodes* des MCU STM32. Je comprends parfaitement le souhait de tout centraliser au sein d'un outil unique universel, mais le principe KISS existe précisément pour éviter ce genre de conséquences. Et ST, pour l'amour du ciel, s'il vous plaît, ajoutez une page de manuel (*manpage*) pour cet outil.

Une solution alternative existe cependant, via le script `create_sdcard_from_flashlayout.sh` se trouvant dans le bien nommé sous-répertoire `scripts/` d'`images/stm32mp1/`. Celui-ci s'utilise comme `STM32_Programmer_CLI` en spécifiant un fichier `.tsv`, mais au lieu d'accéder au matériel via USB-C, il produira une image `.raw` qu'on pourra transférer ensuite sur le support avec `dd`. Petit problème cependant, le script utilise `sgdisk` (outil permettant l'édition de tables de partitions GPT) qui se trouve dans `/sbin`, un chemin n'étant normalement pas dans le `PATH` des utilisateurs standards. Comme il est absolument hors de question d'exécuter, en `root`, un script non vérifié de plus de 880 lignes, mieux vaudra alors le modifier en remplaçant toutes occurrences de `sgdisk` en `/sbin/sgdisk`. Ceci nous permettra alors de l'utiliser en toute confiance sans `sudo` :

```
$ ./scripts/create_sdcard_from_flashlayout.sh \
  flashlayout_st-image-weston/trusted/ \
  FlashLayout_sdcard_stm32mp157f-dk2-trusted.tsv

Create Raw empty image:
flashlayout_st-image-weston/trusted/../../
  FlashLayout_sdcard_stm32mp157f-dk2-trusted.raw
  of 1536MB
Create partition table:
[CREATED] part 1:    fsbl1 [partition size 256.0 KiB]
[CREATED] part 2:    fsbl2 [partition size 256.0 KiB]
[CREATED] part 3:      fip [partition size 4.0 MiB]
[CREATED] part 4:     boot [partition size 64.0 MiB]
[CREATED] part 5: vendorfs [partition size 16.0 MiB]
[CREATED] part 6:   rootfs [partition size 744.5 MiB]
[CREATED] part 7:   userfs [partition size 707.0 MiB]

Partition table from flashlayout_st-image-weston/trusted/
  ../../FlashLayout_sdcard_stm32mp157f-dk2-trusted.raw

Populate raw image with image content:
[ FILLED ] part 1:    fsbl1, image:
   arm-trusted-firmware/tf-a-stm32mp157f-dk2-sdcard.stm32
[ FILLED ] part 2:    fsbl2, image:
   arm-trusted-firmware/tf-a-stm32mp157f-dk2-sdcard.stm32
[ FILLED ] part 3:      fip, image:
   fip/fip-stm32mp157f-dk2-trusted.bin
[ FILLED ] part 4:     boot, image:
   st-image-bootfs-openstlinux-weston-stm32mp1.ext4
```

```
[ FILLED ] part 5: vendorfs, image:
   st-image-vendorfs-openstlinux-weston-stm32mp1.ext4
[ FILLED ] part 6:   rootfs, image:
   st-image-weston-openstlinux-weston-stm32mp1.ext4
[ FILLED ] part 7:   userfs, image:
   st-image-userfs-openstlinux-weston-stm32mp1.ext4
[...]
```

Nous obtenons un fichier `.raw` de quelques 1,5 Go étant l'image de la microSD, ainsi qu'un fichier du même nom, suffixé `how_to_update.txt` résumant la structure des partitions et les informations permettant de mettre à jour ce qu'elles contiennent.

1.2 Developer Package : un SDK

Le système binaire, autrement appelé « *Starter Package* », est une simple démonstration technique des fonctionnalités du MPU, du *devkit* et du support logiciel. Ce n'est pas réellement un environnement destiné à une utilisation courante, et c'est bien normal, la plateforme elle-même ne l'est pas. Nous n'avons pas là une tentative de création d'un ordinateur ARM *desktop* généraliste, mais bel et bien un système embarqué. Certes, vous pouvez si vous le souhaitez administrer le système, supprimer et ajouter des paquets (APT), personnaliser l'environnement, la configuration, les services, etc., mais ce n'est clairement pas l'objectif.

En cela, développer pour cette plateforme ne consiste donc pas à installer un environnement dédié, comme un IDE et une chaîne de compilation sur celle-ci (ça ne devrait d'ailleurs jamais être le cas), mais bel et bien de développer sur une machine performante à destination de cette cible. Généralement, ceci passe par l'installation d'une chaîne de compilation et la confection d'un environnement dédié comprenant les éléments logiciels indispensables (*headers*, lib, outils, *debugger*, etc.). Pour simplifier cela, ST met à disposition un « *Developer Package* » constitué d'un SDK basé sur celui du projet Yocto, un ensemble de sources des éléments constituant le système (noyau, *bootloader*, TF-A, OP-TEE), un paquet STM32Cube pour le Cortex-M4 intégré et un IDE spécifique, STM32CubeIDE.

Là encore, cette notion de « *package* » est, à mon sens, peu appropriée, puisqu'il ne s'agit pas d'un tout monolithique, mais de différents composants téléchargeables et installables individuellement, et pour certains, optionnellement. Le principal composant est bien entendu le SDK, téléchargeable via le site officiel **[7]** sous la forme d'une archive `en.SDK-x86_64-stm32mp1-openstlinux-5.10-dunfell-mp1-21-11-17.tar_v3.1.0.xz` à décompresser avec `tar xfJv` et fournissant un répertoire `stm32mp1-openstlinux-5.10-dunfell-mp1-21-11-17` qui n'est pas le SDK, mais un installeur contenu dans un sous répertoire `sdk/` sous la forme d'un imposant script shell qu'il vous suffira d'exécuter. Celui-ci vous demandera simplement de spécifier un répertoire de destination pour l'installation (et de confirmer) :

```
ST OpenSTLinux - Weston - (A Yocto Project Based Distro)
SDK installer version 3.1.11-openstlinux-5.10-dunfell-mp1-21-11-17
==========================================================================
Enter target directory for SDK (default: /usr/local/oecore-x86_64):
```

À ce stade, on peut commencer très sérieusement à se demander si ST n'a pas un problème avec les répertoires en général (et je ne plaisante qu'à moitié). Pourquoi vouloir installer le SDK par défaut dans `/usr/local`, ce qui nécessiterait les permissions du super-utilisateur, alors que l'environnement de développement est précisément fait pour éviter de torturer inutilement son système comme nous allons le voir plus loin ?

Quoi qu'il en soit, une fois le SDK installé dans le répertoire désigné plus judicieusement par vos soins (et, oui, on peut spécifier un chemin relatif), ces fichiers ainsi que l'archive seront inutiles et peuvent être supprimés.

Comme le précise le message en fin d'installation, le script `environment-setup-cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi`, se trouvant à la racine de l'arborescence créée, devra être « sourcé » avec « `.` » ou la commande `source` suivie de son nom. Dès lors, un certain nombre de variables d'environnement auront été définies ou complétées, parmi lesquelles `PATH`, `CC`, `LD`, `OBJCOPY`, `GDB` ou encore `CFLAGS` et `LDFLAGS`. Cette évaluation du script fourni est décrite dans la documentation officielle (wiki ST) comme constituant un « démarrage du SDK » (« *Starting up the SDK* »).

Une fois cet environnement « démarré », on pourra très simplement vérifier son bon fonctionnement en quelques lignes. Avec un peu de C :

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
        printf("Bonjour STM32MP1 !\n");
        return(EXIT_SUCCESS);
}
```

Un `Makefile` basique :

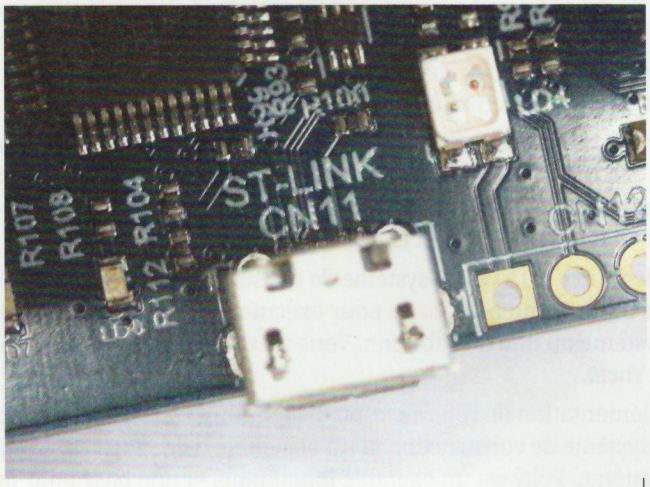
```
TARGET   := main
WARN     := -Wall
CFLAGS   += ${WARN}

all: ${TARGET}

${TARGET}.o: ${TARGET}.c
${TARGET}: ${TARGET}.o

clean:
  rm -rf ${TARGET}.o ${TARGET}
```

Et quelques commandes :

```
$ make
$ scp main stm32mp1.local:~/
$ ssh stm32mp1.local ./main
Bonjour STM32MP1 !
```

Ceci, en supposant que vous avez : connecté la carte au réseau Ethernet, ajouté un utilisateur standard et ajouté sa clé dans le `~/.ssh/authorized_keys` distant pour faciliter la copie de fichiers avec `scp`. Ceci est, bien entendu, un exemple simpliste n'utilisant que la libc. Pour des développements plus avancés, vous trouverez dans l'arborescence du SDK un répertoire `sysroots` contenant deux sous-répertoires :

- `cortexa7t2hf-neon-vfpv4-ostl-linux-gnueabi/` : le *sysroot* ou racine système de l'environnement cible est littéralement le contenu du système de fichiers de la cible embarqué. C'est donc là que nous trouvons les bibliothèques et les *headers* adéquats.
- `x86_64-ostl_sdk-linux/` : est, en quelque sorte, l'équivalent pour le système sur lequel nous développons, contenant ce dont nous aurons besoin en cours de développement (comme la libncursesw pour le `menuconfig` du noyau, le binaire `cmake`, `ninja` ou encore `pkg-config`).



Le devkit propose une interface ST-LINK/V2-1 intégrée permettant à la fois de déboguer et de fournir une console série pour le système installé sur la carte microSD.

Ces deux éléments découlent directement de l'utilisation du dernier « *package* » que nous allons voir dans un instant et qui n'est autre que le système de construction OpenEmbedded du projet Yocto. OpenEmbedded, accompagné des *layers* spécifiques au STM32MP1, permet de générer un système pour le *devkit*, mais également le SDK correspondant. Le *Starter Package* et le *Developer Package* que nous venons de télécharger, d'installer et de mettre en œuvre ont été produits par ce système de construction. Il y a donc une dépendance entre la configuration utilisée pour cette construction et les deux « *packages* » en question.

Tout l'intérêt du système, et de cette apparente complexité inutile, est précisément de permettre la construction d'un écosystème complet où, sur la base d'une distribution élaborée avec OpenEmbedded, on sera en mesure de construire non seulement un système pour la cible, mais le SDK qui lui est associé pour des développements futurs.

Nous aurions encore d'autres choses à détailler concernant le *Developer Package*, comme la cross-compilation et la réinstallation d'un noyau Linux ou de U-Boot, mais non seulement ceci est relativement bien détaillé sur le wiki ST **[8]**, mais repose également sur un environnement de développement fonctionnel comme celui que nous venons de mettre en œuvre et sur des procédures qui sont en réalité décrites dans des `README.HOW_TO.txt` qui font partie intégrante des recettes OpenEmbedded **[9]** et **[10]**.

1.3 Distribution Package : ce dont nous avons vraiment besoin

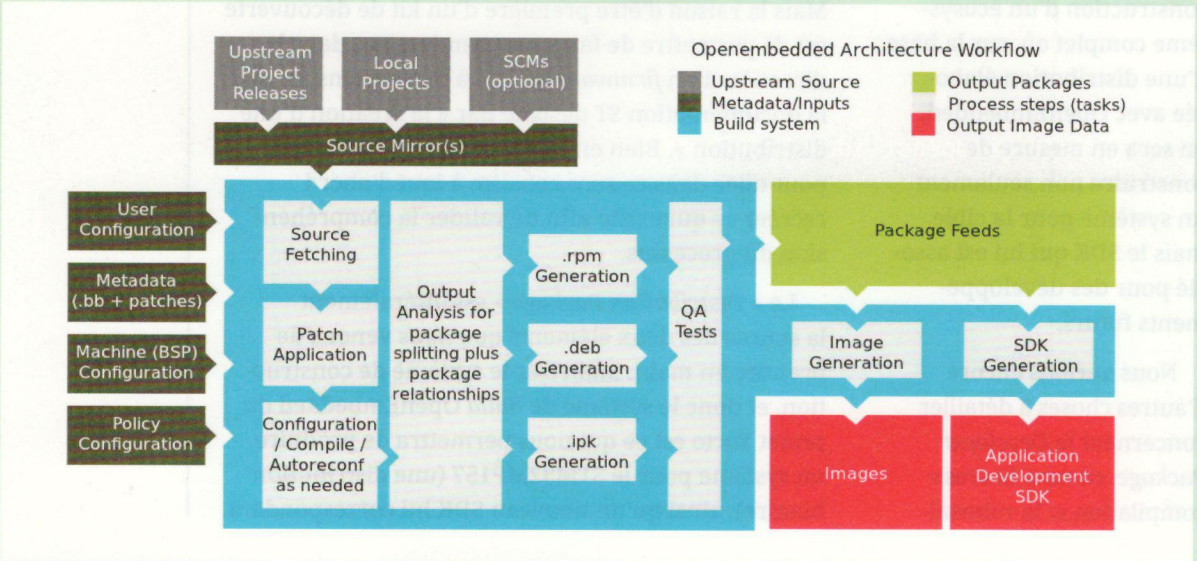
Le SDK permet de faire des choses intéressantes, qu'il s'agisse de développement original ou de modification de quelques composants choisis du système. Mais la raison d'être première d'un kit de découverte est de permettre de faire ses premiers pas dans la direction d'un *firmware* adapté à nos besoins, ce que la documentation ST désigne par « la création d'une distribution ». Bien entendu, la démarche logique pour aller dans ce sens consiste à tout d'abord recréer ce qui existe afin de valider la compréhension du processus.

Le « *Distribution Package* » est littéralement la source des deux éléments que nous venons de prendre en main. Maîtriser le système de construction, et donc le système de *build* OpenEmbedded du projet Yocto est ce qui nous permettra de produire un système pour le STM32MP157 (une distribution binaire), ainsi qu'un nouveau SDK lui correspondant.

Terminologie : Yocto, OpenEmbedded, Poky, etc.

Ces trois mots désignent des choses très différentes, mais cependant liées entre elles. Ils sont parfois utilisés indifféremment, tantôt pour désigner un projet, un système, une version ou un environnement de construction, mais ceci une erreur. Chacun de ces termes à une définition claire qu'il est important de connaître. Il en va de même pour d'autres désignations utilisées avec OpenEmbedded :

- **Yocto** est le projet responsable du développement d'OpenEmbedded et Poky. Ce n'est ni un système, ni une distribution, ni un outil, pas plus que Mozilla est le navigateur Firefox ou que Google est votre client mail. La confusion entre Yocto et OpenEmbedded provient du fait qu'OpenEmbedded existait en tant que projet avant que Yocto ne prenne forme et chapeaute l'ensemble.
- **OpenEmbedded** (ou « EO ») est le système de construction reposant sur des recettes, des fichiers de configuration et des outils comme `bitbake` pour exécuter les tâches de construction. OpenEmbedded n'est pas non plus un système ou une distribution. Vous n'exécutez pas OpenEmbedded sur la cible, pas plus que vous exécutez Yocto.
- **Poky** est l'implémentation de référence, ou distribution de référence, du projet Yocto. Une distribution comprend un système de construction et un ensemble de recettes, regroupées en *layers*, permettant de produire un système. Poky est un exemple fonctionnel et une base pour la construction d'une distribution personnalisée.
- **metadata** ou *meta* désigne n'importe quel fichier configurant le processus de construction par le jeu d'outils accompagnant OpenEmbedded. L'appellation de métadonnées découle du fait qu'il s'agit d'éléments satellites aux briques construisant réellement un système (les sources).
- Les **recipes** ou recettes sont des *meta* d'un type spécifique constituées d'instructions permettant de télécharger, de configurer et de compiler les codes sources des composants du système d'exploitation à créer.
- Les **layers** regroupent des *metadata* par lots ou ensembles, sous la forme de répertoires contenant des fichiers *metadata*.
- Un BSP ou *Board Support Package* est un type particulier de *layer* définissant la manière dont le système de construction doit agir pour construire un système à destination d'une carte spécifique. Un BSP est généralement créé et maintenu par une entité externe au projet Yocto, le plus souvent un fabricant de processeurs, SoC, ou *devkits*.
- Une **distribution** (ou « distro ») est un ensemble basé sur le système de construction OpenEmbedded, contenant un jeu de *layers*, où nous trouvons le BSP propre à la cible, les différents composants nécessaires au système et les éléments personnalisés du développeur. Il est possible, avec un même ensemble de *layers*, de fournir plusieurs distros, chacune d'elles en mesure de construire plusieurs images à flasher sur la cible.



Installer et utiliser OpenEmbedded pour construire un système pour la carte STM32MP157F-DK2 peut se faire de plusieurs façons. En effet, le constructeur met à disposition, sur GitHub, un ensemble de *layers* **[11]** dont un BSP, mais également la distribution OpenSTLinux que nous avons installée en début d'article. Le contenu du répertoire `images/stm32mp1/` du « *Starter Package* » constitue, en effet, le résultat de la construction de l'image `st-image-weston` fournie par la distribution `openstlinux-weston`. Reconstruire cette image et le SDK est une excellente façon de faire connaissance avec OpenEmbedded et les *layers* fournis par ST.

Pour cela, deux voies sont possibles :

- suivre les indications du constructeur en utilisant les éléments mis à disposition par ST ;
- ou partir sur la base de Poky et ajouter les *layers* nécessaires (ST ou autres).

Nous allons explorer ces deux approches, la première dans le but de reproduire l'existant et la seconde pour arriver à quelque chose de plus standard et facilement personnalisable. Notez que nous nous permettrons quelques écarts par rapport à la documentation officielle, ne serait-ce que pour rendre l'article un peu plus lisible (oui, je parle des noms de répertoires à nouveau).

1.3.1 Approche ST

Composer sa distribution passe par l'obtention de différents *layers*, le plus souvent via Git et depuis plusieurs dépôts de sources différents. Pour simplifier les choses dans ce genre de situations, un outil Python est généralement utilisé : *Repo* **[2]**. Initialement développé pour les sources AOSP d'Android, Repo est désormais utilisé par bon nombre de projets et c'est également le cas ici. Votre distribution GNU/Linux dispose certainement d'une version de l'outil disponible sous forme de paquet, mais une version à jour pourra facilement être téléchargée et installée n'importe où dans le système (il n'est pas même nécessaire de l'ajouter au `PATH`).

Récupérer les éléments de construction d'OpenSTLinux sera donc relativement simple :

```
$ cd quelque_part
$ mkdir Distribution-Package
$ cd Distribution-Package
$ mkdir openstlinux
$ cd openstlinux

$ ~/bin/repo init -u \
https://github.com/STMicroelectronics/oe-manifest.git \
-b refs/tags/openstlinux-5.10-dunfell-mp1-21-11-17

Downloading Repo source from https://gerrit.googlesource.com/git-repo
Downloading manifest from https://github.com/STMicroelectronics/oe-manifest.git
remote: Enumerating objects: 61, done.
remote: Counting objects: 100% (27/27), done.
remote: Compressing objects: 100% (18/18), done.
remote: Total 61 (delta 15), reused 20 (delta 9), pack-reused 34
Dépaquetage des objets: 100% (61/61), fait.

Your identity is: Denis Bodor <xxxxxxx@xxxxxx.xxx>
If you want to change this, please re-run 'repo init' with --config-name

repo has been initialized in /mnt/SSD2T/ST/Distribution-Package/openstlinux
```

Notez que Repo utilise Git et que votre configuration courante (`~/.gitconfig`) sera utilisée, expliquant l'éventuelle présence de votre nom et adresse mail. Ceci fait, nous pouvons synchroniser les fichiers avec :

```
$ ~/bin/repo sync
Fetching: 100% (8/8), done in 3m5.852s
Garbage collecting: 100% (8/8), done in 0.079s
Checking out: 100% (8/8), done in 1.450s
repo sync has finished successfully.
```

L'opération est relativement longue, mais vous vous retrouverez, à terme, avec un répertoire `layers` à l'emplacement courant, contenant :

- `openembedded-core` : la base ou noyau (*core*) d'OpenEmbedded qui est le strict minimum pour le système de construction et ne dispose de support que pour des machines émulées ;
- `meta-openembedded` : également appelé « meta-oe », ce layer contient des *metadata* supplémentaires, regroupées en *layers*, fournissant des paquets supplémentaires (Python, Perl, démons réseau, outils de base, etc.) ;
- `meta-qt5` : regroupe tous les éléments (hôte et cible) pour supporter des applications utilisant le *toolkit* Qt ;
- `meta-st` : un ensemble de *layers* provenant directement de ST et incluant, entre autres, le BSP pour les différentes cartes STM32MP157 (EV et DK).

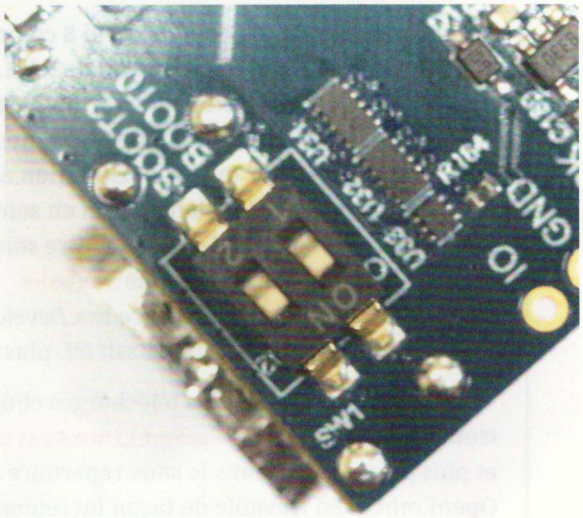
Chaque *layer* contient un répertoire `conf` lui-même contenant un fichier `layer.conf` décrivant sa priorité, sa compatibilité, ses dépendances, etc. On trouve également, dans chacun d'eux, un ensemble de recettes (*recipes*) contenant les instructions permettant de construire des paquets, de la récupération des sources à l'intégration dans le système, en passant par l'application de patchs, les configurations, la compilation, etc.

Normalement, avec OpenEmbedded, on commence par initialiser l'environnement de construction en sourçant le fichier `oe-init-build-env` (`openembedded-core/`) en lui passant en argument un répertoire de construction. Dans le cas présent, ST utilise un script « maison » permettant de définir la distribution et la machine pour configurer l'environnement. Il faut donc sourcer (avec `source` ou `.`) ce script depuis le répertoire d'où vous avez lancé `repo` :

```
$ DISTRO=openstlinux-weston \
  MACHINE=stm32mp1 \
  source layers/meta-st/scripts/envsetup.sh
```

Un message (interface `dialog`) s'affichera éventuellement, vous signalant que vous n'utilisez pas une distribution GNU/Linux officiellement supportée (pour l'heure Ubuntu 16.04, 18.04 ou 20.04) et vous pourrez simplement choisir « *IGNORE WARNING* » pour poursuivre. J'utilise une Debian 9.11 relativement ancienne et la procédure se déroule sans le moindre problème. Les *layers* ST incluent des éléments qui sont soumis à une licence spécifique, vous serez donc invité à lire les textes (EULA) et en accepter les termes. Ceci fait, vous vous retrouverez automatiquement placé dans `build-openstlinuxweston-stm32mp1/` qui constitue le répertoire de construction.

Le message à l'écran précise que deux images peuvent être construites, `st-image-weston` qui est précisément ce qui fonctionne actuellement sur votre *devkit* et `st-image-core`, une version minimaliste beaucoup plus légère. À votre disposition également, toutes les commandes de configuration et de construction d'OpenEmbedded : `bitbake`, `bitbake-layers`, `oe-pkgdata-util`, `devtool`, etc. Ceci vient du fait que vous avez modifié votre environnement en sourçant le script et que si vous fermez votre terminal, vous devrez obligatoirement le sourcer à nouveau avant de pouvoir utiliser OpenEmbedded.



Ces deux micro-interrupteurs se trouvant sous la carte (sous le connecteur HDMI) permettent de signifier au bootloader de passer le port USB-C en mode DFU. Il est alors possible de programmer le contenu de la microSD avec STM32CubeProgrammer ou un autre outil compatible avec ce protocole.

Notez que les notions de « DISTRO » et de « MACHINE » sont fondamentales avec OpenEmbedded. « DISTRO » désigne la distribution, mais dans la terminologue Yocto, il est plus exact de parler de politique de configuration, un ensemble de règles qui déterminent comment est composée la construction et le système obtenu. « MACHINE », pour sa part, fait référence aux spécifications et aux configurations provenant du BSP, fournissant une ou plusieurs cibles matérielles utilisables. La modularité fournie par les *layers* est ici la clé, il est parfaitement possible d'utiliser une même distribution avec plusieurs machines. Poky, par exemple, la distribution de référence pourra parfaitement être utilisée avec notre STM32MP157, une machine x86 émulée avec QEMU, ou n'importe quelle autre cible.

Pour construire une image, tout ce que vous avez à faire ici est d'utiliser la commande `bitbake st-image-weston` et d'attendre que le processus se termine, ce qui peut prendre un temps **considérable**. Pour optimiser la construction, vous pouvez éditer le fichier `conf/local.conf` pour ajouter quelques lignes :

```
# Nombre de tâches bitbake en parallèle
BB_NUMBER_THREADS = "4"

# Nombre de jobs parallèles pour make
PARALLEL_MAKE = "-j 4"

# Activer l'archiveur ST
# Permet de créer les paquets sources
# du Developer Package (noyau Linux, U-Boot, TF-A)
ST_ARCHIVER_ENABLE = "1"
```

Disposant d'un bi-Xeon E5520, j'ai 8 cœurs et 16 *threads* à ma disposition, le tout avec 24 Go de RAM. Vous devrez adapter ces valeurs à votre configuration. Même avec une configuration relativement musclée comme celle-ci, il faudra plusieurs heures pour que la construction arrive à terme. Il est donc recommandé de lancer cela en soirée et laisser la machine faire son travail pendant votre sommeil. Vous pourrez ensuite utiliser la commande `bitbake -c populate_sdk st-image-weston` pour produire le « *Developer Package* » sous la forme d'un gros script Bash (cf. plus haut).

L'ensemble des éléments téléchargés et/ou produits se trouveront dans `build-openstlinuxweston-stm32mp1` et plus précisément dans le sous-répertoire `tmp-glibc`. OpenEmbedded travaille de façon incrémentale, ce qui signifie qu'en lançant une seconde fois `bitbake st-image-weston`, seul le nécessaire sera traité. Cela veut également dire qu'en cas d'erreur (comme un site temporairement indisponible), vous pourrez relancer la commande et la construction reprendra là où elle s'est arrêtée.

Les éléments qui vous intéressent se trouveront dans le sous-répertoire `tmp-glibc/deploy/` du répertoire de construction. Là, vous trouverez :

- `deb/` : l'ensemble des paquets construits. OpenEmbedded peut utiliser plusieurs formats de paquet (configuré dans `local.conf`), RPM, DEB ou IPK. Dans le cas d'OpenSTLinux, ce sera DEB ;
- `images/` : vous retrouvez ici, peu ou prou, ce que vous avez téléchargé en récupérant le « *Starter Package* », les images des systèmes de fichiers, les `manifest` contenant la liste des paquets et les fichiers `.tsv` à utiliser avec `STM32_Programmer_CLI` (ou `create_sdcard_from_flashlayout.sh` qui se trouve également là) ;
- `licenses/` : l'ensemble des fichiers de licence pour tous les paquets (MIT, GPL, BSD ou autres) ;
- `sdk/` : le « *Developer Package* » correspondant à l'image construite ;
- `sources/` : les archives sources des différents éléments accompagnant le « *Developer Package* » (noyau, U-Boot, TF-A, OP-TEE, etc.).

À ce stade, vous obtenez littéralement exactement la même chose que ce que met à disposition ST, à la nuance près que vous pouvez désormais personnaliser l'ensemble à souhait.

Bien entendu, l'image `st-image-weston` est avant tout une démonstration incluant bien trop d'éléments pour servir de base pour un projet. Une alternative possible est `st-image-core`, plus basique, mais reposant toujours sur la distribution OpenSTLinux. Une autre approche possible, pour disposer d'une fondation plus standard, consiste à construire sa distribution sur la base de Poky.

1.3.2 Approche générique

Partir d'une distribution déjà très riche est souvent un problème puisqu'on se retrouve à « déconstruire » ce qui a été fait, tout en prenant soin de corriger les problèmes de dépendance au fur et à mesure. C'est un peu comme jouer à Jenga, on enlève des briques en tentant de faire en sorte que rien ne s'écroule. Il est bien plus simple de partir sur une base la plus minimaliste possible et d'ajouter le nécessaire petit à petit. Nous allons donc ici faire précisément cela, en partant de Poky, la distribution de référence.

À ce stade, vous avez compris qu'une distribution est avant tout une accumulation cohérente de *layers*, nous commençons donc par récupérer le BSP fourni par ST via GitHub :

```
$ cd quelque_part
$ mkdir -p MP1perso/layers
$ cd MP1perso/layers

$ git clone https://github.com/STMicroelectronics/meta-st-stm32mp.git
$ cd meta-st-stm32mp/
$ git branch
* dunfell
$ cd ..
```

Notez l'utilisation de `git branch` afin de savoir exactement sur quelle *release* nous travaillons, car celle-ci devra être identique pour tous les *layers*. *Dunfell* est le nom de code de la *release* 3.1 du projet Yocto datant de novembre 2021 (3.1.12) et étant une LTS (*Long Term Support*).

À cette date, la dernière stable est *Hardknott* (3.3) et la future LTS est *Kirkstone* (3.5). La lecture du `README.md` du BSP nous indique une dépendance vers `openembedded-core` ou `poky`, mais également vers `meta-python` et `meta-oe`. Nous récupérons donc ces *layers* dans la foulée :

```
$ git clone git://git.yoctoproject.org/poky.git
$ cd poky
$ git branch
* master
$ git branch -r | grep dunfell
  origin/dunfell
  origin/dunfell-next
$ git checkout dunfell
La branche 'dunfell' est paramétrée pour suivre la
branche distante 'dunfell' depuis 'origin'.
Basculement sur la nouvelle branche 'dunfell'

$ cd ..
$ git clone git://github.com/openembedded/meta-openembedded.git
$ cd meta-openembedded
$ git checkout dunfell
La branche 'dunfell' est paramétrée pour suivre la
branche distante 'dunfell' depuis 'origin'.
Basculement sur la nouvelle branche 'dunfell'
$ cd ../..
```

Nous disposons de tous les *layers* nécessaires, mais ceux-ci ne sont pas encore intégrés dans notre *build*. Mais avant toute chose, nous devons initialiser l'environnement, cette fois en utilisant le script officiel fourni par Poky :

```
$ . layers/poky/oe-init-build-env
```

Le répertoire de construction par défaut est `build` et nous y sommes automatiquement placés. Là, nous pouvons utiliser les commandes d'OpenEmbedded pour, tout d'abord, lister les *layers* intégrés, puis ajouter ceux dont nous avons besoin :

```
$ bitbake-layers show-layers
NOTE: Starting bitbake server...
layer          path                                                  priority
==========================================================================
meta           /mnt/SSD2T/STM32MP1/MP1perso/layers/poky/meta             5
meta-poky      /mnt/SSD2T/STM32MP1/MP1perso/layers/poky/meta-poky        5
meta-yocto-bsp /mnt/SSD2T/STM32MP1/MP1perso/layers/poky/meta-yocto-bsp   5

$ bitbake-layers add-layer ../layers/meta-openembedded/meta-oe
NOTE: Starting bitbake server...

$ bitbake-layers add-layer ../layers/meta-openembedded/meta-python
NOTE: Starting bitbake server...

$ bitbake-layers add-layer ../layers/meta-st-stm32mp
NOTE: Starting bitbake server...
```

Une nouvelle occurrence de la commande `bitbake-layers show-layers` nous montrera que tout a été intégré correctement. Notez que l'ordre d'intégration est important en raison des dépendances et si vous tentez de commencer par `meta-st-stm32mp`, vous serez rappelé à l'ordre. Remarquez également la colonne `priority` dans la sortie. Ceci peut s'avérer excessivement important, car si plusieurs recettes identiques sont fournies par des *layers*, la priorité de ces derniers sera utilisée pour déterminer laquelle devra être utilisée. Ainsi, si vous souhaitez remplacer une recette d'un *layer* par une version personnalisée, il vous suffit de créer un nouveau *layer* la contenant et de lui attribuer une valeur de priorité plus importante (voir plus loin dans l'article).

Nous sommes presque prêts pour lancer la construction, mais devons tout d'abord nous pencher sur la configuration en éditant le fichier `conf/local.conf`. Le premier élément à ajuster est `MACHINE` qui est actuellement réglé sur l'une des cibles prises en charge nativement par Poky, `qemux86-64`. Nous pouvons consulter le contenu du répertoire `conf/machine/` du BSP pour avoir cette information. Le fichier `stm32mp1.conf` indique clairement :

```
#@TYPE: Machine
#@NAME: stm32mp1
#@DESCRIPTION: Configuration for all STM32MP1 boards (EV, DK, ...)
```

Nous ajustons donc `MACHINE = "stm32mp1"` dans `local.conf` et en profitons pour ajouter, en fin de fichier, les paramètres que nous avons utilisés précédemment :

```
ACCEPT_EULA_stm32mp1 = "1"
BB_NUMBER_THREADS = "4"
PARALLEL_MAKE = "-j 4"
ST_ARCHIVER_ENABLE = "1"
```

`ACCEPT_EULA_stm32mp1` est mentionné dans le `README.md` du BSP et nous permet de signifier l'acceptation des conditions d'utilisation de ST et de ses partenaires. D'autres éléments peuvent être ajustés dans le fichier et les commentaires intégrés détaillent clairement leur signification. `DISTRO` est réglé sur `poky`, ce qui nous convient parfaitement, mais `PACKAGE_CLASSES` peut être changé. Cet élément détermine le format de paquet utilisé et est, par défaut, réglé sur `package_rpm`. Étant utilisateur de Debian de très longue date, je préfère `package_deb`, mais c'est là un choix tout personnel.

Le fichier `local.conf` correspondant à vos attentes et préférences, vous pouvez alors passer à la construction avec :

```
$ bitbake core-image-minimal
[...]
```

Cette opération durera un temps conséquent, mais bien moins long que celle construisant l'image officielle. Vous pouvez directement poursuivre avec la construction du SDK via la commande `bitbake -c populate_sdk core-image-minimal`. On retrouve alors dans `tmp/deploy/` une arborescence assez similaire à celle de l'approche précédente et on pourra flasher la carte directement avec :

```
$ STM32_Programmer_CLI -c port=usb1 -w \
flashlayout_core-image-minimal/trusted/\
FlashLayout_sdcard_stm32mp157f-dk2-trusted.tsv

[...]
Time elapsed during download operation: 00:00:38.014

RUNNING Program ...
  PartID:       :0x13
Start operation done successfully at partition 0x13
Flashing service completed successfully
```

Bien entendu, le système une fois démarré sur le *devkit* sera bien différent de celui de la distribution OpenSTLinux. Nous avons là une configuration minimale sans SSH, sans Bash, sans systemd (est-ce vraiment un mal ?), sans mDNS (avahi), sans Wayland, etc. Ce sera à vous d'ajouter des *layers*, ou des recettes dans un *layer* personnalisé, pour composer votre système en vous inspirant, éventuellement, de ceux mis à disposition par ST pour leur OpenSTLinux **[12]**.

1.3.3 Petit supplément : ajuster selon vos besoins

Plus haut dans l'article, nous avons soulevé un petit problème concernant le script `create_sdcard_from_flashlayout.sh` et le fait que la commande `sgdisk` devrait être référencée avec son chemin complet pour une utilisation par un utilisateur standard. Nous pouvons corriger ce problème et, par la même occasion, voir comment ajouter une recette et un *layer* personnalisé à notre projet.

Durant la phase de programmation de la microSD, le déroulement des opérations est visible directement sur l'écran LCD, mais également via le port série accessible via l'interface ST-LINK/V2-1. Si l'une des étapes d'écriture échoue, nos expérimentations montrent que la cause probable sera certainement un câble USB-C de médiocre qualité.

Nous nous assurons, premièrement, d'avoir initialisé l'environnement de construction avec `oe-init-build-env`, puis nous nous plaçons dans le répertoire racine (et non `build`) pour utiliser la commande :

```
$ bitbake-layers create-layer layers/meta-denis
NOTE: Starting bitbake server...
Add your new layer with 'bitbake-layers add-layer layers/meta-denis'
```

Un sous-répertoire `meta-denis` est automatiquement créé et peuplé dans `layers/`. Nous y trouvons une licence simple de type MIT/BSD, un `README`, une recette exemple et un répertoire `conf` contenant le fichier de configuration du layer, `layer.conf` :

```
# We have a conf and classes directory, add to BBPATH
BBPATH .= ":${LAYERDIR}"

# We have recipes-* directories, add to BBFILES
BBFILES += "${LAYERDIR}/recipes-*/*/*.bb \
            ${LAYERDIR}/recipes-*/*/*.bbappend"

BBFILE_COLLECTIONS += "meta-denis"
BBFILE_PATTERN_meta-denis = "^${LAYERDIR}/"
BBFILE_PRIORITY_meta-denis = "7"

LAYERDEPENDS_meta-denis = "core stm-st-stm32mp"
LAYERSERIES_COMPAT_meta-denis = "dunfell"
```

Nous pouvons ajuster ici les dépendances et le niveau de priorité. Étant donné que notre *layer* a pour but de remplacer l'une des recettes fournies par `meta-st-stm32mp`, et `sdcard-raw-tools` en particulier, nous pouvons rendre ce nouveau *layer* dépendant de `stm-st-stm32mp`, mais surtout augmenter sa priorité à `7`, puisque le fichier `meta-st-stm32mp/conf/layer.conf` indique `6`. Les deux lignes en question deviennent donc :

```
BBFILE_PRIORITY_meta-denis = "7"
LAYERDEPENDS_meta-denis = "core stm-st-stm32mp"
```

Il ne nous reste plus ensuite qu'à copier `meta-st-stm32mp/recipes-devtools/sdcard-raw-tools` dans notre *layer*. Notez que le chemin complet est important. Vous ne pouvez pas simplement copier `sdcard-raw-tools`, ce répertoire doit être dans `recipes-devtools`. Enfin, nous éditons `create_sdcard_from_flashlayout.sh` pour procéder à nos corrections.

Notre layer est maintenant prêt pour être intégré à notre *build*. Nous retournons donc dans le répertoire de construction et utilisons :

```
$ bitbake-layers add-layer ../layers/meta-denis
NOTE: Starting bitbake server...
```

Nous pouvons vérifier la bonne prise en compte avec un `bitbake-layers show-layers` ou en listant le contenu de `conf/bblayers.conf`. Comme tout est correct, nous pouvons lancer une nouvelle construction :

```
$ bitbake st-image-core
[...]
Build Configuration:
BB_VERSION           = "1.46.0"
BUILD_SYS            = "x86_64-linux"
NATIVELSBSTRING      = "universal"
TARGET_SYS           = "arm-poky-linux-gnueabi"
MACHINE              = "stm32mp1"
DISTRO               = "poky"
DISTRO_VERSION       = "3.1.13"
[...]
Initialising tasks: 100% |###############| Time: 0:00:03
Sstate summary: Wanted 13 Found 0 Missed
13 Current 1183 (0% match, 98% complete)
NOTE: Executing Tasks
NOTE: Tasks Summary: Attempted 3185 tasks of which 3159
didn't need to be rerun and all succeeded.
```

Nos modifications sont minimales, et même si notre configuration a changé, OpenEmbedded a déjà fait la très grande majorité du travail. Seules 13 tâches doivent donc être exécutées pour compléter l'objectif de construction en prenant en compte les dépendances. Et effectivement, nous pouvons constater que `tmp/deploy/images/stm32mp1/scripts/` contient notre version modifiée du script.

Bien entendu, notre nouveau layer peut inclure des recettes pour faire bien plus que cela. Nous pouvons y intégrer de quoi produire d'autres images, ajouter nos propres outils et applications, ou tout simplement « surcharger » la configuration pour intégrer davantage de services, d'utilitaires, de bibliothèques, etc.

2. CONCLUSION TRÈS TEMPORAIRE

J'aime beaucoup le STM32MP157F et son *Discovery kit*, car il permet pleinement de faire connaissance avec un système de *build* massivement utilisé dans l'embarqué. Mieux encore, le STM32MP157F intègre énormément de fonctionnalités intéressantes comme le *boot* sécurisé ou l'utilisation de concert du Cortex-A7 et du Cortex-M4. Deux points que nous creuserons probablement dans de prochains articles.

Bien entendu, tout n'est pas tout rose et, sans vouloir me montrer trop critique, on constate une nette différence de « rigueur » ou de perfectionnisme technique entre des projets comme OpenEmbedded et, par exemple, STM32CubeProgrammer. Cela fonctionne, bien sûr, mais il est relativement difficile de ne pas y voir un aspect un peu « brut de décoffrage », en particulier en contraste avec des choses soignées comme OpenEmbedded.

Cependant, tout ceci n'enlève rien à l'aspect le plus important lorsqu'on se penche sur une telle plateforme : la documentation. Le wiki de ST, une fois les points de terminologie assimilés et la philosophie générale comprise, est une mine d'informations sans fin. Et je ne parle même pas des *datasheets*, les manuels de référence et autres *application notes* qui contiennent, par définition, l'information dont on a besoin. Nous sommes loin des documents avares et parcellaires comme ceux de Broadcom ou totalement incompréhensibles (voir inexistant en anglais) de certains fabricants chinois.

Une autre bonne raison, mais tout à fait non technique et toute personnelle, qui me fait apprécier cette plateforme (ainsi que le reste de la famille STM32) est tout simplement le fait que STMicroelectronics soit non seulement une multinationale européenne (franco-italienne avec un siège en Suisse), mais qu'elle fabrique elle-même ses puces. Il a d'ailleurs été annoncé dernièrement un objectif visant à doubler les capacités de production en Europe d'ici 2025. Il est réellement rassurant de voir qu'on peut encore, en France et en Europe, innover et produire dans le domaine des semi-conducteurs... **DB**

RÉFÉRENCES

[1] **https://github.com/bootlin/buildroot-external-st**

[2] **https://gerrit.googlesource.com/git-repo/**

[3] **https://www.st.com/en/development-tools/stm32cubeprog.html**

[4] **https://www.st.com/en/embedded-software/stm32mp1starter.html**

[5] **https://wiki.st.com/stm32mpu/wiki/Example_of_directory_structure_for_Packages**

[6] **https://wiki.st.com/stm32mpu/wiki/STM32MP15_Discovery_kits_-_Starter_Package#Image_download**

[7] **https://www.st.com/en/embedded-software/stm32mp1dev.html**

[8] **https://wiki.st.com/stm32mpu/wiki/STM32MP1_Developer_Package**

[9] **https://github.com/STMicroelectronics/meta-st-stm32mp/blob/dunfell/recipes-bsp/u-boot/u-boot-stm32mp/README.HOW_TO.txt**

[10] **https://github.com/STMicroelectronics/meta-st-stm32mp/blob/dunfell/recipes-kernel/linux/linux-stm32mp/README.HOW_TO.txt**

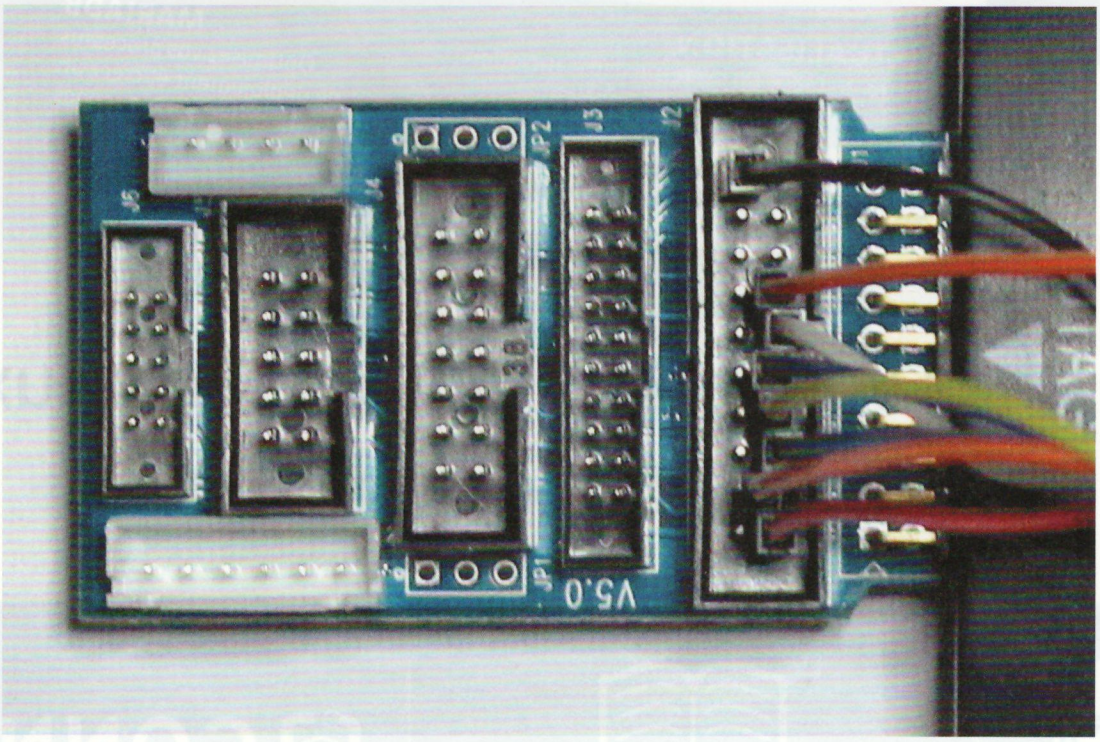
[11] **https://github.com/orgs/STMicroelectronics/repositories?q=meta+stm32mp**

[12] **https://github.com/STMicroelectronics/meta-st-openstlinux**

DÉVELOPPEMENT BAREMETAL SUR PI 3 : LES PERFORMANCES

Denis BODOR

Dans l'article précédent, nous avons été capables d'afficher à l'écran une image, traitée par l'application Gimp et intégrée à notre code. Mais si vous avez reproduit ces essais, en particulier avec une image d'une dimension non négligeable, vous avez sans le moindre doute remarqué un petit problème de performances. Si nous voulons utiliser le framebuffer de façon acceptable, nous devons régler ce problème.



La première chose à faire pour améliorer la situation consiste à mesurer l'étendue du problème. Or, à ce stade, nos fonctions utilitaires sont très limitées. Nous n'avons, en particulier, absolument rien qui nous permette de mesurer une durée ou même de temporiser le code avec un délai quelconque. Certes, la configuration de l'UART (miniUART) inclut une petite boucle de `NOP` (*No OPeration*), mais ceci est bien insuffisant.

1. MESURONS LE PROBLÈME

Le SoC Broadcom contient de nombreux périphériques et parmi eux se trouvent plusieurs *timers*, constitués de compteurs cadencés par un signal d'horloge. L'un d'entre eux, le *System Timer* est précisément conçu pour nos besoins puisqu'il est cadencé à 1 MHz (avec donc une période d'une microseconde). La *datasheet* du BCM2835 **[1]** nous apprend, en page 172, que ce *timer* utilise un compteur (*free running counter*) de 64 bits dont la valeur est accessible via deux registres. Nous avons donc sous la main un compteur s'incrémentant à 1 MHz qu'il nous suffit de lire pour obtenir quelque chose de similaire à la fonction `micros()` d'Arduino.

Commençons par créer un nouveau *header* (`delay.h`) contenant les macros désignant les registres en question ainsi que le prototype de notre future fonction :

```
#pragma once

#define STC_LOW (*(volatile unsigned *)(IO_BASE + 0x3004))
#define STC_HIGH (*(volatile unsigned *)(IO_BASE + 0x3008))
uint64_t micros();
```

Notre code source, stocké dans `delay.c` et ajouté au `Makefile`, contiendra peu de choses :

```
#include <stdint.h>
#include "mmio.h"
#include "delay.h"

uint64_t micros()
{
  uint32_t high, low;
  high = STC_HIGH;
  low  = STC_LOW;
  if(high != STC_HIGH)
  {
    high = STC_HIGH;
    low  = STC_LOW;
  }
  return ((uint64_t)high << 32) | low;
}
```

Nous avons là une petite astuce nous permettant d'éviter un problème potentiel. La lecture du compteur se fait en deux temps, via le registre de 32 bits haut (`STC_HIGH`) contenant les bits de poids les plus forts et le registre 32 bits bas (`STC_LOW`). Ces duos d'opérations ne sont pas atomiques, ce qui signifie qu'au moment où nous lisons l'un des deux registres, l'autre peut parfaitement avoir changé. Ceci n'est généralement pas réellement un problème, sauf si au moment de la lecture de `STC_HIGH`, `STC_LOW` contient `0xFFFFFFFF` et que le prochain front montant de l'horloge est sur le point d'arrivée. Dans ce cas, par exemple, si nous venons de lire `0x00000001` dans `STC_HIGH`, et qu'au moment de lire `STC_LOW`, celui-ci vient juste de passer de `0xFFFFFFFF` à `0x00000000`, nous obtenons un compteur sur 64 bits à `0x0000000100000000` et non `0x0000000200000000`, ce qui représente une différence de plus de 4 milliards de microsecondes (soit plus d'une heure). Pour éviter le problème, nous lisons donc le contenu de `STC_HIGH` une seconde fois, après la lecture de `STC_LOW`. En cas de changement, nous lisons à nouveau les deux registres. Les deux valeurs 32 bits sont ensuite assemblées sous la forme d'un `uint64_t` qui est retourné par la fonction.

Nous pouvons alors utiliser `micros()` dans notre `hello.c` ainsi :

```
unsigned char *ptr=fb.fbp;
uint64_t avant = micros();

int x,y;
char *imgdata=header_data;
char pixel[4];

// centrage
  ptr += (fb.height-height)/2*fb.pitch + (fb.width-width)*2;

// RGB ou BGR ?
if(fb.isrgb) {
  // RGB
  for(y=0;y<height;y++) {
    for(x=0;x<width;x++) {
      HEADER_PIXEL(imgdata, pixel);
      *((unsigned int*)ptr) = *((unsigned int *)&pixel);
      ptr+=4;
    }
    ptr+=fb.pitch-width*4;
  }
} else {
  // BGR
  for(y=0;y<height;y++) {
    for(x=0;x<width;x++) {
      HEADER_PIXEL(imgdata, pixel);
      *((unsigned int*)ptr) =
         (unsigned int)(pixel[0]<<16|pixel[1]<<8|pixel[2]);
```

```
            ptr+=4;
        }
        ptr+=fb.pitch-width*4;
    }
}
    puts("Time: ");
    printdec((micros()-avant));
    puts("µs\r\n");
    printdec(1000000/(micros()-avant));
    puts(" FPS\r\n");
```

Notez que notre double boucle a changé par rapport à l'article précédent. Il s'agit d'une modeste optimisation puisque le test de `fb.isrgb` est maintenant fait une unique fois et non plus à chaque pixel de l'image affichée. Il s'agit ici plus d'une bonne pratique que d'une réelle quête d'optimisation. En particulier au regard de ce que nous allons ensuite faire...

Le résultat de ce test est tout bonnement affligeant, puisque le terminal série nous affiche quelque chose comme :

```
Bonjour Hackable !
Framebuffer address: 0x3E402000
Serial number = 0000000015D0C02E
ARM memory start at 0x00000000 - 948 Mo
Videocore memory start at 0x3B400000 - 76 Mo
SoC temperature: 36476
ARM current clock: 600000000 Hz
Exception level: 2
Framebuffer width: 1920
Framebuffer height: 1080
Framebuffer pitch: 7680
Framebuffer pixel format: BGR
Time: 876996µs
1 FPS
```

Presque 900 ms pour afficher une image de 376×501 pixels est tout bonnement pathétique. Mais comment font donc Linux et X11/Wayland pour rafraîchir l'écran avec un *framebuffer* aussi lent. La réponse est simple : leur code est bien meilleur (mais pas seulement), mais nous pouvons arranger cela...

Notez au passage que l'utilisation du terme « FPS » (*Frames Per Second*) ici n'est pas absolument exacte puisque nous prenons en considération le temps d'affichage de l'image et non du rafraîchissement complet de l'écran de 1920×1080 pixels. Mais les deux étant proportionnels, ceci nous donne une bonne estimation de la vitesse du *framebuffer* (à condition de ne pas changer l'image entre les tests).

Les Raspberry modèles 3 B+ sont équipées du SoC BCM2837B0 et non du BCM2837. Les deux sont presque totalement identiques, même au niveau de leur architecture et des cœurs ARM Cortex A53. Ils ne se différencient que par leur fréquence maximum de fonctionnement, poussée de 1,2 GHz à 1,4 GHz, rendant le modèle B+ quelque 17 % plus rapide.

2. 600 MHZ ?

Peut-être avez-vous remarqué un petit détail intéressant dans la sortie écran/série précédente. Notre `kernelmain()`, en plus de configurer le *framebuffer* et la sortie HDMI, utilise une poignée de tags nous permettant d'obtenir des informations sur la configuration. Numéro de série (`MBOX_TAG_GETSERIAL`), taille de la mémoire allouée au CPU (`MBOX_TAG_GETARMMEM`) et GPU (`MBOX_TAG_GETVCOREMEM`), température du SoC (`MBOX_TAG_GETCORETEMP`) et... fréquence utilisée par le(s) CPU (`MBOX_TAG_GETCLK` et ID périphérique 3). Et c'est précisément là que le bât blesse :

```
ARM current clock: 600000000 Hz
```

Nous avons là une Raspberry Pi 3B avec un SoC BCM2837 capable de fonctionner à 1,2 GHz, mais le *firwmare* du GPU a configuré la fréquence à la moitié de sa valeur maximum. Étant à présent familiarisés avec la *mailbox*, quelques lignes de code et un simple coup d'œil à la documentation **[2]** nous permettent de trouver le tag `0x00038001`, alias « *Set clock state* », qui nous permettrait de définir une nouvelle fréquence pour l'horloge avec l'ID `0x000000003` (*ARM*)... ou pas.

En effet, pour l'instant nous utilisons une facilité nous permettant de communiquer des informations depuis la Pi : le miniUART, qui n'est pas une implémentation complète compatible 16650 (voir page 10 de la *datasheet*). Pire encore, ce miniUART est lié à l'horloge du GPU (ou VPU dans la documentation) et si nous changeons la fréquence du CPU, celle du GPU va automatiquement être ajustée et notre pseudo-UART ne fonctionnera plus correctement (ceci a malheureusement été vérifié lors de la rédaction du présent article).

Pour nous offrir une plus grande liberté de configuration, nous devons donc nous défaire du miniUART et configurer, à sa place, l'UART « standard » décrit dans la *datasheet* en page 175 et nommé PL011 UART, utilisant une source d'horloge dédiée, indépendante et configurable. Avant de nous pencher sur la reconfiguration de l'UART, facilitons-nous la tâche en créant quelques fonctions utilitaires concernant les horloges. Comme d'habitude, commençons par le *header* `clock.h` :

```
#pragma once

#define CLK_EMMC        0x00000001
#define CLK_UART        0x00000002
#define CLK_ARM         0x00000003
#define CLK_CORE        0x00000004
#define CLK_V3D         0x00000005
#define CLK_H264        0x00000006
#define CLK_ISP         0x00000007
#define CLK_SDRAM       0x00000008
#define CLK_PIXEL       0x00000009
#define CLK_PWM         0x0000000a
#define CLK_HEVC        0x0000000b
#define CLK_EMMC2       0x0000000c
#define CLK_M2MC        0x0000000d
#define CLK_PIXEL_BVB   0x0000000e

unsigned int getclock(uint32_t clkid, uint8_t max, volatile unsigned int *mbox);
void printclock(volatile unsigned int *mbox);
void setclock(uint32_t clkid, uint32_t freq, volatile unsigned int *mbox);
void setmaxclock(uint32_t clkid, volatile unsigned int *mbox);
```

Rien de bien exceptionnel ici, nous définissons quelques macros pour désigner les ID d'horloges et les prototypes de nos fonctions permettant : d'obtenir une fréquence courante, d'afficher toutes les fréquences de toutes les horloges (actuelle et maximum), de définir une nouvelle fréquence et de configurer une horloge avec sa valeur maximum.

L'implémentation de ces fonctions est faite dans `clock.c` que nous ajoutons à notre `Makefile` :

```
#include <stdint.h>
#include "uart.h"
#include "mbox.h"
#include "util.h"
#include "clock.h"

unsigned int getclock(uint32_t clkid, uint8_t max, volatile unsigned int *mbox)
{
  mbox[0]=8*4;
  mbox[1]=MBOX_REQUEST;
  mbox[2]= max>0 ? MBOX_TAG_GETMAXCLK : MBOX_TAG_GETCLK;
  mbox[3]=8;
  mbox[4]=0;
  mbox[5]=clkid;
  mbox[6]=0;
```

```
  mbox[7]=MBOX_TAG_LAST;
  if (mbox_call(MBOX_CH_PROP, mbox)) {
    return(mbox[6]);
  } else {
    puts("Error getting clock!\r\n");
    return(0);
  }
}

void setclock(uint32_t clkid, uint32_t freq, volatile unsigned int *mbox)
{
  mbox[0]=9*4;
  mbox[1]=MBOX_REQUEST;
  mbox[2]=MBOX_TAG_SETCLK;
  mbox[3]=12;
  mbox[4]=0;
  mbox[5]=clkid;
  mbox[6]=freq;
  mbox[7]=0;
  mbox[8]=MBOX_TAG_LAST;
  // ATTENTION : erreur silencieuse
  mbox_call(MBOX_CH_PROP, mbox);
}

void printclock(volatile unsigned int *mbox)
{
  static const char *clocktxts[] = {
    "reserved", "EMMC", "UART", "ARM", "CORE",
    "V3D", "H264", "ISP", "SDRAM", "PIXEL", "PWM",
    "HEVC", "EMMC2", "M2MC", "PIXEL_BVB"
  };

  puts("Clocks:\r\n");
  for(int i=1; i<0xf; i++) {
    puts("  ");
    puts(clocktxts[i]);
    puts(": ");
    printdec(getclock(i, 0, mbox));
    puts(" / ");
    printdec(getclock(i, 1, mbox));
    puts(" Hz\r\n");
  }
}
```

```
void setmaxclock(uint32_t clkid, volatile unsigned int *mbox)
{
  uint32_t maxfreq = getclock(clkid, 1, mbox);
  setclock(clkid, maxfreq, mbox);
}
```

Si vous avez lu l'article sur la *mailbox* et le *framebuffer* dans le numéro 40 **[3]**, ainsi que la page « *Mailbox property interface* » du wiki Raspberry Pi **[2]**, tout ceci n'a rien de très nouveau.

Nous pouvons maintenant revoir nos sources, à commencer par `uart.h` avec de nouveaux registres :

```
#pragma once

#include "mmio.h"

/* PL011 UART */
#define UART0_DR        (*(volatile unsigned *)(IO_BASE + 0x00201000))
#define UART0_FR        (*(volatile unsigned *)(IO_BASE + 0x00201018))
#define UART0_IBRD      (*(volatile unsigned *)(IO_BASE + 0x00201024))
#define UART0_FBRD      (*(volatile unsigned *)(IO_BASE + 0x00201028))
#define UART0_LCRH      (*(volatile unsigned *)(IO_BASE + 0x0020102C))
#define UART0_CR        (*(volatile unsigned *)(IO_BASE + 0x00201030))
#define UART0_IMSC      (*(volatile unsigned *)(IO_BASE + 0x00201038))
#define UART0_ICR       (*(volatile unsigned *)(IO_BASE + 0x00201044))

void init_uart();
void putc(char c);
```

Nos prototypes de fonctions ne changent pas, mais leur implémentation, oui. L'approche est la même qu'avec le miniUART, si ce n'est dans une variation des registres utilisés ainsi que sur les bits concernés (pour la fonction alternative des GPIO). Voici notre nouveau fichier `uart.c` :

```
#include <stdint.h>
#include "uart.h"
#include "gpio.h"
#include "mbox.h"
#include "clock.h"

void init_uart()
{
  // Attente du périphérique
  while(UART0_FR & (1<<3)) { ; }
```

```
  // arrêt UART0
  UART0_CR = 0;

  // GPIO 14 & 15 en alt0
  GPFSEL1 &= ~((7<<12)|(7<<15));
  GPFSEL1 |= (4<<12)|(4<<15);
  GPPUD = 0;
  for(uint8_t i = 0; i<150; i++)
    asm volatile ("nop");
  GPPUDCLK0 = (1<<14)|(1<< 5);
  for(uint8_t i = 0; i<150; i++)
    asm volatile ("nop");
  GPPUDCLK0 = 0;

  // Clear interruption
  UART0_ICR = 0x7FF;

  // Calcul diviseur pour 115200 bps à 3 MHz
  UART0_IBRD = 1;       // part entière
  UART0_FBRD = 40;      // part fractionnaire
  UART0_LCRH = 0b11<<5; // format 8n1
  UART0_CR = 0x301;     // active Tx, Rx & FIFO
}

void putc(char c)
{
  // FIFO plein ?
  while(UART0_FR & (1<<5)) { ; }
  UART0_DR = c;
}
```

La partie nouvelle réside dans l'utilisation des registres `UART0_IBRD` et `UART0_FBRD` configurant la valeur du diviseur nous permettant, à partir d'une fréquence d'horloge donnée pour l'UART, d'obtenir la vitesse souhaitée. Le calcul est le suivant : diviseur = fréquence / (16 x bps). Ceci est décrit sommairement dans la *datasheet* en page 183 du BCM2835, mais également dans les références techniques ARM **[4]** puisque le périphérique intégré au SoC est une implémentation classique de l'UART ARM PL011 (*PrimeCell UART PL011*).

Sur la base de ces informations et des exemples ARM, nous pouvons calculer *diviseur* = 3000000 / (115200 x 16) = 1,627. `UART0_IBRD` sera donc égal à `1` et pour `UART0_FBRD`, nous calculons (0,627 x 64) + 0,5 = 40,628 et donc `40`. Notez que la documentation ARM fournit des valeurs typiques avec un diviseur entier pour une fréquence de 7,3728 MHz ou encore différentes valeurs de division pour une fréquence de 4 MHz avec un taux d'erreur dépendant de la vitesse en bits par seconde. Dans notre cas, nous avons un diviseur effectivement à ((40 - 0,5) / 64) + 1 = 1,6171875, et donc une vitesse effective à 3 MHz de 3000000 / 1,6171875 / 16 = 115942 bps, soit une erreur de (115942 / 115200 x 100) - 100 = 0,644 %, ce qui est acceptable, sans être idéal.

Après ces changements, nous pouvons revoir le début de notre `hello.c` qui ressemblera maintenant à :

```
int kernelmain(void)
{
  struct fb_t fb;

  setmaxclock(CLK_ARM, mbox);
  setclock(CLK_UART, 3000000, mbox);
  init_uart(mbox);

  puts("\r\n\r\n\r\nBonjour Hackable !\r\n");
[...]
```

Après compilation et exécution sur la Pi, nous voyons apparaître l'effet obtenu :

```
[...]
Clocks:
  EMMC: 200000000 / 200000000 Hz
  UART: 3000000 / 1000000000 Hz
  ARM: 1200002000 / 1200000000 Hz
  CORE: 400000000 / 400000000 Hz
[...]
Framebuffer pitch: 7680
Framebuffer pixel format: BGR
Time: 643144µs
1 FPS
```

Le CPU est effectivement à 1,2 GHz et le GPU à 400 MHz et nous sommes magnifiquement passés de 876996 µs à 643144 µs, ce qui représente un gain de presque 240 ms, certes, mais ne nous permet toujours pas de dépasser une seule image par seconde. Nous sommes donc toujours dans le domaine du pathétiquement poussif...

3. MERCI LE COMPILATEUR !

Et si je vous disais que changer un unique caractère dans l'un de nos fichiers pouvait diviser par 30 le temps d'affichage de notre image ? Me croiriez-vous ? Non, et pourtant :

```
[...]
Framebuffer pitch: 7680
Framebuffer pixel format: BGR
Time: 21481µs
44 FPS
```

Souvenez-vous, dès le premier article (Hackable 40 **[5]**) de cette petite série, nous avons posé les bases en créant quelques fichiers, dont notre `Makefile`. Or, celui-ci contient une ligne spécifiant les options à utiliser par le compilateur :

```
CFLAGS=-Wall -O0 -g -ffreestanding
```

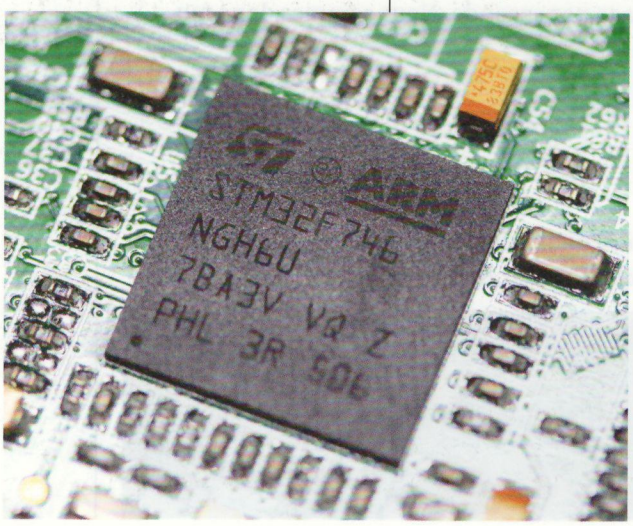
Correspondant à :

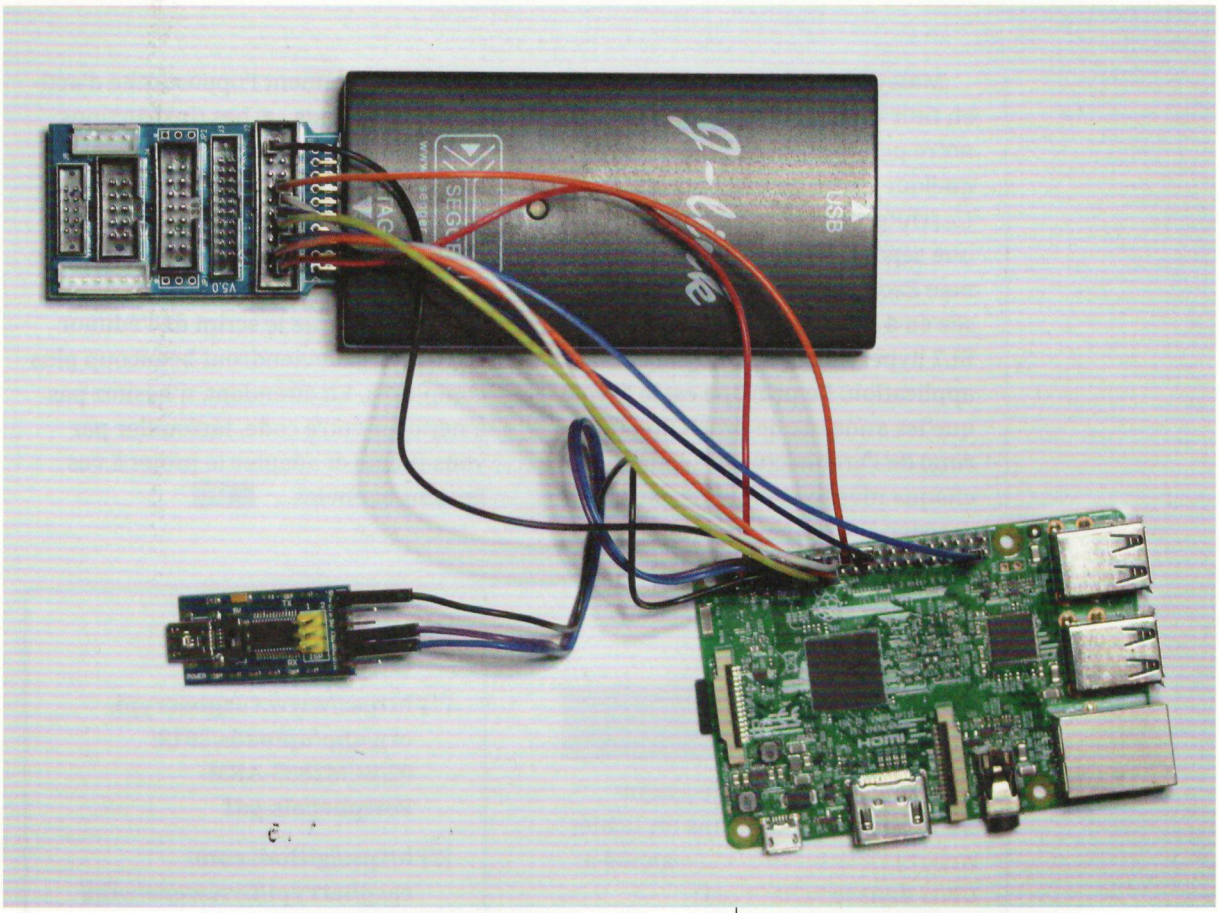
- `-ffreestanding` : indique au compilateur de ne pas considérer que les fonctions standard sont implémentées avec leurs prototypes habituels ;
- `-Wall` : active tous les avertissements du compilateur. On pourra également ajouter `-Wextra` pour davantage de vérification et d'avertissements ;
- `-g` : demande au compilateur et à l'éditeur de liens de conserver les symboles dans le binaire afin de faciliter la mise au point (*debug*) ;
- et surtout `-O0` qui indique le niveau d'optimisation du compilateur avec ici `0` correspondant à « aucune » (presque aucune).

C'est ce dernier argument qu'il vous faudra changer de `0` en `3`. Vous pouvez afficher le détail des optimisations actives en fonction de la valeur de l'option `-O` avec la commande `aarch64-none-elf-gcc -c -Q -O0 --help=optimizers | grep enabled`. Ce faisant, vous remarquerez que le compilateur, même avec `-O0` procède tout de même à quelques optimisations, en particulier sur les boucles.

Cette simple modification de notre `Makefile` nous apporte un gain phénoménal de performances, mais il a un coût et c'est précisément pour cette raison que `-O0` est généralement utilisé lors du développement : la compilation est plus lente et le code assembleur généré est bien plus complexe puisque le compilateur met en œuvre toute une ribambelle d'astuces pour accélérer le code. En cas de problème, il est bien plus difficile de mettre au point le code et de trouver un bug avec un niveau d'optimisation important (« *Prototype before polishing. Get it working before you optimize it.* » dans « *The Art of UNIX Programming* » de Eric S. Raymond **[6]**).

Le développement baremetal sur Pi en 32 bits n'a de sens que si l'on souhaite tirer pleinement profit de la puissance du ou des CPU ARM présents sur ces cartes. Pour des applications moins « gourmandes », même implémentant des fonctionnalités graphiques, mieux vaut se tourner vers des microcontrôleurs comme ce STM32F7 à cœur ARM Cortex-M7 incluant un accélérateur ChromART DMA2D et ayant un réel support du constructeur (datasheet complète, devkit, exemples, IDE, etc.).





À un moment ou un autre, dans votre exploration du développement baremetal sur Pi surgira l'étape où la question de la mise au point de programmes devra se poser sérieusement. Les Pi disposent d'un port JTAG permettant de connecter une sonde qui, couplée à OpenOCD et GNU GDB, permet un contrôle total de l'environnement d'exécution. C'est un point que nous traiterons dans un futur article...

Remarquez également que si le compilateur est en mesure de savoir précisément pour quelle architecture il fait son travail, l'optimisation n'en sera que plus efficace. Nous pouvons donc ajouter les options `-march=armv8-a+crc` et `-mcpu=cortex-a53` désignant explicitement l'ARM Cortex A53 (architecture ARM V8) comme cible pour notre Pi équipée d'un BCM2837. Le gain est minime, mais existant (21288 µs).

4. C'EST TOUT POUR LE MOMENT

Nous avons encore un certain nombre de choses à voir concernant le développement *baremetal* sur Raspberry Pi 3. Trois points pourront encore être potentiellement abordés : la mise en œuvre d'autres périphériques simples (SPI, I²C, etc.), l'utilisation d'une libC (Newlib) pour faciliter nos développements ou encore l'intégration d'une bibliothèque graphique comme HAGL ou LVGL.

Mais avant cela, nous devons en finir avec l'optimisation, car notre petit projet fait l'impasse sur quelque chose de très important. L'architecture ARM V8 implémente une séparation de privilèges nommée *Exception levels* (ou EL) divisée en 4 niveaux : EL3 *firmware*, EL2 hyperviseur, EL1 noyau et EL0 application. Le principe est le même que les anneaux de protection (ou *ring*) de l'architecture X86/amd64 où chaque niveau peut ou ne peut pas accéder à certaines fonctionnalités.

Dans l'une des sorties d'écran précédentes, vous avez peut-être remarqué la ligne :

```
Exception level: 2
```

Ceci provient du code assembleur suivant stocké dans un fichier `armutils.S` (`armutils.o` ajouté à `OBJS` dans le `Makefile`) :

```
.global get_el
get_el:
        mrs x0, CurrentEL
        lsr x0, x0, #2
        ret
```

Et appelé depuis `hello.c` :

```
puts("Exception level: ");
printdec(get_el());
puts("\r\n");
```

Ceci nous montre que notre code s'exécute en EL2 alors que, techniquement, nous devrions être en EL1 pour ce type de développement (nous n'écrivons pas un hyperviseur, mais un pseudonoyau). Changer de niveau nous permettra de corriger ce point, mais nous donnera également l'opportunité d'activer les différents caches (instructions et *data*) qui devraient sensiblement accélérer notre code, et en apprendre davantage sur le démarrage de la carte et du SoC. Ceci est un gros morceau et nous en profiterons pour revoir notre `crt0.S` ainsi que le script de l'éditeur de liens, qui deviendront beaucoup plus complexes. En attendant, n'hésitez pas à nettoyer votre code, farfouiller par vous-même et adapter le projet à vos besoins et envies... **DB**

RÉFÉRENCES

[1] **https://www.raspberrypi.org/app/uploads/2012/02/BCM2835-ARM-Peripherals.pdf**

[2] **https://github.com/raspberrypi/firmware/wiki/Mailbox-property-interface**

[3] **https://connect.ed-diamond.com/hackable/hk-040/developpement-baremetal-sur-pi3-mailbox-et-framebuffer**

[4] **https://developer.arm.com/documentation/ddi0183/g/programmers-model/register-descriptions/fractional-baud-rate-register--uartfbrd**

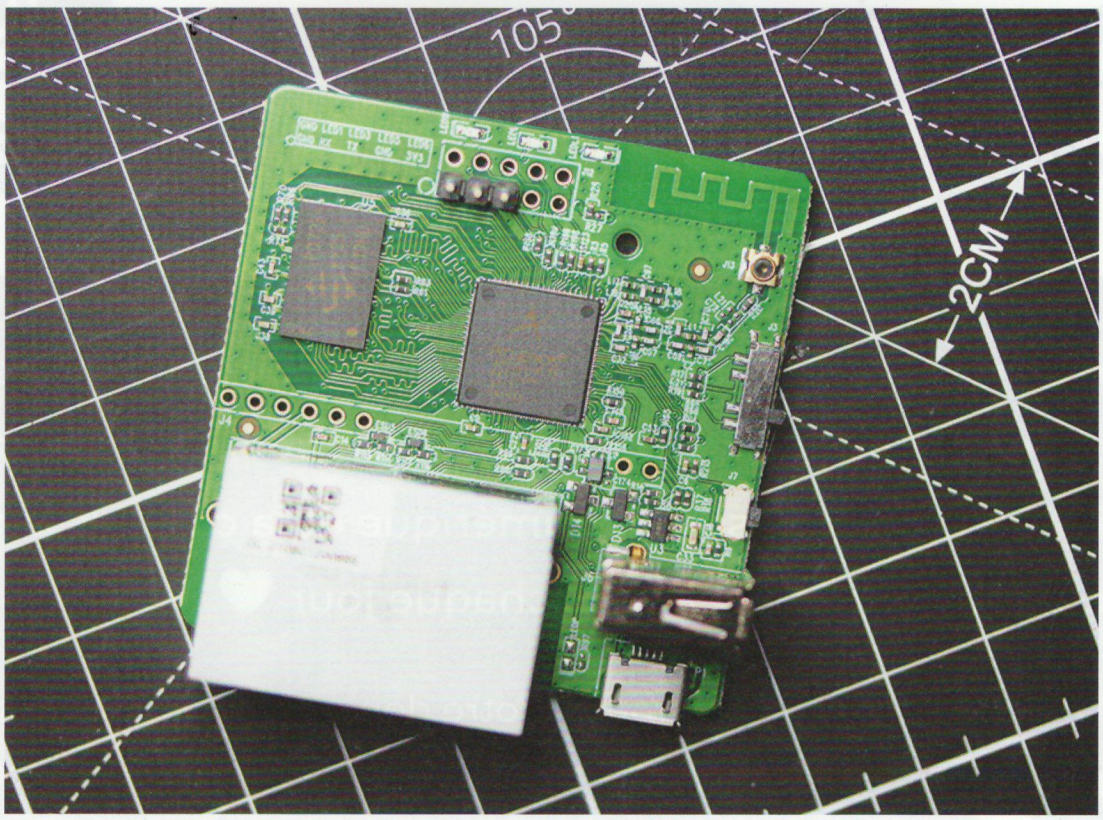
[5] **https://connect.ed-diamond.com/hackable/hk-040/developpement-baremetal-sur-raspberry-pi-3**

[6] **http://catb.org/esr/writings/taoup/html/**

OPENWRT : UN FIRMWARE ET DES APPLICATIONS

Laurent FOUCHER
Consultant informatique en IoT et SDR

Pour certaines applications légères, OpenWrt tournant sur un équipement réseau low cost est une alternative solide face à un OS classique sur une Raspberry Pi. Nous allons voir comment intégrer une application dans un firmware destiné à un petit routeur Wi-Fi coûtant moins de 30 euros.



La plateforme de choix pour une petite application IoT pourrait être une Raspberry Pi avec une distribution GNU/Linux classique (Ubuntu, Debian, etc.). Il s'agit d'une plateforme répandue pour laquelle il existe une documentation abondante et une communauté nombreuse prête à aider. Elle est de surcroît réputée abordable financièrement.

D'un autre côté, les équipements réseaux tels que les box internet et routeurs Wi-Fi disposent désormais d'une certaine puissance et font tourner des applications tel qu'un *media center* ou le partage de fichiers, il peut être tentant d'y substituer notre propre application.

Nous avions un réseau de passerelles IoT à construire pour remonter des mesures fournies par des capteurs Wi-Fi spécifiques, un prototype a donc été réalisé en Python avec une Raspberry Pi.

En regardant les passerelles ZigBee vendues en grande surface pour gérer des capteurs et ampoules connectées (par exemple, les passerelles Tuya chez Lidl), je me suis demandé s'il n'y avait pas moyen de « détourner » de petits points d'accès Wi-Fi pour notre usage.

1. PRÉSENTATION DE OPENWRT

OpenWrt est un système d'exploitation *open source* basé sur Linux, il était initialement destiné aux équipements réseau tels que les routeurs et points d'accès Wi-Fi, il s'agissait d'avoir une alternative *open source* aux systèmes propriétaires qui animaient ces équipements (OpenWrt tient d'ailleurs son nom du routeur Wi-Fi Linksys WRT54GL qui était un matériel de choix pour les bidouilleurs).

Certains constructeurs tels que GL.iNet (**https://www.gl-inet.com/**) vendent du matériel fonctionnant sous une version « maison » de OpenWrt, il est alors possible de les passer sous OpenWrt « *open source* » de façon simple.

D-Link, Netgear, Linksys ou TP-Link utilisent leurs propres *firmwares*, cependant leurs matériels sont supportés par OpenWrt grâce à la communauté et/ou le support du constructeur.

OpenWrt utilise de très faibles ressources, le minimum requis pour OpenWrt 21.02 est de 64 Mo de RAM et 8 Mo de mémoire flash.

Afin d'éviter les risques de corruption de système de fichiers, les répertoires au contenu « volatile » (`/var/run`, `/var/log`...) sont des liens vers `/tmp` qui est de type `tmpfs` (c'est-à-dire stocké en RAM). Le système de fichiers racine est un *overlay* qui se superpose au système de fichiers de base (qui est en lecture seule et stocké compressé en mémoire flash).

OpenWrt est configurable grâce à une interface web appelée LuCI, il est possible de développer des modules spécifiques qui seront rajoutés à cette interface. Il est également possible de se connecter en SSH pour paramétrer ou mettre à jour un équipement. LuCI et SSH sont des options qui ne sont disponibles que si elles ont été activées lors de la génération du *firmware*.

La mise à jour de l'OS se fait en installant une nouvelle version de *firmware*, la mise à jour est possible avec ou sans conservation des paramètres.

OpenWrt dispose d'un gestionnaire de *packages* qui lui est propre, « `opkg` » (**https://openwrt.org/docs/guide-user/additional-software/opkg**), qui gère classiquement les dépendances entre *packages* et permet de les installer

ou de les mettre à jour. La mise à jour d'un *package* provoque l'écriture dans l'*overlay* et consomme du stockage supplémentaire (puisque la version « ROM » incluse dans le *firmware* et la version installée cohabitent sur le stockage).

Afin de consommer le moins de ressources possible, le système est minimal et n'embarque que les drivers propres à un équipement donné. La plupart des équipements utilisent des systèmes intégrés sur une puce « SoC » (**https://fr.wikipedia.org/wiki/Système_sur_une_puce**) et donc le *firmware* ne contient le support que pour un seul SoC.

2. CHOIX D'UN ÉQUIPEMENT

La page « *Table of Hardware* » (**https://openwrt.org/toh/start**) donne la liste des équipements compatibles avec OpenWrt.

Outre qu'il faut choisir un équipement compatible avec OpenWrt, le choix s'effectue en fonction des critères classiques pour un équipement réseau :

- nombre et type des interfaces réseau filaires ;
- présence du Wi-Fi (2.4 GHz, 5 GHz) et type d'antenne (interne, fixe, externe) ;
- alimentation électrique (5V, 12V, PoE passif, PoE actif) ;
- présence de ports USB ;
- présence de connectivité 3G/4G/5G, type de carte SIM (physique ou eSIM) ;
- disponibilité du GNSS (GPS, Galileo...) ;
- puissance (mémoire, flash, processeur) en fonction du type d'usage ou d'application ;
- type du boîtier (étanche, plastique, métallique, compatibilité avec un rail DIN).

Un petit routeur Wi-Fi coûte environ 30 €, un routeur Wi-Fi étanche supportant le PoE coûte entre 50 et 100 €, un routeur 4G en boîtier métallique supportant une alimentation de 9 à 35 volts coûte environ 100 €.

Bien qu'il soit possible d'utiliser de vieux équipements présents dans vos tiroirs, il est déconseillé d'utiliser un routeur ADSL, en effet la génération du firmware OpenWrt est plus compliquée, car les chipsets ADSL ne sont généralement pas *open source*. Générer un *firmware* OpenWrt nécessite souvent des manipulations pour extraire le *firmware* du chipset ADSL d'un *firmware* du constructeur.

Pour cet article, nous allons utiliser un GL.iNet GL-AR150 (**https://www.gl-inet.com/products/gl-ar150/**) disponible sur Internet à moins de 30 €.

Notez que les systèmes de GL.iNet disposent de 2 ou 3 GPIO qui sont repérés sur le circuit imprimé dans le cas où votre application en aurait besoin.

3. CRÉATION D'UN ENVIRONNEMENT DE GÉNÉRATION DE FIRMWARE

Il est possible de compiler des programmes qui sont exécutés sous OpenWrt. Comme pour les *firmwares*, le processus de compilation doit prendre en compte l'équipement auquel il est destiné (en particulier, quel SoC).

Il n'est pas possible de compiler directement sur la machine cible, car les équipements n'ont pas les ressources nécessaires (CPU/mémoire/disque). On

utilise donc une machine sous GNU/Linux (poste de travail ou serveur) qui dispose des ressources nécessaires, on parle alors de « compilation croisée », ou de « *cross-compilation* » en anglais (**https://fr.wikipedia.org/wiki/Chaîne_de_compilation#Chaîne_de_compilation_croisée**).

La génération d'un *firmware* nécessite de compiler :

- tous les outils de compilation (la « *toolchain* » **https://fr.wikipedia.org/wiki/Chaîne_de_compilation**) ;
- le *kernel* Linux ;
- les différents programmes et bibliothèques utilisables sous OpenWrt ou utilisés par le système pour fonctionner.

Le plus simple est d'utiliser un conteneur Docker, ceci va permettre de s'affranchir du système d'exploitation de la machine sur laquelle on compile.

Exemple de `Dockerfile` pour construire un environnement de génération OpenWrt `v19.07.7` :

```
FROM ubuntu:20.04

RUN apt-get update \
 && DEBIAN_FRONTEND=noninteractive apt-get install -y \
    build-essential ccache ecj \
    fastjar file g++ gawk \
    gettext git java-propose-classpath \
    libelf-dev libncurses5-dev libncursesw5-dev \
    libssl-dev python python2.7-dev python3 \
    python3-distutils python3-setuptools \
    python3-dev rsync subversion swig \
    time unzip wget xsltproc zlib1g-dev \
 && useradd buildbot -m -k /dev/null -d /home/buildbot

USER buildbot

WORKDIR /home/buildbot
RUN git clone https://git.openwrt.org/openwrt/openwrt.git source \
 && cd /home/buildbot/source \
 && git checkout v19.07.7 \
 && make distclean \
 && ./scripts/feeds update -a \
 && ./scripts/feeds install -a

WORKDIR /home/buildbot
```

Ce `Dockerfile` permet de créer une image contenant tout le nécessaire pour générer un *firmware* ou de compiler des programmes pour OpenWrt.

```
$ docker build . -t openwrt_buildbot_atelier
```

4. RÉCUPÉRATION DES INFORMATIONS PRÉALABLES

Afin de pouvoir générer un *firmware* adapté à notre équipement, nous devons vérifier quelle est son architecture interne.

Attention : certains constructeurs déclinent leurs équipements en plusieurs versions (parfois notées v1, v2...), ces versions peuvent avoir des architectures complètement différentes. Très peu de vendeurs en ligne précisent la version sur leur site.

Attention également aux clones présents sur les sites tels « AliExpress » et sur le Marketplace d'Amazon, ils peuvent ne pas avoir la même architecture que le produit officiel.

Dans le « tableau du hardware supporté » (**https://openwrt.org/toh/start**), nous tapons le modèle de notre équipement « GL-AR150 ». Ceci nous amène à une liste restreinte, si le modèle dispose de plusieurs versions supportées, elles sont listées. En cliquant sur le modèle dans la colonne « *device page* », nous accédons à la page dédiée à notre équipement (**https://openwrt.org/toh/gl.inet/gl-ar150**).

La page de l'équipement donne ses caractéristiques ainsi que la dernière version de OpenWrt supportée. Concernant la version de OpenWrt, trois cas sont à distinguer :

- L'équipement est pleinement supporté : la version listée est la dernière version « officielle » de OpenWrt (à la date d'aujourd'hui 21.02.1, mais pour cet article nous allons utiliser la 19.07.7).
- Le support de l'équipement est en cours de développement : elle est alors marquée « *snapshot* », c'est-à-dire la version courante non taguée dans GitHub.
- L'équipement n'est pas supporté par les dernières versions de OpenWrt : une ancienne version de OpenWrt est alors listée. C'est notamment le cas pour les équipements qui n'ont pas assez de RAM ou de mémoire flash pour supporter les dernières versions de OpenWrt.

Dans le `Dockerfile` précédent, notez la ligne `git checkout v19.07.7`, elle permet d'extraire une version particulière d'OpenWrt.

Dans la rubrique « Installation » de la page de l'équipement, on trouve un lien vers le *firmware* le plus récent disponible, il est possible de le télécharger et de l'installer directement, mais nous allons en compiler un nous-mêmes.

L'URL de téléchargement du *firmware* nous donne des informations importantes **http://downloads.openwrt.org/releases/19.07.7/targets/ath79/generic/openwrt-19.07.7-ath79-generic-glinet_gl-ar150-squashfs-sysupgrade.bin** :

- *version* : `19.07.7` ;
- *target* : `ath79` ;
- *subtarget* : `generic` ;
- *target profile* : `glinet_gl-ar150`.

La *target* est le SoC utilisé par l'équipement, *subtarget* représente des spécificités d'architecture (en particulier au niveau de la mémoire flash) et *target profile* est le modèle de l'équipement.

Attention : vous verrez dans le chapitre suivant qu'il existe une *target Atheros AR7xxx/AR9xxx* qui semble parfaitement adaptée à notre équipement listé comme étant basé sur le SoC *Atheros AR9331*, cependant il semble que cette *target* soit obsolète et elle est inutilisable, d'où l'intérêt de se baser sur le nom du *firmware* pour déterminer ces trois éléments.

5. FIRMWARE SYSUPGRADE, RECOVERY, FIRMWARE FACTORY

Certains équipements sont livrés avec un *firmware* basé sur OpenWrt, dans ce cas vous pouvez installer un *firmware* standard appelé *sysupgrade* en utilisant la procédure décrite par le constructeur, mais en utilisant l'un de ceux que vous avez générés.

Certains équipements sont livrés avec des *firmwares* complètement propriétaires et n'acceptent d'installer que ceux mis à disposition par le constructeur. Dans ce cas, il peut être nécessaire d'ouvrir l'équipement, voire de faire de la soudure pour se connecter à un port console ou JTAG. Cette procédure dite de « *recovery* » est généralement décrite sur la page du site OpenWrt dédiée à l'équipement.

Pour certains équipements utilisant des *firmwares* propriétaires, la procédure de génération va générer en plus du *firmware sysupgrade* un *firmware factory* qui sera reconnu comme un *firmware* du constructeur et sera donc utilisable en utilisant la procédure décrite par le constructeur.

6. GÉNÉRATION D'UN FIRMWARE

La procédure pour générer un *firmware* est similaire à celle pour compiler un *kernel* Linux, elle est basée sur un « menu de configuration ».

Nous devons commencer par créer un *container* Docker à partir de l'image générée précédemment.

```
$ docker run -ti --name atelier openwrt_buildbot_atelier
buildbot@de3f92b60bb7:~$
```

Nous sommes maintenant à l'intérieur du *container* qui contient tout le nécessaire pour générer un *firmware*, `buildbot@de3f92b60bb7:~$` est l'invite de notre shell.

Note : si vous êtes sorti accidentellement du shell de votre *container*, vous pouvez le relancer et vous y réattacher :

```
$ docker restart atelier
$ docker attach atelier
$ buildbot@de3f92b60bb7:~$
```

La commande `make menuconfig` permet de configurer l'image désirée.

```
buildbot@de3f92b60bb7:~$ cd source
buildbot@de3f92b60bb7:~/source$ make menuconfig
```

Lancer `menuconfig` et mettre les valeurs suivantes pour les 3 paramètres matériels vus plus haut :

- ***target: Atheros ATH79 (DTS)*** ;
- ***subtarget: generic*** ;
- ***target profile: GL.iNet GL-AR150***.

Ajouter l'interface de configuration web :

- ***LuCI ---> 1. Collections ---> <*> luci*** ;
- ***LuCI ---> 2. Modules ---> <*> luci-compat***.

Notez que sélectionner « *LuCI* » va automatiquement sélectionner ses dépendances.

Ensuite, sauvegarder et quitter `menuconfig`. Un fichier `.config` est créé avec les options choisies et les options par défaut. Voici le début de ce fichier :

```
$ buildbot@de3f92b60bb7:~/source$ head .config -n 13
#
# Automatically generated file; DO NOT EDIT.
# OpenWrt Configuration
#
CONFIG_MODULES=y
CONFIG_HAVE_DOT_CONFIG=y
# CONFIG_TARGET_sunxi is not set
# CONFIG_TARGET_apm821xx is not set
# CONFIG_TARGET_ath25 is not set
# CONFIG_TARGET_ar71xx is not set
CONFIG_TARGET_ath79=y
# CONFIG_TARGET_brcm2708 is not set
# CONFIG_TARGET_bcm53xx is not set
```

Il ne faut pas le modifier à la main, mais il est possible d'en créer un minimal, puis d'y rajouter automatiquement les valeurs par défaut. Nous verrons cela ultérieurement.

On lance le téléchargement des fichiers nécessaires :

```
buildbot@de3f92b60bb7:~/source$ make -j $(nproc) download
```

Le déroulement des opérations s'affiche lors de l'exécution de la commande :

- `tools` : correspond aux fichiers nécessaires pour compiler les outils génériques ;
- `toolschain` : les fichiers nécessaires pour compiler la *toolchain* (les compilateurs) ;
- `feed` : les sources de *packages*, LuCI n'est pas intégré au système OpenWrt de base, il est dans des sources distinctes ;
- `package` : les différents *packages* qui vont composer le système.

La commande suivante va compiler tout ce qui est nécessaire (la *toolchain*, tous les *packages* demandés dans `menuconfig`) et va générer un *firmware* installable directement sur notre équipement :

```
buildbot@de3f92b60bb7:~/source$ make -j $(nproc) world
```

L'exécution de cette commande peut durer plus d'une heure en fonction de la puissance de la machine sur laquelle vous compilez.

Si par la suite, vous enlevez ou rajoutez des fonctionnalités à l'aide de `menuconfig`, cela sera largement plus rapide, car il n'est pas nécessaire de compiler les éléments non modifiés.

Le *firmware* généré est disponible dans notre *container* : `/home/buildbot/source/bin/targets/ath79/generic/openwrt-ath79-generic-glinet_gl-ar150-squashfs-sysupgrade.bin`.

7. COPIE DU FIRMWARE GÉNÉRÉ

Le *firmware* a été généré à l'intérieur du conteneur Docker « atelier » que nous avons créé.

Nous allons maintenant le copier pour qu'il soit utilisable en dehors de notre *container*.

Lancée sur le serveur Docker, la commande `docker cp` permet de copier des fichiers entre le serveur Docker et les *containers* (dans les deux sens).

Nous considérons que vous travaillez depuis un PC sous GNU/Linux, la commande est à adapter pour Windows (changer le chemin de destination de la copie) :

```
$ docker cp atelier:/home/buildbot/source/bin/targets/ath79/generic/openwrt-
ath79-generic-glinet_gl-ar150-squashfs-sysupgrade.bin /tmp/firmware.bin
$ ls -l /tmp/firmware.bin
-rw-r--r-- 1 takeshiba takeshiba 4195091 août  18 22:25 /tmp/firmware.bin
```

8. APPLICATION DU FIRMWARE

Nous supposons que vous disposez d'un routeur GL-AR150 neuf, ou remis en configuration d'usine.

De même, vous disposez d'un ordinateur avec une interface Ethernet paramétrée en DHCP.

Connectez votre ordinateur au port marqué « *LAN* » du GL-AR150, mettez-le sous tension et attendez 2 minutes que le routeur ait fini de s'initialiser.

Lancez un navigateur web et allez à l'adresse **http://192.168.8.1/** (notez le `8.1` de l'adresse) :

- Sélectionner ***English*** comme langue ;
- Choisissez un mot de passe.

Vous devez arriver sur ***Admin Panel*** comme illustré dans la Figure 1, page suivante.

Cette interface d'administration n'est pas celle de OpenWrt, elle est spécifique à GL.iNet.

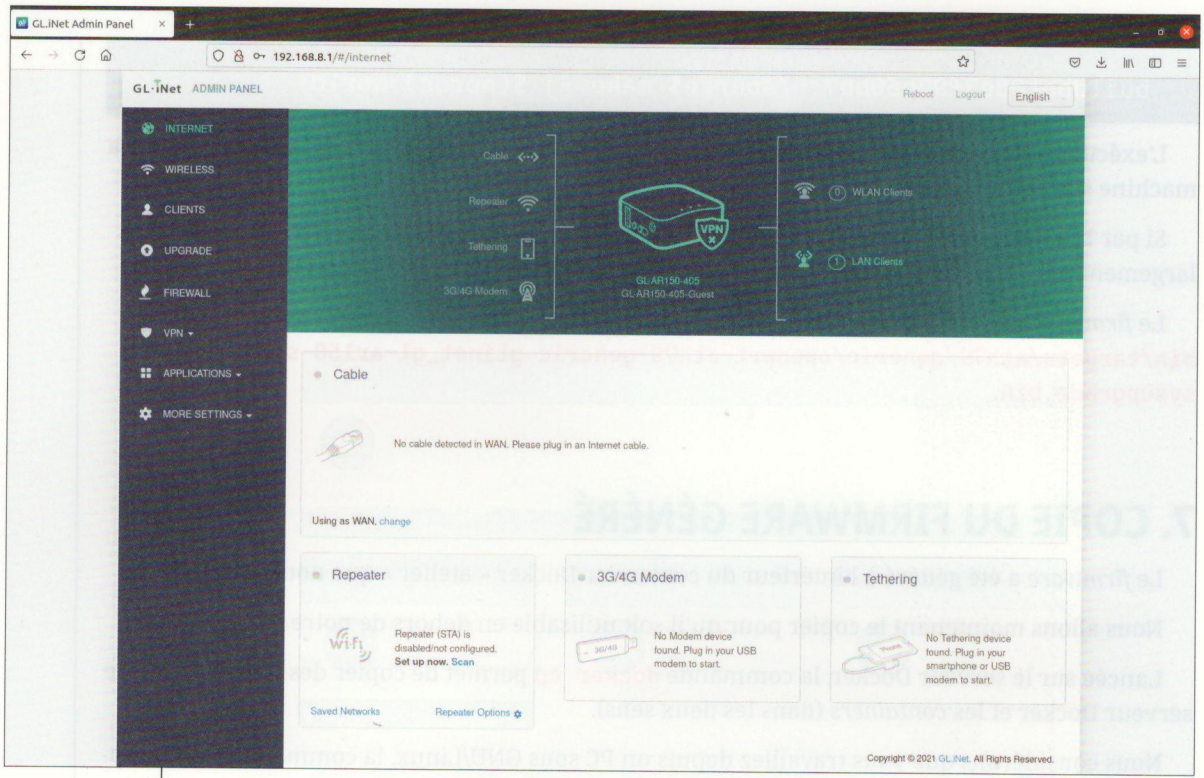


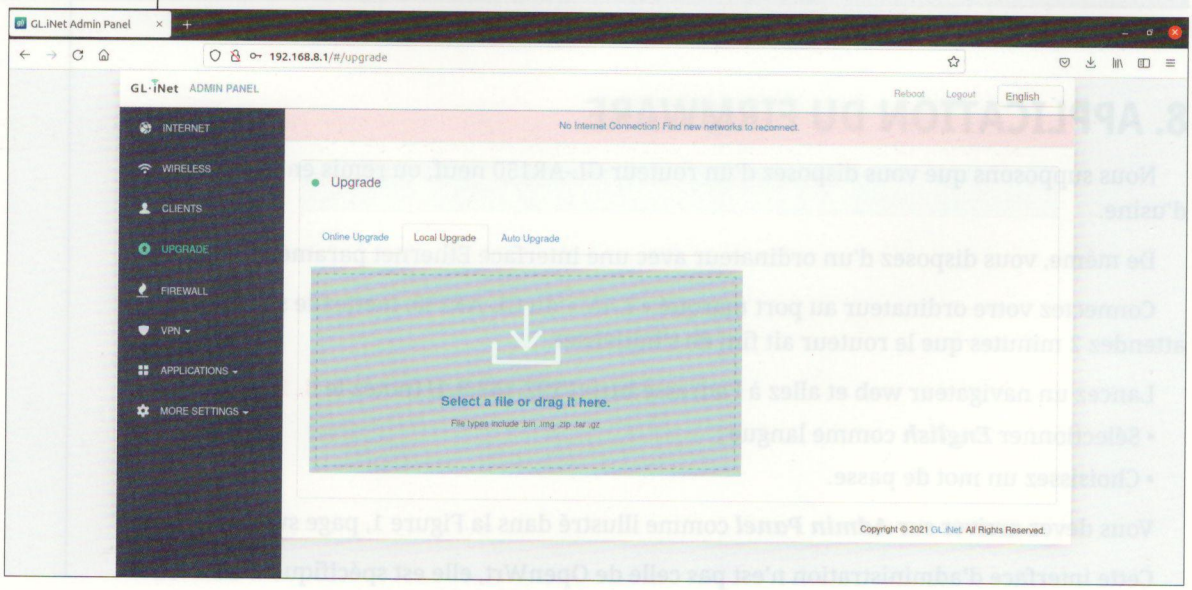
Figure 1

Allez dans le menu ***UPGRADE***, puis l'onglet ***Local Upgrade*** (Figure 2).

Vous pouvez au choix :

- Faire un glisser-déplacer du fichier `/tmp/firmware.bin` entre votre explorateur de fichiers et la zone entourée de pointillés sur la page « Upgrade ».

Figure 2



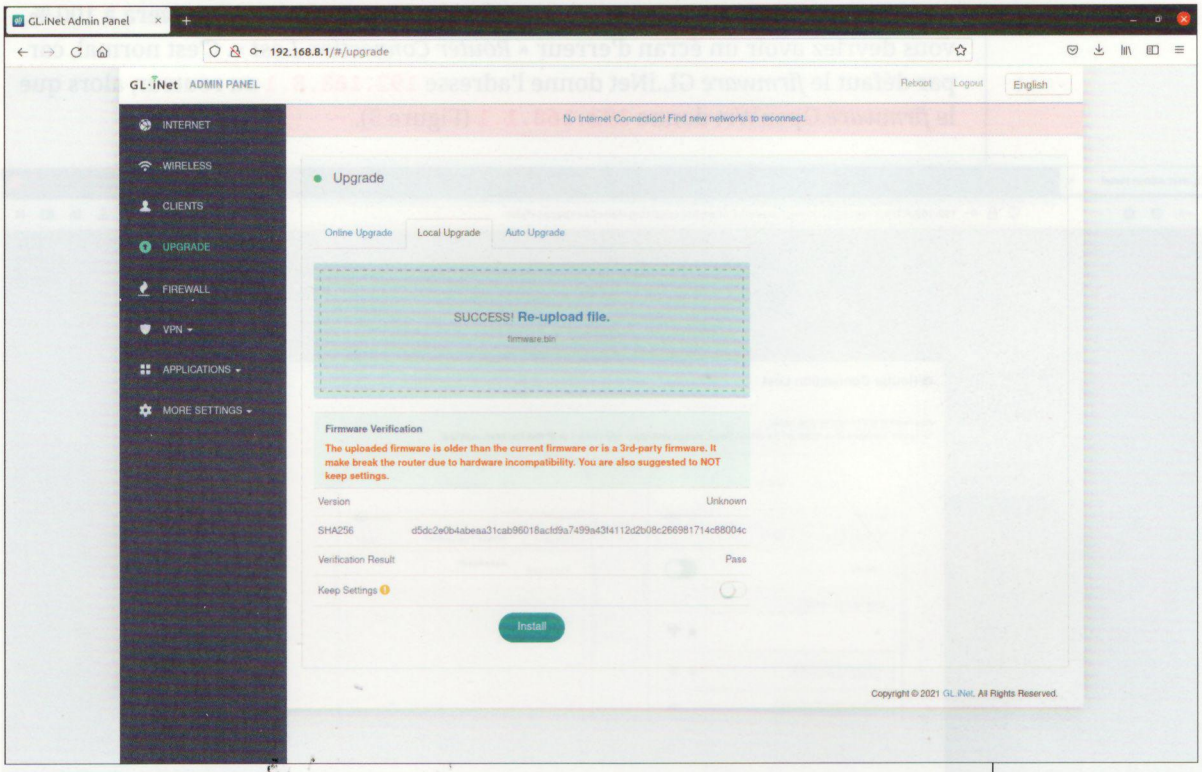


Figure 3

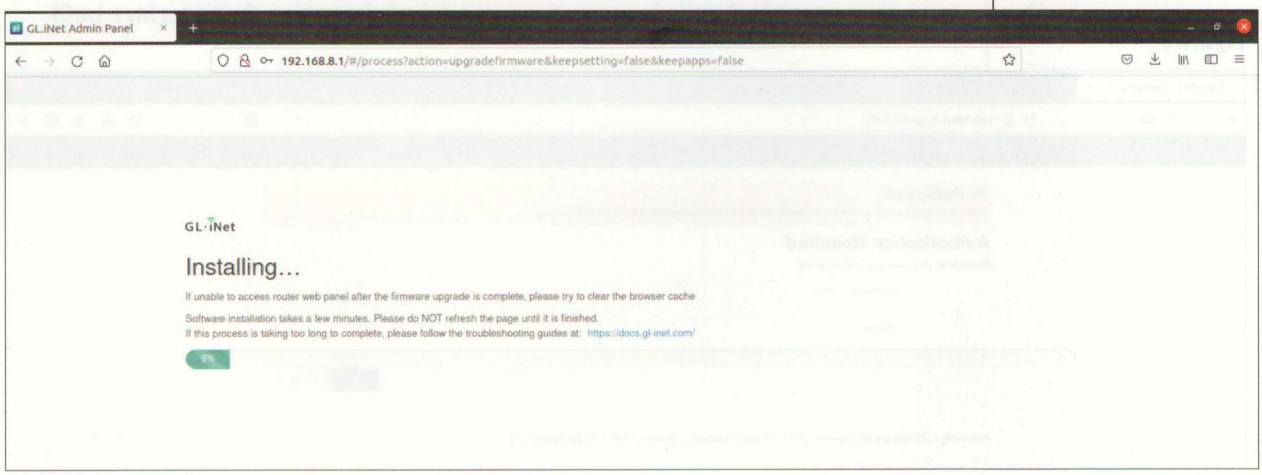
- Cliquer sur ***Select a file or drag it here*** et naviguer pour sélectionner le fichier `/tmp/firmware.bin`.

Le *firmware* est envoyé vers le routeur, et il est contrôlé.

Désactivez ***Keep Settings*** et cliquez sur ***Install*** (Figure 3).

Laisser la mise à jour se faire (Figure 4).

Figure 4



Attention, la barre de progression est fictive. Lorsque la barre arrivera à 100 %, vous devriez avoir un écran d'erreur « *Router Connection Lost* ». C'est normal, car par défaut le *firmware* GL.iNet donne l'adresse `192.168.8.1` au routeur, alors que le *firmware* OpenWrt donne `192.168.1.1` (Figure 5).

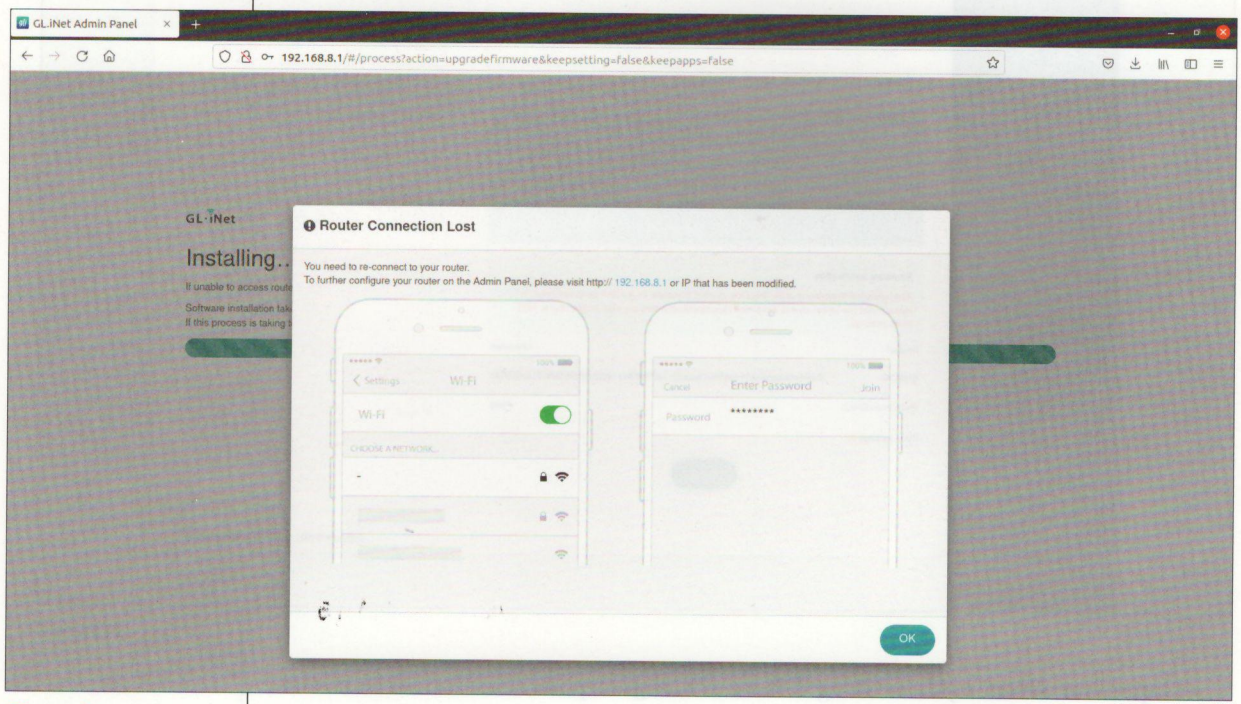
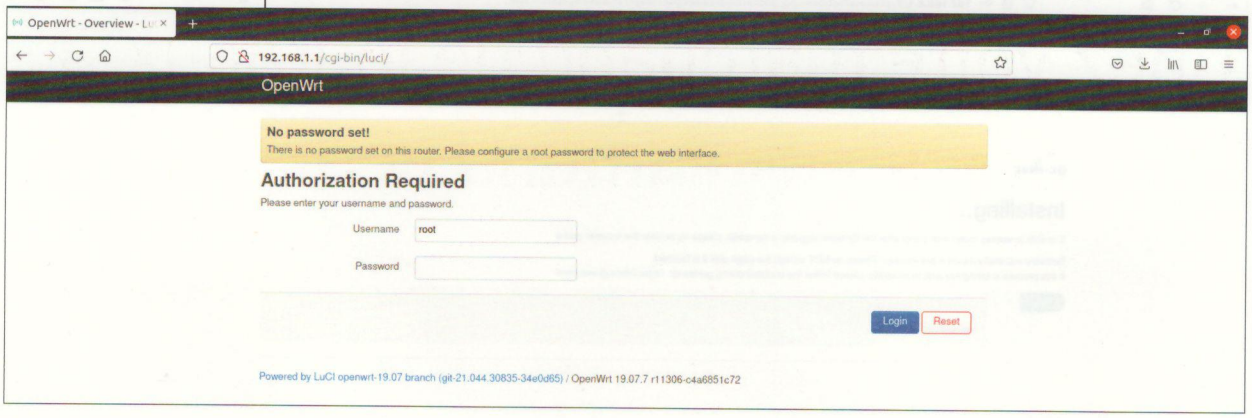


Figure 5

Fermez la fenêtre du navigateur, ouvrez en une nouvelle et allez à l'adresse **http://192.168.1.1/**, vous devriez voir apparaître l'interface de paramétrage standard de OpenWrt appelée « LuCI ».

Notez que par défaut, il n'y a pas de mot de passe pour l'utilisateur administrateur « `root` ». Il faut bien évidemment en définir un avant la mise en réseau du routeur, c'est possible depuis le menu ***System->Administration*** de l'interface LuCI.

Figure 6



9. AJOUT DE NOTRE APPLICATION À OPENWRT

Nous allons ajouter l'application « *HelloServer* », il s'agit d'un serveur web qui affiche « `Hello World!` » lorsqu'on se connecte dessus, il sera possible de modifier « *Hello* » par autre chose à partir de l'interface « LuCI » dans laquelle un onglet « *Hello Server* » sera rajouté.

En fait pour ceci, nous allons créer un *package* `helloserver` et nous le rajouterons au système `menuconfig` que nous avons vu précédemment.

9.1 Structure du package

On commence par récupérer le code source de notre application dans notre *container* :

```
$ cd /home/buildroot
$ mkdir U03
$ cd U03
$ git clone https://github.com/U03/OpenWrt_hk
$ mv OpenWrt_hk helloserver
```

Les fichiers suivants composent notre *package* :

- `helloserver.c` : le programme lui-même, il est écrit en C ;
- `Makefile` : le fichier décrivant la compilation du programme, la construction et l'installation du *package* ;
- `etc_init_d_helloserver` : script permettant de lancer le programme ;
- `usr_share_luci_menu_d_helloserver.json` : définition du menu *HelloServer* dans LuCI ;
- `usr_lib_lua_luci_model_cbi_helloserver_configuration.lua` : définition de l'écran de configuration *HelloServer* dans LuCI.

Un fichier de configuration permettant de mémoriser le message à afficher sera créé lors du premier lancement. Ce fichier est lu par notre programme et mis à jour dans LuCI.

Le fichier de configuration est un fichier texte, il aurait été facile de l'ajouter au *package*, mais il aurait été écrasé à chaque mise à jour du *package* ou du *firmware*, du coup sa création (et les éventuels ajouts de paramètres) seront gérés dans le script de lancement :

```
start() {
   uci get helloserver 2> /dev/null
   if [[ ! $? == 0 ]]; then
       uci import helloserver < /dev/null
       uci add helloserver server
       uci set helloserver.@server[0].message='Hello'
       uci commit helloserver
   fi
```

9.2 Ajout de notre source de packages

Nous pouvons maintenant ajouter la définition de notre « *feed* » au système « *menuconfig* » :

```
$ cd /home/buildbot/source
$ cat << EOF > feeds.conf
src-link U03 /home/buildbot/U03
EOF

$ ./scripts/feeds update U03
$ ./scripts/feeds install -a -p U03
```

Les deux dernières commandes peuvent durer une bonne dizaine de secondes, la deuxième commande provoque le rajout dans le système `menuconfig`, elle doit afficher les deux lignes suivantes (ignorer le *warning* sur la librairie manquante `kmod-crypto-arc4`) :

```
Installing all packages from feed U03
Installing package 'helloserver' from U03
```

Nous avons vu que la commande `make menuconfig` génère un fichier `.config` décrivant l'équipement et ce que doit contenir le *firmware*. Nous avions dit qu'il ne fallait pas le modifier manuellement.

Il est néanmoins possible d'en créer automatiquement (c'est utilisé par les systèmes de génération automatisée de *firmware*).

Nous créons un fichier `.config` minimal (ou nous écrasons le fichier existant). Il contient la définition de notre *target*, ainsi que l'ajout de notre *package* personnel (précédemment ajouté grâce au *feed*), on ajoute l'interface web de paramétrage LuCI.

Nous appelons ensuite la commande `make defconfig` qui va analyser notre fichier minimal, placer les options par défaut ainsi que les dépendances induites.

```
$ cd /home/buildbot/source
$ cat << EOF > .config
CONFIG_TARGET_ath79=y
CONFIG_TARGET_ath79_generic=y
CONFIG_TARGET_ath79_generic_DEVICE_glinet_gl-ar150=y
# Ajout de notre application
CONFIG_PACKAGE_helloserver=y
# Ajout de l'interface LuCI
CONFIG_PACKAGE_luci=y
CONFIG_PACKAGE_luci-base=y
CONFIG_PACKAGE_luci-compat=y
EOF

$ make defconfig
```

Si vous lancez `make menuconfig`, vous verrez que la *target* est définie de la même manière que lorsque nous l'avions définie manuellement, un menu ***U03*** est apparu dans lequel ***helloserver*** est sélectionné.

10. COMPILATION ET GÉNÉRATION DU PACKAGE

Nous pouvons désormais compiler notre programme et générer le *package* d'installation correspondant :

```
$ cd /home/buildbot/source
$ make package/helloserver/compile
```

Le système va éventuellement tenter de recompiler les dépendances de notre programme, puis il va générer le *package* d'installation de notre programme, il sera disponible à cet emplacement : `/home/buildbot/source/bin/packages/mips_24kc/U03/helloserver_1.0-0_mips_24kc.ipk`.

11. INSTALLATION DU PACKAGE

Nous pouvons transférer directement le *package* vers notre routeur depuis le *container* de génération :

```
$ scp /home/buildbot/source/bin/packages/mips_24kc/U03/helloserver_1.0-0_
mips_24kc.ipk \
root@192.168.1.1:/tmp/
```

Lors du premier transfert, la commande `scp` affichera une demande de confirmation, taper `yes` puis la touche Entrée :

```
The authenticity of host '192.168.1.1 (192.168.1.1)' can't be established.
RSA key fingerprint is SHA256:8miIqQTCpxwejNJ3NtFESYvN3SrvcKFHH5iKHWyj7O4.
Are you sure you want to continue connecting (yes/no/[fingerprint])?
```

Il est possible de se connecter en SSH à notre routeur à l'aide du mot de passe défini précédemment :

```
$ ssh root@192.168.1.1
```

Nous sommes accueillis par la jolie bannière OpenWrt :

```
BusyBox v1.30.1 () built-in shell (ash)

  _______                     ________        __
 |       |.-----.-----.-----.|  |  |  |.----.|  |_
 |   -   ||  _  |  -__|     ||  |  |  ||   _||   _|
 |_______||   __|_____|__|__||________||__|  |____|
          |__| W I R E L E S S   F R E E D O M
 -----------------------------------------------------
 OpenWrt 19.07.7, r11306-c4a6851c72
 -----------------------------------------------------
```

Le *package* d'*install* est visible dans le répertoire `/tmp` ; on voit qu'il est de taille très réduite :

```
$ ls -lh /tmp/helloserver_1.0-0_mips_24kc.ipk
-rw-r--r--    1 root     root        3.8K Feb 15 19:33 /tmp/
helloserver_1.0-0_mips_24kc.ipk
```

Il est désormais possible de l'installer :

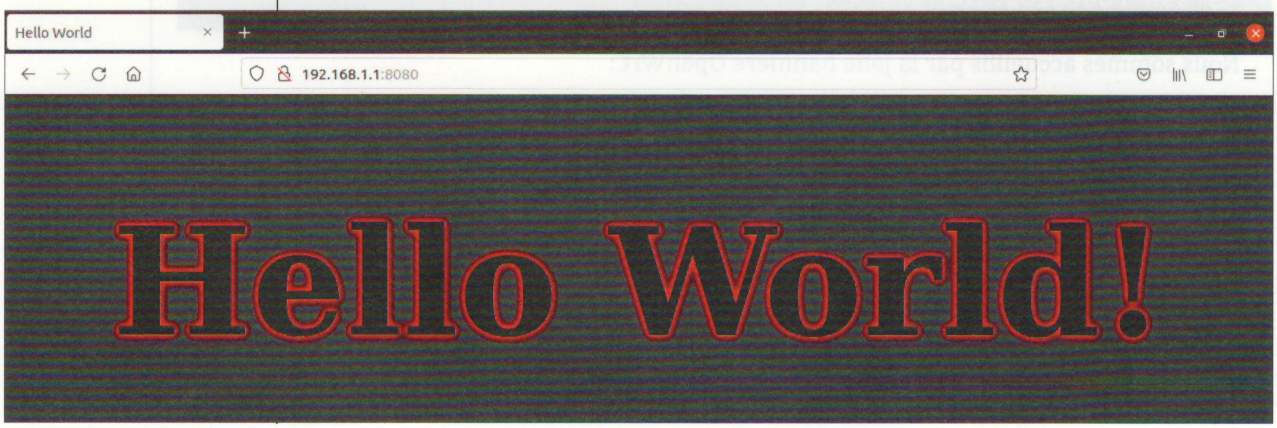
```
$ opkg install /tmp/helloserver_1.0-0_mips_24kc.ipk
      Installing helloserver (1.0-0) to root...
      Configuring helloserver.
      cfg01769c
```

Les commandes `ps` et `netstat -tlnp` permettent de vérifier que le programme est bien lancé et qu'il a bien ouvert le port TCP sur lequel il attend les connections (port 8080) :

```
root@OpenWrt:~# ps
  PID USER       VSZ STAT COMMAND
    1 root      1564 S    /sbin/procd
    2 root         0 SW   [kthreadd]
  .../...
 2848 root      1116 S    /usr/bin/helloserver

root@OpenWrt:~# netstat -tlnp
Active Internet connections (only servers)
Proto Recv-Q Send-Q Local Address  Foreign Address  State   PID/Program name
tcp        0      0 0.0.0.0:8080   0.0.0.0:*       LISTEN  2848/helloserver
```

Ouvrez un navigateur et allez à l'adresse **http://192.168.1.1:8080/**, vous devriez voir un écran marqué « `Hello World!` » (Figure 7).

Figure 7



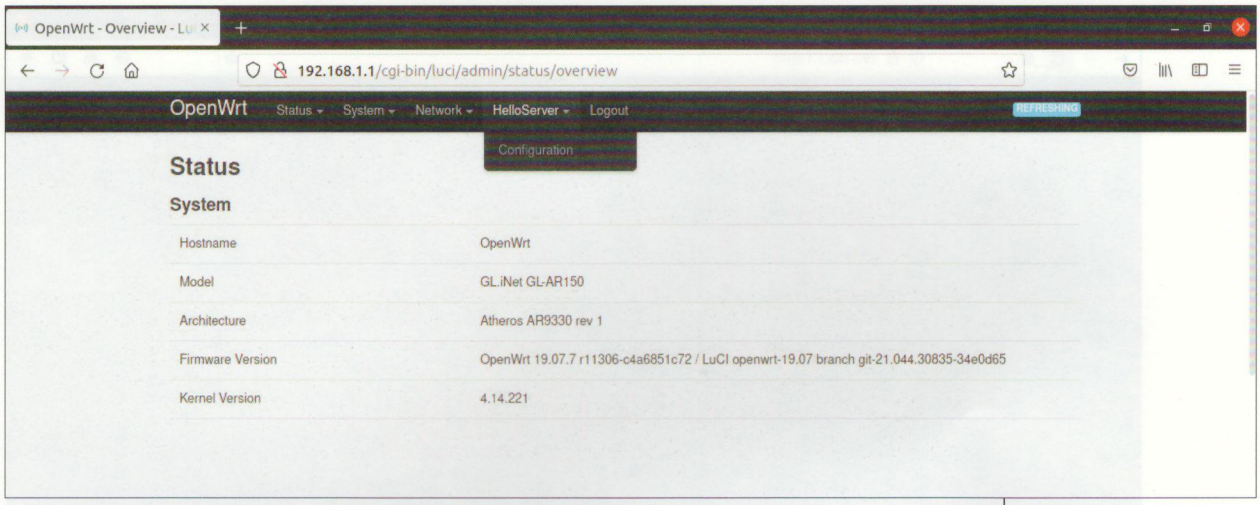
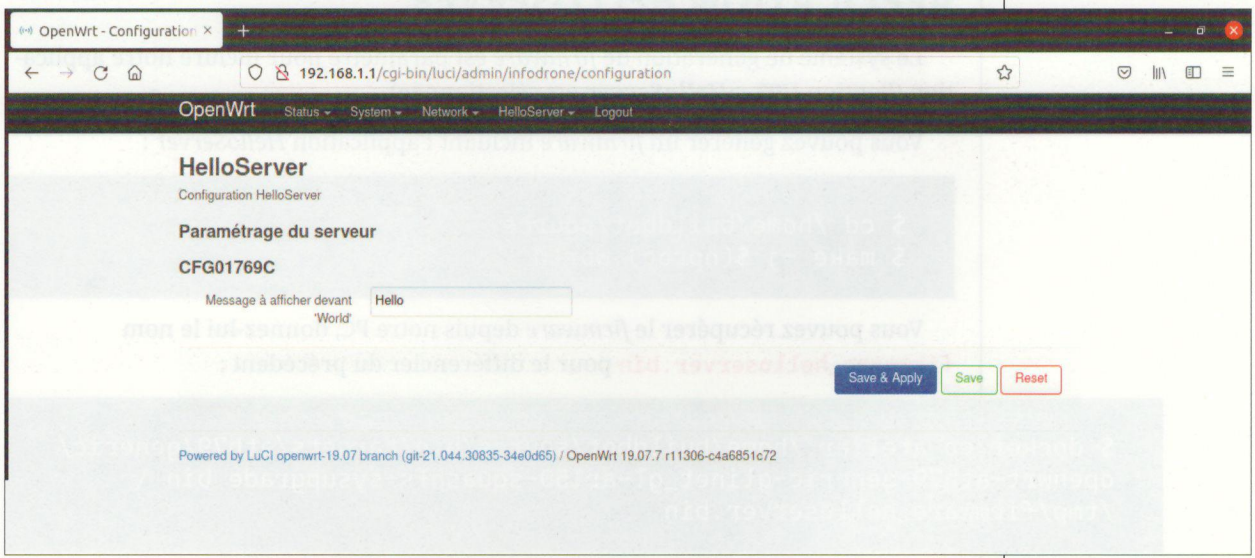


Figure 8

Dans l'interface de configuration LuCI, un nouveau menu ***HelloServer*** et un menu ***Configuration*** sont apparus (Figure 8).

Si vous sélectionnez ***HelloServer/Configuration***, vous obtenez un écran permettant de modifier la salutation affichée devant le mot « `World` », mettez par exemple « `Goodbye` » et cliquez une fois sur ***Save & Apply***, puis cliquez une deuxième fois sur ***Save & Apply*** :

Figure 9



Un bug est présent dans LuCI qui nécessite de cliquer 2 fois successives sur ***Save & Apply***.

Si vous actualisez la page « *Hello World* », vous devez maintenant voir s'afficher « `Goodbye World!` » (Figure 10, page suivante).

Figure 10

12. GÉNÉRATION D'UN FIRMWARE AVEC NOTRE APPLICATION HELLOSERVER

Le système de génération de *firmware* est paramétré pour inclure notre application (l'option ***U03-->HelloServer*** est sélectionnée).

Vous pouvez générer un *firmware* incluant l'application *HelloServer* :

```
$ cd /home/buildbot/source
$ make -j $(nproc) world
```

Vous pouvez récupérer le *firmware* depuis notre PC, donnez-lui le nom `firmware_helloserver.bin` pour le différencier du précédent :

```
$ docker cp atelier:/home/buildbot/source/bin/targets/ath79/generic/
openwrt-ath79-generic-glinet_gl-ar150-squashfs-sysupgrade.bin \
/tmp/firmware_helloserver.bin
```

Vous pouvez maintenant charger le *firmware* de la même manière que précédemment, installez-le sans conserver les paramètres (veillez à décocher la case ***Keep settings and retain the current configuration***).

Une fois que le routeur a *rebooté*, si vous actualisez la page **http://192.168.1.1:8080/**, vous devriez à nouveau voir « `Hello World!` ».

13. SPÉCIFICITÉS DES ÉQUIPEMENTS

13.1 Le chien de garde

Un chien de garde (ou « *watchdog* ») est un mécanisme qui permet de détecter un blocage du système ou d'une application. C'est un minuteur qui active le signal « *reset* » du processeur une fois la temporisation échue. Le système ou l'application doivent réinitialiser régulièrement le minuteur pour qu'il n'arrive pas à échéance. En cas de blocage, le système est réinitialisé de façon *hardware* par le *watchdog*.

Physiquement, il peut être intégré sur la puce du processeur, il peut aussi s'agir d'un petit composant distinct, il en existe également connecté en USB dans le cas où il n'y en a pas d'intégré dans le système.

OpenWrt gère le *watchdog hardware* si l'équipement en dispose, un *process* `watchdog` est lancé au démarrage de l'OS, il réinitialise le *watchdog* physique à intervalles réguliers.

Notre application de passerelle IoT nécessite pour fonctionner de placer l'interface Wi-Fi en « mode *monitor* », sans que l'on sache pourquoi le « mode *monitor* » arrête de fonctionner de façon aléatoire.

Pour limiter l'impact d'une telle désactivation intempestive, nous désactivons la gestion du *watchdog* par l'OS, et notre programme se charge de réinitialiser le *watchdog* lorsqu'il traite un paquet Wi-Fi.

Ci-dessous, la commande pour désactiver la gestion du *watchdog* par l'OS (l'équipement va *rebooter* au bout de quelques dizaines de secondes, si l'application ne prend pas le relais pour la gestion du *watchdog*) :

```
ubus call system watchdog '{"magicclose":true}'
ubus call system watchdog '{"stop":true}'
```

Extrait de code C :

```
#include <linux/watchdog.h>
#include <sys/ioctl.h>

// Ouverture du watchdog
//
int    watchdog_fd;
watchdog_fd = open("/dev/watchdog", O_RDWR);

   .../...

// Réinitialisation du watchdog lors de la réception d'une trame Wi-Fi.
//
if(ioctl(watchdog_fd, WDIOC_KEEPALIVE, NULL)) {
    cerr << "Erreur Watchdog WDIOC_KEEPALIVE\n" << endl;
}
```

13.2 L'horloge temps réel

Les PC et les serveurs disposent sur la carte mère d'une puce RTC (*Real Time Clock*) alimentée par une pile bouton, elle est utilisée pour récupérer l'heure courante lors du lancement de l'OS, ceci permet d'avoir une heure « à peu près correcte » en attendant qu'un système tel que NTP se charge d'obtenir l'heure exacte depuis un serveur de temps centralisé ou localement en utilisant le GNSS ou un récepteur DCF-77.

Nos équipements ne disposent pas de RTC (tout comme la Raspberry Pi), et donc l'heure au lancement de l'OS n'est pas correcte, habituellement les OS se lancent avec une heure au 01/01/1970 à 00:00. OpenWrt utilise la date de modification la plus grande dans le répertoire `/etc` comme date initiale.

Mettre une date « relativement plausible » peut sembler une bonne idée, mais en fait en attendant que la synchro de temps soit effectuée, il peut être utile pour l'application de savoir que la date n'est pas correcte.

Le script `/etc/init.d/sysfixtime` se charge de placer cette heure « plausible » au lancement de l'OS, il fait partie du *package* `base-files`, les commandes suivantes permettent de supprimer le script du *package* juste avant de générer le *firmware* :

```
# Supprimer sysfixtime
#
rm -f /home/buildbot/source/package/base-files/files/etc/init.d/sysfixtime
make package/base-files/clean
```

Désormais, si la date est inférieure à 2021, l'application sait que l'horloge n'est pas encore correcte et elle peut adapter ses traitements.

13.3 La 4G

Certains équipements disposent de la 4G, il est conseillé de prendre un équipement utilisant un module au format Mini PCIe, il faut également vérifier qu'il utilise bien les bandes de fréquences utilisées dans le pays. Certains modules fournissent en plus le GNSS (positionnement par satellites) si l'on ajoute une antenne GPS.

Si vous utilisez une carte SIM d'un opérateur téléphonique classique, vous aurez un abonnement mensuel d'une dizaine d'euros à payer, votre forfait se comptera peut-être en giga-octets, par contre vous aurez une adresse IP variable. Si vous avez plusieurs équipements, ils devront passer par un serveur pour communiquer entre eux et vous devrez utiliser un VPN pour vous connecter dessus à distance.

Les opérateurs IoT fournissent un parc de cartes SIM qui ont des adresses IP fixes dans leur propre réseau privé (par contre, elles partagent toute la même adresse IP internet). Ainsi avec un routeur 4G utilisant une SIM de votre parc, vous pouvez vous connecter directement à chacun des équipements de votre parc. Ce type d'opérateurs est réservé aux « entreprises ».

Par exemple, l'opérateur 1NCE propose pour 10 € des cartes SIM valables 10 ans, sans abonnement mensuel, avec un total de 500 Mo à utiliser (ou 250 SMS) pendant la durée de vie de la carte. Il est possible de rajouter du crédit à une carte SIM de façon manuelle ou automatique. L'avantage est qu'il s'appuie sur les accords d'itinérance de Deutsche Telekom, le crédit est utilisable mondialement sans frais supplémentaires et en France, votre équipement se connectera indifféremment aux réseaux d'Orange, Bouygues ou SFR en fonction de la couverture réseau locale.

Par contre, il faut économiser les octets échangés (c'est général avec de l'IoT, les octets coûtent cher, en euro ou en pourcentage de batterie).

Pour ceci, deux solutions sont possibles :

- paramétrer le *firewall* de l'équipement pour n'autoriser que les flux applicatifs utiles (et éventuellement les flux de maintenance) ;
- éviter l'activation de services tels que NTP, DNS, etc. qui consomment du réseau (certes peu, mais c'est tout de même non négligeable sur la durée).

CONCLUSION

Initialement, il semblait difficile de passer d'une application Python sur Raspberry Pi utilisant quelques centaines de méga-octets de bibliothèques Python à une application écrite en C/C++ sur un système nécessitant de générer un *firmware* personnalisé tenant dans 16 Mo de mémoire flash, application comprise.

Finalement, OpenWrt est bien documenté et l'opération a été plus simple que prévu (même si je dois admettre qu'à force de faire du Python, j'étais quelque peu rouillé en C/C++).

Finalement, j'ai retenu 3 modèles en fonction du cas d'usage :

- GL.iNet GL-AR150 : un modèle à moins de 30 €, idéal pour le labo et organiser des démonstrations.
- TP-Link EAP225-Outdoor : un modèle étanche fonctionnant en extérieur, alimenté en PoE passif (à l'aide d'un injecteur fourni) ou en PoE actif sur un *switch* PoE classique.
- GL.iNet GL-X300B : un modèle en boîtier métallique supportant 4G et GNSS et une alimentation de 9 à 35 volts.

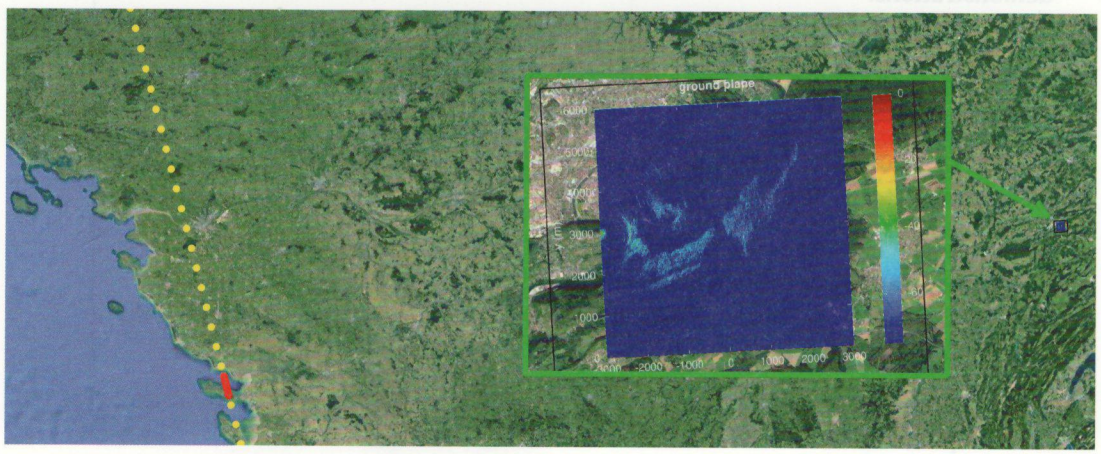
L'application et l'OS se révèlent stables, l'utilisation du *watchdog* permet de relancer le système quand l'application ne semble plus fonctionner, l'utilisation d'un système de fichiers en lecture seule doublée d'un *overlay* évite le risque de corruption du système de fichiers.

Avoir un équipement comprenant tout le nécessaire dans un boîtier professionnel est aussi un plus. LF

RADAR PASSIF BISTATIQUE
AU MOYEN D'UNE RASPBERRY PI 4, D'UNE RADIO LOGICIELLE ET DU SATELLITE SENTINEL-1

J.-M FRIEDT
FEMTO-ST, département temps-fréquence, Besançon, France
& W. FENG
Early Warning and Detection Department, Air Force Engineering University, Xi'an, Chine

Nous exploitons les signaux émis à intervalles connus et documentés sur le site Copernicus de l'ESA par les satellites Sentinel-1 pour une mesure au sol de RADAR passif bistatique. Deux antennes au sol, une antenne de référence qui observe le signal direct émis par le satellite, et une seconde antenne de surveillance qui observe les réflexions par les cibles illuminées par le satellite sont connectées à un récepteur de radio logicielle pour collecter à 5405 MHz les signaux de Sentinel-1. La détection de cibles à plusieurs kilomètres du récepteur est démontrée avec un système simple composé d'une radio logicielle Ettus Research B210 et d'une Raspberry Pi 4 programmée efficacement. La diversité spatiale introduite par le mouvement du satellite le long de son orbite permet de cartographier les cibles en distance et en azimut.



– RADAR passif bistatique au moyen d'une Raspberry Pi 4, d'une radio logicielle et du satellite Sentinel-1 –

Le détournement de signaux radiofréquences à des fins autres que ceux pour lesquels ils sont initialement prévus (un *hack* dans le sens le plus noble du terme) a récemment été diffusé dans l'« actualité » avec l'utilisation de la constellation de satellites en orbite basse Starlink pour la géolocalisation. Malheureusement, cette prouesse technique est associée à deux drames pour la science : la diffusion du résultat technique au travers d'un communiqué de presse aussi concis que dénué de contenu scientifique rigoureux **[1]**, et sa reprise par des journalistes que certains n'hésiteraient pas à qualifier d'aussi incompétents que stupides **[2]** puisque capables d'affubler la prouesse technique d'adjectifs tels que « piratage » et « absence de gain financier pour Starlink » alors que la constellation n'avait jamais été conçue pour la géolocalisation. On ne pourra que regretter que le compte rendu technique ne soit pas librement disponible **[3]**, mais seul un résumé de quelques lignes qui nous donne l'envie de découvrir les techniques mises en œuvre : patientons un peu le temps que l'article soit libéré des éditeurs commerciaux **[4]**. Nous allons ici aborder le « piratage » – ou en termes plus nuancés l'exploitation passive de signaux radiofréquences issus de RADAR spatioportés – pour une détection de position de réflecteurs au sol en exploitant le signal brut (analogique) reçu directement du satellite et après réflexion par des cibles au sol. Cette application est surprenamment peu commune puisque nous avons trouvé une unique référence bibliographique **[5]** abordant ce problème.

Nous avons décrit dans ces pages la « bonne » façon de développer sur Raspberry Pi 4 et en particulier des applications gourmandes en ressources telles que le traitement logiciel de signaux radiofréquences grâce à Buildroot **[6]** qui permet de ciseler l'exécutable aux ressources disponibles. Nous avons décrit dans ces pages comment Sentinel-1 **[7]**, RADAR spatioporté de l'ESA, observe la surface de la Terre en l'illuminant autour de 5,405 GHz et comment ces données sont mises à disposition pour en tirer le meilleur parti, éventuellement au-delà des applications initialement prévues. Nous avions déjà abordé les préceptes du RADAR bistatique passif (Fig. 1) dans lequel un émetteur non coopératif, généralement émettant un signal bien plus puissant que ce que le civil amateur moyen aurait le droit de diffuser, est observé sur une voie dite de référence tandis qu'une seconde voie de mesure radiofréquence dite de surveillance observe les réflexions retardées dans le temps (distance) et décalées en fréquence par effet Doppler (vitesse) des cibles. Nous allons fusionner l'ensemble de ces connaissances en exploitant une Raspberry Pi 4 pour enregistrer les signaux radiofréquences d'un récepteur de radio logicielle à deux voies cohérentes (cadencées par le même oscillateur fc pour garantir la même phase et donc fréquence) afin de traiter les signaux bruts de Sentinel-1 pour une application de RADAR passif (Fig 1).

Il est en effet tout à fait fascinant de non seulement constater qu'il est relativement « simple » de capter un signal venant de l'espace

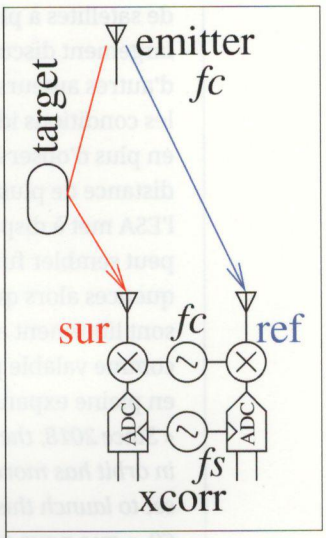


Figure 1 : Architecture d'un récepteur de RADAR passif avec une voie de référence (« ref ») et une voie de mesure des signaux réfléchis par les cibles, dite de surveillance (« sur »).

de satellites à plus de 700 km de distance – nous l'avions déjà largement discuté avec les satellites météorologiques **[8, 9]** et d'autres auteurs dépassent les 300 millions de kilomètres dans les conditions idéales de propagation en espace libre **[10]** – mais en plus d'observer les réflexions de ces signaux au sol à une distance de plusieurs kilomètres du récepteur. Bien entendu, l'ESA met à disposition les mesures de Sentinel-1 et l'exercice peut sembler futile – pourquoi détourner des signaux radiofréquences alors que les « vraies » mesures reçues par le satellite sont librement accessibles – mais nous considérons cet exercice comme valable pour tous les RADAR spatioportés, une faune en pleine expansion si l'on en croit **[11]** qui nous informe que « *Since 2018, the number of civil and commercial SAR satellites in orbit has more than doubled. And at least a dozen more are set to launch this year, which would bring the total to more than 60.* » que nous confirme notamment l'annonce du lancement de NISAR, le projet indo-américain qui semble s'aligner sur la distribution libre des données acquises **[12]**.

1. OBJECTIFS DE L'ACQUISITION DE DONNÉES

Sentinel-1 émet en bande C, et en attendant le lancement de NISAR (prévu 2023) ou Sentinel-12 (peut-être en 2027) qui émettront en bande L en deçà de 2 GHz, il faut utiliser un récepteur fonctionnant au-dessus de 5 GHz pour recevoir ses signaux analogiques. Pour le moment, seuls les récepteurs de radio logicielle équipés de détecteur (*frontend*) Analog Devices AD936x et dont les deux voies sont câblées seront utilisables : nous exploiterons le récepteur Ettus Research B210 **[13, 14]** à ces fins, même si l'E312 plus cher a aussi été démontré fonctionnel dans de telles applications **[15]**, tandis que la PlutoSDR de Analog Devices ne proposant qu'une unique voie ne peut être utilisée, et les récepteurs à base de Lime Microsystems LMS7002 étant limités à 3 GHz ne seront pas utilisables pour l'application qui nous intéressera ici. Par ailleurs, les signaux de Sentinel-1 occupent une bande passante de 100 MHz, imposant d'échantillonner les signaux au maximum de la bande passante disponible sur le récepteur de radio logicielle. Une bande passante de 100 MHz suppose un bus de communication de plus de 400 Mb/s en continu si toute la résolution (deux octets) des deux voies doit être conservée. Peu d'interfaces embarquées permettent un tel débit – embarquées, car nous verrons que les sites intéressants de réception sont souvent isolés et sans infrastructure pour l'alimentation d'un poste fixe – et nous choisissons le port USB 3 de la Raspberry Pi 4 pour atteindre le maximum de la bande passante de la B210 de 30 Méchantillons/s.

Le scénario est donc ficelé : acquérir aussi rapidement que possible un flux de données de deux voies d'une radio logicielle B210 au moyen d'un ordinateur Raspberry Pi 4 en vue de transférer les données vers un ordinateur portable pour du post-traitement visant à identifier les cibles illuminées par Sentinel-1 lors de son survol. Notons que l'heureux possesseur d'un ordinateur portable équipé d'un port USB 3 pourra s'affranchir de l'intermédiaire de la Raspberry Pi 4, mais 1) ce n'est pas le cas de notre Panasonic CF19 qui n'est équipé que de ports USB 2 et 2) il est intéressant d'appliquer les préceptes du développement de systèmes embarqués à ce projet, ne serait-ce que dans un objectif de mesure long terme autonome en énergie sur un site isolé. En effet, la Raspberry Pi 4 et la B210 consomment à pleine vitesse de 1,55 GHz globalement 1,55 A sous 5 V, tandis qu'une acquisition dure 1 minute : il est donc envisageable d'effectuer environ 85 mesures sur une batterie de

– RADAR passif bistatique au moyen d'une Raspberry Pi 4, d'une radio logicielle et du satellite Sentinel-1 –

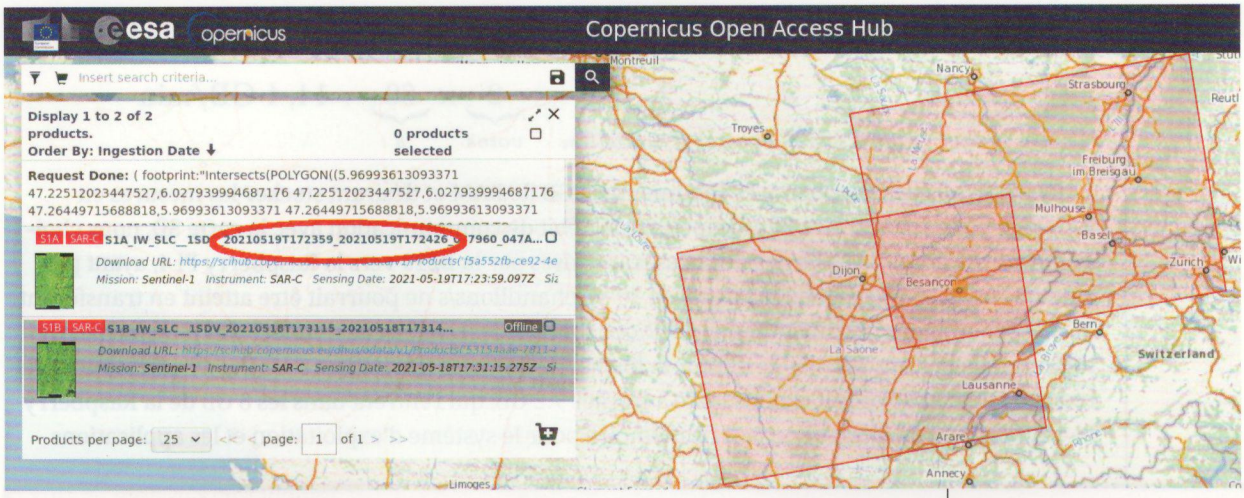


Figure 2 : Le site Copernicus de l'ESA permet de rechercher pour un site géographique les dates et horaires de passage passés des satellites Sentinel-1. Sachant que leur orbite est conçue pour se répéter exactement tous les 12 jours, nous déduisons de la nomenclature des fichiers l'horaire de début et de fin de réception du signal au sol pour l'emplacement du récepteur, ici Besançon. Ce passage, le 19 mai 2021 de 17 h, 23 min et 59 sec à 17 h, 24 min et 26 secondes, se répétera au même horaire, à quelques secondes près, le 31 mai, puis le 12 juin, etc.

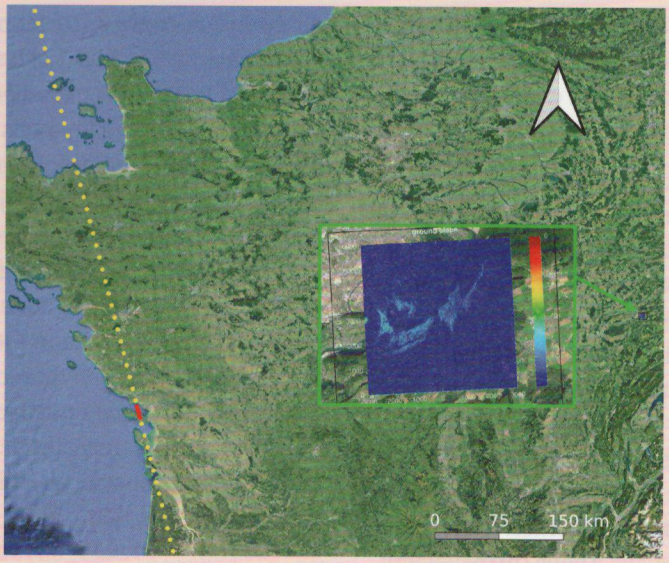
capacité 2200 mA h telle que nous avons utilisée ici, excluant le traitement embarqué ou le stockage.

Les deux satellites Sentinel-1 survolent tout point de la Terre selon la même orbite tous les 12 jours, mais dans la pratique aux latitudes de la France métropolitaine, nous pouvons espérer observer un passage tous les jours ou tous les deux jours, selon des orbites et donc une élévation différentes. Alors que Heavens Above (**https://www.heavens-above.com**) nous informe qu'un passage de Sentinel-1 d'horizon à horizon dure 9 minutes, l'observation des signaux radiofréquences démontre que Sentinel-1 n'illumine un point donné de la surface de la Terre que pendant quelques secondes, et son mode de balayage de faisceau (*swath*) séquencé en impulsions successives (*burst*) ne fournira finalement que des séquences d'une fraction de seconde exploitables. Une différence majeure entre les observations systématiques de Sentinel-1 et les mesures ponctuelles de RADARSAT canadien, TerraSAR-X allemand ou ALOS-2 japonais est que le lieu et la date de passage sont prédictibles et se déduisent des observations publiées 12 jours auparavant. Nous pouvons donc connaître à la seconde près (Fig. 2) la date de début et de fin du passage du satellite au-dessus d'une zone géographique donnée : cette information est fortement dépendante de la localisation du récepteur et doit être répétée pour chaque nouveau site d'écoute. Ainsi, la nomenclature des fichiers bruts obtenus sur le site Copernicus Hub de l'ESA est de la forme `S1A_IW_RAW__0SDV_20210519T172356_20210519T172429_037960_047AF4_8DCD` pour le fichier brut de niveau 0 qui indique une acquisition le 19 mai 2021 de 17:23:56 à 17:24:29 UTC donc une durée de 33 s, ou le fichier traité de niveau 1 IW SLC (*Interferometric Wide – Single Look Complex*) `S1B_IW_SLC__1SDV_20210705T173118_20210705T173144_027662_034D27_3FBE` indique une acquisition plus ciblée le 7 mai 2021 de 17:31:18 à 17:31:44, soit une durée de 16 s. Cependant, compte tenu de l'incertitude sur l'exactitude de l'horloge locale et la durée du lancement de l'acquisition, il est prudent d'acquérir une minute de données et d'extraire les informations utiles *a posteriori*. Au débit de 30 Méchantillons/s complexes sur deux voies, des données sur 16 bits occupent :

$$\underbrace{30}_{débit} \times \underbrace{2}_{complexe} \times \underbrace{2}_{octets/échantillon} \times \underbrace{2}_{voies} \times \underbrace{60}_{s} = 14,4 \text{ GB/min}$$

Ayant acquis des Raspberry Pi 4 munies de 8 GB de RAM et désirant stocker les mesures en RAMdisk pour éviter d'être handicapés par le débit de communication avec la carte SD, nous choisissons de tronquer la taille des données transmises par la B210 vers la Raspberry Pi 4, ayant par ailleurs constaté que le débit maximal de 30 Méchantillons/s ne pourrait être atteint en transférant 16 bits/échantillons : la bibliothèque UHD qui contrôle le transfert de données des plateformes de radio logicielle Ettus permet de définir le format de données transmises *over the wire* (otw) et de le réduire à 8 bits. De ce fait, il nous reste à stocker 7.2 GB, qui rentrent dans les 8 GB de la Raspberry Pi 4 appropriée en laissant un peu de mémoire pour le système d'exploitation et les applications.

Il peut être intéressant de se demander où se trouve le satellite lorsqu'il illumine une région donnée de la Terre. En effet avec une illumination oblique à environ 45°, le satellite à une altitude de 700 km survole à une distance de 700 km à l'est ou à l'ouest selon que son passage soit ascendant ou descendant par rapport à la zone illuminée. À titre d'exemple lors de l'illumination de Besançon au cours d'un passage ascendant, le satellite survole l'ouest de la France tel que nous l'indique le *Ground Track Generator* de **https://github.com/anoved/Ground-Track-Generator**, avec un signal électromagnétique qui aura parcouru 700/cos(45°)=980 km avant d'atteindre sa cible.



2. RASPBERRY PI 4 POUR ACQUISITION « RAPIDE »

Ettus Research propose, sur le dépôt de la bibliothèque UHD **https://github.com/EttusResearch/uhd/tree/master/host/examples**, des programmes d'exemple et notamment `rx_multi_samples.cpp` qui ne travaille cependant que sur une voie et transfère des données sur 16 bits. Ainsi partant de cet exemple, nous dupliquons toutes les définitions des propriétés des canaux pour passer d'une à deux voies, et indiquons à UHD de transmettre des données en format complexe 8 bits par :

```
uhd::stream_args_t stream_args(string"sc8",string"sc8");
stream_args.channels             = channel_nums;
uhd::rx_streamer::sptr rx_stream = usrp->get_rx_stream(stream_args);
```

Ainsi que de communiquer les informations en flux tendu et non sous forme d'un unique paquet de données :

```
uhd::stream_cmd_t stream_cmd(uhd::stream_cmd_t::STREAM_MODE_START_CONTINUOUS);
```

Ainsi, nous recevrons deux flux de données que nous stockons aussi vite que possible dans deux fichiers qui seront traités comme les deux voies d'observation synchrones puisque les convertisseurs analogiques numériques sont cadencés par la même horloge f_s (Fig. 1), l'une issue d'une antenne pointée vers le ciel qui fournit le signal de référence illuminant la scène observée à la surface de la Terre, et la seconde pointant vers cette scène pour observer les cibles réfléchissant les signaux radiofréquences. Nous gagnerons en rapport signal à bruit en nous efforçant d'isoler l'antenne de mesure du signal de référence selon le montage proposé en Fig. 3 où le plan de masse de l'antenne hélicoïdale observant le signal de référence cache autant que possible l'antenne de mesure du signal direct. L'implémentation finale est disponible à **https://github.com/jmfriedt/sentinel1_pbr/blob/main/b210_to_file/rx_multi_samples.cpp**.

On pensera à passer la Raspberry Pi 4 en mode performance pour fournir la puissance de calcul nécessaire à l'acquisition de données :

```
echo performance > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

Gauche : antenne hélicoïdale formée d'un fil de cuivre émaillé entouré sur un cylindre de teflon creux usiné d'une gorge de la forme appropriée. Droite : le cylindre de teflon est taraudé afin d'accueillir une vis en plastique pour le maintenir plaqué au plan de masse du circuit imprimé en FR4. Un connecteur SMA est soudé au plan de masse pour faire le lien entre l'antenne et le récepteur B210.

Bien que nous ayons déjà développé les principes de conception de l'antenne hélicoïdale, nous en reprenons ici les grandes lignes par souci d'autonomie de cette présentation, selon les consignes fournies dans **[16**, section 10.3.1**]**.

Chaque spire plaquée sur un cylindre de diamètre D doit présenter une longueur $C=\pi D$ comprise entre 3/4 et 4/3 de la longueur d'onde, dans notre cas $\lambda=300/5450$ m = 5,55 cm donc $D\in[1{,}32:2{,}36]$ cm. L'angle

– RADAR passif bistatique au moyen d'une Raspberry Pi 4, d'une radio logicielle et du satellite Sentinel-1 –

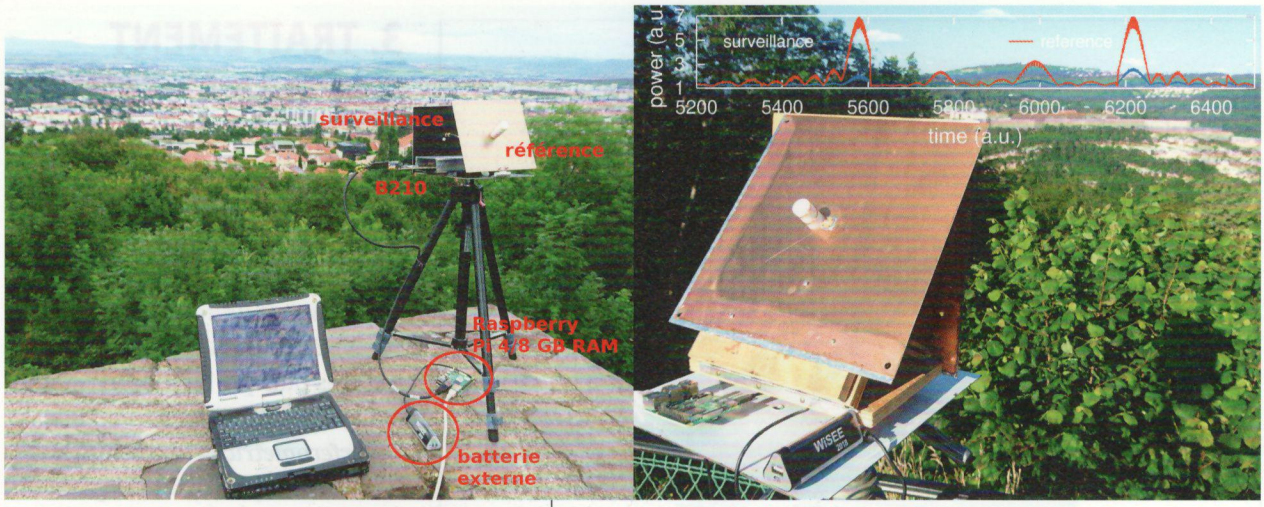


Figure 3 : Gauche : le système de mesure déployé au-dessus de Clermont-Ferrand, sur le point de vue menant au Puy-de-Dôme. Le port USB de l'ordinateur CF19 fournit un courant insuffisant pour alimenter la Raspberry Pi 4 et le récepteur B210 qui lui est connecté sur port USB 3 : une batterie externe est utilisée pour alimenter ces deux dispositifs. Le plan de masse de l'antenne de référence s'efforce de cacher l'antenne de surveillance du signal direct. Droite : un système de mesure identique déployé devant la Citadelle de Besançon depuis le Fort Chaudanne.

Buildroot plaçant par défaut le processeur en économie d'énergie en le cadençant à 800 MHz au lieu des 1500 MHz du mode performance. Une fois assemblé, le système est transportable et prêt pour l'acquisition de données (Fig. 3).

de la spirale doit être compris entre 12 et 14° : des contraintes d'usinage au tour imposent un espacement entre spires de 1 cm et en choisissant un diamètre standard de D=1,5 cm, une gorge de la profondeur du diamètre du fil émaillé qui fera office de conducteur (0,7 mm dans notre cas) est usinée avec un pas de 1 cm donc un angle des spires de 12°. Ces caractéristiques géométriques déterminent l'impédance $Z=140 \times C/\lambda \simeq 140\ \Omega$ et au mieux 100 Ω et nécessiterait une adaptation d'impédance par ligne de transmission se comportant comme transformateur pour correctement coupler avec l'entrée 50 Ω du récepteur B210 : nous accepterons de perdre une fraction de la puissance incidente en connectant directement une embase SMA en bout de fil pour relier l'antenne au récepteur de radio logicielle. Le connecteur SMA est soudé sur une plaque carrée de FR4 de 16 × 16 cm faisant office de plan de masse de dimensions importantes devant la longueur d'onde. L'ouverture angulaire (moitié de la puissance rayonnée) de l'antenne est annoncée comme $52\lambda 3/2/C\sqrt{NS}$ degrés avec N le nombre de tours, dans notre cas N=4 pour une ouverture de 72°, un angle important qui est particulièrement utile pour l'antenne observant les réflexions sur la scène s'étendant devant elle et pour laquelle il est désirable de recevoir les signaux provenant de toutes les directions. **https://www.antenna-theory.com/antennas/travelling/helix.php** prédit une ouverture angulaire de 90° selon une formule quelque peu différente toujours inversement proportionnelle à la racine du nombre de spires, et un gain de 3 ou 5 dB, modeste mais nécessaire, compte tenu de la large ouverture angulaire nécessaire à visualiser l'ensemble de la scène devant l'antenne de surveillance.

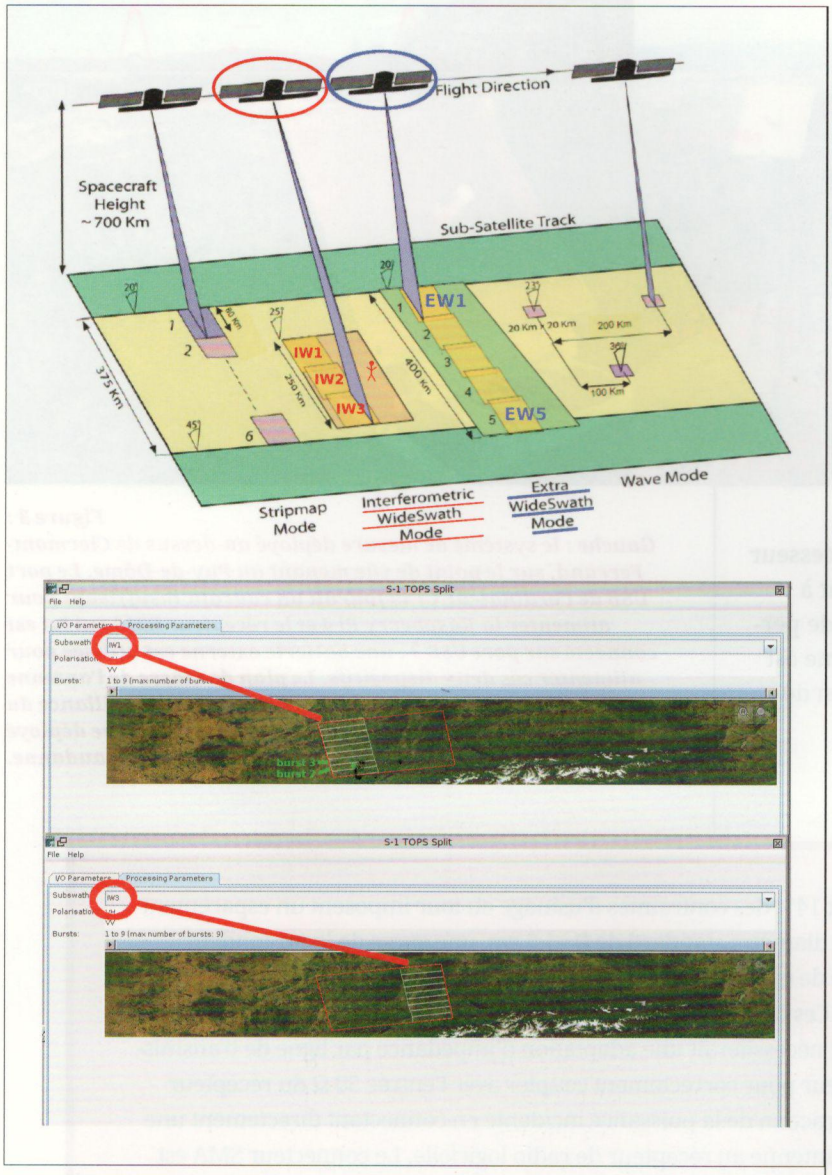


Figure 4 :
Les divers modes de fonctionnement de Sentinel-1 (haut), avec l'illustration des IW et EW qui nous seront utiles ici. La problématique tient au fait que les divers faisceaux IW1 à IW3 ou EW1 à EW5 présentent des paramètres d'illumination différents qu'il nous faut identifier, ici par exemple si la cible matérialisée par le bonhomme rouge se trouve en IW2. Le logiciel SNAP de l'ESA (bas) peut fournir une aide pour identifier quel faisceau a illuminé quelle zone (interface graphique du traitement Split du menu TOPSAR). Illustration annotée de https://sentinels.copernicus.eu/web/sentinel/technical-guides/sentinel-1-sar/sar-instrument/acquisition-modes.

3. TRAITEMENT SUR UNE ANTENNE UNIQUE : TAUX DE RÉPÉTITION DES IMPULSIONS

Une scène illuminée au sol peut recevoir des signaux de diverses natures puisque Sentinel-1 balaie trois faisceaux (*swath*) en mode IW (*Interferometric Wide*) utilisé aux latitudes moyennes et cinq faisceaux en mode EW (*Extra Wide*) utilisé aux latitudes élevées et au-dessus des océans, un cas qui nous intéresse lors de l'utilisation de ce système au Spitzberg (79°N) où les satellites en orbite polaire sont visibles deux fois chaque jour, augmentant les chances de mesures répétées (Fig. 4). Ces divers faisceaux ne sont pas caractérisés par les mêmes paramètres d'émission et en particulier le taux de répétition des impulsions radiofréquences émises par le RADAR : PRI pour *Pulse Repetition Interval*.

Alors que le décodage des signaux de niveau 0 fournis par l'ESA **[20]** nous informe des diverses valeurs possibles de PRI, nous ne savons pas *a priori* quel faisceau illumine l'antenne au sol à un instant donné. Soit nous testons tous les cas possibles (Fig. 5), soit une autocorrélation du signal directement reçu depuis le satellite (dit de référence) nous fournira cette information qui sera nécessaire pour découper la séquence continue de mesure dans le

– RADAR passif bistatique au moyen d'une Raspberry Pi 4, d'une radio logicielle et du satellite Sentinel-1 –

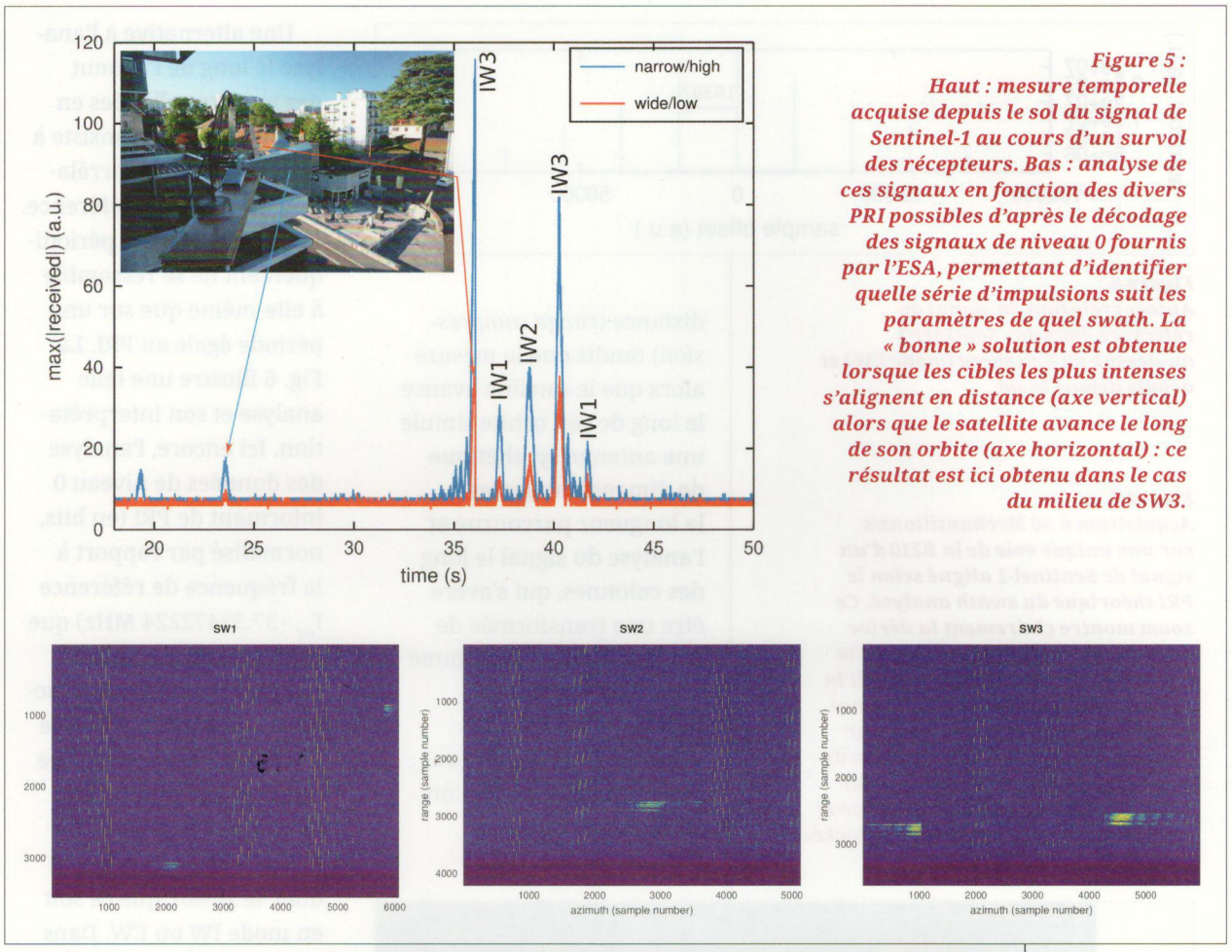


Figure 5 : Haut : mesure temporelle acquise depuis le sol du signal de Sentinel-1 au cours d'un survol des récepteurs. Bas : analyse de ces signaux en fonction des divers PRI possibles d'après le décodage des signaux de niveau 0 fournis par l'ESA, permettant d'identifier quelle série d'impulsions suit les paramètres de quel swath. La « bonne » solution est obtenue lorsque les cibles les plus intenses s'alignent en distance (axe vertical) alors que le satellite avance le long de son orbite (axe horizontal) : ce résultat est ici obtenu dans le cas du milieu de SW3.

temps en échos observés suite à chaque illumination. Heureusement, les PRI associés à chaque *swath* restent constants d'un survol à l'autre et il n'est pas nécessaire de systématiquement décoder tous les fichiers de niveau 0 pour trouver cette information : nous avons retrouvé les mêmes paramètres au cours de tous les survols étudiés dans ce document, avec un PRI de 22777 pour *swath* 14 (EW1), 19355 pour *swath* 15 (EW2), 22779 pour *swath* 16 (EW3), 19777 pour *swath* 17 (EW4) et 23018 pour *swath* 18 (EW5), ou encore 21859 pour IW1, 25857 pour IW2 et 22265 pour IW3 (ces paramètres doivent être divisés par la fréquence de référence de 37.53472224 MHz pour être convertis en intervalle temporel, par exemple 1647.9 Hz pour EW1). Ainsi, connaissant la fréquence d'échantillonnage f_s, nous découperons le signal temporel acquis par chaque canal en matrice de $M = f_s \times$ PRI points par ligne et autant de colonnes que le permet la durée de la mesure. Cette matrice contiendra donc l'information temporelle selon l'abscisse qui se répète le long des colonnes alors que le satellite avance sur son orbite. Nous avons largement discuté dans ces pages que la corrélation le long de l'abscisse permettra de retrouver la distance aux cibles et se nomme la compression en

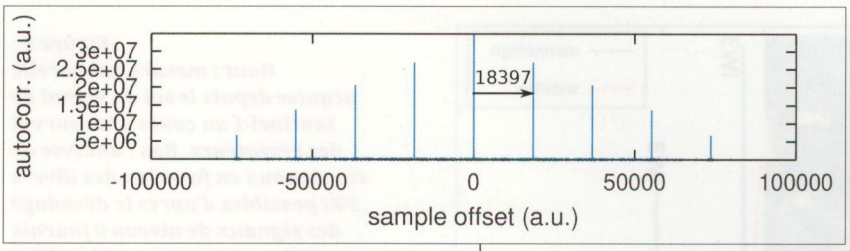
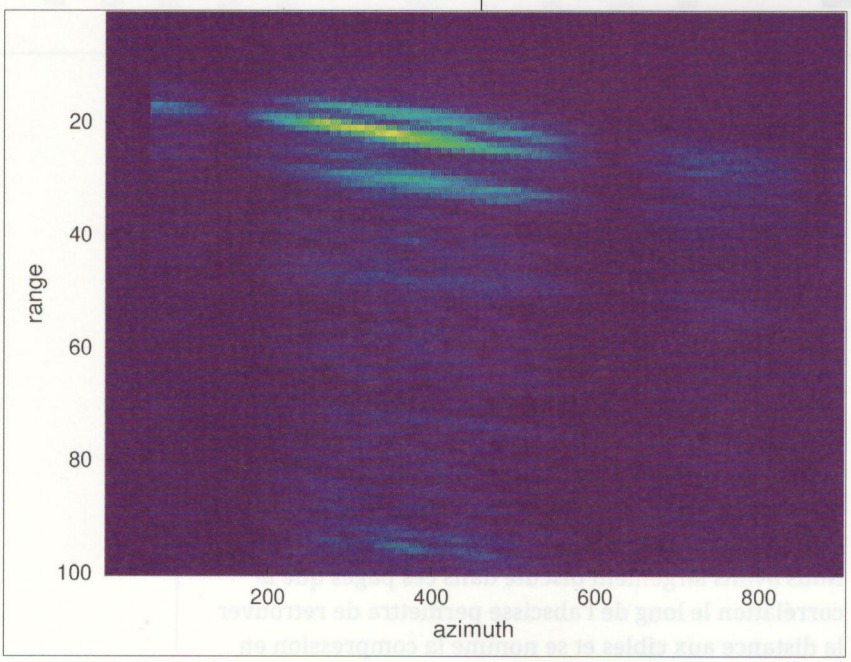


Figure 6 : Autocorrélation du signal de référence émis par Sentinel-1 au-dessus du Spitzberg (mode EW) et acquis depuis le sol.

Figure 7 : Acquisition à 56 Méchantillons/s sur une unique voie de la B210 d'un signal de Sentinel-1 aligné selon le PRI théorique du swath analysé. Ce zoom montre clairement la dérive de la distance entre le satellite et la cible intense observée qui interdit la compression en azimut en l'absence de correction qui sera amenée par l'acquisition sur une seconde voie du signal de référence, diminuant par conséquent la vitesse d'acquisition et doublant le volume de données stockées à analyser.

distance (*range compression*) tandis que la mesure alors que le satellite avance le long de son orbite simule une antenne synthétique de dimensions égales à la longueur parcourue et l'analyse du signal le long des colonnes, qui s'avère être une transformée de Fourier inverse, se nomme la compression en azimut (*azimuth compression*). Nous reprendrons ces points dans la section suivante.



Une alternative à l'analyse le long de l'azimut des mesures alignées en distance (*range*) consiste à considérer l'autocorrélation du signal de référence. L'impulsion émise périodiquement ne se ressemble à elle-même que sur une période égale au PRI. La Fig. 6 illustre une telle analyse et son interprétation. Ici encore, l'analyse des données de niveau 0 informant de PRI (en bits, normalisé par rapport à la fréquence de référence f_{ref} = 37.53472224 MHz) que nous avions entreprise auparavant, couplée à l'acquisition à une fréquence d'échantillonnage connue de 30 MHz, nous permet de trouver la période T des pics de corrélation et donc le *swath*, que ce soit en mode IW ou EW. Dans cet exemple, la mesure au cours d'un passage au-dessus du Spitzberg en mode EW du satellite indique une autocorrélation tous les 18397 échantillons acquis à 30 Méchantillons/s soit par rapport à f_{ref} un PRI de 23018 qui s'avère égal à la valeur de la télémétrie du faisceau EW5 que nous avons décodé sur `S1B_EW_RAW__0SDH_20210925T063129_20210925T063237_028851_037165_3786.SAFE` du 25 septembre 2021.

Nous pourrions déduire de cette analyse que connaissant les paramètres des impulsions émises par le satellite depuis l'espace tel que documenté dans les télémesures de niveau 0, nous pourrions exploiter les mesures acquises par une unique antenne visant les cibles et qu'il serait inutile de recevoir le signal de référence. Ce faisant, nous diminuerions d'un facteur deux les contraintes de bande passante, d'espace de stockage et multiplierions le nombre de plateformes de radio logicielles compatibles avec cette analyse. Malheureusement, il n'en est rien : l'analyse fine des mesures acquises sur une unique voie et alignées selon le PRI théorique démontre une dérive excessive alors que le satellite avance le long de son orbite. En effet, le déplacement est tellement rapide (7,5 km/s) que pendant le temps d'acquisition, même si nous n'exploitons que 200 ms du fichier acquis, le satellite s'est déplacé de 1,5 km, loin d'être négligeable devant la distance des cibles que nous visons (Fig. 7). Il faut donc absolument acquérir sur une seconde voie le signal de référence émis par le satellite pour s'affranchir de l'impact de la distance variable entre le satellite et la cible et ainsi permettre à tous les échos réfléchis par une même cible d'être alignés sur la même ordonnée de la Fig. 7, condition nécessaire à la compression en azimut.

L'acquisition de la seconde voie de référence abaisse le débit maximum de données que peut transmettre la B210 de 56 Méchantillons/s (simple voie) à 30 Méchantillons/s (doubles voies), fournissant tout de même une résolution en distance de 5 m.

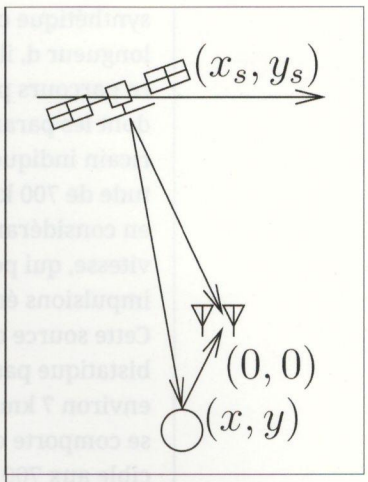


Figure 8 : Paramètres géométriques de l'analyse permettant de séparer les directions de distance à la cible et azimutale lors du traitement des données acquises.

4. TRAITEMENT DES SIGNAUX ACQUIS SUR DEUX ANTENNES...

4.1 ... dans un référentiel arbitraire

Nous avions développé, avec quelques erreurs à la relecture, le modèle classique de RADAR à antenne synthétique dans lequel l'émetteur et le récepteur sont colocalisés et se déplacent tous deux linéairement, ou dans le cas qui nous avait intéressés où l'émetteur restait fixe et le récepteur de déplaçait linéairement pour simuler une antenne de dimensions égales au chemin parcouru par le récepteur et ainsi présentant une résolution angulaire nettement améliorée par rapport à l'antenne réceptrice elle-même. On rappelle en effet que l'ouverture angulaire d'une antenne de dimensions d est de l'ordre de acos(λ/d) et que la résolution azimutale d'un RADAR est donc directement liée à la taille d'antenne d, dans la limite de ce qu'il est raisonnablement réalisable mécaniquement (Fig. 8). Pour d trop grand, l'antenne ne devient plus réalisable, mais le concept d'ouverture

synthétique consiste à considérer qu'en déplaçant l'antenne le long d'un chemin de longueur d, il est possible de recombiner les signaux acquis successivement le long de ce parcours pour former une antenne équivalente de taille d. Dans le cas de Sentinel-1 dont les paramètres orbitaux (TLE pour *Two Line Element*) diffusés par le NORAD américain indiquent qu'il parcourt 14.592 orbites autour de la Terre chaque jour à une altitude de 700 km, la vitesse linéaire est donc de $(40000+700\times2\pi)\times14.592/24=27000$ km/h en considérant la circonférence de la Terre égale à 40000 km, soit 7,5 km/s. Cette vitesse, qui pourrait sembler incroyable au sol, permet de déduire qu'entre deux impulsions émises au rythme d'environ 1900 Hz, le satellite parcourt environ 4 m. Cette source qui se déplace à 700 km d'altitude est reçue par notre montage de RADAR bistatique passif au sol pour observer des cibles à une distance que nous prendrons à environ 7 km au maximum du récepteur, donc par le théorème de Thalès le satellite se comporte comme une source mobile s'avançant de 4 cm – le rapport des 7 km à la cible aux 700 km à la source – vérifiant les contraintes de l'équivalent spatial du théorème d'échantillonnage pour ne pas présenter d'incertitude de repliement spectral lors de l'acquisition des données. Cette différence de géométrie – une source très loin des récepteurs de référence et de surveillance ainsi que des cibles illuminées – nous impose de développer un nouveau modèle de propagation des signaux qui justifiera de la transformée de Fourier inverse pour la compression en azimut, mais qui fournira les constantes nécessaires à graduer les axes en distance connue et non en unités arbitraires.

La démonstration se développe comme suit :

- la compression en distance se déduit d'une corrélation entre le signal de référence ref et le signal de surveillance sur le long de l'axe temporel de la matrice que nous avons formé en alignant $f_s\times$PRI échantillons les uns après les autres, et la corrélation xcorr se calcule dans le domaine de Fourier tel que le démontre le théorème de convolution par $FFT(xcorr(ref,sur))=FFT(ref)\cdot FFT^*(sur)$ avec FFT la transformée de Fourier rapide de complexité algorithmique Nlog(N) lors du traitement de N points et non N^2 comme le ferait une transformée de Fourier classique, avec * le complexe conjugué ;
- la compression en azimut se déduit d'une transformée de Fourier inverse le long de la colonne où le signal se répète puisque nous savons **[17]** que la transformée de Fourier d'un signal de la forme $\exp(j2\pi k_x\cdot x)\exp(j2\pi k_y\cdot y)$ se comprime en un Dirac en (x,y) par transformée de Fourier. Dans cette expression, k_x et k_y sont les grandeurs duales de x et y que les physiciens nomment habituellement vecteur d'onde. La question est donc de savoir si nous sommes capables d'exprimer le signal reçu et contenu dans la matrice (échantillon(t),position satellite) sous une forme qui sépare explicitement k_x et k_y : si c'est le cas, alors une transformée de Fourier inverse bidimensionnelle après avoir effectué la transformée de Fourier selon l'axe du temps doit fournir l'image dans le domaine spatial (x,y) qui s'apparentera à la géographie des cibles ;
- supposons (Fig. 8) le satellite en position (x_s,y_s) se déplaçant le long de l'axe x, les récepteurs à l'origine (0, 0), et une cible en (x,y) ;

– RADAR passif bistatique au moyen d'une Raspberry Pi 4, d'une radio logicielle et du satellite Sentinel-1 –

- nous pouvons exprimer la distance entre le satellite et les récepteurs comme $R_1 = \sqrt{(x_s)^2 + (y_s)^2} \simeq y_s + 1/2(x_s)^2/y_s$ puisque $y_s \gg x_s$ qui permet d'exploiter le développement limité de $\sqrt{u^2+v^2}$ avec $u \ll v$ qui s'écrit $v\sqrt{1+(u/v)^2} \simeq v(1+1/2(u/v)^2) = v + u^2/(2v)$;
- de la même façon, la distance entre le satellite et une cible au sol s'exprime par $R_2 = \sqrt{(x_s-x)^2+(y_s-y)^2} \simeq y_s - y + 1/2(x_s-x)^2/y_s - y \simeq y_s - y + 1/2(x_s-x)^2/y_s$ selon le même argumentaire ;
- enfin, la distance entre la cible et les récepteurs est $R_3 = \sqrt{x^2+y^2}$ sans approximation, mais qu'on notera être indépendant de x_s ;
- un signal de bande relativement étroite est impacté lors de sa propagation par un déphasage qui s'exprime soit en fonction du temps τ t soit de la distance parcourue R à vitesse c comme $\phi = 2\pi f\tau = 2\pi/\lambda R$ puisque $\tau = R/c$;
- finalement, la différence de chemin entre le signal arrivant directement au récepteur de référence et le signal qui a rebondi sur une cible avant d'atteindre le récepteur de surveillance s'exprime comme $R_3 + R_2 - R_1$ qui induit donc un déphasage :

$$2\pi f(\underbrace{R_3}_{\text{distance cible}}/c) + \frac{2\pi}{\lambda}(R_2 - R_1) = 2\pi f R_3/c +$$

$$\frac{2\pi}{\lambda}\left(-y + \frac{1}{2y_s}\underbrace{((x_s-x)^2 - (x_s)^2)}_{a^2-b^2=(a+b)\cdot(a-b)}\right) = \underbrace{2\pi f(R_3-y)/c}_{\text{FFT en distance}} + \frac{2\pi}{\lambda}\left(-\frac{x}{2y_s}(\underbrace{2x_s}_{\text{FFT en azimut}} - \underbrace{x}_{\text{indépendent de } x_s})\right)$$

Qui sépare bien la distance et l'azimut et justifie la compression d'impulsion par transformée de Fourier inverse le long de f et x_s avec l'expression des grandeurs duales pour retrouver une graduation quantitative.

Ce calcul ne nécessite en aucun cas la connaissance précise des paramètres orbitaux du satellite ou sa position dans l'espace : seule sa vitesse et donc l'intervalle entre deux mesures est nécessaire, qui se déduit d'une analyse simple des fichiers TLE (nombre de rotations autour de la Terre par jour) et de la connaissance de l'altitude à laquelle il survole la Terre, en plus de PRI. La seule hypothèse que nous avons faite pour autoriser le développement limité qui sépare les variables est que la distance satellite-sol est très grande devant la distance cibles-récepteur. Ainsi, le traitement se réduit à une transformée de Fourier inverse en deux dimensions, dans l'axe de la distance sur le produit des transformées de Fourier qui permet de produire une intercorrélation, et selon l'axe de l'azimut pour la compression tel que le démontre l'application de ce traitement sur le tout premier jeu de données acquis et présenté sur la Fig. 9, pag suivante.

Bien que la présence de motifs géométriques sur cette image soit plus qu'encourageante et prouve que nous sommes sur la bonne voie, une analyse détaillée notamment juste à gauche de la flèche qui pointe vers le nord laisse présager d'un problème lorsque nous plaquons nos mesures sur un fond de carte : les motifs géométriques sont proches, mais ne se superposent pas avec la carte, avec un écart d'autant plus grand que l'on s'écarte de l'axe de visée de l'antenne de surveillance. Il faut donc améliorer le modèle de projection.

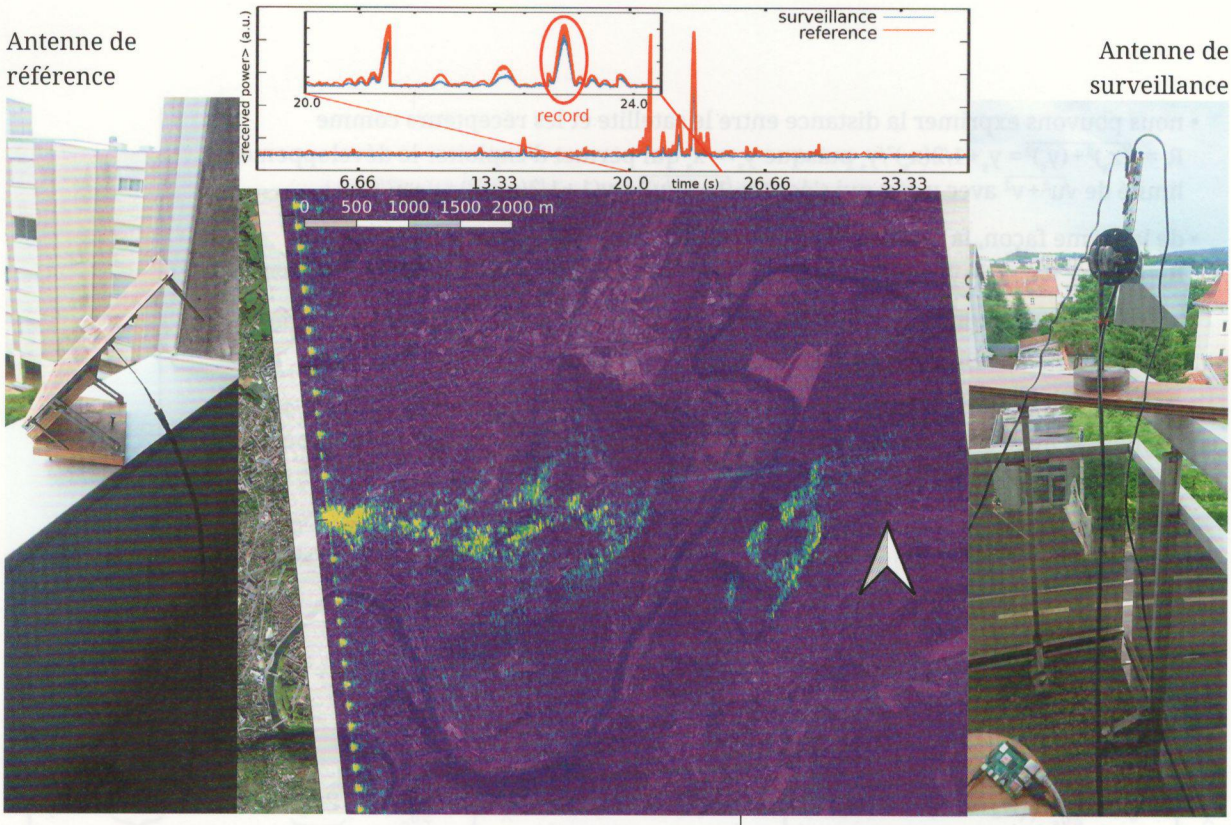
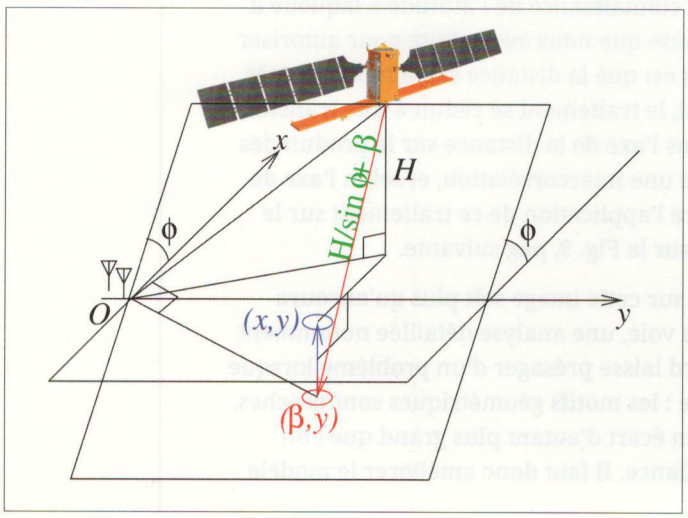


Figure 9 : Premiers essais avec une antenne cornet d'ouverture angulaire étroite comme antenne de surveillance et une antenne hélicoïdale de gain réduit et d'ouverture importante comme référence, un choix peu judicieux qui a pourtant permis d'obtenir une première image exploitable dans un cône d'observation relativement étroit.

Figure 10 : Propriétés géométriques de la projection du plan contenant les antennes (O) et la trajectoire du satellite (yOβ) vers le plan tangentiel à la surface de la Terre (xOy).



4.2 ... en coordonnées géographiques

La dernière subtilité est que ce calcul de compression en distance et azimutale se fait dans le plan contenant la trajectoire du satellite et les récepteurs au sol, alors que nous voudrions une projection dans le plan tangent à la surface de la Terre qui permettrait de se raccorder à un système de coordonnées géographiques (Fig. 10). Connaissant l'altitude de vol du satellite et l'angle d'illumination, il est possible de migrer chaque pixel du premier plan vers le second, une opération un peu longue en calcul qui nécessite de trouver une solution numérique pour chaque point, mais qui permettra ci-dessous de fournir des images dans un format compatible avec un outil de Système d'Information Géographique (SIG) tel que QGIS sans aucun degré de liberté

autre que la rotation des images. S'agissant de géométrie en trois dimensions, la représentation graphique sur une feuille en deux dimensions n'est pas aisée et nous laisserons le lecteur se convaincre que connaissant l'altitude H=693 km du satellite et l'angle d'illumination $\phi \simeq 45°$ dont la valeur est fournie avec précision pour chaque survol par le site web Heavens Above, alors la condition d'égalité de la somme des temps de vol satellite-cible et cible-récepteur dans le plan trajectoire-récepteur (α,β) dans le premier cas, et le plan tangent à la surface de la Terre (x,y) dans le second membre, s'exprime par :

$$\underbrace{\sqrt{y^2+(H/\sin\varphi+\beta)^2}}_{satellite-cible}+\underbrace{\sqrt{y^2+\beta^2}}_{cible-référence}=\underbrace{\sqrt{(H/\tan\varphi+x)^2+y^2+H^2}}_{satellite-cible}+\underbrace{\sqrt{x^2+y^2}}_{cible-référence}$$

En choisissant arbitrairement α=y (ne connaissant pas la position du satellite le long de sa trajectoire, cela suppose « simplement » que l'origine de la carte que nous traçons est centrée sur les récepteurs au sol... dont nous connaissons la position et qui impose donc l'emplacement de la carte sur un système de projection géographique) et en éliminant à droite et à gauche le terme commun de temps de vol du satellite à l'antenne de référence. Cette équation impose donc de trouver β du plan de départ qui vérifie pour chaque point (x,y) du plan d'arrivée l'égalité : après avoir cherché maintes approximations et solutions numériques (par exemple par méthode de Newton **[21]**... qui diverge ici compte tenu du très grand écart des valeurs numériques des différents termes) qui ont échoué, Wolfram Alpha (**https://www.wolframalpha.com/**) nous informe à notre plus grande surprise qu'il existe une solution analytique. En effet, en réduisant l'égalité à $\sqrt{A+(B+\beta)^2}+\sqrt{A+\beta^2}-C=0$ avec A, B et C des termes dépendant de x, y, H et ϕ uniquement et donc de valeur numérique constante pour chaque itération, il est possible de trouver la solution β par une équation certes longue, mais simple à implémenter dans un langage tel que GNU/Octave ou Python (Fig. 11).

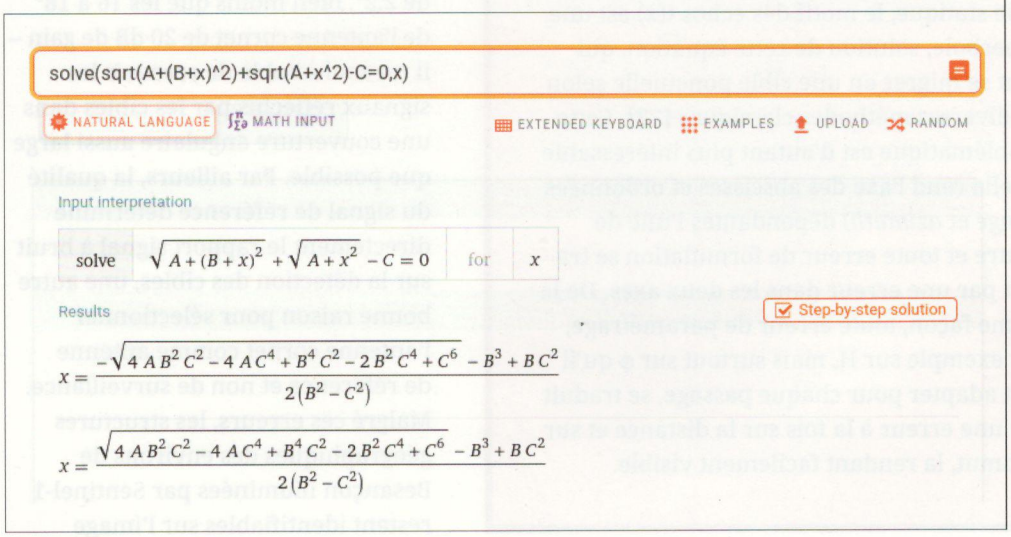


Figure 11 : Solution fournie par Wolfram Alpha à l'équation qu'il faut résoudre pour projeter la carte des cibles sur un plan tangentiel à la surface de la Terre.

Cette étape de migration peut paraître triviale en fin de chaîne de traitement, mais permet de convertir des mesures en unités arbitraires dans le plan contenant la trajectoire du satellite et le récepteur vers un plan tangent à la surface de la Terre. Le problème pourrait paraître simple puisqu'il s'agit d'une projection qui respecte le théorème de Pythagore, mais en incidence oblique l'expression est un peu ardue bien qu'une solution analytique ait été identifiée. Cette étape de projection peut ou ne pas être utile : lors de mesures subsurfaces par RADAR de sol (GPR pour *Ground Penetrating RADAR*), il est souvent plus aisé d'analyser les hyperboles produites par une cible ponctuelle alors que l'émetteur et le récepteur se déplacent à la surface pour identifier des cavités ou interfaces sub-surfaces, au détriment de la rigueur sur la géométrie des réflecteurs. Les hyperboles du GPR sont très similaires au problème qui nous intéresse ici, puisque résultant du motif solution de $c^2 \times t^2 = (x - x_c)^2 + z_c^2$ avec (x_c, z_c) la position du réflecteur enterré, x la position du RADAR à la surface et t le temps de vol de l'onde électromagnétique se propageant à vitesse c dans le sol. Lors du mouvement du RADAR x en surface au-dessus de la cible statique, le motif des échos t(x) est une hyperbole, solution de cette équation, qui peut se migrer en une cible ponctuelle selon les diverses méthodes classiques **[22]**. Cette problématique est d'autant plus intéressante qu'elle rend l'axe des abscisses et ordonnées (*range* et *azimuth*) dépendantes l'une de l'autre et toute erreur de formulation se traduit par une erreur dans les deux axes. De la même façon, toute erreur de paramétrage, par exemple sur H, mais surtout sur ϕ qu'il faut adapter pour chaque passage, se traduit par une erreur à la fois sur la distance et sur l'azimut, la rendant facilement visible.

5. RÉSULTATS

Nos premiers essais avec une architecture à deux antennes sont illustrés en Fig. 9 et 12 pour mettre en évidence les erreurs de débutant. Dans ce montage, deux antennes sont utilisées, l'une visant le satellite (référence, gauche) et l'autre les cibles (surveillance, droite). Inquiet de ne pas avoir assez de gain du signal rétrodiffusé par les cibles tout en considérant le signal venant du satellite puissant, nous avions sélectionné une antenne cornet (*horn antenna*) de fort gain (20 dB) A-Infomw LB-159-20-C-SF comme antenne de surveillance et une antenne hélicoïdale comme antenne de référence. Ces deux choix sont erronés : alors que le signal direct du satellite ne sera reçu que pendant un temps très court et donc dans une ouverture angulaire étroite compatible avec l'antenne cornet – le satellite parcourt en 5 secondes une distance de 37,5 km qui à une distance de 700/cos(45°) = 980 km se traduit par une ouverture angulaire de 2,2°, bien moins que les 16 à 18° de l'antenne cornet de 20 dB de gain – il est souhaitable de recevoir les signaux réfléchis par les cibles dans une couverture angulaire aussi large que possible. Par ailleurs, la qualité du signal de référence détermine directement le rapport signal à bruit sur la détection des cibles, une autre bonne raison pour sélectionner l'antenne cornet comme antenne de référence et non de surveillance. Malgré ces erreurs, les structures géographiques des environs de Besançon illuminées par Sentinel-1 restent identifiables sur l'image

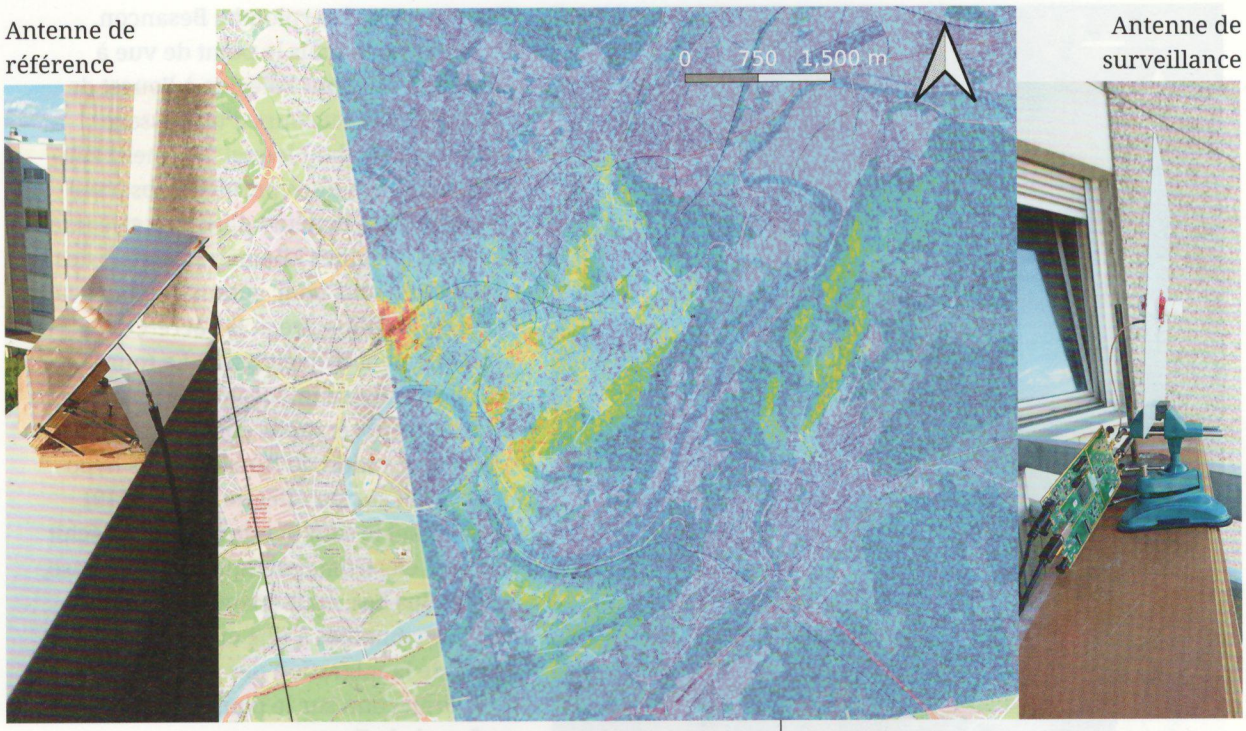


Figure 12 : Mesures depuis le même site – le balcon de l'appartement de l'auteur – cette fois avec une antenne hélicoïdale d'ouverture angulaire importante comme antenne de surveillance. Noter la cohérence des motifs des réflecteurs avec ceux de la Fig. 9. L'antenne de référence reste une antenne hélicoïdale plus facile à installer rapidement dans la cage d'escalier sans risquer de blesser un passant maladroit qui trébucherait sur le câble coaxial et se blesserait avec l'antenne cornet plus imposante.

(Fig. 9 et 12, milieu). On notera que la distance d'une quinzaine de mètres entre le balcon qui supporte l'antenne de surveillance (droite) et la cage d'escalier dans laquelle se trouve l'antenne de référence (gauche) ne pourra être couvert à 5,405 GHz par un câble coaxial de qualité médiocre aux fréquences micro-ondes tel que RG-58 (plus de 30 dB de pertes), mais qu'un câble de bonne qualité tel que LMR-400 (moins de 6 dB de pertes sur cette longueur) utilisé ici sera indispensable.

Dans ce premier exemple de la Fig. 9, non seulement la géométrie des antennes choisies est peu judicieuse, mais en plus l'image obtenue par transformée de Fourier inverse a « simplement » été déformée par homothétie en abscisse et ordonnée – en l'absence de coordonnées géographiques avant projection sur le plan tangent à la surface de la Terre, tel qu'expliqué en section 5.1 – et tournée en imposant la localisation des récepteurs sur le balcon de l'appartement de l'auteur.

Les mesures ont été répétées depuis divers sites, toujours en nous positionnant en altitude afin de limiter l'impact d'obstacles entre les cibles et le récepteur, sans pour autant tenir compte de cette différence d'élévation dans le modèle de projection : Fig. 13 (page suivante) illustre le cas de Clermont-Ferrand et le point de vue de la Pierre Carrée qui surplombe la ville, ainsi que le Fort

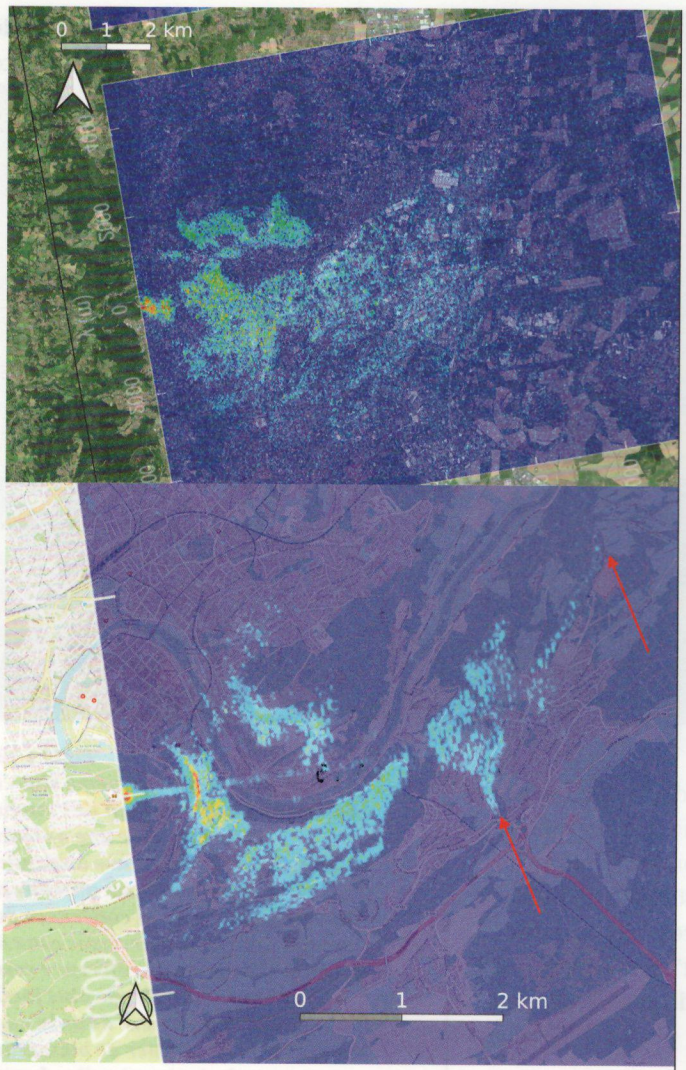
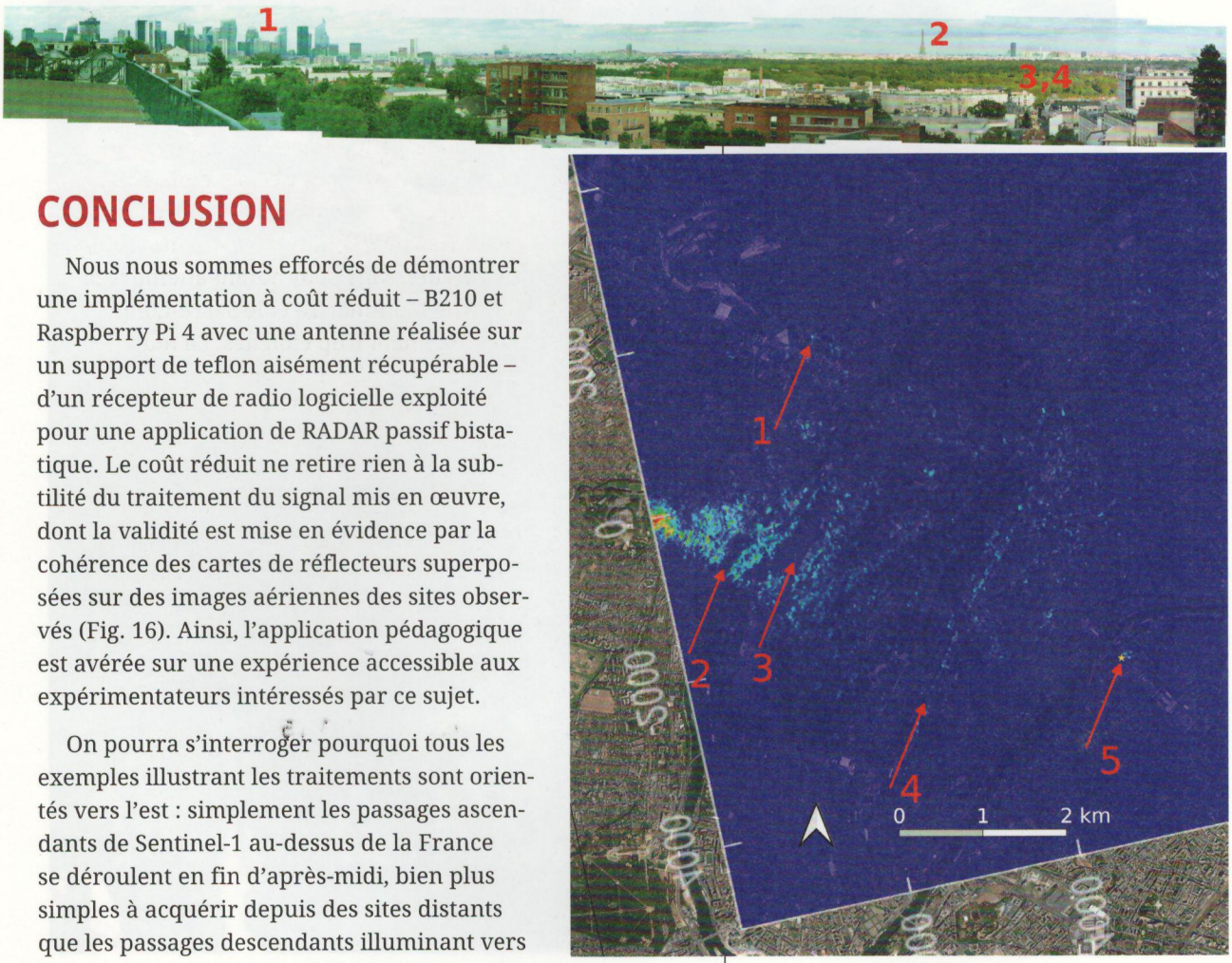


Figure 13 :
Haut : mesure acquise à Clermont-Ferrand depuis le point de vue de la Pierre Carrée sur la route menant au Puy-de-Dôme avec le montage illustré en Fig. 3 (gauche).
Bas : mesure acquise à Besançon depuis le Fort de Chaudanne, face à la Citadelle et ses falaises faisant office de réflecteurs efficaces, au moyen du système de mesure illustré en Fig. 3 (droite). Les deux flèches rouges mettent en évidence l'excellent accord entre la position de Morre (bas) et les falaises de Montfaucon (haut) avec les réflecteurs observés.

Chaudanne qui surplombe Besançon. L'acquisition depuis le point de vue à proximité du mont Valérien à l'ouest de Paris (Fig. 14), au cours d'un passage ascendant pendant lequel Sentinel-1 illumine vers l'est, nous permet d'observer les réflecteurs majeurs dans une circonférence de quelques kilomètres autour des récepteurs, sans pour autant atteindre la butte de Montmartre et le Sacré-Cœur, probablement trop éloignés. La tour Eiffel, une masse imposante de métal est clairement observable, comme un réflecteur à sa position nominale, et nos hypothèses **[18]** sur un effet de *layover* **[19]** dans lequel le sommet de la tour réfléchit le signal électromagnétique avant sa base, 360 m plus bas, était erronées et uniquement attribuables à une erreur dans le calcul de la migration des données lors de la projection sur le plan tangentiel à la surface de la Terre.

Pour conclure cette série de démonstrations expérimentales, la portabilité et l'efficacité du système sont validées au Spitzberg, dans un environnement où le poids et le volume des bagages sont contraints. Le montage tient dans un sac à dos et se déploie en quelques minutes sur site (Fig. 15, haut) pour un résultat de mesure passive par RADAR bistatique tout à fait en accord avec le signal obtenu par Sentinel-1 lors de son survol du site d'observation (Fig. 15, bas). Sur cette image, l'arrière-plan fournit le contexte géographique de Google Earth, l'image en tons de gris est la carte de réflectivité des cibles obtenue par Sentinel-1, et la couleur indique les réflecteurs détectés par le RADAR passif. Les montagnes imposantes visibles sur Fig. 15, en haut à droite, sont clairement visibles sur la mesure par RADAR passif à un emplacement en accord avec la mesure de Sentinel-1.

– RADAR passif bistatique au moyen d'une Raspberry Pi 4, d'une radio logicielle et du satellite Sentinel-1 –



CONCLUSION

Nous nous sommes efforcés de démontrer une implémentation à coût réduit – B210 et Raspberry Pi 4 avec une antenne réalisée sur un support de teflon aisément récupérable – d'un récepteur de radio logicielle exploité pour une application de RADAR passif bistatique. Le coût réduit ne retire rien à la subtilité du traitement du signal mis en œuvre, dont la validité est mise en évidence par la cohérence des cartes de réflecteurs superposées sur des images aériennes des sites observés (Fig. 16). Ainsi, l'application pédagogique est avérée sur une expérience accessible aux expérimentateurs intéressés par ce sujet.

On pourra s'interroger pourquoi tous les exemples illustrant les traitements sont orientés vers l'est : simplement les passages ascendants de Sentinel-1 au-dessus de la France se déroulent en fin d'après-midi, bien plus simples à acquérir depuis des sites distants que les passages descendants illuminant vers l'ouest qui se déroulent tôt le matin.

Une zone dégagée est nécessaire pour obtenir des échos réfléchis par les cibles au sol sans avoir été atténués par des obstacles à proximité des récepteurs, d'où la recherche de points élevés (point de vue de Clermont-Ferrand, Mont Valérien à Paris, colline de Planoise ou Chaudanne à Besançon) lors de ces mesures. Le système est suffisamment léger et peu gourmand en ressources pour être emmené par drone : cette perspective n'a pas (encore) été mise en œuvre, mais semble envisageable pour s'affranchir de la contrainte sur le site d'observation, le vecteur de vol ne devant rester stationnaire en altitude que quelques secondes.

L'ensemble des logiciels et scripts de traitement décrits dans ce document est disponible à **https://github.com/jmfriedt/sentinel1_pbr**.

Figure 14 : Mesures depuis le mont Valérien à l'ouest de Paris au cours d'un passage ascendant de Sentinel-1 qui illumine donc vers l'est. En haut le panorama annoté pris depuis le site d'observation, et en bas à gauche le résultat du traitement : 1 représente le quartier de La Défense avec ses tours, 2 la Seine, 3 la plaine de jeux du bois de Boulogne et ses stades de football et rugby, 4 une des avenues rectilignes du Bois de Boulogne, et 5 la tour Eiffel dont la position est marquée par une étoile.

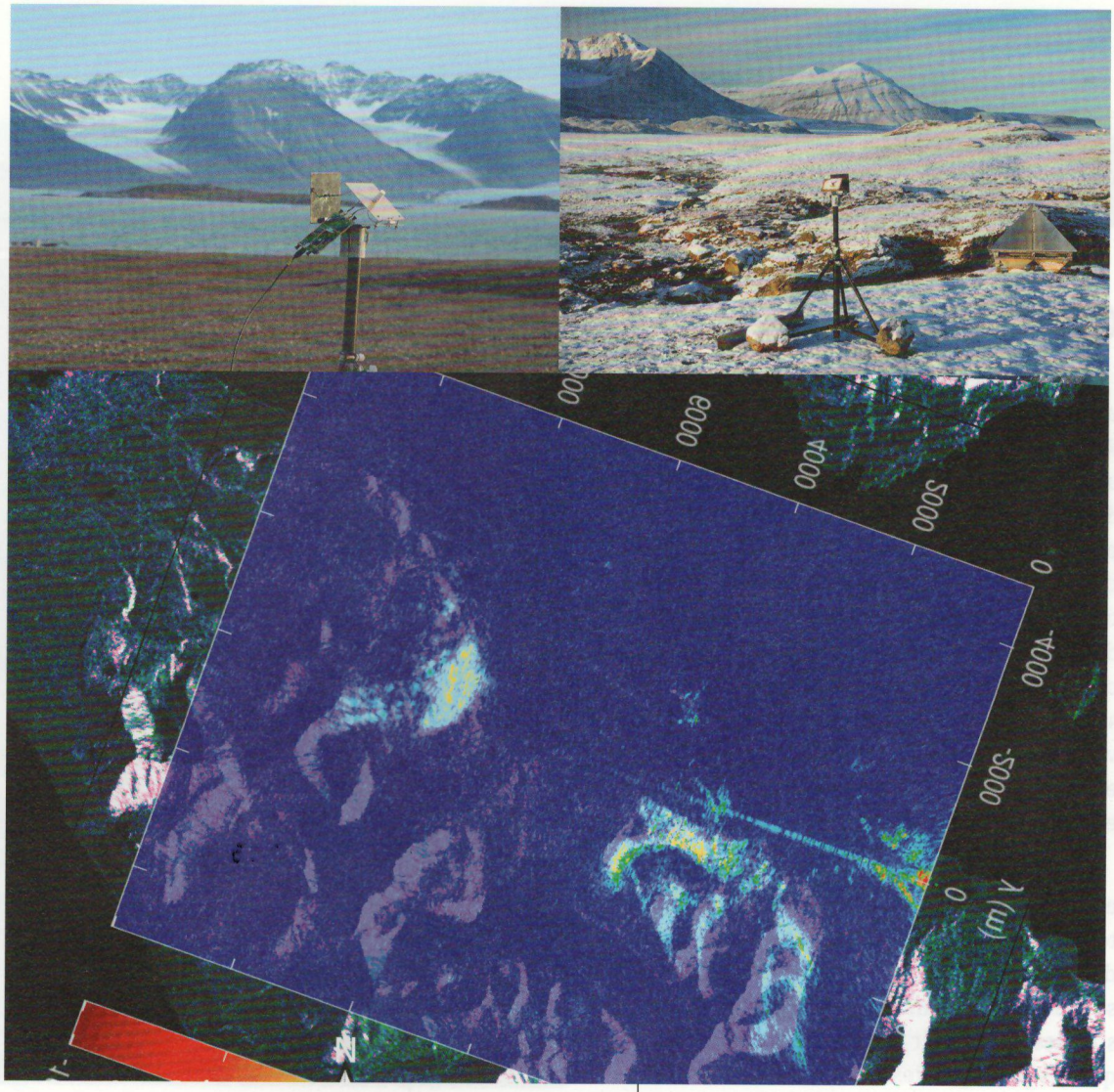


Figure 15 :
Haut gauche : le montage de RADAR bistatique déployé au Spitzberg, avec l'antenne de référence à droite orientée vers le ciel et l'antenne de surveillance face aux réflecteurs potentiels. La B210 est connectée aux deux antennes prêtes pour acquérir une minute de signal. Haut droite : les montages de la presqu'île de Brøgger feront office de réflecteurs, au même titre que le catadioptre visible en premier plan, mais dont le signal ne sera pas analysé ici. Bas : carte de la puissance du signal réfléchi, en tons de gris le signal détecté par Sentinel-1 en mode IW (Interferometric Wide, donc « haute résolution ») le 3 octobre 2021, et en couleur la mesure par RADAR passif bistatique lors du survol du 4 octobre. Nous supposons que les réflecteurs dans la mer sont des glaçons vêlés par les glaciers à front marin au fond de la baie (fjord), sans certitude en l'absence de photographie prise simultanément. Contexte géographique en arrière-plan : Google Earth.

REMERCIEMENTS

Les recherches bibliographiques pour nos activités de développement ne sauraient aboutir sans Library Genesis **[4]**, une ressource incontournable. L'acquisition du matériel ayant permis ce projet a été en partie financée par le Centre National d'Études Spatiales (CNES) au travers du projet R-S18/LN-0001-036 tandis que le voyage au Spitzberg a été financé par l'Institut polaire Paul Émile Victor (IPEV) dans le cadre du projet PRISM. MF

– RADAR passif bistatique au moyen d'une Raspberry Pi 4, d'une radio logicielle et du satellite Sentinel-1 –

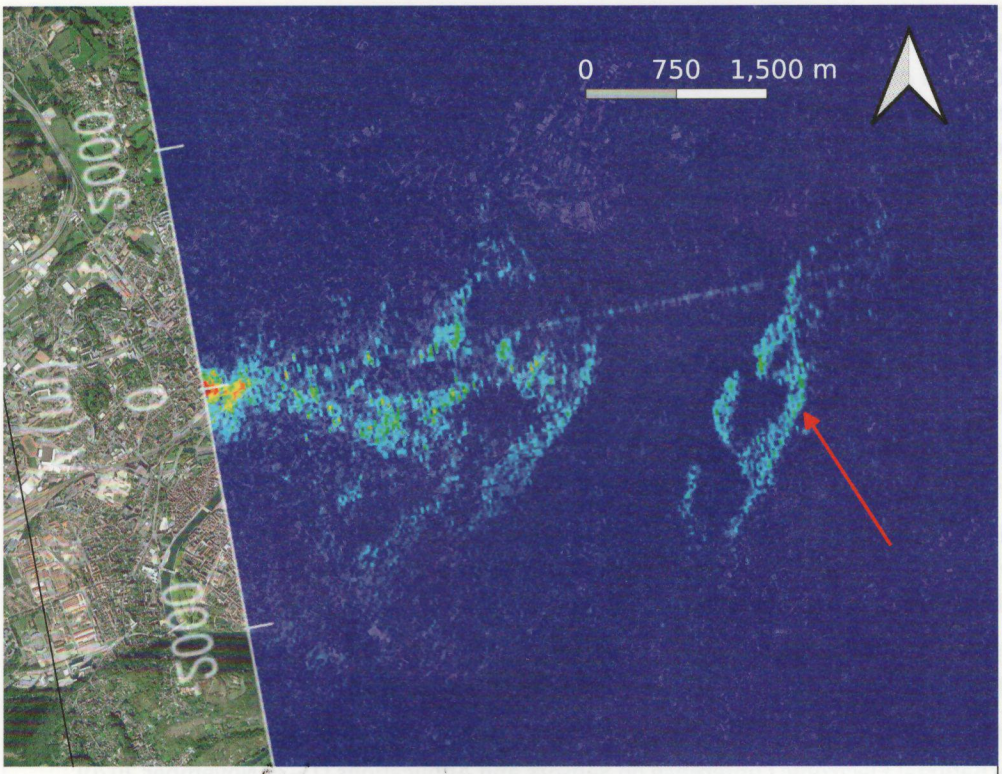


Figure 16 : Jeu de données de la Fig. 9 correctement projeté : la flèche rouge attire l'œil sur la zone qui portait à contestation avant projection. Comparer avec Fig. 12 pour observer la cohérence des deux mesures prises à des dates distinctes avec des antennes différentes depuis le même site.

RÉFÉRENCES

[1] L. Arenschield, *SpaceX satellite signals used like GPS to pinpoint location on Earth* (22/09/2021) à **https://news.osu.edu/spacex-satellite-signals-used-like-gps-to-pinpoint-location-on-earth**

[2] C. Deluzarche, « *Des chercheurs créent leur propre système de GPS en piratant des satellites* » (07/10/2021) à **https://www.futura-sciences.com/tech/actualites/piratage-chercheurs-creent-leur-propre-systeme-gps-piratant-satellites-93962/**

[3] M. Neinavaie & al., *Exploiting Starlink Signals for Navigation: First Results*, Proc. ION GNSS (2021) à **https://www.ion.org/publications/abstract.cfm?articleID=18122**

[4] Library Genesis à **gen.lib.rus.ec**

[5] A. Anghel & al., *Bistatic SAR imaging with Sentinel-1 operating in TOPSAR mode*, 2017 IEEE Radar Conference.

[6] G. Goavec-Merou, J.-M Friedt, *« On ne compile jamais sur la cible embarquée » : Buildroot propose GNU Radio sur Raspberry Pi (et autres), Hackable n°37* (avril-mai-juin 2021) **https://connect.ed-diamond.com/hackable/hk-037/on-ne-compile-jamais-sur-la-cible-embarquee-buildroot-propose-gnu-radio-sur-raspberry-pi-et-autres**

[7] J.-M Friedt, P. Abbé, *Parler à un RADAR spatioporté : traitement et analyse des données de Sentinel-1, GNU/Linux Magazine France n°246* (mars 2021) **https://connect.ed-diamond.com/GNU-Linux-Magazine/glmf-246/parler-a-un-radar-spatioporte-traitement-et-analyse-des-donnees-de-sentinel-1**

[8] S. Guinot, J.-M Friedt, *La réception d'images météorologiques issues de satellites : utilisation d'un système embarqué, GNU/Linux Magazine France Hors Série n°24* (2006).

[9] J.-M. Friedt, *Décodage d'images numériques issues de satellites météorologiques en orbite basse : le protocole LRPT de Meteor-M2, GNU Linux Magazine France* (2019), en 3 parties.

[10] D. Estevez, *Decoding Interplanetary Spacecraft*, tutoriel GNU Radio Conference (2019) à **https://www.youtube.com/watch?v=RDbs6l4rMhs**

[11] J. Rosen, *Shifting ground*, Science 371 (6532), pp. 876-880 (2021) annonce plus de 60 RADAR spatioportés dans les années à venir.

[12] J. Blumenfeld, *Getting Ready for NISAR - and for Managing Big Data using the Commercial Cloud* (2 oct. 2017) à **https://earthdata.nasa.gov/learn/articles/tools-and-technology-articles/getting-ready-for-nisar**

[13] S. Prager & al., *Ultrawideband Synthesis for High-Range-Resolution Software-Defined Radar*, IEEE Trans. Instrum. Meas. 69 3789–3803 (2020).

[14] O. Toker & al., *A Synthetic Wide-Bandwidth Radar System Using Software Defined Radios*, 7th International Electronic Conference on Sensors and Applications (15-30 November 2020).

[15] S.T. Peters & al., *In Situ Demonstration of a Passive Radio Sounding Approach Using the Sun for Echo Detection*, IEEE Trans. Geosci. and Remote Sensing 56 (12) 7338 (Dec. 2018).

[16] C.A. Balanis, *Antenna Theory – Analysis and design*, 3rd Ed., Wiley Interscience (2005).

[17] **https://hforsten.com/synthetic-aperture-radar-imaging.html**

[18] J.-M Friedt, W. Feng, *Passive bistatic RADAR using spaceborne Sentinel-1 non-cooperative source, a B210 and a Raspberry Pi 4*, GNU Radio Conference (2021) à **https://youtu.be/BaRgx5ehdOw?t=7960**

[19] *Remote sensing and SAR radar images processing* – Physics of radar, p.40, ESA/CNES à **https://earth.esa.int/c/document_library/get_file?folderId=226458&name=DLFE-2127.pdf**

[20] **https://github.com/jmfriedt/sentinel1_level0**

[21] J.-M Friedt, *Arithmétique sur divers systèmes embarqués aux ressources contraintes : les nombres à virgule fixe, GNU/Linux Magazine France Hors Série n°113* (mars 2021) **https://connect.ed-diamond.com/GNU-Linux-Magazine/glmfhs-113/arithmetique-sur-divers-systemes-embarques-aux-ressources-contraintes-les-nombres-a-virgule-fixe**

[22] G. Charvat, *The Range Migration Algorithm (RMA)*, section 4.2 de Small and Short-Range Radar Systems, CRC Press (2014).

eurobot

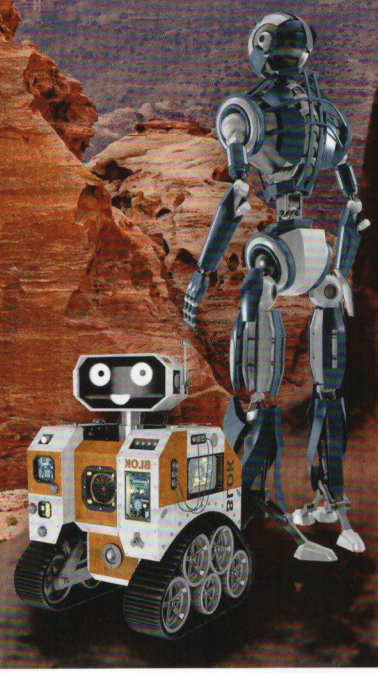
Coupe de France DE ROBOTIQUE JUNIOR

COUPE DE FRANCE DE ROBOTIQUE

DU 26 AU 28 MAI 2022

LA ROCHE-SUR-YON

PARC EXPO DES OUDAIRIES



www.coupederobotique.fr #CDR2022

ORYON

Planète Sciences — une aventure pour les jeunes

La Roche-sur-Yon Agglomération — Le cœur Vendée

Région PAYS DE LA LOIRE

EXOTEC

14ème ÉDITION JOURNÉES RÉSEAUX

DE L'ENSEIGNEMENT ET DE LA RECHERCHE

MARSEILLE

DU 30 NOVEMBRE AU 2 DÉCEMBRE

17 au 20 Mai 2022

CONNECTONS NOS DIFFÉRENCES !

jres MARSEILLE 2021

www.jres.org

#jres2021

ORGANISÉE PAR : RENATER, Aix*Marseille université

EN COLLABORATION AVEC : UNIVERSITÉ DE TOULON, AVIGNON UNIVERSITÉ, CENTRALE MARSEILLE, sciencespo.aix

AVEC LE SOUTIEN DE : MINISTÈRE DE L'ENSEIGNEMENT SUPÉRIEUR, DE LA RECHERCHE ET DE L'INNOVATION