ÉLECTRONIQUE | EMBARQUÉ | RADIO | IOT

HACKABLE

L'EMBARQUÉ À SA SOURCE

N° 52
JAN. / FÉV. 2024

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

L 19338 - 52 - F: 14,90 € - RD

CPPAP : K92470

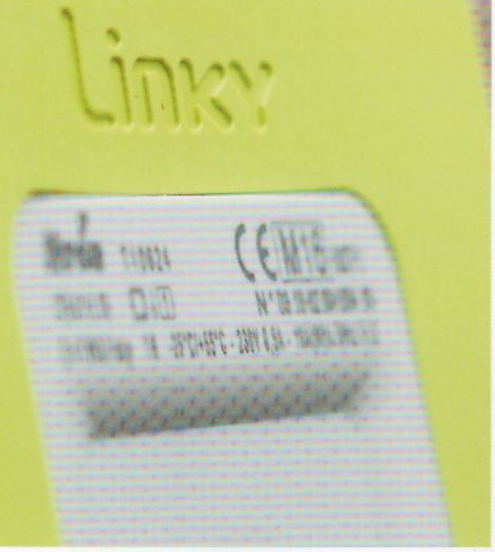
Compteur / Domotique / Home Assistant

Comment garder facilement un œil sur votre consommation ?

SURVEILLEZ VOTRE LINKY !

p.54

- Se brancher à la prise dédiée du compteur (TIC)
- Collecter les informations via Wi-Fi
- Suivre sa consommation sur Raspberry Pi



LE SALON ONE TO ONE MEETINGS DES RÉSEAUX, DU CLOUD, DE LA MOBILITÉ ET DE LA CYBERSÉCURITÉ

IT AND CYBERSECURITY

MEETINGS BY WEYOU GROUP

WWW.IT-AND-CYBERSECURITY-MEETINGS.COM

19, 20 & 21 MARS 2024

PALAIS DES FESTIVALS ET DES CONGRÈS DE CANNES

ILS SONT DÉJÀ INSCRITS

6cure · adista · AISI · allentis · alternativ · ANOZR WAY · APIXIT · aqua · arcserve · ARMIS · Avanidee Cliquer · BCS Solutions · betaSystems · betaobe · BeyondTrust · Bitdefender · BlackBerry Cybersecurity · BlueFiles · BlueTrusty · board of cyber · bouygues · bso · CATO · cegedim · ClariLog · CLOUDFLARE · COHESITY · conscio · consoleconnect · CROWDSTRIKE · CRYPT.ON IT · CUSTOCY · cybereason · Cyberwatch · CYBLEX · dbSQWare · DATACORE · Delinea · devensys · DIGITAL REALTY · dust MOBILE · EASYVISTA · EATON · efficient IP · EGERIE · EPOS · ERCOM · eset · EV Observe · exabeam · Extelent Systems · Otenable · EXCUBE · FILIGRAN · Forcepoint · FORTINET · GATEWATCHER · GitGuardian · CONVERGENCE · HarfangLab · HEXANET · HUMAN · ilex · INFINIGATE · IRON MOUNTAIN · ITS Ibelem · Jabra GN · juniper · KADISKA · KDDI KDDI France · LastPass · Linkt · Lutessa · MAKE IT SAFE · Mediaserv · ManageEngine · MANHATTAN · Mantra · MIEL · mindflow · MY · monacodigitalgroup · nameshield · NeoYAD · netskope · NOZOMI NETWORKS · HYBBLE · ceil sauf ceil · okta · ONE IDENTITY · opentext Cybersecurity · Orange Cyberdefense · OVERSOC · OWN · PatrOwl · PENTERA · Ping · platform.sh · pradeo · PRIMX · proofpoint · PURESTORAGE · pythear · QOS · Qualys · radware · RingCentral · Riot · rubycat · SailPoint · sea tpi · SentinelOne · sewan · Skyhigh · SNS SECURITY · SOPHOS · sosafe · STORMSHIELD · Suricate · synetis · sysdig · TANIUM · tdf · TEHTRIS · TELEHOUSE · Tenacy · TRENDS · TrustHQ · TrustBuilder · TYREX · UBIKA · vade · VARONIS · VEEAM · wavetel · WITH · xmco · ZeroTrust · Zertø · zpe

by weyou Group

Professional Exhibitions and One to One Meetings Exhibitions

ÉDITO

« De nos jours, la programmation est une course entre les développeurs s'efforçant de créer des programmes plus gros et à l'épreuve des imbéciles, et l'Univers essayant de créer de meilleurs et de plus gros imbéciles. »

Cette phrase est tirée du livre « *The Wizardry Compiled* » de Rick Cook, datant de 1990 et contant l'histoire d'un programmeur devenu sorcier, capturé par les forces du mal et devant être secouru par une bande d'informaticiens recrutés pour l'occasion. Si vous aimez Douglas Adams et Terry Pratchett, vous aimerez certainement Rick Cook, même si ses livres sont difficiles à trouver et malheureusement non traduits en français.

Je me demande ce que penserait aujourd'hui cet auteur, en voyant que les frontières entre « développeurs » et « imbéciles » tendent à devenir de plus en plus floues et, dans le même temps, que la bêtise humaine, elle, semble définitivement infinie comme le pensait Einstein. La notion même de « développeur » est potentiellement en voie d'extinction, avec des IA de plus en plus compétentes à produire non seulement du code, mais aussi des « entités » purement virtuelles (pour ne pas dire « fake ») adulées par des dizaines de milliers de... heu... humains (cherchez « Aitana Lopez », vous verrez). Ajoutez à cela la « *dead Internet theory* » provenant tout droit de 4chan et les articles récurrents sur l'« épidémie de solitude », et vous obtenez une approximation dystopique relativement inquiétante d'un futur aussi proche qu'étrange et déprimant.

2024 risque d'être une nouvelle année aussi intéressante que les trois précédentes. Il devient de plus en plus difficile de deviner, ou plutôt d'estimer, comment tout cela va tourner, mais une chose est sûre, ça va être absolument fascinant à observer. Bonne année 2024 à vous, ou bonnes fêtes si vous avez attrapé ce numéro dès 2023 !

Denis Bodor

Note : photo de couverture par pixarno - stock.adobe.com

Hackable Magazine
est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND
STIMULANT DE MATIÈRE GRISE

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

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

Suivez-nous sur Twitter
@hackablemag

Alsace

SOMMAIRE

À PROPOS DE HACKABLE...

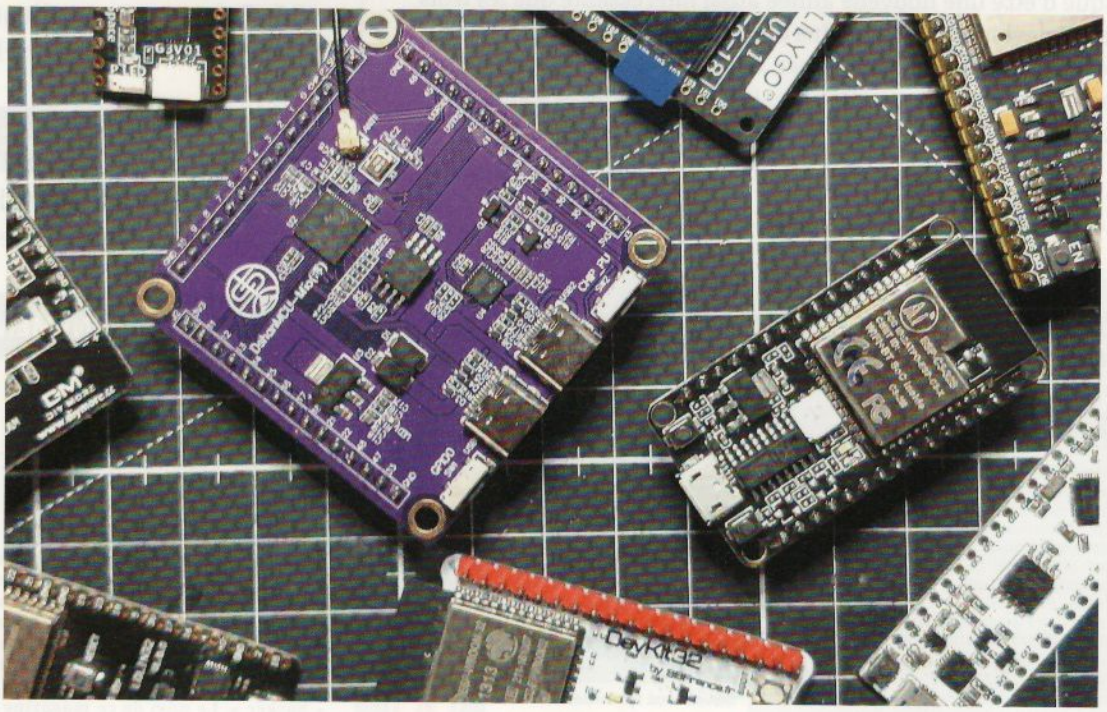
HACKS, HACKERS & HACKABLE

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

ESP32 : EXPLORATION, MISE À JOUR, CRYPTO ET PORTAGE DEPUIS RP2040

Denis BODOR

La carte Raspberry Pi Pico et son RP2040 est une plateforme très intéressante et proposant nombre de fonctionnalités, mais lorsqu'il s'agit de caractéristiques plus avancées telles que le chiffrement, la signature électronique ou le stockage sécurisé, celle-ci n'est clairement pas adaptée. Presque tous les ESP32, en revanche, disposent de ce type de ressources et c'est là précisément le point de départ du présent article : le portage d'un précédent projet, PicoTOTP (voir numéro 50), du RP2040 vers l'ESP32-S2.

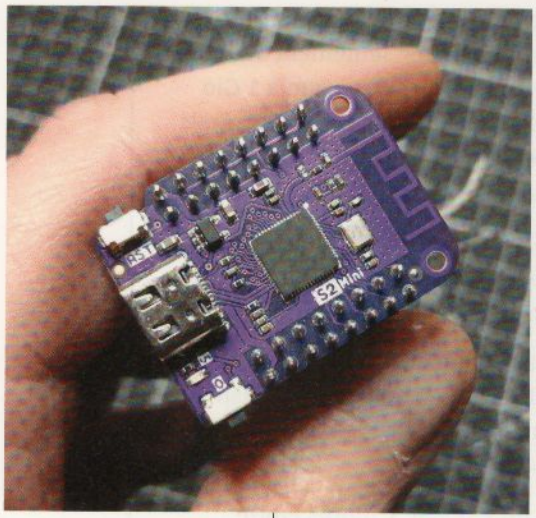


L'idée de ce qui va suivre n'est pas exactement de vous embarquer avec moi pour réimplémenter entièrement le projet complet d'une plateforme à une autre, mais plus exactement de mettre en avant certains points intéressants rencontrés au cours de cette opération, les problèmes et, bien entendu, les solutions. Les ESP32 d'Espressif Systems sont désormais une vaste famille de microcontrôleurs, pour certains basés sur un processeur Tensilica Xtensa et pour d'autres, sur du RISC-V 32 bits. Mais le plus intéressant concerne le SDK associé, prenant la forme d'un environnement de développement appelé ESP-IDF (pour *ESPressif IoT Development Framework*), reposant sur un compilateur C/C++, le système de construction CMake et un script Python pour orchestrer le tout. C'est un unique *framework* permettant de développer pour l'ensemble des composants et modules de la famille ESP32, dont :

- ESP32 : le premier du nom (2016) avec son Xtensa LX6 *dual-core* 32 bits à 240 MHz, 520 Kio de SRAM, jusqu'à 16 Mio de flash externe (4 Mio interne selon série), Wi-Fi 802.11 b/g/n (alias Wi-Fi 4), Bluetooth v4.2 BR/EDR et BLE ;
- ESP32-S2 : toujours du Xtensa à 240 MHz, mais en version LX7 cette fois et monocœur, avec seulement 320 Kio de SRAM, 0, 2 ou 4 Mio de flash interne et jusqu'à 1 Gio de flash externe, Wi-Fi 802.11 b/g/n, sans Bluetooth, mais avec USB OTG 1.1, coprocesseur *low-power* ULP-RISC-V (PicoRV32), et DAC (2 canaux 8 bits) ;
- ESP32-C3 : processeur RISC-V 32 bits cadencé à 160 MHz, 400 Kio de SRAM, 4 Mio de flash interne et jusqu'à 16 Mio en externe, Wi-Fi 802.11 b/g/n toujours mais avec BLE 5.0 ;
- ESP32-S3 : retour au Xtensa 32 bits LX7 double-cœur à 240 MHz, avec 512 Kio de SRAM, 0 ou 8 Mio de flash interne et jusqu'à 1 Gio en externe, Wi-Fi 802.11 b/g/n, BLE 5.0, USB OTG 1, ULP-RISC-V, interface pour caméra DVP ;
- ESP32-C6 : plus ou moins un ESP32-C3, moins étoffé mais plus récent, avec Bluetooth LE v5.3 et surtout Wi-Fi 802.11 b/g/n/ax (Wi-Fi 6) ;
- ESP32-H2 : plus modeste que ses cousins (RISC-V à 96 MHz, sans Wi-Fi, 320 Kio de SRAM, etc.) ce modèle se spécialise dans une autre forme de connectivité *wireless* 2,4 GHz, le LR WPAN 802.15.4 (*Low Rate Wireless Personal Area Network*) sur lequel repose des protocoles dont le nom est plus connu comme ZigBee ou Thread ;
- ESP32-P4 : pour l'instant uniquement annoncé, ce modèle peut être vu comme « la grosse bête » de la famille, orientée performance avec son double cœur RISC-V à 400 MHz (plus un cœur *Low Power* à 40 MHz), 768 Kio de SRAM, avec interface MIPI-CSI (caméra) et MIPI-DSI (écrans), accélération matérielle pour l'encodage vidéo (H.264) et même une sorte de GPU pour le développement d'interfaces utilisateur (PPA pour *Pixel Processing Accelerator*).

Devant cette avalanche de solutions et déclinaisons aux noms divers et variés, le constructeur propose judicieusement un comparateur permettant de rapidement se faire une idée de ce qui est et n'est pas intégré dans chaque SoC : **https://products.espressif.com/#/product-comparison**. Bien entendu, tous ne sont pas disponibles sous la forme de module et pour jeter votre dévolu sur l'un ou l'autre modèles, vous devrez également jongler, en parallèle, avec les offres présentées sur Mouser, Farnell, DigiKey, Amazon, eBay ou encore AliExpress et consorts.

Notez au passage que l'ESP-IDF est également à la base du support Arduino pour l'ESP32, alias *Arduino ESP32 Core*, très



Le module ou carte Wemos S2 Mini est relativement minimaliste et repose entièrement sur les fonctions fournies par le SoC ESP32-S2FN4R2 intégrant un contrôleur USB (USB CDC ACM), la flash et la PSRAM. En dehors d'un quartz, d'un régulateur de tension et de quelques composants passifs, cet ESP32 n'a besoin d'aucun autre élément pour fonctionner.

récemment passé en version 3.0.0 et basé sur ESP-IDF 5.1. Pour ma part, j'avoue que je préfère, pour les projets sérieux, utiliser directement l'environnement Espressif et faire l'impasse sur Arduino, en particulier depuis que je suis tombé sur un petit ajout qui rend les choses vraiment plus simples à composer (cf. plus loin dans l'article). Autre point intéressant, les ESP8266 disposent de leur propre environnement, généralement basé sur Arduino justement, mais peuvent également être utilisés avec un SDK spécifique « dans le style » ESP-IDF : l'ESP8266 RTOS SDK **[1] [2]**. Ceci dit, je trouve que les ESP8266, vieillissants (2016), perdent peu à peu leur intérêt face aux modules ESP32 vendus quasiment au même prix aujourd'hui et bien plus capables.

À propos de capacité justement, ce qui nous intéresse ici, ce sont les fonctionnalités orientées sécurité et l'accélération cryptographique, en particulier. Ceci est présent dans presque tous les ESP32 récents (Xtensa et RISC-V) et en particulier l'accélérateur HMAC. Dans l'article sur l'authentification à deux facteurs via Raspberry Pi Pico, j'avais évoqué le fait que notre projet n'était pas utilisable en production, car il était très facile de récupérer la clé présente au sein du code sous la forme d'un simple tableau, parfaitement lisible en flash. Avec un périphérique dédié comme celui présent dans les ESP32, les choses sont très différentes, car cette clé peut être inscrite dans une zone mémoire protégée et configurée pour autoriser l'accès en lecture uniquement par certaines fonctions internes à l'accélérateur HMAC.

Du fait de la polyvalence de l'ESP-IDF et de l'existence de cette fonctionnalité sur de nombreux modèles d'ESP32, le choix de l'ESP32-S2 en particulier n'est pas important. Tout ce qui va suivre fonctionnera tout aussi bien avec les ESP32-C3 ou ESP32-S3 qui avec l'ESP32-S2 sont aujourd'hui les modèles qu'on trouve le plus fréquemment sur les modules courants et économiques (entre 3 € et 6 €). Notez que l'ESP32 (tout court) possède également un accélérateur cryptographique, mais il est bien moins évolué que celui des ESP32 plus récents, ne proposant par exemple que les fonctions de hachage SHA-* et non l'algorithme **cryptographique** HMAC SHA-*. De ce fait, les fonctions dont nous avons besoin ne sont pas disponibles dans l'API pour ce modèle, il vous faut un ESP32 moderne.

Le matériel utilisé ici sera un module Wemos S2 Mini construit autour d'un ESP32-S2FN4R2 disposant de 4 Mio de flash intégrés ainsi que de 2 Mio de PSRAM interne. C'est une solution compacte qui se résume au SoC, un circuit d'alimentation, un quartz et quelques composants passifs, mais pas de flash ou PSRAM SPI externe, ni de convertisseur USB/série type CP2102 ou CH340 (ce qui soulève un intéressant problème que nous allons régler).

1. CONFIGURATION GÉNÉRALE : ESP-IDF 5.1

L'utilisation des SoC et microcontrôleurs Espressif n'est pas une nouveauté dans les pages de ce magazine, mais cela fait quelque temps que nous n'avons pas parlé de l'environnement de développement ESP-IDF (Hackable 34, juillet 2020), de son installation et de sa configuration. La dernière version en date à ce jour est la 5.1.2 (16/11/2023) et celle-ci est installable aussi bien sous Windows avec un installeur dédié **[3]**, sous macOS et bien entendu, sous GNU/Linux. Notez qu'il existe également un plug-in pour l'IDE Eclipse, ainsi qu'une extension pour VSCode si vous appréciez ce type d'environnement. Personnellement, mon système de prédilection pour ce type de développement est GNU/Linux, complété d'un bon vieux shell et d'un éditeur de code type Vi (NeoVim, pour être précis), mais adapter les explications qui vont suivre ne devrait pas poser de problème, étant donné la nature universelle d'ESP-IDF.

L'installation de l'environnement Espressif ne nécessite aucune modification système ni aucun privilège particulier. Quelle que soit la distribution GNU/Linux que vous utilisez, ceci devrait donc parfaitement fonctionner avec votre utilisateur courant. Tout ce qu'il vous faut, c'est Git, Python avec le module `venv` pour la création d'environnements virtuels (paquet `python3.11-venv` avec une Debian/Devuan/Raspbian actuelle) et CMake. La première chose à faire consistera à cloner le dépôt Git officiel hébergé sur GitHub avec :

```
$ cd
$ mkdir ESP32
$ cd ESP21
$ git clone --recurse-submodules https://github.com/espressif/esp-idf.git
Clonage dans 'esp-idf'...
remote: Enumerating objects: 529213, done.
remote: Counting objects: 100% (4671/4671), done.
[...]
```

La version ainsi téléchargée sera celle de développement et non une *release* stable. Notez l'utilisation de l'option `--recurse-submodules` afin de récupérer également les éléments provenant d'autres dépôts Git dont l'environnement dépend (bibliothèques, pile réseau, système de fichiers en flash, etc.). J'ai pour habitude de maintenir sur mon PC plusieurs versions d'ESP-IDF pour certains projets un peu particuliers et, de ce fait, je renomme habituellement le répertoire obtenu pour clairement faire mention de la version stable utilisée et je bascule sur la version stable désirée :

```
$ mv esp-idf esp-idf51
$ cd esp-idf51
$ git checkout v5.1.2
M       components/bt/controller/lib_esp32
M       components/bt/controller/lib_esp32c3_family
M       components/bt/controller/lib_esp32c6/esp32c6-bt-lib
M       components/bt/controller/lib_esp32h2/esp32h2-bt-lib
M       components/bt/host/nimble/nimble
M       components/esp_wifi/lib
M       components/heap/tlsf
M       components/lwip/lwip
Note : basculement sur 'v5.1.2'.
```

Une autre approche pour obtenir un environnement dans une version stable précise est de, tout simplement, pointer son navigateur sur la page **https://github.com/espressif/esp-idf/releases** et télécharger l'archive en ZIP ou `.tar.gz`. Une autre option est de directement cloner le dépôt dans la bonne version avec l'option `-b v5.1.2` de `git clone` par exemple. Il est également parfaitement possible d'utiliser tout cela dans la version actuelle de développement (*master*), sans basculer sur une version *release* et de mettre régulièrement à jour avec (`git pull` et `git submodule update`).

Les quelque 1,4 Gio de fichiers ainsi récupérés constituent l'environnement presque complet. En effet, pour être utilisable, il faudra également récupérer les différentes chaînes de compilation (Xtensa et RISC-V) permettant de produire des binaires à destination des ESP32 de son choix. Ceci peut se faire très facilement en invoquant le script `./install.sh` présent dans le répertoire courant :

```
$ ./install.sh
Detecting the Python interpreter
Checking "python3" ...
Python 3.11.2
"python3" has been detected
Checking Python compatibility
Installing ESP-IDF tools
Current system platform: linux-amd64
Updating /home/denis/.espressif/idf-env.json
[...]
All done! You can now run:
  . ./export.sh
```

Cela aura pour effet de vérifier l'environnement Python et compléter l'installation (localement avec `venv`) avec différents modules puis de télécharger quelque 2,6 Gio supplémentaires stockés dans `~/.espressif/`. À ce stade, tout est prêt à être utilisé et il ne restera qu'à activer l'environnement avec la commande `. ./export.sh` ou `source ./export.sh` :

```
$ source ./export.sh
Setting IDF_PATH to '/home/denis/ESP32/esp-idf51'
Detecting the Python interpreter
Checking "python3" ...
Python 3.11.2
"python3" has been detected
Checking Python compatibility
Checking other ESP-IDF version.
Adding ESP-IDF tools to PATH...
[...]
Done! You can now compile ESP-IDF projects.
Go to the project directory and run:

  idf.py build
```

L'objectif de ce projet est de créer un token TOTP « fixe » (sans batterie) dans sa déclinaison ESP32. Des produits commerciaux existent dans ce domaine, mais sont généralement limités quant au nombre de chiffres constituant le code utilisé pour l'authentification (6 sur les 10 que supporte le standard TOTP).

Ceci a pour but de vérifier l'installation, de définir un certain nombre de variables d'environnement et d'ajouter des chemins de recherche des binaires (`PATH`) pour rendre le tout utilisable dans le shell **courant**. Il vous faudra, en effet, *sourcer* le fichier `export.sh` à chaque ouverture de terminal virtuel pour initialiser l'environnement. Vous n'êtes pas obligé de vous trouver dans le répertoire en question et pouvez sourcer en spécifiant le chemin complet avec, par exemple, `source ~/ESP32/esp-idf51/export.sh`. Ajouter cette ligne à votre `~/.bashrc` est également une possibilité, tout comme définir un alias pour vous simplifier la vie (`alias initidf51='source ~/ESP32/esp-idf51/export.sh'` par exemple).

ESP-IDF utilise CMake pour construire vos projets, mais le script Python `idf.py` est là pour vous faciliter la vie et servir d'interface pour toutes sortes de tâches autres que la compilation. Nous pouvons procéder à un petit test rapide pour découvrir son utilisation.

Placez-vous dans un répertoire quelconque de votre système de fichiers, créé pour l'occasion (`~/SRC/ESP` par exemple) et initiez un nouveau projet avec :

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

Ceci aura pour effet de créer un répertoire `premier/` contenant le strict minimum pour démarrer un développement. On retrouve là un `CMakeLists.txt` spécifiant le nom du projet, ainsi qu'un sous-répertoire `main/` contenant le fichier source C `premier.c` et un autre `CMakeLists.txt` précisant les fichiers sources à utiliser. Personnellement, je renomme immédiatement le fichier C en `main.c` et modifie le `CMakeLists.txt` en conséquence, mais c'est juste une préférence personnelle (sans oublier un `git init && git add -A && git commit -m "first commit"` pour partir du bon pied).

Pour pouvez ensuite utiliser votre nouveau meilleur ami, `idf.py`, pour spécifier la cible à utiliser :

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

```
esp32c6
esp32h2

$ idf.py set-target esp32s2
Adding "set-target"'s dependency "fullclean"
  to list of commands with default set of options.
Executing action: fullclean
Build directory '/home/denis/SRC/premier/build'
  not found. Nothing to clean.
Executing action: set-target
[...]
-- Configuring done
-- Generating done
-- Build files have been written to:
/home/denis/SRC/premier/build
```

Suite à cette commande, un sous-répertoire `build/` a fait son apparition (à ajouter au `.gitignore`), ainsi qu'un fichier `sdkconfig` dont le contenu n'est pas sans rappeler celui d'un fichier de configuration d'un noyau Linux (et pour cause, puisque c'est effectivement *Kconfig* qui est à l'œuvre). Ce fichier contient la configuration courante du SDK pour le projet, mais avant de nous pencher sur le sujet, tentons une compilation :

```
$ idf.py build
Executing action: all (aliases: build)
Running make in directory /home/denis/SRC/premier/build
Executing "make -j 10 all"...
[  0%] Built target custom_bundle
[  0%] Generating memory.ld linker script...
[  0%] Generating ../../partition_table/partition-table.bin
[  0%] Generating project_elf_src_esp32s2.c
[  0%] Built target memory_ld
[  0%] Built target _project_elf_src
[...]
esptool.py v4.7.dev3
Creating esp32s2 image...
Merged 2 ELF sections
Successfully created esp32s2 image.
[...]
[100%] Built target app_check_size
[100%] Built target app
[...]
```

Cette première construction prend un certain temps puisque tous les éléments actuellement configurés doivent être compilés, mais le *build* est incrémental, de ce fait une nouvelle occurrence de la commande ne déclenchera pas un processus aussi long. Le résultat de la compilation se trouve dans `build/` sous la forme des fichiers `premier.elf` et `premier.bin` (nommés d'après le nom du projet et non du fichier source qui est maintenant `main.c`). Un

message en fin de construction vous précise que vous n'avez pas besoin d'utiliser un outil tiers pour placer ce programme dans la flash de la cible, `idf.py` sait parfaitement utiliser `esptool.py`, livré avec ESP-IDF, pour vous. Il vous suffit de connecter votre module ESP32 (éventuellement en le basculant en mode *bootloader* via GPIO0 à la masse) et utiliser `idf.py -p /dev/ttyACM0 flash`. L'option `-p` permet de spécifier le port série à utiliser (`ttyACM*` ou `ttyUSB*` selon le module et l'interface) et vous devez vous assurer de disposer des permissions adéquates pour y écrire (placer l'utilisateur dans le groupe `dialout` est normalement suffisant).

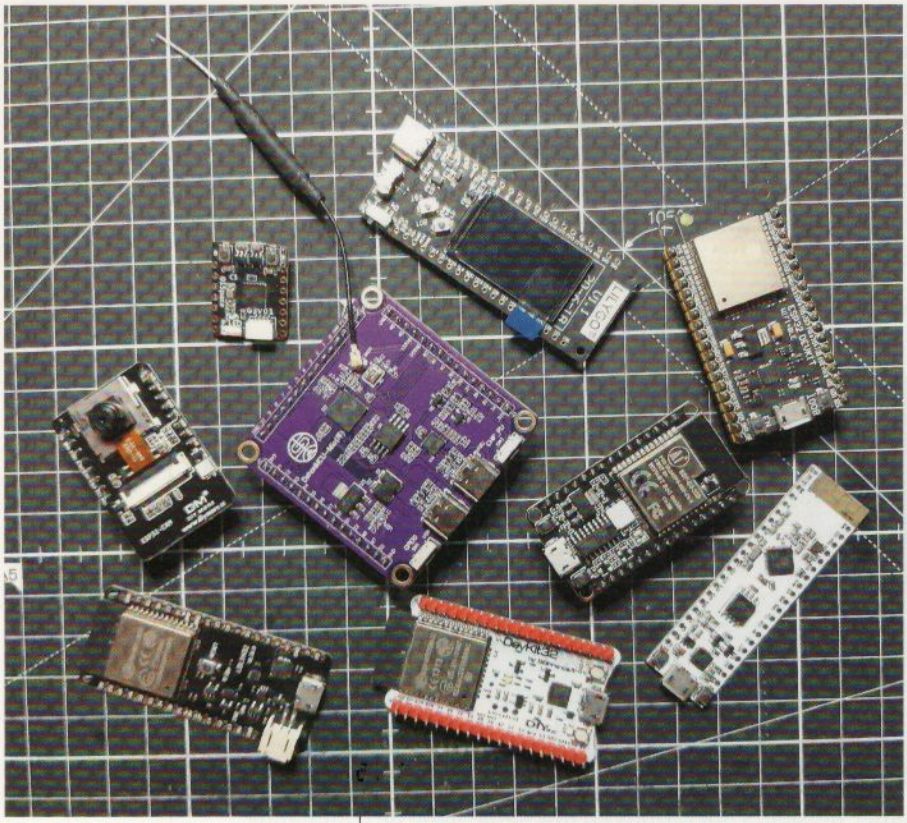
Une autre commande intéressante de `idf.py` est `size`, vous affichant différentes informations concernant l'occupation mémoire de votre petite création (SRAM et flash) :

```
Total sizes:
Used stat D/IRAM:   54167 bytes ( 121961 remain, 30.8% used)
      .data size:    6540 bytes
      .bss  size:    2000 bytes
      .text size:   44600 bytes
   .vectors size:    1027 bytes
Used Flash size :  118355 bytes
           .text:   82991 bytes
         .rodata:   35108 bytes
Total image size:  170522 bytes (.bin may be padded larger)
Built target size
```

Vous pouvez également combiner plusieurs opérations en une seule ligne de commande, `idf.py -p /dev/ttyACM0 clean build size flash` par exemple. Mais, avant de conclure cette partie, notons que la commande la plus intéressante est `menuconfig`, vous permettant, via une sympathique interface en mode texte, de configurer des éléments de votre projet et du SDK. Là, vous pourrez choisir quelles fonctionnalités activer ou non, spécifier le niveau de verbosité, utiliser le *secure boot*, régler les paramètres de la flash externe SPI, ou encore indiquer vos options de compilation favorites.

Dans le cas du module Wemos ESP32-S2, assez minimaliste, c'est via cette interface que nous allons régler deux problèmes. En effet, la connexion USB est directement prise en charge par le SoC et ne passe pas par un convertisseur USB/série comme sur d'autres cartes/modules. Ceci signifie que la sortie console est disponible pour votre code (`printf()`) lorsque celui-ci est en fonction et qu'en mode *bootloader*, c'est ce dernier qui la contrôle. Mais entre les deux, le port série disparaît, ce qui est rapidement pénible en phase de développement (impossible d'avoir un Minicom ouvert dans un second terminal). Mieux vaut alors utiliser un convertisseur USB/série pour les entrées/sorties d'une part et le port USB du module pour la programmation de l'autre.

En lien avec ce problème, et vous l'avez sans doute remarqué si vous avez fait l'essai, pour flasher l'ESP32, le passage en mode *bootloader* se fait manuellement avec une Wemos S2 Mini qui dispose d'un bouton « 0 » (pour GPIO0) et un bouton « RST » pour *reset*. En utilisant « RST » tout en maintenant « 0 » enfoncé, nous passons sur le *bootloader* et une pression sur « RST » seul démarre votre code. Le problème qui se pose concerne la sortie du



Les cartes ESP32 forment désormais un écosystème très diversifié. Forme, architecture, couleur, fonctionnalité, connectivité, richesse fonctionnelle… il y en a vraiment pour tous les goûts. Au point d'ailleurs qu'il devient presque difficile de s'y retrouver lorsqu'il s'agit de choisir une base pour un nouveau projet.

mode *bootloader* puisqu'il n'y a pas de ligne DTR sur ce port série afin de provoquer le *reset* automatiquement, comme c'est le cas avec un module utilisant une puce FT232R, CP2102 ou CH340. De ce fait, `esptool.py` est incapable de redémarrer après écriture de la flash et un message d'erreur est affiché.

Ces deux petits soucis peuvent être réglés via un simple `idf.py menuconfig`. Nous devons tout d'abord spécifier au SDK que la « console » standard ne doit plus être le périphérique CDC ACM USB du SoC, mais une liaison série standard (UART) connectée via deux GPIO. Rendez-vous donc dans ***Component config***, ***ESP System Settings*** et ***Channel for console output*** puis choisissez ***Custom UART***. Ne reste ensuite plus qu'à préciser les GPIO utilisées via ***Component config***, ***ESP System Settings*** et ***UART TX on GPIO#***, ainsi que ***UART RX on GPIO#***, en indiquant respectivement 37 et 38 (qu'il faudra bien entendu connecter à votre adaptateur USB/série).

Se débarrasser du message d'erreur post-écriture de la flash est tout aussi simple et se passe dans ***Serial flasher config*** et ***After flashing*** où il faut choisir ***Stay in bootloader***. Vous pouvez aussi en profiter pour ajuster ***Before flashing*** à ***No reset*** puisque tout est manuel et que cette action est parfaitement inutile.

Pour enregistrer ces changements, utilisez simplement la touche « q » et confirmez l'enregistrement. Votre fichier `sdkconfig` est mis à jour et la précédente version conservée sous le nom `sdkconfig.old`. Attention, changer la plateforme cible réinitialisera totalement la configuration et vous perdrez vos réglages. Ce qui est, somme toute, normal puisque ce choix est généralement fait au départ et ne devrait pas être changé par la suite.

Tout est maintenant prêt pour débuter un nouveau code sur la base de ce projet vide.

2. ARDUINO VS LE RESTE DU MONDE

Le choix d'une plateforme pour un nouveau projet repose, à mon sens, sur trois éléments importants : les fonctionnalités de la plateforme matérielle (RAM, périphériques, vitesse, etc.), le langage et l'environnement de développement et enfin l'écosystème complet. J'entends par là l'existence d'une communauté importante, active, produisant du code sous une licence digne de ce nom (BSD, MIT, GPL, etc.) et aux objectifs diversifiés. L'idée est de pouvoir reposer sur le code existant, que ce soit sous forme d'implémentation ou de bibliothèques, et bien entendu, de contribuer soi-même de la même manière par la suite.

C'est un point sur lequel Arduino excelle généralement avec une très très vaste collection de « bibliothèques », certes de qualité assez variable, mais permettant de prendre en charge une quantité phénoménale de périphériques, composants et circuits. Ce n'est pas tant l'inclusion dans l'environnement (ou l'IDE, dans le cas Arduino) qui importe, mais la facilité d'intégration dans son propre code. Côté Raspberry Pi Pico, en revanche, et en particulier lors de son lancement, prendre en charge une RTC, un afficheur, un capteur ou un écran LCD SPI était plus ou moins synonyme de « porter le code provenant d'Arduino » ou de « lire la *datasheet* et implémenter ». Les choses se sont un peu améliorées aujourd'hui, mais sont loin d'être au niveau d'Arduino (si ce n'est en utilisant la Pico avec Arduino).

Quid des ESP32 ? Bien sûr, il y a, là encore, le support Arduino et il est même possible d'utiliser cette compatibilité directement avec un projet reposant sur ESP-IDF (voir Hackable 24 **[4]**), mais je parle ici d'utiliser le SDK seul. Le problème était le même que pour Raspberry Pi Pico jusqu'à l'arrivée de l'*ESP-IDF Components library* de Ruslan V. Uss (alias UncleRus).

Ce sublime dépôt GitHub **[5]** est un ensemble de composants logiciels pour ESP-IDF et l'ESP8266 RTOS SDK cité précédemment, permettant de prendre en charge près d'une centaine de composants et périphériques, parmi les plus courants **[6]** : lm75, bmp180, bmp280, ds3502, mcp342x, max1704x, pcf8574, tsl2561, hd44780... et surtout le max7219 et le ds3231/ds1307, utilisés pour mon projet PicoTOTP **[7]**. Tout ceci est activement développé et complété au fil du temps, et s'intègre très facilement à votre environnement ESP-IDF.

Il vous suffit, en effet, de cloner le dépôt non loin de l'ESP-IDF (dans `~/ESP32` donc) et d'ajouter une simple ligne dans le `CMakeLists.txt` à la racine de votre projet :

```
set(EXTRA_COMPONENT_DIRS $ENV{HOME}/ESP32/esp-idf-lib/components)
```

Ceci fait, vous pourrez inclure dans votre code C n'importe quel *header* de cette collection de supports et directement utiliser les fonctionnalités. Notez qu'un certain nombre d'options sont également ajoutées dans la configuration (`menuconfig`), dans le menu ***Component config***. Ceci concerne généralement des points spécifiques pour certains composants et/ou bus purement logiciels.

Forcément, le code développé/rassemblé par UncleRus ne couvrira pas nécessairement tous vos besoins, mais les contributions ne cessent d'être ajoutées et rien ne vous empêche de soumettre, vous aussi, du code pour apporter votre pierre à l'édifice.

3. L'HEURE : UN PROBLÈME, DEUX SOLUTIONS

3.1 La classique RTC

Fort de notre nouvelle capacité à prendre en charge une RTC du type DS3231, nous allons immédiatement en faire usage en guise d'exemple. Rappelons au passage que HOTP est un algorithme basé sur HMAC SHA-1 permettant de générer un code entre 4 et 10 chiffres à partir d'une clé secrète ainsi que de la valeur d'un compteur et que ce code est destiné à servir de second facteur d'authentification. TOTP est une déclinaison de HOTP utilisant la date et heure UNIX en guise de compteur, le code changeant ainsi automatiquement toutes les 30 secondes. TOTP étend également l'utilisation d'un HMAC pouvant utiliser SHA-256 et SHA-512.

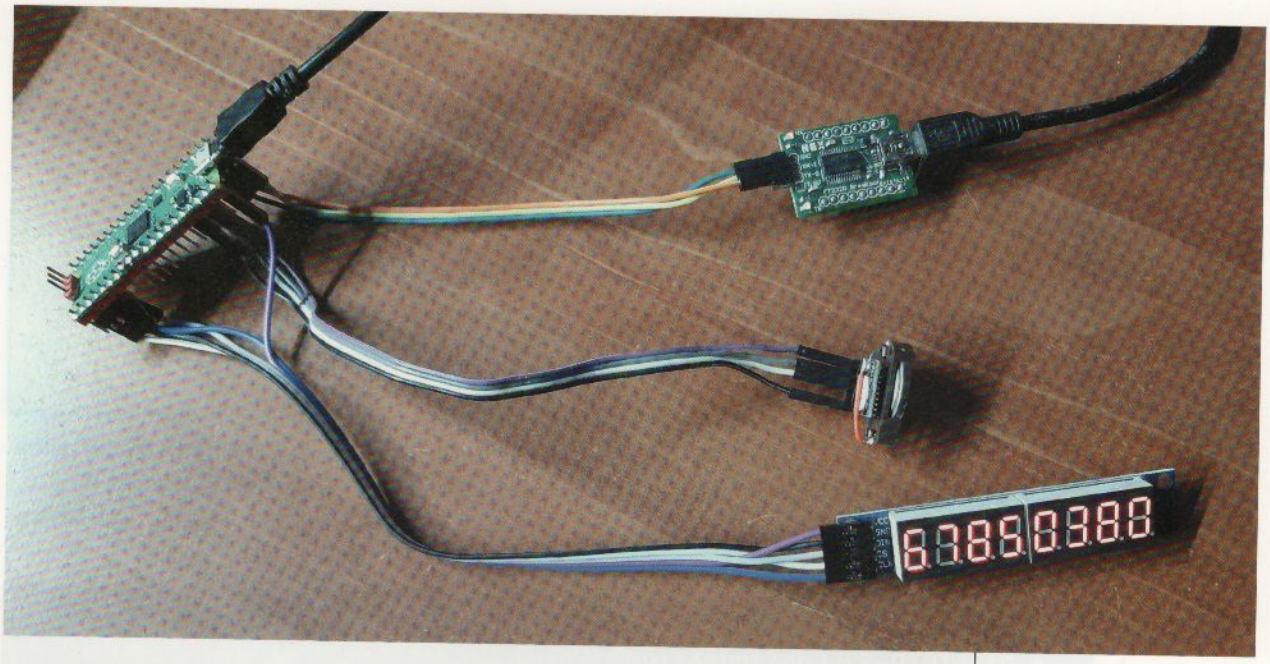
L'approche utilisée pour le projet basé sur Raspberry Pi Pico mettait en œuvre un module RTC construit autour d'une puce DS3231, utilisable avec n'importe quelle plateforme 0/3,3 V (Raspberry Pi incluse) et nous ferons de même ici. La première chose à faire est d'inclure le *header* adapté, spécifier deux macros pour désigner les GPIO utilisées pour le bus i²c et déclarer la variable permettant d'accéder au bus en question globalement :

```
#include <ds3231.h>

#define TOTP_I2C_SCL 35
#define TOTP_I2C_SDA 33

i2c_dev_t dev;
```

Avec ESP-IDF, nous n'avons pas de `main()` mais un `app_main()`, car même si cela est parfaitement transparent dans le cas de l'utilisation d'ESP32 avec Arduino, votre code n'est pas *baremetal* ou, en d'autres termes, exécuté directement par le silicium. Les ESP32 font fonctionner un système temps réel appelé FreeRTOS et votre code est en réalité une « application » et une tâche dans ce système. Le point d'entrée de cette application est `app_main()` et c'est cette fonction qui est déclarée de base quand vous créez un nouveau projet. Ici, nous aurons donc :

```
void app_main(void)
{
    if (i2cdev_init() != ESP_OK) {
        printf("Unable to init i2c device!\n");
        abort();
    }

    if (ds3231_init_desc(&dev, 0,
      TOTP_I2C_SDA, TOTP_I2C_SCL) != ESP_OK) {
        printf("Unable to init RTC device!\n");
        abort();
    }
[...]
```

Notez que je ne conclus pas le code présenté ici et la portée de la fonction, car dans l'état, si la seule tâche présente s'arrête, on se retrouve dans une situation qui ne devrait pas avoir lieu. Si vraiment vous tenez à développer pas à pas et tester au fil de l'eau, vous ne pouvez pas laisser



La version originale du projet PicoTOTP faisait usage d'une carte Raspberry Pi Pico, complétée d'une RTC et d'un afficheur 8 fois 7 segments pour la présentation du code. En version ESP32, du fait de la présence du Wi-Fi, nous pouvons éliminer la RTC et opter pour une synchronisation via SNTP.

votre processus quitter. Écartez de suite l'idée d'ajouter une boucle infinie avec un simple `while(1) { ; }`, ceci ne fonctionnera pas non plus. Là, votre processus ne laisse plus aucune chance au système de faire quoi que ce soit et/ou de passer la main aux autres tâches (FreeRTOS est multitâche) et le *watchdog* entre directement en action pour redémarrer l'ESP32. Pour faire les choses proprement, vous devez ajouter un délai pour mettre votre processus en pause, ce qui nous donne quelque chose comme :

```
while(1) {
    vTaskDelay(1000 / portTICK_PERIOD_MS);
}
```

`portTICK_PERIOD_MS` correspond à la période d'un *tick* système dont la fréquence est par défaut de 100 Hz (`CONFIG_FREERTOS_HZ` dans la configuration). De ce fait, le délai de pause que nous mettons en place est d'une seconde (1000 *ticks* divisés par la période d'un *tick* de 100 Hz en millisecondes nous donnent 1 seconde).

Avant d'aller plus loin concernant la RTC, revenons un instant sur ces deux macros pour les GPIO utilisées. Ceci n'est pas très pratique, car si je souhaite mon bus i²c sur d'autres broches, me voici obligé d'éditer mon code. Il existe une solution bien plus intéressante et celle-ci repose sur le système de configuration existant que nous pouvons tout simplement étendre en fonction de nos besoins. Nous pouvons par exemple ajouter un menu spécifique à notre projet où nous pourrons spécifier les GPIO pour SCL et SDA. Pour ce faire, il suffit de créer un fichier supplémentaire dans `main/`, appelé `Kconfig.projbuild` :

```
menu "PicoTOTP-S2 Configuration"
        config TOTP_I2C_MASTER_SCL
                int "i2c master SCL GPIO"
                default 35
                help
                        This GPIO is used for SCL signal

        config TOTP_I2C_MASTER_SDA
                int "i2c master SDA GPIO"
                default 33
                help
                        This GPIO is used for SDA signal
endmenu
```

Ceci fait, un `idf.py menuconfig` nous présentera notre entrée ***PicoTOTP-S2 Configuration*** dans le menu racine et nous pouvons saisir nos valeurs (si elles doivent être différentes de celles par défaut spécifiées ici). Une fois la configuration utilisée, le fichier `sdkconfig` reflétera nos choix :

```
$ grep TOTP sdkconfig
# PicoTOTP-S2 Configuration
CONFIG_TOTP_I2C_MASTER_SCL="35"
CONFIG_TOTP_I2C_MASTER_SDA="33"
# end of PicoTOTP-S2 Configuration
```

Les noms qui apparaissent sur ces deux lignes sont ceux des macros qui seront définies pour nous. Il nous faudra alors supprimer celles que nous avons insérées manuellement dans le code, et remplacer les occurrences de `TOTP_I2C_SCL` et `TOTP_I2C_SDA` par `CONFIG_TOTP_I2C_MASTER_SCL` et `CONFIG_TOTP_I2C_MASTER_SDA`. Ceci peut, bien entendu, être utilisé avec n'importe quel élément configurable utilisé dans le code et les GPIO sont généralement de bons candidats. Prenez cependant garde au type spécifié dans le `Kconfig.projbuild` (ici, `int`), sinon vous aurez de mauvaises surprises. Les types utilisables sont `bool`, `tristate`, `string`, `hex`, et `int`.

Revenons à nos moutons en récupérant l'heure configurée dans la RTC :

```
struct tm dsts;
[...]
    if (ds3231_get_time(&dev, &dsts) != ESP_OK) {
        printf("Could not get time\n");
        abort();
    }
```

Et affichons-la :

```
char buf[80];
[...]
    strftime(buf, sizeof(buf), "%b %d %Y %H:%M:%S", &dsts);
    printf("RTC set to: %s UTC (%llu)\n", buf, mktime(&dsts));
```

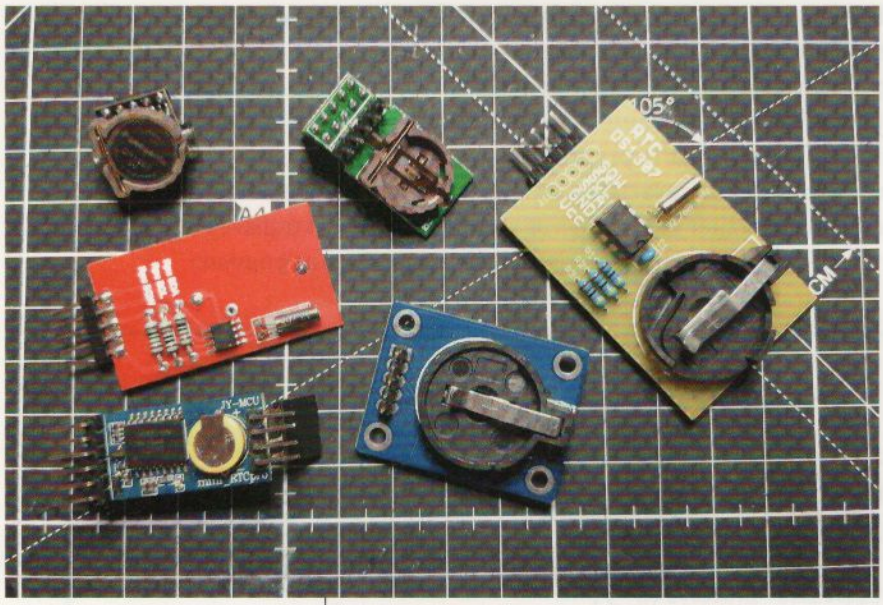
Les fonctions et structures utilisées sont parfaitement standard et POSIX. À terme, je pourrais procéder exactement comme avec la version Pico et permettre un réglage de l'heure via la liaison série (mais il y a beaucoup mieux, cf. plus loin). En attendant, une bonne solution pour définir l'heure dans une RTC qui pense être en 1900, faute de pile en état, est de reposer sur deux macros présentes et prédéfinies pour nous par le préprocesseur : `__DATE__` et `__TIME__`. Celles-ci contiennent, sous la forme de chaînes de caractères, respectivement la date et l'heure du moment où le préprocesseur a ouvert le fichier C. Ces macros sont généralement utilisées accompagnées de `__FILE__` et `__LINE__` pour certains messages d'erreur, mais nous pouvons les détourner à notre avantage :

```
time_t epoch;
struct tm buildts;
[...]
    if (dsts.tm_year < 123) {
        printf("BAD YEAR ! Setting RTC from build date/time.\n");
        if (strptime(__DATE__ " " __TIME__,
                     "%b %d %Y %H:%M:%S", &buildts) == NULL) {
            printf("strptime ERROR!\n");
        } else {
            epoch = mktime(&buildts);
            epoch -= 3600; // CET -> UTC
            buildts = *localtime(&epoch);
            if (ds3231_set_time(&dev, &buildts) != ESP_OK) {
                printf("Could not set time!\n");
                abort();
            }
            if (ds3231_get_time(&dev, &dsts) != ESP_OK) {
                printf("Could not get time (after setting)\n");
                abort();
            }
            strftime(buf, sizeof(buf), "%b %d %Y %H:%M:%S", &dsts);
            printf("RTC reset to: %s UTC\n", buf);
        }
    }
```

Attention cependant, `__TIME__` est l'heure **locale** du système alors que nous souhaitons stocker l'heure UTC. Et, non, la solution ici (`epoch -= 3600`) n'est pas parfaite, car dans l'absolu, il faudrait prendre en compte ces maudits passage à l'heure d'hiver/d'été (CET vs CEST) qui perturbe deux fois pas an le cycle circadien de tous et le mien en particulier.

3.2 Et pourquoi pas en Wi-Fi avec NTP ?

Avec un ESP32, nous avons une connectivité Wi-Fi (sauf certains rares modèles). Pourquoi donc ne pas abandonner la RTC et profiter de cela pour connaître l'heure exacte à chaque démarrage ? L'ESP32-S2 et le SDK disposent de tout le nécessaire. Nous avons non seulement une RTC embarquée au SoC (mais pas de fonctionnement autonome sur pile comme avec



Les modules RTC existent également en toutes formes, couleurs et qualités, mais seule une poignée de modèles de circuits intégrés sont généralement utilisés (DS1307, DS1338, DS3132, etc.). Le critère de choix est généralement la tension d'alimentation à utiliser, car certains composants, comme le DS1307 ne fonctionnent qu'en 0/5 V et sont incompatibles, de base, avec des plateformes utilisant 0/3,3 V comme les ESP32, les RP2040 ou même les Raspberry Pi.

un module DS3231), mais FreeRTOS et le SDK savent gérer cela tout à fait correctement. Inutile donc de quémander l'information à chaque calcul, que ce soit à un composant satellite ou un serveur.

NTP ou plus exactement SNTP (*Simple Network Time Protocol*) ne date pas d'hier (RFC 1361), c'est un protocole très connu et utilisé permettant de très simplement mettre à l'heure et synchroniser des horloges de multiples systèmes sur la base d'une référence fournie par un serveur (généralement public). SNTP est une déclinaison simplifiée de NTP et celle-ci est parfaitement adaptée à un microcontrôleur comme l'ESP32. Mais avant de pouvoir en faire usage, nous devons d'abord nous connecter à notre point d'accès Wi-Fi et ici, les choses sont très différentes de la simplicité à laquelle on est habitué, par exemple, avec un environnement Arduino (ESP8266 et ESP32).

La complexité est cependant mitigée par deux éléments. Premièrement, il suffit d'utiliser le code de démonstration fourni avec le SDK ESP-IDF **[8]**. C'est simple, direct, et on peut comprendre l'architecture générale de l'initialisation et de la configuration, sans pour autant entrer excessivement dans le détail. Et deuxièmement, c'est quelque chose à n'implémenter qu'une fois, si l'on approche le problème de façon modulaire. Ainsi, plutôt que d'intégrer brutalement le code dans son principal (et unique, à ce stade) fichier source C, il me parait plus judicieux de le stocker dans un fichier source distinct qui, reposant en partie sur le mécanisme de configuration que nous venons de voir, sera donc parfaitement générique et susceptible d'être copié « bêtement » dans tous nos nouveaux projets. Ainsi, j'ai pour habitude de créer (et de réutiliser) un fichier `wifistuff.c` contenant la gestion de la connectivité Wi-Fi et celui-ci est accompagné d'un `wifistuff.h` contenant simplement le prototype de la fonction d'initialisation :

```
#pragma once
void wifi_init_sta(void);
```

Le reste sera, bien sûr, implémenté dans le fichier source C `wifistuff.c` que nous survolerons rapidement, à commencer par les habituels *headers*, variables et macros :

```
#include <esp_wifi.h>
#include <esp_event.h>
#include <esp_log.h>
#include <lwip/ip_addr.h>
#include <lwip/err.h>
#include <lwip/sys.h>
#include <freertos/event_groups.h>

#define WIFI_CONNECTED_BIT BIT0
#define WIFI_FAIL_BIT      BIT1

static EventGroupHandle_t s_wifi_event_group;
static const char *TAG = "picototpS2";
static int s_retry_num = 0;
```

Tout le système est structuré sur la notion d'événements avec un processus chargé de procéder à la connexion, déclenchant différents événements que nous devons capturer et traiter. Ici, seuls deux d'entre eux nous intéressent, à savoir une connexion réussie ou un échec. Ceci est traité par une fonction dédiée, le *event handler* :

```
static void event_handler(void* arg, esp_event_base_t event_base,
        int32_t event_id, void* event_data)
{
    if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_START) {
        esp_wifi_connect();
    } else if (event_base == WIFI_EVENT && event_id == WIFI_EVENT_STA_DISCONNECTED) {
        if (s_retry_num < CONFIG_ESP_MAXIMUM_RETRY) {
            esp_wifi_connect();
            s_retry_num++;
            ESP_LOGI(TAG, "retry to connect to the AP");
        } else {
            xEventGroupSetBits(s_wifi_event_group, WIFI_FAIL_BIT);
        }
        ESP_LOGI(TAG,"connect to the AP fail");
    } else if (event_base == IP_EVENT && event_id == IP_EVENT_STA_GOT_IP) {
        ip_event_got_ip_t* event = (ip_event_got_ip_t*) event_data;
        ESP_LOGI(TAG, "got ip:" IPSTR, IP2STR(&event->ip_info.ip));
        printf("got ip:" IPSTR "\n", IP2STR(&event->ip_info.ip));
        s_retry_num = 0;
        xEventGroupSetBits(s_wifi_event_group, WIFI_CONNECTED_BIT);
    }
}
```

Ce gestionnaire d'événements sommaire est à l'écoute et analyse l'information qui lui parvient en relayant au reste du « système » via un *event group*. Ce mécanisme assez touffu est la base de la communication interprocessus dans FreeRTOS et est largement documenté sur le site officiel **[9]**. Notez l'utilisation récurrente des macros `ESP_LOGI` et `ESP_LOGE` permettant

d'envoyer des messages (respectivement d'information et d'erreur) dans le journal d'activité et donc sur la console. La configuration du SDK (via `menuconfig`) nous permet d'ajuster le niveau de verbosité : menu ***Component config***, ***Log output*** et ***Default log verbosity*** avec un choix allant de ***No output*** à ***Verbose***, en passant par ***Error***, ***Warning***, ***Info*** et ***Debug***. Le choix par défaut est réglé sur ***Info***, ce qui est intéressant en phase de développement, mais sera rapidement ramené à ***Error*** pour limiter l'avalanche de messages. Ceci explique pourquoi nous avons deux mentions de l'adresse IP obtenue, un envoi dans les *logs* avec `ESP_LOGI` et un affichage plus sobre via `printf()` si la verbosité est réduite.

S'Y RETROUVER DANS LES MENUS

Vous n'êtes pas obligé de mémoriser l'emplacement de chaque élément de configuration dans l'arborescence de menus. Chacun d'eux possède un nom qu'on retrouve dans le fichier `sdkconfig` préfixé de `CONFIG_`. Ce nom peut être utilisé pour une recherche avec le raccourci « / » et l'interface vous listera les choix possibles et vous conduira au bon endroit avec une simple validation.

Dans la suite de l'article, je ne préciserai donc plus les menus, mais directement le nom (et macro) correspondant.

L'initialisation, la configuration et la mise en place du gestionnaire d'événements se feront dans `wifi_init_sta()` :

```
void wifi_init_sta(void)
{
    s_wifi_event_group = xEventGroupCreate();

    ESP_ERROR_CHECK(esp_netif_init());

    ESP_ERROR_CHECK(esp_event_loop_create_default());
    esp_netif_create_default_wifi_sta();

    wifi_init_config_t cfg = WIFI_INIT_CONFIG_DEFAULT();
    ESP_ERROR_CHECK(esp_wifi_init(&cfg));

    esp_event_handler_instance_t instance_any_id;
    esp_event_handler_instance_t instance_got_ip;
    ESP_ERROR_CHECK(esp_event_handler_instance_register(WIFI_EVENT,
                ESP_EVENT_ANY_ID, &event_handler, NULL, &instance_any_id));
    ESP_ERROR_CHECK(esp_event_handler_instance_register(IP_EVENT,
                IP_EVENT_STA_GOT_IP, &event_handler, NULL, &instance_got_ip));

    wifi_config_t wifi_config = {
        .sta = {
            .ssid = CONFIG_ESP_WIFI_SSID,
            .password = CONFIG_ESP_WIFI_PASSWORD,
```

```
            .threshold.authmode = WIFI_AUTH_WPA_PSK,
            .sae_pwe_h2e = WPA3_SAE_PWE_BOTH,
            .sae_h2e_identifier = "",
        },
    };

    ESP_ERROR_CHECK(esp_wifi_set_mode(WIFI_MODE_STA) );
    ESP_ERROR_CHECK(esp_wifi_set_config(WIFI_IF_STA, &wifi_config) );
    ESP_ERROR_CHECK(esp_wifi_start() );

    ESP_LOGI(TAG, "wifi_init_sta finished.");

    EventBits_t bits = xEventGroupWaitBits(s_wifi_event_group,
            WIFI_CONNECTED_BIT | WIFI_FAIL_BIT,
            pdFALSE, pdFALSE, portMAX_DELAY);

    if (bits & WIFI_CONNECTED_BIT) {
        ESP_LOGI(TAG, "connected to ap SSID:%s", CONFIG_ESP_WIFI_SSID);
        printf("Connected to AP SSID: %s\n", CONFIG_ESP_WIFI_SSID);
    } else if (bits & WIFI_FAIL_BIT) {
        ESP_LOGI(TAG, "Failed to connect to SSID:%s", CONFIG_ESP_WIFI_SSID);
    } else {
        ESP_LOGE(TAG, "UNEXPECTED EVENT");
    }
}
```

`CONFIG_ESP_WIFI_SSID` et `CONFIG_ESP_WIFI_PASSWORD`, tout comme `CONFIG_ESP_MAXIMUM_RETRY` de la fonction précédente, proviennent du mécanisme de configuration, avec le fichier `main/Kconfig.projbuild` suivant :

```
menu "PICOTOTPS2 Configuration"
        config ESP_WIFI_SSID
                string "WiFi SSID"
                default "myssid"
                help
                        SSID (network name) to connect to.

        config ESP_WIFI_PASSWORD
                string "WiFi Password"
                default "mypassword"
                help
                        WiFi password to use.

        config ESP_MAXIMUM_RETRY
                int "Maximum retry"
                default 5
                help
                        Maximum retry to avoid infinite loop.
endmenu
```

Un autre critère de choix très important pour les modules RTC est la présence d'une pile remplaçable et non d'un élément soudé au circuit. Dans le cas contraire, lorsque la pile est incapable de fournir le courant adéquat, il n'y a d'autre solution que de bidouiller quelque chose de vaguement présentable pour pouvoir continuer à utiliser le module.

Remarquez l'utilisation de la macro `ESP_ERROR_CHECK`, similaire à un `assert()` et évitant des `if (fonction(arg1, arg2) != ESP_OK) { }` à répétition. Ceci est cependant à utiliser avec parcimonie, car en cas d'erreur, c'est `abort()` qui entre en jeu et *reboote* brutalement l'ESP32, ce qui n'est pas forcément ce qu'on préfère (certains projets peuvent parfaitement tolérer une erreur de connexion, par exemple).

Pour utiliser ce code, on n'oubliera pas d'ajouter le fichier source `wifistuff.c` dans le `main/CMakeLists.txt` en compagnie de `main.c`. Il ne nous restera plus ensuite qu'à inclure `wifistuff.h` dans ce dernier pour, très simplement, utiliser :

```
#include <nvs_flash.h>
[...]
    // Initialisation NVS
    esp_err_t ret = nvs_flash_init();
    if (ret == ESP_ERR_NVS_NO_FREE_PAGES
     || ret == ESP_ERR_NVS_NEW_VERSION_FOUND) {
        ESP_ERROR_CHECK(nvs_flash_erase());
        ret = nvs_flash_init();
    }
    ESP_ERROR_CHECK(ret);

    // Connexion Wi-Fi
    printf("Connecting to wifi...\n");
    wifi_init_sta();
[...]
```

Nous sommes ici obligés d'initialiser la mémoire non volatile (NVS pour *Non-volatile storage*), c'est-à-dire une partie de la flash utilisée comme zone de stockage persistant. En effet, par défaut le SDK est configuré pour stocker non seulement la configuration concernant la connexion, mais également les paramètres de calibration afin d'éviter une calibration complète de la couche physique (liaison radio) à chaque connexion. Nous devons donc soit initialiser cet espace de stockage en début de programme et **avant** la configuration du Wi-Fi, soit désactiver ces fonctionnalités (`ESP_WIFI_NVS_ENABLED` et `ESP_PHY_CALIBRATION_AND_DATA_STORAGE`) via *menuconfig*.

Une fois ceci compilé et chargé en flash, nous pouvons constater que nous avons « du Net ». Nous pouvons donc passer à la suite et nous en prendre au serveur SNTP. Pour cela, rien de plus simple puisque tout est déjà à disposition dans le SDK :

```
    time_t now;
    struct tm ntpts;
[...]
    // Configuration du serveur
    esp_sntp_config_t config = ESP_NETIF_SNTP_DEFAULT_CONFIG("pool.ntp.org");
    esp_netif_sntp_init(&config);

    // Demande et attente
    printf("Syncing time...\n");
    if (esp_netif_sntp_sync_wait(pdMS_TO_TICKS(10000)) != ESP_OK) {
        printf("Failed to update system time within 10s timeout");
        abort();
    }

    // Affichage
    time(&now);
    localtime_r(&now, &ntpts);
    strftime(buf, sizeof(buf), "%b %d %Y %H:%M:%S", &ntpts);
    printf("NTP time set to: %s UTC (%llu)\n", buf, mktime(&ntpts));
```

Les structures et types de variables utilisés sont les mêmes qu'en reposant sur une RTC (ou d'ailleurs qu'avec un système POSIX) et `esp_netif_sntp_sync_wait()` règle l'horloge interne. Parce que, oui, l'ESP32 dispose d'une RTC (un *RTC timer*) parfaitement gérée par FreeRTOS, exactement comme si nous étions en train de travailler avec un système GNU/Linux ou [Free|Open|Net]BSD. `time()` nous retourne donc l'heure POSIX (*epoch*) dans `now` et nous l'utilisons pour remplir une structure `tm`, qui est ensuite utilisée pour formater une belle chaîne de caractères. Toutes les fonctions habituelles sont là (`gettimeofday`, `localtime`, `mktime`, etc.). C'est bien plus simple qu'un module RTC, mais oui, vous exposez votre ESP32 au réseau…

Notez que rien ne vous empêche d'utiliser les deux approches, SNTP et RTC, en établissant la connexion Wi-Fi au *boot*, pour uniquement récupérer l'heure via SNTP pour mettre à jour la RTC et couper la connexion par la suite. Les RTC de type DS3231 sont fort pratiques, mais une dérive d'horloge est toujours possible en fonction des conditions environnementales (température). TOTP est tolérant lors de la vérification d'un code soumis par l'utilisation et n'a donc pas besoin d'une horloge absolument exacte, mais cela peut être une approche intéressante.

4. ON DEVAIT PAS PARLER DE HMAC ?

La plupart des ESP32, dont l'ESP32-S2, disposent de fonctionnalités cryptographiques sous la forme d'un accélérateur dédié. Ceci est un avantage en termes de performances, puisque c'est le silicium qui va se charger directement des opérations mathématiques coûteuses en ressources CPU, mais ne présente pas d'avantage direct en termes de sécurité. Ce qui est important dans ce genre de situation est précisément le problème que nous avons rencontré avec la version Raspberry Pi Pico : le secret (une clé généralement) est parfaitement accessible et absolument pas protégé.

Avec l'ESP32, un mécanisme particulier est à l'œuvre, reposant sur la notion de *eFuse*. Dans la littérature technique Espressif, un eFuse est un champ d'un bit pouvant être programmé à 1, mais ne pouvant par revenir à 0 par la suite. C'est une sorte de PROM directement intégrée au composant. Ceci ne règle pas notre problème de secret, mais nous allons y venir. Ces eFuses sont utilisés en interne par quelques périphériques, mais une partie est accessible aux développeurs sous la forme de blocs de 256 bits. Dans le cas de l'ESP32-S2, il existe 11 blocs :

- `EFUSE_BLK0`, `EFUSE_BLK1` et `EFUSE_BLK2` sont réservés au système et chaque bit désigne une fonctionnalité ou un élément que l'utilisateur ne peut modifier directement (version, adresse MAC, calibration, etc.) ou des paramètres de configuration impactant l'usage des autres blocs ;
- `EFUSE_BLK3` (`EFUSE_BLK_USER_DATA`) est dédié à une utilisation par le développeur à sa discrétion ;
- `EFUSE_BLK4` à `EFUSE_BLK9` également référencés sous `EFUSE_BLK_KEY0` à `EFUSE_BLK_KEY5` permettent de stocker des clés pour les fonctions cryptographiques, le démarrage sécurisé (*secure boot*) ou le chiffrement de la flash ;
- `EFUSE_BLK10` (`EFUSE_BLK_SYS_DATA_PART2`) est un autre bloc réservé au système.

L'utilité de chaque bit des blocs « système » est décrite dans un fichier CSV intégré du SDK (`components/efuse/esp32s2/esp_efuse_table.csv`), mais ce qui nous intéresse ici, ce sont les blocs `EFUSE_BLK_KEY0` à `EFUSE_BLK_KEY5` que nous pouvons programmer avec la clé devant être utilisée pour le calcul HMA-SHA256. Notez que ceci est le seul algorithme disponible pour les plateformes ESP32 actuelles, nous n'avons pas de HMAC SHA1 ou SHA-512 (peut-être dans l'ESP32-P4 ?).

Enregistrer une clé dans un bloc est relativement simple. Exemple :

```
#include <esp32s2/rom/efuse.h>
#include <esp_efuse.h>
[...]
const uint8_t key_data[32] = {
    0x21, 0x8f, 0xc3, 0x19, 0x98, 0x37, 0x75, 0x83,
    0x0c, 0xca, 0xc7, 0xde, 0xec, 0xb3, 0x01, 0x56,
    0x7e, 0x97, 0xe4, 0xe7, 0x61, 0x78, 0x65, 0x5a,
    0xf0, 0x26, 0xc2, 0x14, 0x3e, 0x06, 0x21, 0xdf
};

[...]
    if (ets_efuse_write_key(ETS_EFUSE_BLOCK_KEY0,
                ETS_EFUSE_KEY_PURPOSE_HMAC_UP,
                key_data, sizeof(key_data)) != ESP_OK) {
        printf("ets_efuse_write_key failed!\n");
        abort();
    }
    printf("key written ok\n");
```

Attention, n'utilisez pas ce code n'importe comment même si, et justement parce que, il est relativement simple. Vous l'aurez compris, l'écriture d'eFuses est une opération irréversible et vous n'avez droit, dans l'absolu, qu'à 6 essais, car vous n'avez que 6 blocs. Le bloc `ETS_EFUSE_BLOCK_KEY0` ainsi écrit passe automatiquement en lecture seule et toute tentative de récidiver retournera une erreur. Il est donc judicieux, je pense, de ne pas permettre cette écriture au travers d'une interface utilisateur (comme avec PicoTOTP) ou éventuellement, de ne le faire qu'avec maintes étapes de confirmation. Remarquez que l'écriture de la clé avec cette fonction prend en argument le type d'utilisation de celle-ci, qui peut être (dans le cadre d'une utilisation en HMAC) :

- `ETS_EFUSE_KEY_PURPOSE_HMAC_UP` : génération de HMAC par l'application utilisateur ;
- `ETS_EFUSE_KEY_PURPOSE_HMAC_DOWN_DIGITAL_SIGNATURE` : signature électronique avec le module dédié où le HMAC est utilisé pour dériver les clés ;
- `ETS_EFUSE_KEY_PURPOSE_HMAC_DOWN_JTAG` : pour sécuriser l'utilisation de l'interface JTAG qui est en parallèle désactivée logiciellement (il est également possible de désactiver matériellement l'interface, mais celle-ci est alors totalement inutilisable) ;
- `ETS_EFUSE_KEY_PURPOSE_HMAC_DOWN_ALL` : pour la signature et le JTAG verrouillé logiciellement.

Une voie possible pour faire évoluer le projet serait de détacher la partie purement cryptographique du montage de base. Ce genre d'approche est déjà utilisée par des périphériques bancaires comme celui-ci, faisant usage d'une carte à puce et étant fabriqué par Gemalto. Le challenge cependant est d'arriver à faire communiquer la smartcard avec l'ESP32...

Pour une utilisation telle que la nôtre, à savoir faire générer des HMAC pour un usage propre à notre applicatif, c'est `ETS_EFUSE_KEY_PURPOSE_HMAC_UP` et, de ce fait, le bloc et la clé ne peuvent être utilisés pour une autre fonction. Il existe d'autres macros spécifiant l'usage d'un bloc, comme le montre l'énumération dans `esp32s2/rom/efuse.h` (chiffrement AES, par exemple), mais ceci sort du cadre du présent article.

Notez également que j'ai bien dit « lecture seule », car par défaut, l'écriture d'un bloc, même s'il est destiné à l'utilisation de fonctions cryptographiques, laisse ce dernier lisible avec `esp_efuse_read_block()`. Pour interdire cela, nous ajoutons donc :

```
if (esp_efuse_set_key_dis_read(ETS_EFUSE_BLOCK_KEY0) != ESP_OK) {
    printf("esp_efuse_set_key_dis_read failed!\n");
    abort();
}
```

Enfin, et même si cela coule de source, le choix de l'usage d'un bloc ainsi que la protection en lecture sont deux éléments stockés dans des eFuses. Les deux opérations sont donc tout aussi irréversibles que l'écriture des données. Utiliser une clé ou un bloc pour un usage qui ne lui correspond pas, tout comme tenter d'écrire une nouvelle fois les données, provoquera une erreur. Lire un bloc protégé, en revanche, fonctionnera sans problème, mais vous ne lirez que des `0x00`.

Pour limiter les risques d'erreur, en particulier en cas de fonctionnement interactif, nous pouvons choisir d'intégrer l'enregistrement d'une clé dans le système de configuration plutôt que de devoir exécuter un premier code pour écrire la clé, puis reflasher l'ESP32 avec le code devant en faire usage. Pour cela, nous ajoutons les éléments suivants dans `main/Kconfig.projbuild` :

```
config TOTP_KEY0_ENABLE
        bool "Register key0"
        default n
        help
                Enable key0 registration for HMAC SHA-256

config TOTP_KEY0
        string "HMAC SHA-256 KEY0"
        depends on TOTP_KEY0_ENABLE
        help
                HMAC key to registrer for Key0 (64 char = 32 bytes).
```

Nous avons ici une nouveauté puisque `TOTP_KEY0` dépend de `TOTP_KEY0_ENABLE` à *VRAI*. Ceci signifie que non seulement l'entrée « *HMAC SHA-256 KEY0* » n'apparaît pas dans le menu si « *Register key0* » n'est pas activé, mais ni `TOTP_KEY0_ENABLE` ni `TOTP_KEY0` ne sera défini dans le code (et dans `sdkconfig`). Nous avons également un problème à régler, car cet élément de configuration est un `string` et non un tableau d'octets. Le type `hex` peut être utilisé, mais est généralement fait pour des valeurs exprimées en hexadécimal et non les 256 bits d'une clé (qu'il faudrait de toute façon traiter également).

Nous avons donc besoin d'une fonction capable de convertir une chaîne de caractères composée de valeurs hexadécimales en un tableau de 32 `uint8_t` :

```
size_t hex2array(const char *line, uint8_t *data, size_t len)
{
    size_t datalen = 0;
    unsigned int temp;
    int indx = 0;
    char buf[5] = { 0 };

    if (strlen(line) < len*2)
        return(0);

    while (line[indx]) {
        if (line[indx] == '\t' || line[indx] == ' ') {
```

```
            indx++;
            continue;
        }
        if (isxdigit((unsigned char)line[indx])) {
            buf[strlen(buf) + 1] = 0x00;
            buf[strlen(buf)] = line[indx];
        } else {
            return(0);
        }
        if (strlen(buf) >= 2) {
            sscanf(buf, "%x", &temp);
            data[datalen] = (uint8_t)(temp & 0xff);
            *buf = 0;
            datalen++;
            if (datalen > len)
                return(0);
        }
        indx++;
    }
    if (strlen(buf) > 0 ) {
        return(0);
    }
    return(datalen);
}
```

Ceci sera stocké dans un fichier `util.c` en ajoutant :

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include <inttypes.h>
#include "util.h"
```

Et sera intégré, comme pour `wifistuff.c`, au fichier `main/CMakeLists.txt`. Dans notre `main.c`, nous pourrons alors agir en fonction de ces nouvelles macros :

```
#ifdef CONFIG_TOTP_KEY0_ENABLE
    if (hex2array(CONFIG_TOTP_KEY0, key_data, 32) != 32) {
        printf("hex2array error!\n");
        abort();
    }

    printf("Writing KEY0 to eFuse...");
    printf("Hex key to write: ");
    for (int i = 0; i < 32; i++)
        printf("[%02X]", key_data[i]);
```

```
    printf("\n");

    if (esp_efuse_get_key_dis_write(ETS_EFUSE_BLOCK_KEY0) ||
            esp_efuse_get_keypurpose_dis_write(ETS_EFUSE_BLOCK_KEY0)) {
        printf("EFUSE_BLK_KEY0 not ready. Key already set ?\n");
    } else {
        if (ets_efuse_write_key(ETS_EFUSE_BLOCK_KEY0,
                    ETS_EFUSE_KEY_PURPOSE_HMAC_UP,
                    key_data, sizeof(key_data)) != ESP_OK) {
            printf("ets_efuse_write_key() failed!\n");
            abort();
        }
        printf("KEY0 written.\n");

        if (esp_efuse_set_key_dis_read(ETS_EFUSE_BLOCK_KEY0) != ESP_OK) {
            printf("esp_efuse_set_key_dis_read() failed!\n");
            abort();
        }
        printf("KEY0 is read protected.\n");
    }
#endif
```

Notez bien que ceci vous impose tout de même de flasher une seconde fois votre ESP32, car du fait de la macro `CONFIG_TOTP_KEY0`, remplacée par la clé par le préprocesseur, celle-ci se trouve sous la forme d'une chaîne dans votre binaire et donc en flash. Il existe des solutions pour traiter ce genre de problème (comme chiffrer la flash, par exemple), mais il est bien plus simple de tout simplement reflasher avec une version où `CONFIG_TOTP_KEY0_ENABLE` n'est pas défini.

À présent que la clé est enregistrée et protégée des regards indiscrets, nous pouvons l'utiliser pour produire un HMAC :

```
    const char *message = "coucou";
    uint8_t hmac[32];
[...]
    esp_err_t result = esp_hmac_calculate(HMAC_KEY0,
        message, msg_len, hmac);

    if (result == ESP_OK) {
        printf("HMAC= ");
        for (int i = 0; i < 32; i++) {
            printf("%02x", hmac[i]);
        }
        printf("\n");
    } else {
        printf("esp_hmac_calculate failed!\n");
    }
```

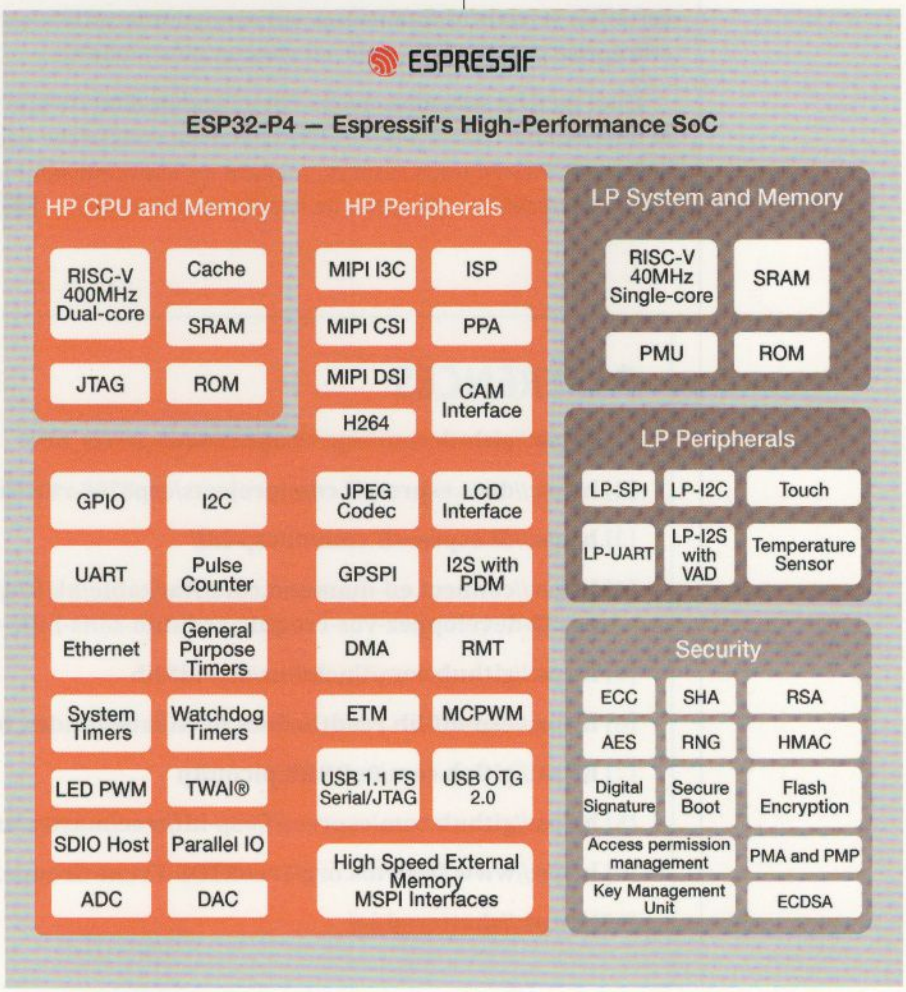
Nous pouvons très simplement vérifier le bon fonctionnement de l'opération en utilisant la commande `openssl` :

```
$ echo -n "coucou" | \
openssl sha256 -hex -mac HMAC -macopt \
hexkey:218fc319983775830ccac7deecb301567e97e4e76178655af026c2143e0621df
```

qui devrait nous afficher exactement la même suite de 32 valeurs que celle calculée par l'ESP32. Comme ce test fonctionne, il ne restera alors plus qu'à porter le reste du code de PicoTOTP que nous avions prévu de façon relativement modulaire à la base. Je ne sais pas si je vais effectivement ajouter un afficheur 8 fois 7 segments, même si supporté par l'*ESP-IDF Components library*, ou du moins pas seulement un tel afficheur. Une idée me trotte dans la tête sous la forme d'une simple question : et si on remplaçait les chiffres de 0 à 9 par des couleurs ? Il suffirait de 4 à 10 LED RGB adressables pour obtenir un résultat très amusant. Non ?

5. LE MOT DE LA FIN

Les fonctions cryptographiques intégrées au silicium sont une carence importante du RP2040 et sont, dans une certaine mesure, très appréciables sur plateforme ESP32. Ceci reste cependant très limité et même si SHA-256 est loin d'être obsolète (contrairement à SHA-1 qui l'est depuis 2011 et pour lequel des collisions ont été trouvées **[10]**), il pourrait être appréciable d'utiliser SHA-512 (d'autant que ceci est évoqué dans la RFC 6238) comme



Le futur ESP32-P4 semble disposer de fonctionnalités cryptographiques bien plus avancées que les modèles précédents, dont un support pour les algorithmes cryptographiques sur courbes elliptiques ECC et ECDSA. Peut-être aurons-nous aussi HMAC SHA-512 ? (Source : annonce Espressif de l'ESP32-P4)

le font les applications et *tokens* TOTP récents. Il en va de même pour le chiffrement et la signature, car même si les ESP32 sont pour le moins riches en fonctionnalités, il ne s'agit pas pour autant de puces spécialisées dans le domaine.

Si ceci est un besoin important, il faudra alors se tourner vers des composants distincts comme les *secure elements* offrant justement possibilité de déléguer les fonctions cryptographiques. C'est le cas par exemple du ATECC508 de MicroChip (ex-Atmel), présent sur la carte Arduino MKR 1010. Une autre option, que je risque d'explorer dans un prochain article, consiste à utiliser ce même principe de délégation, mais avec une carte à puce (ou SIM). La difficulté ici réside dans la communication entre le microcontrôleur et la carte. Des bibliothèques existent comme la *libiso7816* du projet WooKey de l'ANSSI ou encore le *fuzzer* Cardstalker (de l'ANSSI aussi, décidément), mais les deux implémentent la démonstration sur une plateforme de développement STM32, connue pour la « densité » de son *framework* STM32Cube/STM32CubeMX. Ceci peut être un challenge intéressant que d'arriver à faire fonctionner une communication entre ESP32 (ou RP2040, ceci dit) et une Java Card se chargeant des fonctions cryptographiques. Nous arriverions ainsi à quelque chose d'assez proche des *tokens* d'authentification que certaines structures financières utilisent pour la validation des opérations bancaires en entreprise par exemple, mais en version plus modeste...

Quoi qu'il en soit, nous avons fait ici le tour d'un certain nombre de points importants et intéressants concernant les ESP32 et je ne vous cache pas que j'ai hâte de voir ce que le futur ESP32-P4 nous réserve, sachant qu'Espressif semble maintenant avoir pris un tournant décisif en direction de l'architecture RISC-V... DB

RÉFÉRENCES

[1] **https://github.com/espressif/ESP8266_RTOS_SDK**

[2] **https://docs.espressif.com/projects/esp8266-rtos-sdk/en/latest/**

[3] **https://dl.espressif.com/dl/esp-idf/**

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

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

[6] **https://esp-idf-lib.readthedocs.io/en/latest/index.html**

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

[8] **https://github.com/espressif/esp-idf/tree/master/examples**

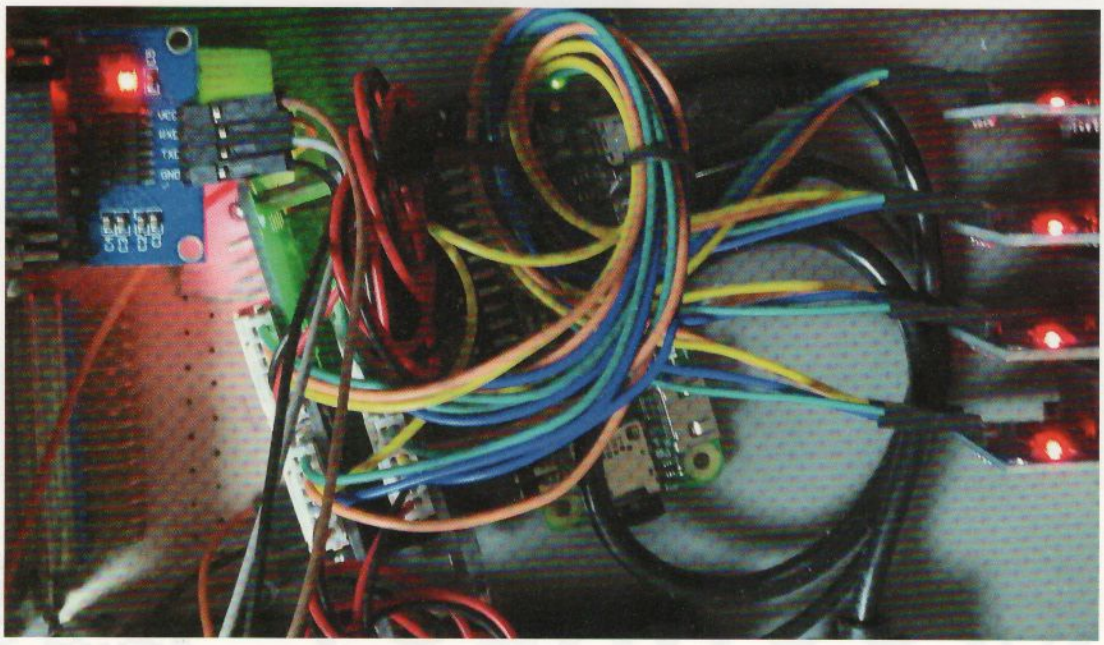
[9] **https://www.freertos.org/FreeRTOS-Event-Groups.html**

[10] **https://shattered.io/**

CRÉEZ VOTRE CONCENTRATEUR RS-232C

Cédric PELLERIN
Directeur technique Hoch Adler, développeur senior chez B<>Com,
utilisateur de GNU/Linux depuis 1993

Au commencement était le port série, tout ce qui a été créé l'a été fait par lui et rien de ce qui a été créé n'a été fait sans lui. Dans les années 90, il était courant de connecter un terminal à une station SUN ou à un serveur x86 afin d'éviter la multiplication des écrans et des claviers. Cette possibilité existe toujours, que ce soit sur une Raspberry Pi, un Arduino ou même un serveur *NIX. Ceci dit, que faire lorsque l'on doit interagir de loin avec plusieurs machines de ce type ? La solution s'appelle concentrateur série.



– Créez votre concentrateur RS-232C –

Lorsqu'on a la possibilité de regrouper la plupart des éléments actifs d'un réseau, j'entends par là serveurs, *switches*, etc., il est plaisant d'avoir un terminal série type Digital VT320, Wyse, Liberty ou autre TeleVideo afin de pouvoir prendre la main sur les équipements et d'intervenir en cas de problème majeur ou de reconfiguration importante à effectuer. En ce qui concerne les serveurs, un simple commutateur KVM fait très bien l'affaire, mais quid de l'accès à la configuration des *switches* ou des routeurs ? Quand en plus on est, comme moi, collectionneur de matériel ancien, on a souvent affaire à des machines qui ne savent parler que RS-232 ou dont la carte graphique et le clavier sont tellement propriétaires qu'il n'existe juste pas de KVM pour eux.

Il est loisible de trouver d'autres intérêts, comme pouvoir se connecter à plusieurs cartes à microcontrôleur disposant d'un vrai port série ou tout autre équipement du même calibre.

C'est pour répondre à ce type de problématique qu'est né le projet présenté ici : un boîtier disposant de N ports série dont il retransmet les données sur Ethernet d'un côté, et vers un terminal passif de l'autre.

1. ARCHITECTURE

1.1 Cahier des charges et composants principaux

Le concentrateur que nous voulons créer doit répondre au cahier des charges ci-dessous :

- être accessible en SSH (tout bon flux est un flux chiffré) ;
- permettre la multisession ;
- être simple d'utilisation ;
- posséder au moins 4 ports série ;
- être utilisable pour se connecter via un terminal série ;
- être abordable financièrement ;
- pouvoir éventuellement servir à autre chose en parallèle.

À ce stade, deux approches sont possibles :

1) Utiliser une carte à microcontrôleur de type Arduino ou plus puissant et tout développer « *from scratch* ».

2) Se baser sur un SoC sous GNU/Linux afin de ne pas avoir à réinventer la roue.

Inutile de dire que la deuxième solution fut celle choisie, la première étant à garder sous le coude pour un projet de fin de BTS, par exemple...

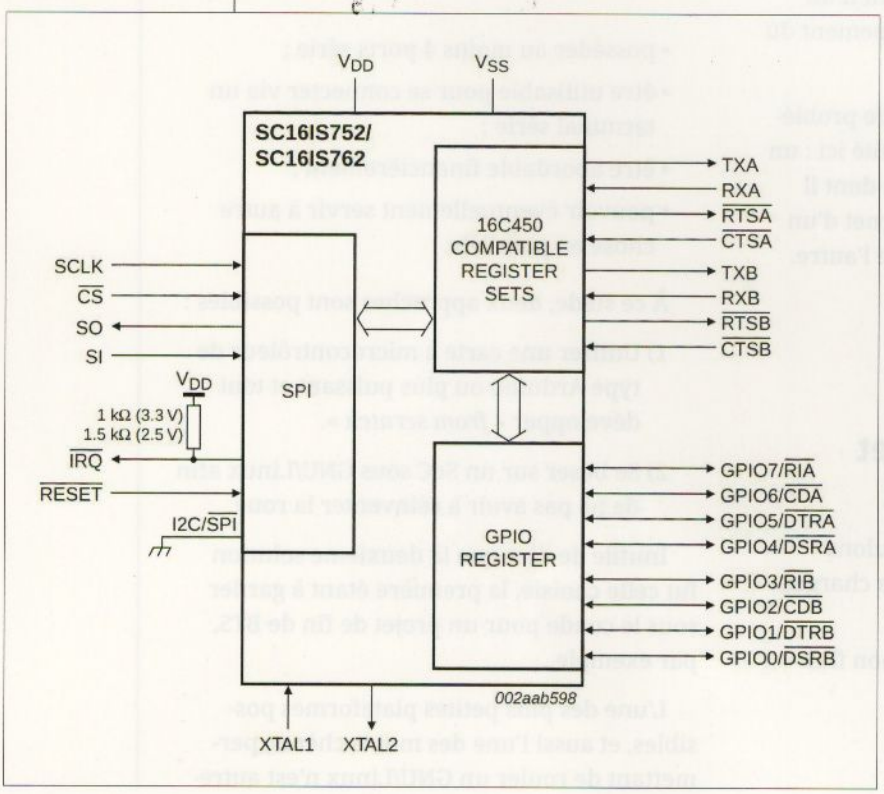
L'une des plus petites plateformes possibles, et aussi l'une des moins chères, permettant de rouler un GNU/Linux n'est autre

que la Raspberry Pi Zero. Elle a comme avantage son prix, ses capacités d'extension et comme inconvénients, son manque de connectivité réseau et sa relative lenteur. Certes, la RPi Zero W existe, mais elle est plus chère et tout baser sur le Wi-Fi n'est pas ce qu'il n'y a de plus fiable ni de plus sécurisé. De plus, j'avais une RPi Zero première du nom qui traînait dans un tiroir, autant l'utiliser.

En ce qui concerne les ports série, la RPi dispose d'un UART *on-board*, mais il nous en faut plus. En cherchant quelques secondes, je suis tombé sur un « *shield* » tout prêt, hébergeant un DUART (*Dual Universal Asynchronous Receiver Transmitter*) nommé SC16IS752. Il s'agit d'une puce de chez NXP possédant les caractéristiques suivantes :

- double UART full-duplex ;
- interface I²C ou SPI au choix ;
- FIFO de 64 octets (Tx et Rx) ;
- totalement compatible avec le standard de fait des 16C450 ou équivalents ;
- *baud rates* jusqu'à 5 Mbit/s ;
- signaux RTS et CTS contrôlés matériellement ;
- jusqu'à 8 GPIO si on n'a pas besoin des signaux de contrôle secondaires (RI, CD, DTR, DSR) ;
- fonctionne en 3,3 V (voire en 2,5) avec les I/O tolérant du 5 V ;
- support des principaux formats de caractères :
  - → 5, 6, 7 ou 8 bits par caractère ;
  - → parité paire, impaire ou sans ;
  - → 1, 1½ ou 2 bits de stop.
- consommation au repos inférieure à 30 µA en 3,3 V ;
- chaînable jusqu'à 16 puces (32 ports) en mode I²C.

SC16IS752 block diagram.



Cette puce fort sympathique est donc disponible sous la forme d'un HAT conçu exprès par Waveshare pour une Raspberry Pi Zero et disponible pour une grosse dizaine d'euros chez tous les bons revendeurs. Ce HAT est prévu pour être chaîné et laisse libre accès au port d'extension, même une fois en place. Son PCB a exactement la taille de celui de la RPi Zero, ce qui permet un montage compact, solide et fort agréable.

1.2 Liste des composants

Partant sur un concentrateur 4 ports, la BOM (*Bill Of Materials*) complète va s'étoffer un peu. En effet, outre la Raspberry Pi Zero et les deux HAT à base de SC16IS752, il nous faut prévoir :

- cinq convertisseurs série 3,3 V ↔ niveaux RS232 : quatre pour les ports vers les machines et un pour connecter un terminal local. La gestion des lignes RTS et CTS est faite par le module, mais rien ne vous oblige à les connecter si vos périphériques n'ont pas besoin de *handshake* matériel ;
- un adaptateur Ethernet sur bus USB pour la connectivité réseau ;
- un câble micro-USB mâle vers *socket* femelle à visser ;
- une alimentation ;
- un boîtier.

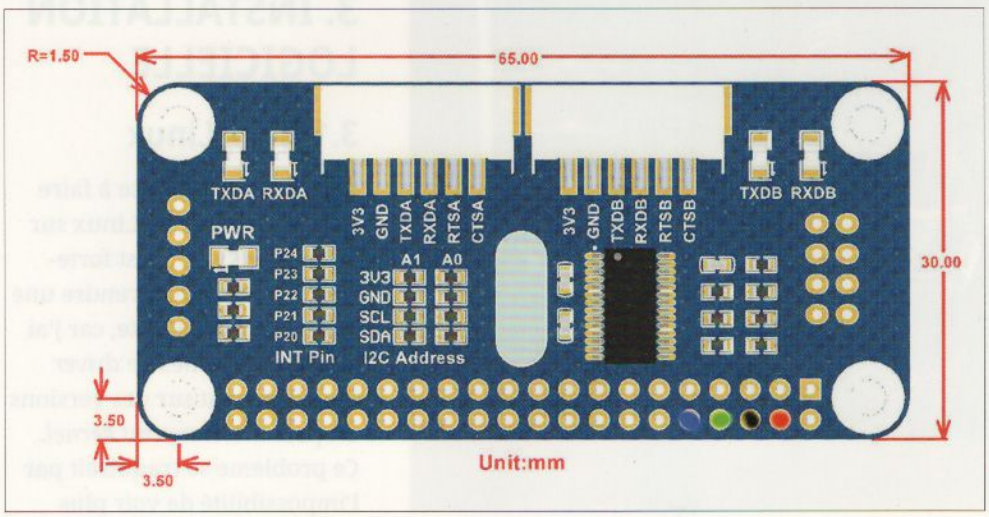
2. INSTALLATION MATÉRIELLE

2.1 Le HAT Waveshare

Si vous décidez d'utiliser plus d'une carte d'extension série, il va falloir les configurer afin qu'elles ne se marchent pas sur les pieds. Cette opération peut être délicate pour les personnes n'ayant pas ou peu d'expérience avec un fer à souder et est à déconseiller formellement si vous avez la tremblote. En vous référant à l'image ci-dessous et au wiki de Waveshare **[WWIKI]**, vous allez devoir déplacer 2 résistances de 0 ohm afin de configurer l'adresse du SC16IS752 sur le bus I²C et le pin d'interruption utilisé.

L'adresse se configure via les lignes A0 et A1 (c'est marqué « I2C Address ») et le pin d'interruption via l'un des pins P20 à P24 (la colonne notée INT Pin). Par défaut, le *shield* est configuré en 0x90 (cf. note page suivante), interruption sur le pin 24. Pour modifier cela, il faut déplacer les résistances CMS en fonction de vos besoins. Il est fortement conseillé de noter sur un bout de papier quelle adresse I²C correspond à quel pin d'interruption, ça va nous servir très vite.

Une fois ces modifications un peu sensibles effectuées, le reste relève plus du jeu de construction que d'autre chose. Il faut commencer par enficher



Configuration du HAT.

les câbles de liaison sur leurs connecteurs et ensuite empiler les *shields* sur la RPi en les sécurisant grâce aux vis et entretoises livrées avec.

> **ATTENTION !**
>
> Une adresse I²C étant sur 7 bits, il va falloir effectuer une rotation vers la droite de l'adresse choisie ici avant de la fournir au *driver*. Par exemple, l'adresse configurée par défaut selon la table donnée par le wiki de Waveshare est 0x90. L'adresse à fournir au *driver* sera 0x90>>1 soit 0x48.

2.2 Le reste

Maintenant que le ou les *shield*(s) sont en place, nous pouvons mettre en place les adaptateurs de niveau RS-232. Il en faut un par port des SC16IS752 plus un sur le port série de la RPi, c'est-à-dire directement sur le bus d'extension comme indiqué sur l'image ci-dessus selon le code couleur :

- rouge (Pin 4) = +5 V ou Pin 1 (celui en carré) si votre adaptateur a besoin de 3,3 V ;
- noir (Pin 6) = GND ;
- vert (Pin 8) = Tx ;
- bleu (Pin 10) = Rx.

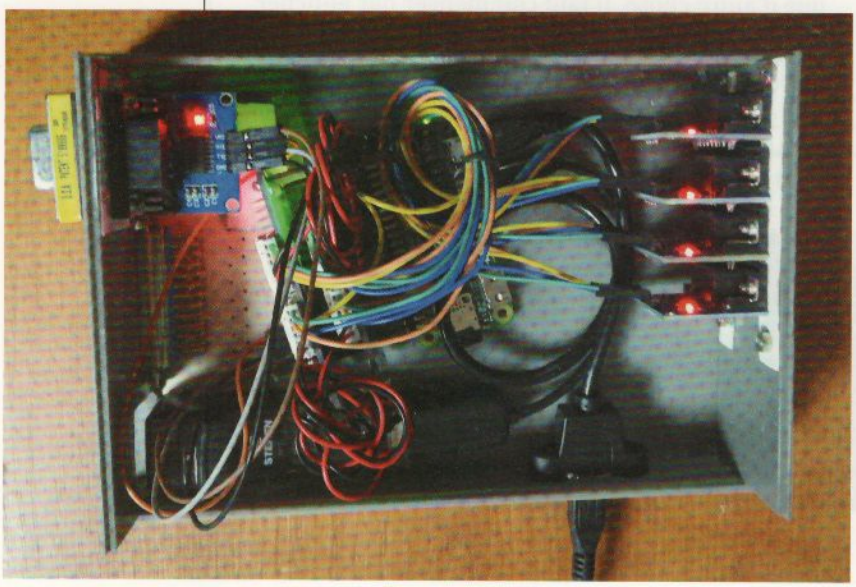
Vous pouvez dès cet instant choisir de croiser Rx et Tx en fonction de vos besoins.

Certains adaptateurs permettent de connecter aussi les signaux RTS et CTS mais ils ne sont pas légion. Si vous en avez, n'hésitez pas à les câbler, ça peut toujours servir.

Il reste à connecter le convertisseur USB → Ethernet sur le port USB qui ne sert pas à l'alimentation de la RPi, un éventuel prolongateur micro-USB sur l'autre port afin de simplifier la connexion de l'alimentation et de tout mettre en boîtier, si vous le désirez.

Ceci conclut l'aspect matériel de la réalisation.

Un peu fouillis, c'est un POC...



3. INSTALLATION LOGICIELLE

3.1 GNU/Linux

La première chose à faire est d'installer GNU/Linux sur la Raspberry Pi. Il est fortement conseillé de prendre une version assez récente, car j'ai eu des problèmes de *driver* non réentrant sur des versions un peu anciennes du *kernel*. Ce problème se traduisait par l'impossibilité de voir plus

d'un seul *shield* série. Comme nous n'avons que faire d'une interface graphique, l'usage d'un RaspberryPi OS Lite s'impose d'elle-même. Je vous laisse choisir votre méthode préférée, de `rpi-imager` à `dd`.

Une fois ceci fait, il va falloir lancer `raspi-config` pour activer, au minimum, les options suivantes :

- le bus I²C ;
- le port série intégré ;
- la connexion SSH.

Profitez-en pour activer aussi tout autre bus que vous voulez, si vous comptez utiliser le montage pour piloter aussi des capteurs ou des actionneurs.

Ceci effectué, il faut indiquer à la RPi l'existence de nos ports série supplémentaires. Là où tout serait pris en charge par l'énumérateur PCI ou USB sur un x86, il va falloir mettre un peu les mains dans le cambouis et modifier le *device tree* de notre carte. Heureusement pour nous, les concepteurs de Raspberry Pi OS ont fortement simplifié la mécanique. Pas besoin de sortir `dtc` (rien à voir ni avec le site web éponyme ni avec de quelconques indications de position), il suffit d'aller rajouter les lignes suivantes à la fin du fichier `/boot/config.txt` :

```
dtoverlay=sc16is752-i2c,int_pin=24,addr=0x48
dtoverlay=sc16is752-i2c,int_pin=23,addr=0x4c
```

à raison d'une ligne par HAT en adaptant les valeurs de `int_pin` et de `addr` en fonction de ce que vous avez noté plus tôt.

Un *reboot* plus tard et vous devriez voir apparaître des *devices* du type `ttySC0`, `ttySC1`, etc., dans `/dev`. Si ce n'est pas le cas, reprenez tranquillement le montage, vérifiez les soudures des résistances 0 ohm, regardez les *logs* du *boot*, etc. Si votre version de Raspberry PI OS est assez récente, il n'y a aucune raison que cela ne fonctionne pas.

3.2 Configuration du port série intégré

Afin d'avoir un shell sur un terminal physique connecté au port de la RPi (`ttyAMA0`), il va falloir modifier la configuration de systemd, car l'autonégociation de la vitesse ne fonctionne pas avec un terminal passif. Cela se fait dans le fichier `/lib/systemd/system/serial-getty\@.service`. Il faut remplacer la ligne :

```
ExecStart=-/sbin/agetty --keep-baud 115200,38400,9600 %I $TERM
```

par :

```
ExecStart=-/sbin/agetty -o '-p -- \\u' --keep-baud 9600 %I $TERM
```

Une fois ceci effectué, il faut recharger le *daemon* systemd :

```
sudo systemctl daemon-reload
```

et activer le service pour le port `ttyAMA0` :

```
sudo systemctl enable serial-getty@ttyAMA0.service
```

On dira ce qu'on voudra, c'est quand même plus simple avec SystemV init, `/etc/inittab` à modifier et zou...

3.3 Automatiser la connexion

La philosophie adoptée, que vous êtes totalement libres de changer, est de créer quatre - ou plus - utilisateurs, un par port série, ce qui permet de choisir le port uniquement en changeant l'utilisateur qui se connecte, que ce soit en SSH ou par le terminal série. Afin de laisser une certaine liberté de manœuvre, j'ai choisi d'utiliser Minicom, car il permet, sans quitter le logiciel, de changer la plupart des paramètres du port série.

Pour commencer, il faut créer un *template* de fichier de configuration pour Minicom, par exemple `/tmp/minirc` et y mettre les lignes suivantes :

```
pu port             /dev/ttySCX
pu baudrate         9600
pu bits             8
pu parity           N
pu stopbits         1
pu rtscts           No
```

Une simple boucle en Bash va nous épargner quelques efforts :

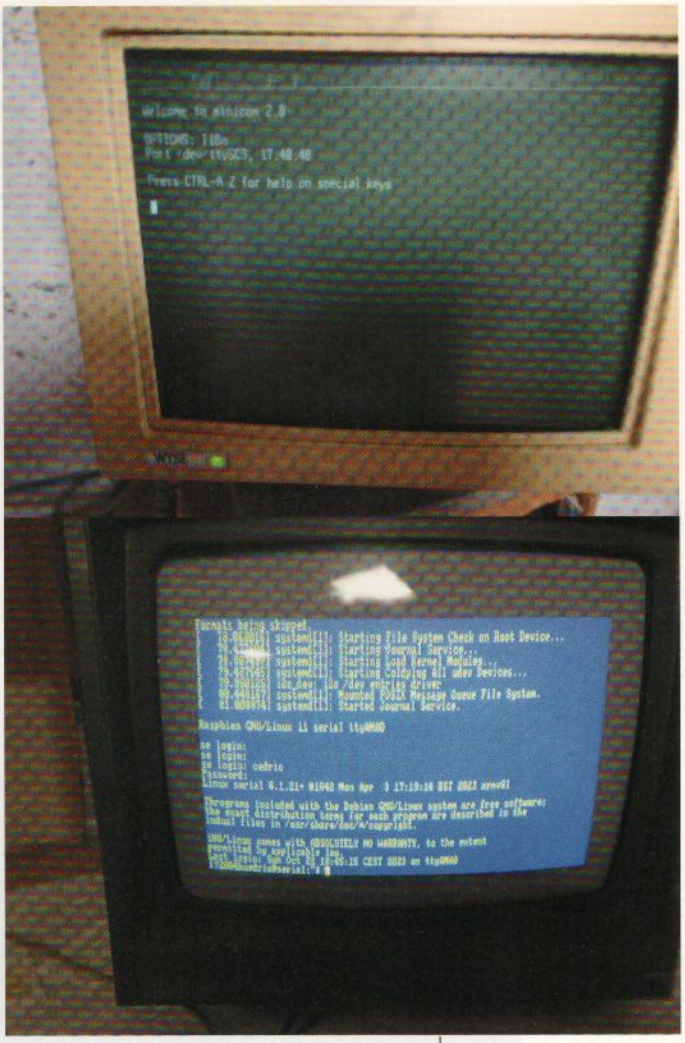
```
for i in `seq 1 4`; do \
useradd -m -s /usr/bin/minicom -d /home/serial${i} -U -G dialout \
-p "`openssl passwd -5 SerialPass${i}`" serial${i}; \
cp /tmp/minirc /home/serial${i}/.minirc; \
j=$((i-1)); \
sed -i "s/ttySCX/ttySC${j}/" /home/serial${i}/.minirc; \
chown toto${i} /home/serial${i}/.minirc; \
done
```

Cette simple ligne crée les utilisateurs nommés `serial1` à `serial4`, leur affecte Minicom comme shell, crée leur *home directory* et leur affecte les mots de passe `SerialPass1` à `SerialPass4`. En outre, on en profite pour copier et modifier adéquatement le fichier de configuration de Minicom. Certes, il y a nettement plus sécurisé, mais c'est une base.

Une fois ceci fait, un `ssh` ou un login via le terminal avec l'utilisateur `serial1`/`SerialPass1` devrait afficher un Minicom connecté au port `ttySC0`. La même chose avec le couple `serial2`/`SerialPass2` devrait afficher un Minicom connecté au port `ttySC1`, etc. Il est bien entendu loisible de reconfigurer Minicom par utilisateur afin de personnaliser la vitesse, le contrôle de flux, etc.

3.4 En option

Utiliser une Raspberry Pi Zero pour faire de temps en temps le passe-plat entre 2 ports série ou entre un port série et le réseau peut sembler un poil surdimensionné. C'est pour cela que j'ai ajouté un adaptateur sous la forme d'un petit PCB avec 25 trous d'un côté et une DB25 de l'autre. L'idée sous-jacente est d'y connecter les bus I²C, le bus SPI et quelques GPIO afin de brancher dessus divers capteurs permettant, par exemple, de surveiller la consommation des serveurs via des capteurs à effet Hall, de mesurer la température de la baie, etc. Quelques petits scripts maison seront ensuite suffisants pour remonter les alertes ou alimenter un traceur de courbes du type Grafana. Là encore, la seule limite réelle reste votre imagination.



Comme quoi il ne faut jamais jeter de matériel...

CONCLUSION

Ce petit montage simple et peu cher rend bien des services au quotidien et peut servir de base à tout un tas de réalisations qui viendront se greffer dessus petit à petit. Il permet d'utiliser intelligemment une carte un peu obsolète qui, sans cela, terminerait au fond d'un tiroir, comme bon nombre de ses sœurs. **CP**

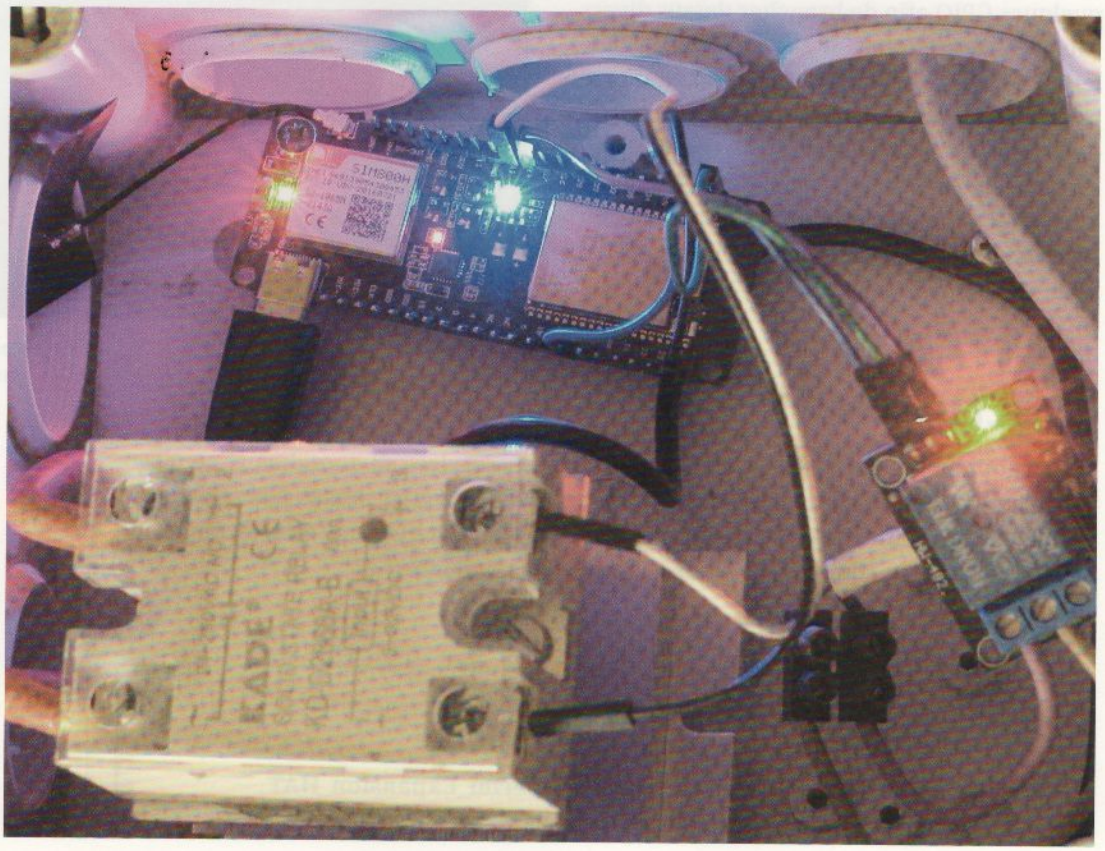
RÉFÉRENCE

[WWIKI] https://www.waveshare.com/wiki/Serial_Expansion_HAT

DOMOTIQUE AVEC DU WI-FI ET DES SMS EN UTILISANT UN ESP32

Antoine LUCAS
Consultant freelance Antoine & Associés, Morlaàs

Vous avez aimé l'article Box Airbnb de Hackable n° 50 [1] ? Je vous propose de remplacer votre Raspberry Pi par un simple ESP32 pour le même résultat : toujours des SMS qui pilotent une domotique à distance, et un Wi-Fi qui ouvre une porte.



– Domotique avec du Wi-Fi et des SMS en utilisant un ESP32 –

Avec un budget de 2 à 20 euros, vous avez des microcontrôleurs programmables qui permettent d'activer une dizaine de relais électriques et qui proposent des connexions Bluetooth et Wi-Fi. Ces microcontrôleurs peuvent être connectés à un modem GSM (téléphone 2G, avec une carte SIM) soudé sur la même carte électronique ou en tant que carte externe. Je vais reprendre la proposition de mon article précédent avec un relais 60 A permettant de couper l'alimentation d'un appartement et un autre relais plus classique pour ouvrir une porte.

Je vais d'abord présenter l'ouverture de porte par Wi-Fi avec un NodeMCU, et dans une deuxième partie la communication par SMS.

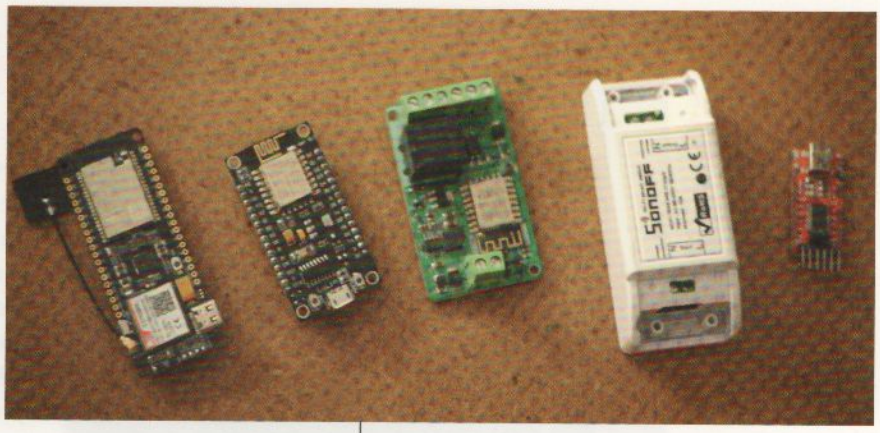


Fig. 1 : Plusieurs cartes possibles pour ce projet, de gauche à droite : le T-Call de LILYGO comporte un ESP32 avec son module SIM800L, le NodeMCU basé sur le microcontrôleur ESP8266, une carte ESP8266 avec un relais intégré, le module Sonoff (ESP8266 également, avec un relais), et un adaptateur FDTI USB vers port COM, nécessaire pour flasher les deux dernières cartes.

1. MATÉRIEL

Le premier composant, c'est votre microcontrôleur comme ceux présentés en figure 1. Le T-Call de LILYGO est bien pratique, car il a le module GSM intégré sur la carte. Si vous ne souhaitez que le verrou connecté, une carte ESP8266 suffit. Attention à certaines cartes qui ne comportent pas de connecteur USB, comme les Sonoff, il vous faudra une petite carte pour communiquer avec un port COM.

En dehors du microcontrôleur, il vous faut toujours des relais comme en figure 2, leur rôle : ouvrir ou fermer un circuit électrique à partir de la commande fournie par le microcontrôleur. La figure 3 montre l'ensemble du projet.

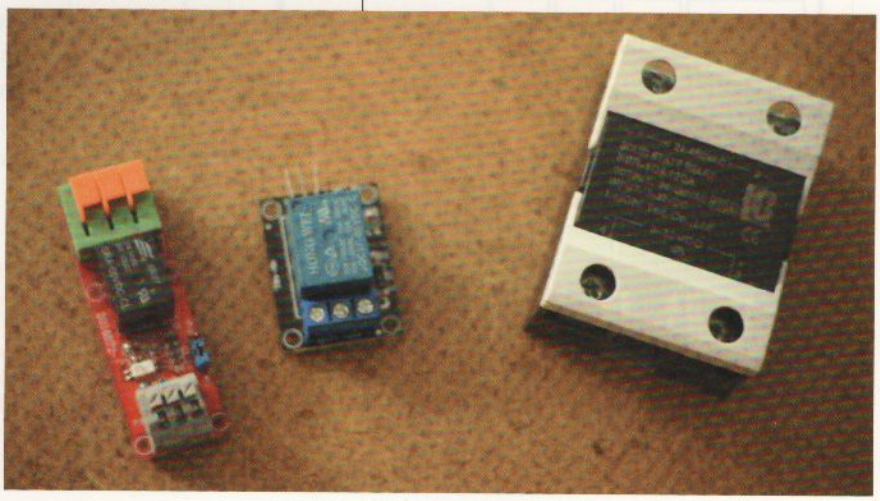


Fig. 2 : Des relais compatibles avec la tension de sortie de nos modules (3 V, avec éventuellement une borne en +5 V non programmable). De gauche à droite : un relais 20 A/14 VDC, un relais 10 A/230 VAC, un relais SSR 120 A/230 VAC. Attention avec ce dernier : il nécessite un radiateur pour fonctionner.

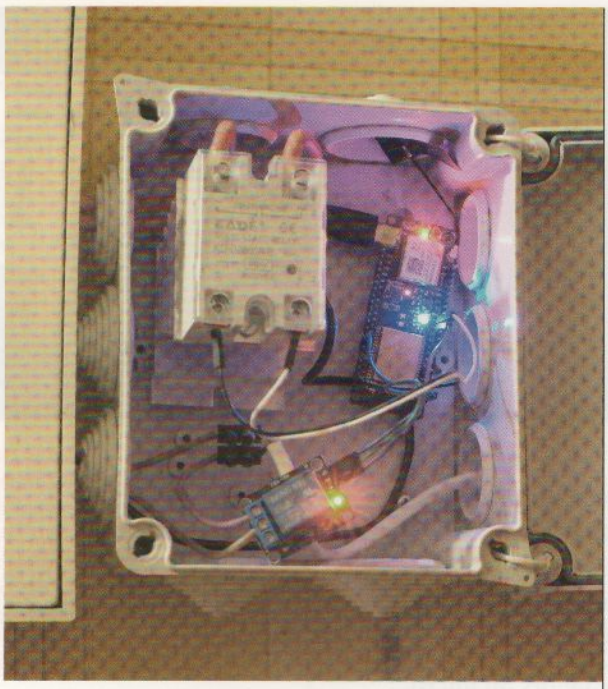


Fig. 3 : L'installation avec notre microcontrôleur connecté aux deux relais.

2. INSTALLATION ÉLECTRIQUE

L'objectif est que votre boîtier reste toujours en ligne (à recevoir des SMS) et qu'il puisse couper le courant de presque tout le logement. Ce ne sera pas donc tous les circuits électriques qui seront éteints (sinon le boîtier lui-même s'éteindra), mais les circuits les plus énergivores, ou quelques rangées du tableau électrique. La figure 4 présente un exemple de configuration du compteur électrique avec une VMC sur la première rangée.

Une alternative, permettant de séparer la partie domotique du tableau électrique, serait de mettre le relais 60 A dans le tableau, et d'avoir dans la partie domotique deux relais classiques à 10 ampères. Votre électricien sera plus à l'aise avec un classique « contacteur de puissance » de chez Legrand, Hager ou Schneider Electric qu'avec un relais SSR de chez AliExpress. Le contacteur de puissance est un relais mécanique avec une bobine alimentée en 230 V. Le relais de 10 A dans la partie domotique pourra actionner le circuit de 230 V de la bobine du contacteur.

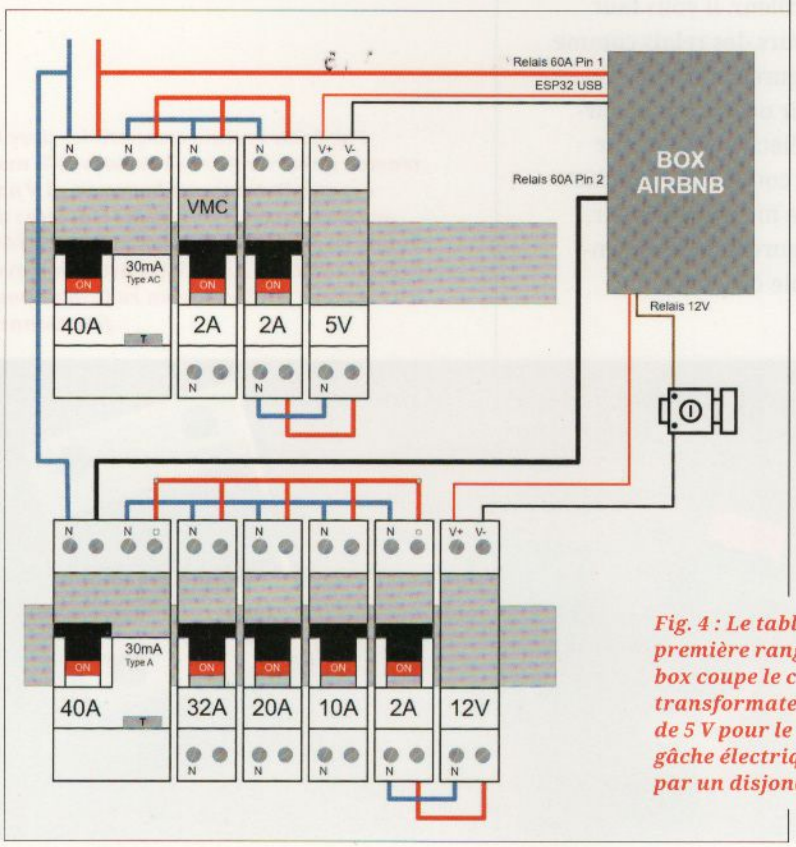


Fig. 4 : Le tableau électrique. Il comporte une première rangée qui reste sous tension lorsque la box coupe le courant dans l'appartement. Il y a deux transformateurs pour les composants de la box, l'un de 5 V pour le microcontrôleur, l'autre de 12 V pour la gâche électrique. Chaque transformateur est protégé par un disjoncteur de 2 ampères.

3. INSTALLATION D'ARDUINO IDE

Plusieurs environnements permettent de programmer et mettre à jour les micro-contrôleurs. J'avais dans un premier temps essayé MicroPython qui, comme son nom l'indique, permet d'utiliser le langage de programmation Python, mais l'IDE d'Arduino s'est révélé finalement plus commode.

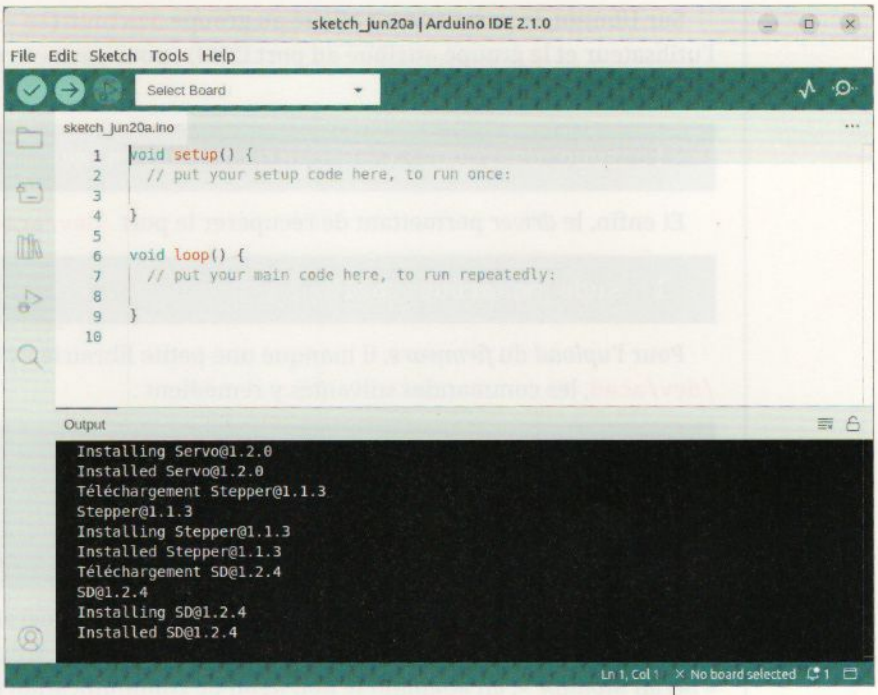


Fig. 5 : Arduino IDE.

Il vous faut le télécharger sur un site chinois **[2]** en cliquant sur « AppImage 64 bits » pour GNU/Linux. Dans le répertoire où vous avez téléchargé le fichier, donnez-lui les droits d'exécution pour le lancer :

```
1: chmod 'a+x' arduino-ide_2.1.0_Linux_64bit.AppImage
2: ./arduino-ide_2.1.0_Linux_64bit.AppImage
```

Cela ouvre l'IDE en figure 5.

Le menu ***File*** > ***Preferences*** vous ouvre une fenêtre de configuration. Dans le champ ***Additional board Manager URLs*** ajouter la ligne (les deux URL en une ligne) :

```
http://arduino.esp8266.com/stable/package_esp8266com_index.json,https://
dl.espressif.com/dl/package_esp32_index.json
```

Dans le menu ***Tools*** > ***Board manager*** recherchez ESP8266, sélectionnez l'item « *Esp8266 Community* » cliquez sur ***Install***. Puis faites la recherche ESP32, sélectionnez l'item « *ESP32 by Espressif System* » et cliquez sur ***Install***.

La connexion avec la carte se fait par un port COM sur USB. Dans GNU/Linux, le *driver* l'attribue à `/dev/ttyUSB0` (NodeMCU) ou `/dev/acm0`.

Sur ma distribution GNU/Linux (Ubuntu) j'ai un conflit entre le port COM `/dev/ttyUSB0` et un utilitaire pour le braille. Pour s'en défaire :

```
1: sudo apt remove brltty
```

Sur Ubuntu, le port COM est alloué au groupe `dialout` (`ls -lh /dev/ttyUSB0` vous donne l'utilisateur et le groupe attribué au port COM) ; pour permettre à l'utilisateur courant d'utiliser ce port COM :

```
1: sudo usermod -a -G dialout <username>
```

Et enfin, le *driver* permettant de récupérer le port `/dev/acm0` se trouve avec la commande :

```
1: sudo modprobe cp210x
```

Pour l'*upload* du *firmware*, il manque une petite librairie Python pour la communication avec `/dev/acm0`, les commandes suivantes y remédient :

```
1: cd /usr/bin
2: sudo ln -s python3 python # create an alias python
3: sudo apt-get install python-pip3
4: sudo pip3 install pyserial
```

Cet IDE vous permet de saisir le code (C ou C++) pour programmer votre microcontrôleur, de le compiler et de flasher votre carte ESP32, ESP8266 ou Arduino. Il y a également un outil « *Serial Monitor* », en adaptant le bon débit de communication pour pourrez voir tous les messages prévus pour les développeurs.

4. LE DÉVERROUILLAGE PAR WI-FI

Cet exercice n'est pas beaucoup plus compliqué que le « hello world » avec un ESP8266. Le schéma en figure 6 présente les composants et les branchements.

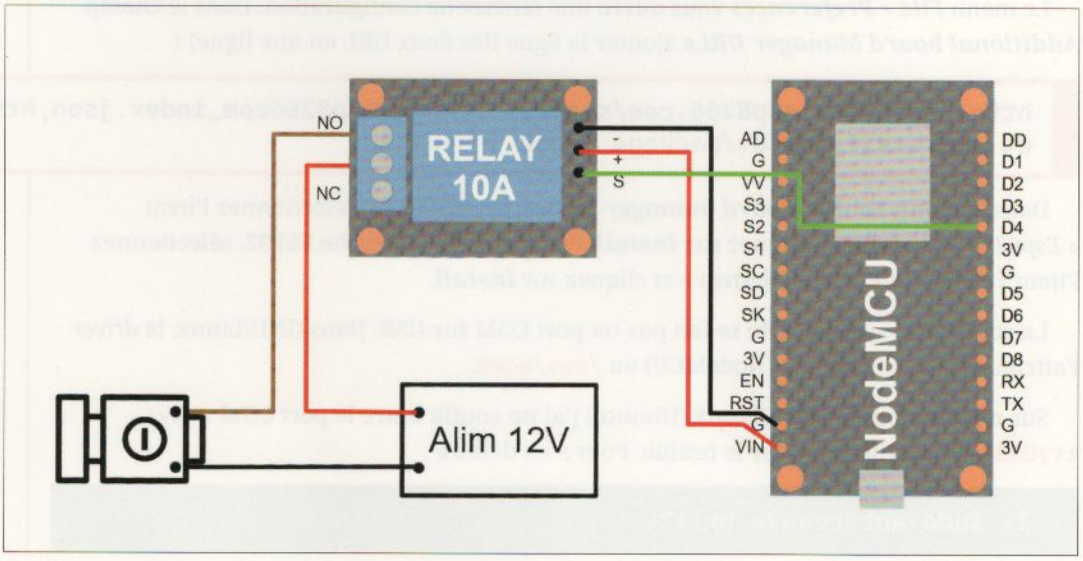


Fig. 6 : Le branchement du NodeMcu avec un relais et une serrure.

L'ensemble du code est disponible sur mon *repository* GitHub **[3]** dans le projet ESPLocker.

Il s'agit de code C ou C++, le fichier commence par l'import des *headers* et quelques variables globales :

```
#ifdef ESP32
#include <WiFi.h>
#define NOTIFICATION_CONNECTION_WIFI ARDUINO_EVENT_WIFI_AP_STACONNECTED
#define LED_BLUE 13
#define LED_ON 1
#define LED_OFF 0
#else
#include <ESP8266WiFi.h>
#define NOTIFICATION_CONNECTION_WIFI WIFI_EVENT_SOFTWARE_DISTRIBUTE_STA_IP
#define LED_BLUE 2
#define LED_ON 0
#define LED_OFF 1
#endif

#define RELAY_DOOR 4

// Variables globales
String ssid = "Revue Hackable";
String password = "DiamondsAreForever";
IPAddress local_IP(192,168,10,10);
IPAddress subnet(255,255,255,0);
bool openTheDoor=false;
```

Vous avez noté les `#ifdef` `#else` et `#endif` qui nous permettent d'utiliser dans la suite le même code alors que de petites nuances diffèrent entre le code pour le processeur ESP32 et celui pour le processeur ESP8266. Ces différences sont dans la librairie Wi-Fi et dans l'énuméré des différents événements de la connexion Wi-Fi.

Pour les numéros des GPIO, et quel GPIO est connecté à une éventuelle LED, cela dépend de chaque carte. Pour celles que j'ai prises (NodeMCU et le T-Call de LILYGO), nous avons les numéros 2 et 13.

Le *framework* Arduino prévoie deux fonctions `setup` pour l'initialisation et `loop` pour un appel récurent, il faut les implémenter :

```
01: void setup() {
02:   Serial.begin(9600);     // Ouvre le port COM et fixe de débit de communication
03:   pinMode(RELAY_DOOR,OUTPUT);
04:   pinMode(LED_BLUE,OUTPUT);
05:   digitalWrite(LED_BLUE,LED_ON);
06:   digitalWrite(RELAY_DOOR,LOW);
07:   Serial.print("Start WiFi ... ");
08:   WiFi.softAPConfig(local_IP, local_IP, subnet); // Création du point d'accès Wi-Fi
```

```
09:  WiFi.onEvent([](WiFiEvent_t a) { openTheDoor=true; }, NOTIFICATION_CONNECTION_WIFI);
10:  WiFi.softAP(ssid,password) ;
11:  delay(500); // Pour voir la LED allumée au démarrage
12:  digitalWrite(LED_BLUE,LED_OFF);
13:}
```

Une petite explication sur les GPIO, ce sont les entrées / sorties de la carte. Il faut dans un premier temps indiquer si on souhaite que le GPIO soit une entrée (lecture d'un capteur), ou une sortie (ordre donné à un composant). C'est la commande `pinMode(RELAY_DOOR, OUTPUT)`. Dans un deuxième temps, on lui assigne la valeur de sortie (0 ou 1) qui donnera 0 ou 3 V respectivement. C'est la commande en ligne 6 `digitalWrite(RELAY_DOOR, LOW)`.

La ligne 9 semble un peu difficile à comprendre. C'est normal : c'est moderne ! il s'agit d'une lambda, c'est-à-dire la définition d'une fonction en une ligne. `[](WiFiEvent_t a) { openTheDoor=true; }` veux dire « je crée une fonction d'un paramètre `a` de type `WiFiEvent`, qui fera ce qui est dans l'accolade ».

Pour l'ouverture de la porte, cela s'écrit de cette façon :

```
void openDoor() {
    digitalWrite(LED_BLUE, LED_ON); // On allume la LED, c'est pour faire joli.
    Serial.println("Ouverture de la porte, 10 sec.\n");
    digitalWrite(RELAY_DOOR, HIGH);
    delay(10000); // 10 secondes
    digitalWrite(RELAY_DOOR, LOW); // Fermeture de la porte (de la serrure en tout cas)
    digitalWrite(LED_BLUE, LED_OFF);
    openTheDoor=false;
}
```

Et la dernière fonction à implémenter, elle est appelée en boucle :

```
void loop() {
  if(openTheDoor) openDoor();
  delay(100); // Attente 100 ms
}
```

5. DOMOTIQUE PAR SMS

Pour cette partie, il vous faut un peu plus de matériel, je propose l'une des trois versions de la carte T-CALL de LILYGO, le branchement se fait suivant la figure 7. Les détails des différents pins se trouvent sur le GitHub de LILYGO **[5]**.

J'ai trouvé deux librairies permettant d'envoyer et de recevoir des SMS avec un ESP32, TinyGSM et Adafruit_FONA **[4]**. J'ai choisi cette dernière qui m'a paru plus simple à utiliser et plus complète (il manque le « readSMS » dans TinyGSM). Il suffit de copier les deux fichiers et le répertoire de cette librairie dans le répertoire de votre projet pour l'installer.

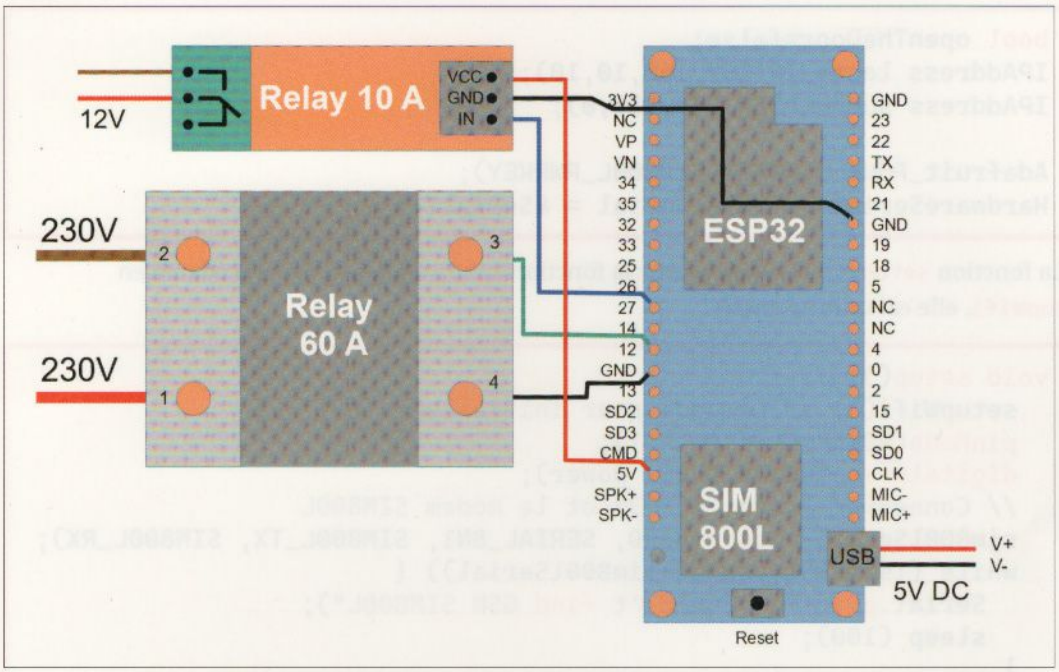


Fig. 7 : Le branchement du microcontrôleur LILYGO T-CALL avec les deux relais.

L'ensemble du code est disponible sur mon *repository* GitHub **[3]** dans le projet BoxAirbnbESP32.

Comme avec la section précédente, il y a des *headers* à spécifier et deux fonctions `setup` et `loop` à implémenter.

Le *header* :

```
#include <WiFi.h>  // Il s'agit d'un ESP32
#include "Adafruit_FONA.h"
#include <string>
#include <cstring>
// GPIO la LED bleue est sur le 13 sur cette carte
#define LED_BLUE 13
#define RELAY_DOOR 2
#define RELAY_ELEC 12
// Le modem SIM800L est lui-même alimenté par un GPIO
#define SIM800L_PWRKEY 4
// Les connexions entre l'ESP32 et SIM800L
#define SIM800L_RX     27
#define SIM800L_TX     26

// Variables globales
String ssid = "Revue Hackable";
String password = "DiamondsAreForever";
int power = HIGH;
```

```
bool openTheDoor=false;
IPAddress local_IP(192,168,10,10);
IPAddress subnet(255,255,255,0);

Adafruit_FONA sim800l(SIM800L_PWRKEY);
HardwareSerial* sim800lSerial = &Serial1;
```

La fonction `setup` (vous renommerez la fonction `setup` de la section précédente en `setupWifi`, elle est utile ici aussi) :

```
void setup(){
  setupWifi(); // Le code pour initialiser le Wi-Fi
  pinMode(RELAY_ELEC,OUTPUT);
  digitalWrite(RELAY_ELEC, power);
  // Connexion entre l'ESP32 et le modem SIM800L
  sim800lSerial->begin(4800, SERIAL_8N1, SIM800L_TX, SIM800L_RX);
  while (!sim800l.begin(*sim800lSerial)) {
    Serial.println("Couldn't find GSM SIM800L");
    sleep (100);
  }
  // Une commande un peu magique qui semble nécessaire...
  sim800lSerial->print("AT+CNMI=2,1\r\n");
  Serial.println("GSM SIM800L Ready");
}
```

La fonction `loop` :

```
void loop(){
  if(openTheDoor) openDoor();
  char buffer[250];
  if (sim800l.available()) {
    int slot = 0;  // Ce devrait être le numéro du SMS dans la carte SIM800
    int charCount = 0;
    // Lecture du flux texte
    for (int i = 0 ; i < sizeof(buffer)-1 ; i++){
      buffer[i] = sim800l.read();
      if(buffer[i] == '\n'){
        buffer[i+1] = 0;
        break;
      }
    }

    // Scan the notification string for an SMS received notification.
    if (1 == sscanf(buffer, "+CMTI: \"SM\",%d", &slot)) {
      digitalWrite(LED_BLUE, HIGH);
      String smsString = "";
```

```
      char phone[32];
      // Retrieve SMS sender address/phone number.
      if (!sim800l.getSMSSender(slot, phone, 31)) {
        Serial.println("Didn't find SMS message in slot!");
      }
      Serial.print("FROM = ");
      Serial.println(phone);

      // Retrieve SMS value.
      uint16_t smslen;

      // Lecture du message
      if (sim800l.readSMS(slot, buffer, 250, &smslen)) {
        smsString = String(buffer);
        Serial.println(smsString);
      }

      if (strstr(smsString.toUpperCase().c_str(), "PASSWORD")!= NULL) {
        std::string copySms(buffer);
        buffer.erase(0, out.find(" ")); // Suppression du premier mot dans
                                        // le message
        String newPassword = String(copySms.c_str());
        // Modification du mot de passe Wi-Fi
        if (WiFi.softAP(ssid, newPassword)) {
            password =  newPassword;
        }
        else {
          Serial.println("incorrect password ");
       }
       sendSMS(phone);
      } else if (strstr(smsString.toUpperCase().c_str(), "ON")!= NULL) {
        power = HIGH;
        digitalWrite(RELAY_ELEC, power);
        sendSMS(phone);
      } else if (strstr(smsString.toUpperCase().c_str(), "OFF")!= NULL) {
        power = LOW;
        digitalWrite(RELAY_ELEC, power);
        sendSMS(phone);
      } else {
        Serial.println("message ignore");
      }
      // Cela devrait supprimer le message
      sim800l.deleteSMS(slot);
      digitalWrite(LED_BLUE, LOW);
    }
  }
  delay(100); // Attente 100 ms
}
```

Il vous manque la petite fonction `sendSms` pour envoyer la notification :

```
void sendSMS(const char * telephone){
  String msg = "Power is ";
  if (digitalRead(RELAY_ELEC) == HIGH) msg += "ON ";
  else msg += "OFF ";
  msg += "password is " + password;
  delay(100);
  sim800l.sendSMS(telephone, msg.c_str());
  delay(100);
}
```

Avec ce code, la porte s'ouvre avec le Wi-Fi, le mot de passe se change par SMS... mais le mot de passe et l'interrupteur général sont re-initialisés en cas de coupure de courant. Pas de panique, ces cartes ont un espace de stockage, il faut rajouter ces deux fonctions, une au *setup*, l'autre pendant le `sendSMS` :

```
#include <Preferences.h>

Preference config;

// À appeler dans le setupWifi, avant la configuration du Wi-Fi
void readConfig() {
  config.begin("config", true);
  password = config.getString("password",password);
  power = config.getInt("power",power);
  config.end();
}


// À appeler dans le sendSMS
void writeConfig(){
  config.begin("config", false);
  config.putString("password", password);
  config.putInt("power", power);
  config.end();
}
```

CONCLUSION

Cet environnement nous permet à peu de frais et avec un code très léger de reprendre l'intégralité des fonctions du projet précédent qui utilisait un OrangePi.

Comme avec l'OrangePi, j'ai encore une déconnexion au bout de 15 jours de la communication GSM, mais cela se corrige avec un redémarrage du module SIM800, sans redémarrer le microcontrôleur. Un exemple est proposé sur le GitHub **[3]**.

Certains en voudront toujours davantage... mon petit frère me demande d'ouvrir également la porte commune de son immeuble. C'est possible, en « hackant » l'interphone. Certains médecins ont un système similaire, on sonne chez le médecin à ses horaires d'ouverture, et la porte s'ouvre.

La première étape consiste à mettre un relais, actionné par la sonnerie et activant l'ouverture de la porte (en dérivation du bouton de votre interphone : tout est dans l'appartement, pas dans les parties communes). Le microcontrôleur pouvant désactiver ce relais à certains horaires, avec un autre relais.

Pour aller plus loin, la communication entre l'ESP32 et le module GSM se fait par un port COM, et la messagerie complète se trouve sur le GitHub de LILYGO **[5]** dans le répertoire `datasheet`. **AL**

RÉFÉRENCES

[1] Antoine Lucas, box Airbnb *Hackable n°50*, septembre-octobre 2023, **https://connect.ed-diamond.com/hackable/hk-050/box-airbnb-domotique-avec-des-sms**

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

[3] GitHub, projets BoxAirbnbESP32 et ESPLocker d'Antoine Lucas, **https://www.github.com/antoinelucas64**

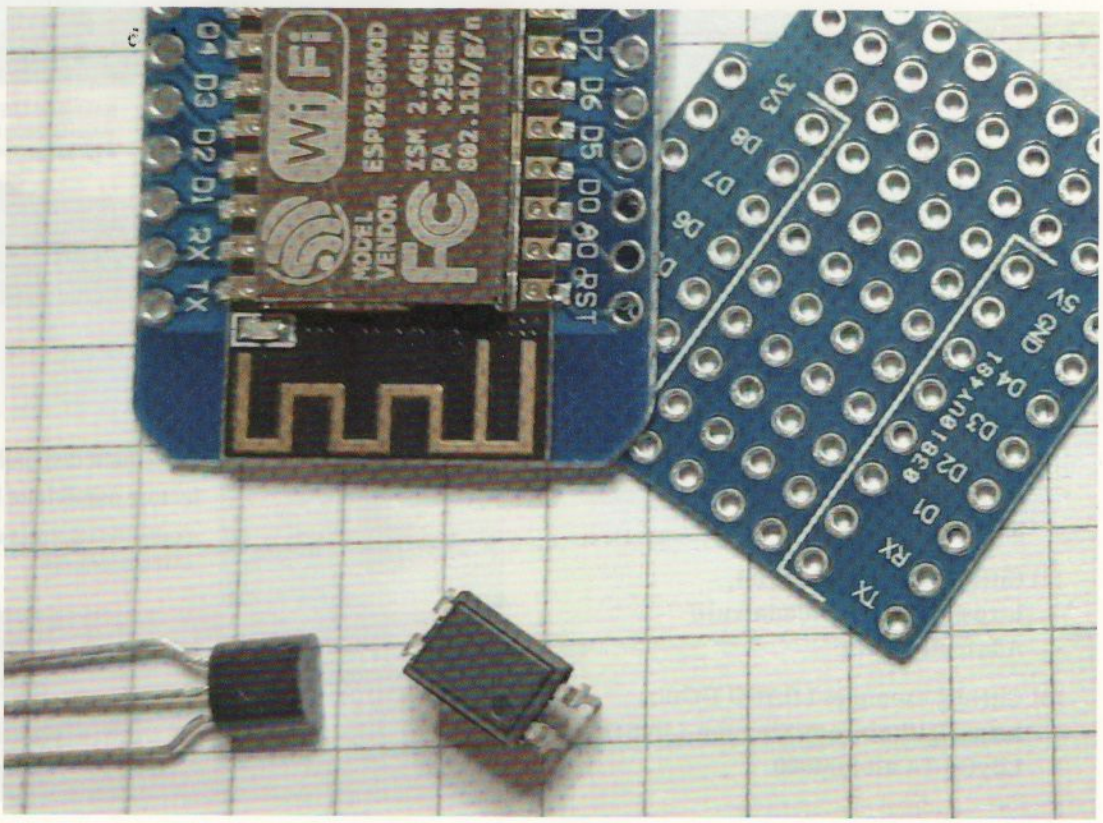
[4] GitHub, librairie Adafruit, **https://github.com/adafruit/Adafruit_FONA**

[5] GitHub, exemples LILYGO T-Call, **https://github.com/Xinyuan-LilyGO/LilyGo-T-Call-SIM800**

LINKY + DOMOTIQUE = ÉCONOMIES

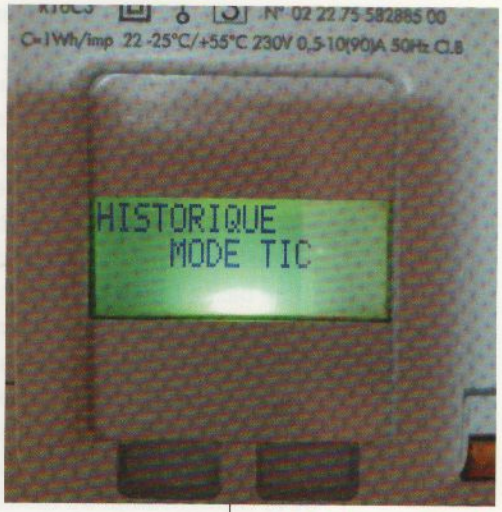
Denis BODOR

Avec le coût de l'énergie qui ne cesse d'augmenter, il est de plus en plus important de garder un œil sur sa consommation et éventuellement de changer ses petites habitudes, sans pour autant sombrer dans l'excès et vivre dans la pénombre et le froid permanents. Il existe maintes solutions pour arriver à faire cela, mais la plus rapide est encore de tout simplement profiter des sources d'information à disposition, à commencer par son compteur électrique : le Linky.



– Linky + domotique = économies –

Écartons de suite les sujets et thématiques à la limite de la superstition, ainsi que ceux, totalement opposés, ventant les bienfaits de la modernisation des installations Enedis. Non, le Linky n'émet pas des ondes nocives de la 5G provoquant le cancer du glyphosate suite au vaccin COVID dans le seul but de dissimuler le fait que la terre est plate (alors que tout le monde sait que c'est un disque cubique creux à 3 faces). C'est juste un compteur et, oui, l'installateur va effectivement changer le réglage du disjoncteur de branchement, car le Linky gère lui-même la puissance maximum (3, 6, 9, 12, 15, 18, 24, 30 ou 36 kVA, 6 étant le plus courant), ce qui ne veut absolument rien dire pour le fonctionnement de votre installation avant ou après changement du compteur (le disjoncteur de branchement est scellé, exactement comme le compteur). Et, oui aussi, le compteur communique avec le réseau en CPL et il est possible de lui adjoindre un module dit « ERL » (« Émetteur Radio Linky », on est très fort pour les acronymes en France) qui communique en Wi-Fi, ZigBee et/ou KNX RF selon le modèle/fabricant. Et non, ce n'est pas un problème sachant, de plus, que la modulation CPL est présente dans les lignes, que vous ayez un Linky ou non. Si vous êtes dans ce genre de « délires », vous lisez clairement le mauvais magazine (la preuve, nous n'utilisons pas Comic Sans).



L'afficheur en façade du Linky permet de connaître différentes informations intéressantes, dont le mode dans lequel fonctionne la TIC (historique ou standard). Ceci n'est cependant pas configurable par l'utilisateur directement, seul le fournisseur d'énergie peut faire basculer le compteur d'un mode à l'autre.

De l'autre côté des avis contrastés sur le Linky, nous avons l'émerveillement concernant la facilité d'utilisation de la liaison domotique et/ou du pilotage de certains appareils en fonction des heures pleines/creuses. Ceci n'est justifié que dans un cas particulier, le mien. Plus généralement (et plus sérieusement), si votre ancien compteur ressemble à un accessoire *steampunk* sortant tout droit d'un roman de Jules Verne (noir, en métal, avec un disque qui tourne et un cadran avec une belle police *vintage*), le passage au Linky est un bond dans le futur. Mais entre le bon vieux compteur électromécanique et le Linky, nous avons aussi le compteur électronique ou CBE (pour « Compteur Bleu Électronique », parfois appelé compteur de seconde génération), offrant déjà la possibilité de collecter des informations de consommation et de piloter son installation en fonction des changements de tarifs (d'où le « Bleu » de CBE).

Ces points étant écartés, entrons dans le vif du sujet en vous parlant de ma vie, mon habitat et mes articles. En « domotisant » mon lieu de vie il y a quelques mois, je n'avais aucune vision claire concernant un changement de compteur. Comme tout le monde, je savais que mon installation allait évoluer et que mon fournisseur d'électricité local et municipal (Vialis) planifiait de changer les compteurs de ses clients, mais rien n'était établi. J'ai donc entrepris, comme nous l'avons vu dans un précédent article dans le numéro 49 **[1]**, le déploiement de capteurs à différents endroits stratégiques (prises et tableau électrique) de manière

à relever puissance et consommation pour des parties que je jugeais importantes (four, PC, électroménager, luminaires sur prises, etc.). L'arrivée du Linky change totalement la donne avec deux options possibles : surveiller uniquement la consommation générale, ou intégrer la différence entre ce que mesure l'ensemble des capteurs installés et le compteur Linky. L'une des approches est excessivement simple et permettra à ceux qui n'ont pas encore d'installation domotique basée sur Home Assistant de se mettre le pied à l'étrier rapidement. L'autre, en revanche, présuppose que vous ayez déjà un Home Assistant « en production » ainsi qu'une surveillance partielle de la consommation et consistera à traiter de l'ajout d'une donnée supplémentaire, globale, sur le tableau de bord et d'une révision complète (et destructive) de la configuration du tableau de bord ***Énergie***.

Dans les deux cas, la première chose à faire est de savoir comment le compteur Linky nous informe de ce qu'il voit (et facture)...

1. TÉLÉ-INFORMATION CLIENT

Le compteur Linky, comme son prédécesseur le CBE, dispose d'un connecteur TIC, pour « Télé-Information Client ». Celui-ci dispose de trois bornes libellées L1, L2 et A et ne doit pas être confondu avec le connecteur à vis, juste dessous, libellé C1 et C2 correspondant au contact sec permettant de piloter les appareils devant fonctionner durant les heures creuses (si vous avez ce type de contrat, ce qui n'est pas mon cas). Les trois broches du connecteur TIC forment deux circuits, L1/L2 pour le circuit d'information et L1/A pour le circuit d'alimentation que nous ne traiterons pas ici, préférant alimenter le montage ESP8266 avec un bloc d'alimentation relié au courant domestique directement dans le tableau électrique (prise sur rail). Si vous êtes intéressé par la possibilité d'alimenter un montage via L1/A, je vous recommande la lecture très intéressante de ce site : **https://www.morbret.fr/linky/alim-bornes-i1-et-a**. Petite précision au passage, ce connecteur TIC n'a absolument rien de secret et est parfaitement décrit par Enedis (Document « Enedis-NOI-CPT_54E » **[2]**). Mais apparemment, qualifier de « secrète » une prise **[3]** permet de vendre plus facilement des produits qu'on peut fabriquer soi-même pour une poignée d'euros...

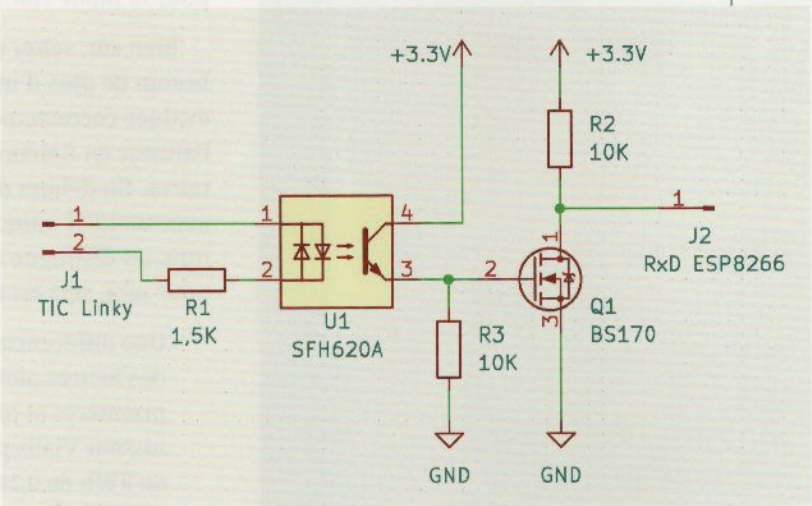
La création d'un adaptateur permettant de connecter une plateforme (PC, Arduino, Raspberry Pi, etc.) est très largement documentée sur le Web. On retrouve un montage relativement simple, décliné en tout un tas de versions plus ou moins similaires. Le principe est tout bête puisque le circuit d'information n'est rien d'autre qu'une liaison série à 1200 ou 9600 bauds. Et c'est ailleurs là que commence ce qu'il faut savoir sur la TIC des Linky. Contrairement au compteur CBE, nous avons deux options possibles :

- La « TIC historique » héritée du CBE et, semble-t-il, configurée par défaut sur les compteurs fraîchement installés. Cette liaison se fait à 1200 bauds et transmet des trames utilisant un jeu relativement réduit d'informations.
- La « TIC Standard », apparue avec le Linky, est assez similaire, mais utilise une liaison à 9600 bauds et transmet beaucoup plus d'informations, dont les « jours de pointe » (EJP), la tension moyenne pour les trois phases, l'énergie réactive, la tension efficace, date et heure courantes, etc. La plupart des trames intègrent également un horodatage, ce qui n'est pas le cas en mode historique.

Dans les deux cas, la communication est unidirectionnelle et la transmission est constante. Le compteur ne cesse de répéter une longue suite d'informations en boucle, dans lesquelles nous devons piocher celles qui nous intéressent. Une trame ainsi transmise, dans un mode ou dans l'autre, se compose d'une série d'octets divisés en une étiquette et une valeur, plus des marqueurs et autres sommes de contrôle. L'étiquette constituée d'une chaîne de caractères (comme « BASE », « IINST », « PAPP », « EAST », « SINSTS », etc.) qualifie la donnée qui suit dans la trame, comme l'index du compteur en Wh, l'intensité du courant fournie en A ou encore la puissance apparente en VA.

Pour arriver à lire ces données, il nous suffit d'utiliser un microcontrôleur ou un ordinateur (SBC ou PC) disposant d'une liaison série capable de gérer 1200 ou 9600 bauds en réception. Cependant, il est hors de question de brancher directement un tel dispositif au connecteur TIC du compteur (bien que des protections soient en place). Nous devons isoler galvaniquement notre circuit de celui du compteur et, pour cela, utiliser un optocoupleur. Le principe est tout simple : le compteur d'un côté de l'optocoupleur transmet les données, celles-ci sont transformées en impulsions lumineuses captées par un phototransistor qui, couplé à un MOSFET va moduler un signal lisible par le microcontrôleur ou le SBC. Notez que je dis ici « SBC » uniquement pour le principe, car il ne serait pas très raisonnable de monopoliser une Raspberry Pi pour ce genre d'usage simpliste.

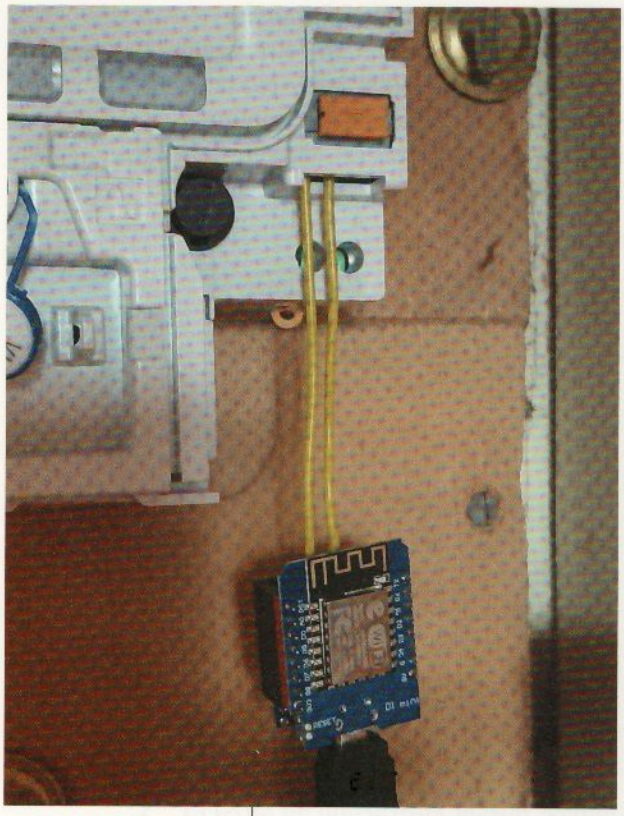
Le montage en lui-même est le suivant :



Il en existe de plus simples, faisant abstraction du MOSFET et reliant simplement le collecteur du phototransistor à RxD (la broche de réception UART du microcontrôleur) à l'aide d'une résistance de rappel à VCC, mais ils sont généralement assez peu stables, car destinés à une utilisation avec une carte Arduino (0/5 V). Mieux vaut faire les choses proprement pour garantir une bonne intégrité du signal. Le BS170 est un modèle courant, d'où son usage ici, mais n'importe quel MOSFET N fera l'affaire. Ceci sera suffisant pour interfacer votre TIC du Linky (ou d'un CBE) avec ce qui vous chante : ESP8266, RPi Pico, ESP32, Raspberry Pi, Orange Pi, convertisseur USB/série, etc.

2. DÉVELOPPONS… OU PAS

Nous disposons maintenant d'une liaison de données fiable entre le compteur et un hypothétique montage pouvant être n'importe quoi capable de recevoir un flux de données série en 1200 ou 9600 bauds. Nous pourrions implémenter un code permettant de traiter les trames et analyser leur contenu. Ce n'est pas bien difficile puisque le protocole est parfaitement décrit



La Télé-Information Client ou TIC est accessible très simplement via un connecteur repérable facilement par ces points de pression orange. Il faut pousser, assez fermement, pour ouvrir les mâchoires et glisser les connecteurs.

dans la documentation, aussi bien pour le mode historique que standard. Il n'y a cependant pas réellement d'intérêt à refaire ce qui a déjà été fait, de maintes façons différentes, par une quantité impressionnante de développeurs. Il existe des dizaines de codes, en différents langages, pour différentes plateformes et, à la base, notre objectif n'est absolument pas d'implémenter tout cela, mais de simplement utiliser les informations à disposition.

À propos d'informations justement, et en particulier si vote compteur est en mode standard (ou que vous avez demandé à votre fournisseur d'électricité de basculer en mode standard), la masse d'informations disponibles est plus qu'il n'en faut pour être heureux. Dans l'absolu, tout ce que nous avons besoin de récupérer est l'index du compteur en watt-heure (Wh) régulièrement. Ceci est identifié par l'étiquette « BASE » pour le mode historique et « EAST » pour le mode standard.

Bien sûr, selon votre contrat, vous avez peut-être besoin de plus d'informations, si vous souhaitez évaluer correctement votre consommation, voire l'ajuster en fonction de paramètres supplémentaires. En dehors du cas de base qui est le mien, avec un tarif unique à toute heure du jour et de la nuit, on distingue deux autres cas (notez bien que pour moi, ceci est totalement théorique) :

- Une différenciation entre des heures creuses et des heures pleines, avec un tarif réduit pour les premières et plein pour les secondes. Mon fournisseur Vialis propose cette option avec un tarif au kWh de 0,2460 € en heures pleines et 0,1828 € en heures creuses (au 20/11/2023). Ceci comparé à un tarif fixe de 0,2276 €/kWh pour un contrat de base. Les heures creuses peuvent être choisies comme étant « de nuit » (de 22 h à 6 h), ou « méridiennes » (de 1 h à 7 h et de 12 h à 14 h).
- Un *dispatch* encore plus poussé est proposé par EDF sous la forme d'un tarif variable, non seulement en fonction des heures creuses/pleines, mais également de la classification de la journée comme étant un « pic de consommation » (jour rouge, 22 par an), un jour « d'assez grande consommation » (jour blanc, 43 par an) ou un de « faible/moyenne consommation » (jour bleu, le reste des jours de l'année). Ceci nous donne donc non pas un tarif au kWh, non pas deux tarifs au kWh, mais six coûts différents d'un kilowatt-heure en fonction du jour et de l'heure.

Les étiquettes correspondant à tous ces tarifs sont décrites dans la documentation officielle Enedis **[2]**, que ce soit pour le mode historique ou le mode standard. Une petite subtilité cependant, le tableau (page 19) pour le mode standard

ne désigne pas explicitement les index correspondant aux heures pleines/creuses et les couleurs des jours. Voici la correspondance (non vérifiée expérimentalement par moi-même, mais utilisée dans de nombreuses configurations sur le Web) :

- « `EASF01` » : énergie soutirée Tempo Bleu heure creuse ;
- « `EASF02` » : énergie soutirée Tempo Bleu heure pleine ;
- « `EASF03` » : énergie soutirée Tempo Blanc heure creuse ;
- « `EASF04` » : énergie soutirée Tempo Blanc heure pleine ;
- « `EASF05` » : énergie soutirée Tempo Rouge heure creuse ;
- « `EASF06` » : énergie soutirée Tempo Rouge heure pleine.

En plus de ces quantités de watts-heures, qui permettent de suivre sa consommation globale, il peut également être intéressant d'avoir une idée de la puissance actuellement supportée par le compteur, en temps réel. Cette information, nommée puissance apparente s'exprime en kilovoltampères (kVA) et est, par définition, liée à la puissance active actuellement demandée par l'ensemble de votre installation électrique, exprimée en kilowatts. Pour une installation domestique, la confusion entre puissance apparente et puissance active est bénigne et on peut s'en tenir à la simple correspondance kVA = kW (en oubliant la puissance réactive et les histoires de facteur de puissance). La puissance apparente indiquée par le compteur (« `PAPP` » en mode historique et « `SINSTS` » en mode standard) représente donc la totalité de la puissance active demandée par votre habitation.

Pourquoi parler de tout cela si nous n'avons pas l'intention d'implémenter nous-mêmes l'analyse des trames ? Tout simplement, car il s'agit là de l'information minimale que vous devez connaître pour utiliser n'importe quelle implémentation existante, dont celle dont je vais vous parler ici : ESPHome.

3. UTILISATION AVEC ESPHOME ET HOME ASSISTANT

Nous avons déjà couvert la réalisation d'une installation domotique dans de précédents numéros (46, 47, 48 et 49) mais si vous venez d'acquérir ce magazine, car le mot « Linky » a attiré votre attention, il va falloir vous mettre à jour et c'est précisément ce que nous allons faire immédiatement, en quelques paragraphes. Notre installation domotique repose sur deux éléments :

- Une carte Raspberry Pi 3 équipée du système HAOS qui n'est autre qu'un GNU/Linux faisant fonctionner Home Assistant (dans un conteneur Docker). Il s'agit de notre centrale domotique, collectant les informations des capteurs via le réseau (ou d'autres biais de communication) et les présentant via une interface web directement accessible avec un simple navigateur. Home Assistant (« HA » dans le reste de l'article), qui est gratuit et *open source* va plus loin, en n'étant pas simplement passif, mais en permettant d'automatiser des actions en fonction des données collectées. Allumer une lumière via une prise connectée en cas de mouvement, déclencher une ventilation en fonction d'une température ou encore mettre en place une simulation de présence via des scripts sont autant de choses qu'il est possible de réaliser très simplement, soit via l'interface web, soit en composant des fichiers au format YAML.

- Des capteurs, installés un peu partout dans votre habitation et faisant généralement fonctionner un *firmware* ESPHome ou Tasmota. Dans le cadre simpliste de cette approche liée uniquement au Linky, nous n'aurons qu'un seul capteur, connecté à la prise TIC du compteur et communiquant avec HA via Wi-Fi/réseau.

HA est en mesure de collecter des informations de capteurs, et d'agir sur certains dispositifs (lumières, pompes, ventilations, etc.), même si ceux-ci ne font pas fonctionner un *firmware open source*. Une vaste gamme de produits commerciaux, utilisant des protocoles propriétaires, semi-propriétaires ou ouverts, est directement supportée. On parle, de manière générale, d'intégrations pour désigner le support pour chacun de ces dispositifs de la part de HA. Une **intégration** supporte un type (ou une famille) de protocoles. Un **équipement** est un dispositif pris en charge. Et une **entité** est une source de données précise. Si un capteur, disons un ESP8266 flashé avec ESPHome, fournit température et hygrométrie relative, cet équipement est pris en charge par l'intégration ESPHome et met à disposition deux entités, la température et un indicateur d'humidité.

Installer HA sur une Pi est un jeu d'enfant puisqu'il suffit d'utiliser l'image SD/MMC directement fournie via le site officiel **[4]**. Le premier démarrage sera relativement lent, mais à terme, vous disposerez d'un accès, en pointant simplement un navigateur sur **http://homeassistant.local:8123** (ceci est modifiable dans la configuration). Vous pourrez faire le tour du propriétaire, ajuster quelques réglages et vous familiariser avec l'interface. Il est également possible que, si des équipements domotiques sont installés chez vous (type lumières connectées, matériels multimédias, etc.), HA les détecte automatiquement et vous propose de les prendre en charge peu après l'installation. À vous de décider si vous voulez le faire de suite ou plus tard.

Côté capteurs, nous jetterons notre dévolu sur ESPHome pour deux raisons, la première étant que son « concurrent », Tasmota, ne gère pas directement la TIC Linky (ou CBE) comme le précise la page officielle **[5]**. Comme Tasmota est orienté « interface graphique » pour la programmation des capteurs avec un *firmware* ouvert, la nécessité de construire (compiler) son Tasmota en activant la fonctionnalité « Teleinfo » peut être un frein. La seconde raison est, tout simplement, qu'ESPHome est clairement mon environnement préféré, car non seulement ce projet à une relation assez intime avec les développeurs de Home Assistant, mais sa philosophie, reposant sur la composition de scripts décrivant les fonctionnalités à intégrer dans le *firmware*, est à mon sens bien plus puissante que la simple aisance d'utilisation de Tasmota (et en plus on peut « versionner » son jeu de scripts avec Git).

Utiliser ESPHome ne nécessite aucune connaissance en programmation, ni même une réelle maîtrise de la plateforme utilisée (ESP8266, ESP32, RP2040/PicoWn ou LibreTiny). Vous pouvez voir cela comme un ensemble de morceaux de codes (les composants) qui seront regroupés selon vos désirs pour être compilés automatiquement et flashés dans la mémoire de la cible. Tout ceci se fait en éditant un fichier de description au format YAML.

Pour installer l'environnement ESPHome et construire vos *firmwares*, la méthode la plus simple est d'utiliser le conteneur Docker dédié. Ceci impose l'utilisation d'un système supportant Docker (typiquement un GNU/Linux), mais fonctionne à merveille tout aussi bien avec une installation sur PC (AMD64) que sur un SBC de type Raspberry Pi 32 ou 64 bits (ARM ou AARCH64). L'installation en elle-même se résume à une simple ligne de commande : `docker pull ghcr.io/esphome/esphome`. Ceci intégrera une image Docker

Le nouveau tableau « Énergie » affiche un graphique bien plus sobre en termes de couleurs puisqu'il n'y a plus qu'une seule source de données, l'index de consommation fournie par la TIC du compteur. Mais le détail est ventilé, en bas, en fonction des capteurs installés dans l'habitation.

ESPHome contenant tout le nécessaire pour produire des images binaires (flash) et programmer vos capteurs. Le principe de fonctionnement est simple : vous écrivez un descriptif de ce que vous souhaitez dans un fichier YAML et vous invoquez le conteneur qui fera tout le travail à votre place.

Voici qui nous amène donc à la pièce maîtresse de l'opération, le script YAML permettant de créer notre capteur interfacé à la TIC du Linky. Pour cela, nous utiliserons un module ESP8266 et plus exactement un clone de Wemos D1 mini, disponible pour un prix entre 3 € et 5 € (plus, c'est du vol) selon la source et le temps que vous acceptez de patienter après commande. Celui-ci sera connecté au PC via un câble USB A vers USB micro B, qui pourra servir ensuite pour alimenter le montage via un adaptateur secteur (type chargeur de smartphone), et apparaîtra comme un port série vis-à-vis du système (`/dev/ttyUSBx` avec `x` étant une valeur `0`, `1`, `2`, etc.).

Il est possible de regrouper l'ensemble de la configuration YAML dans un seul fichier, mais je préfère personnellement diviser cela pour gagner en modularité et pouvoir réutiliser des éléments pour d'autres capteurs. L'arborescence utilisée est :

```
├── d1linky.yaml
└── inc
    ├── auth.yaml
    ├── secrets.yaml
    └── wifi.yaml
```

`d1linky.yaml` est le script que nous allons écrire et le répertoire `inc/` (comme *include*) contient les éléments réutilisables avec d'autres scripts, avec dans l'ordre :

• La configuration du Wi-Fi (client et point d'accès de secours en cas de problème) `wifi.yaml` :

```
wifi:
  ssid: !secret wifi_ssid
  password: !secret wifi_password

  # AP de secours
  ap:
    ssid: $devicename Fallback AP
    password: !secret ap_password
```

• Les informations d'authentification `auth.yaml` :

```
# Authentification API Home Assistant
api:
  encryption:
    key: !secret api_key

# Mise à jour du firmware OTA (over wifi)
ota:
  password: !secret ota_password
```

• Les informations sensibles `secrets.yaml` :

```
# Point d'accès de la maison
wifi_ssid: "monPointdAcces"
# Mot de passe wifi
wifi_password: "mot2passeWifi"
# Mot de passe de l'AP de secours
ap_password: "mot2passeAP"
# Mot de passe de mise à jour OTA
ota_password: "123456"
# Clé secrète partagée avec HA
api_key: "mMjIyNMGRjZDjYTQyZTJhN2U2ZTMUwZTMyMmJMxOGRm="
```

Les seules informations à adapter à votre situation se trouvent dans ce dernier fichier et les commentaires parlent d'eux-mêmes. Le point d'accès de secours sera créé par le capteur s'il n'arrive pas à se connecter au réseau domestique et ceci devrait vous permettre de mettre à jour le *firmware* sans avoir à le désinstaller physiquement pour le connecter en USB. Notez que la première programmation de l'ESP8266 passe obligatoirement par une connexion physique mais qu'ensuite, un changement de configuration se fera par Wi-Fi, soit en ligne de commande, soit directement dans l'interface web de HA (alias *Lovelace*). La clé référencée par `api_key` est une valeur 32 bits encodée en base64. Pour générer une telle clé aléatoirement, vous pouvez par exemple utiliser la commande `openssl rand -hex 16 | base64` (ou `openssl rand -base64 32`).

Et nous pouvons enfin nous occuper du fameux script en commençant par la partie générique :

```
# Le nom du capteur est une variable
substitutions:
  devicename: wemoslinky

# Nom du capteur
esphome:
  name: $devicename

# Architecture et plateforme
esp8266:
  board: d1_mini

# Désactivation du journal d'activité
logger:
  baud_rate: 0
  hardware_uart : UART1
  level: info
  esp8266_store_log_strings_in_flash: false

# Inclusion des configurations génériques
<<: !include inc/wifi.yaml
<<: !include inc/auth.yaml
```

La partie importante ici est celle débutant par `logger` qui désactive la console série sur le port série de l'ESP8266. Par défaut, il est généralement intéressant d'obtenir des informations sur l'activité « brute » du capteur (comme une sorte de *syslog*) mais ici, cela perturbe la lecture des informations puisque le port est utilisé pour la TIC. Nous passons ensuite à la configuration du port en tant que tel, et de l'objet représentant la source de données :

```
# Configuration port série
uart:
  id: uart_bus
  rx_pin: GPIO3
  tx_pin: GPIO1 # non utilisé
  baud_rate: 1200
  parity: EVEN
  data_bits: 7

# Source de données
teleinfo:
  id: mateleinfo
  update_interval: 30 s
  historical_mode: true
```

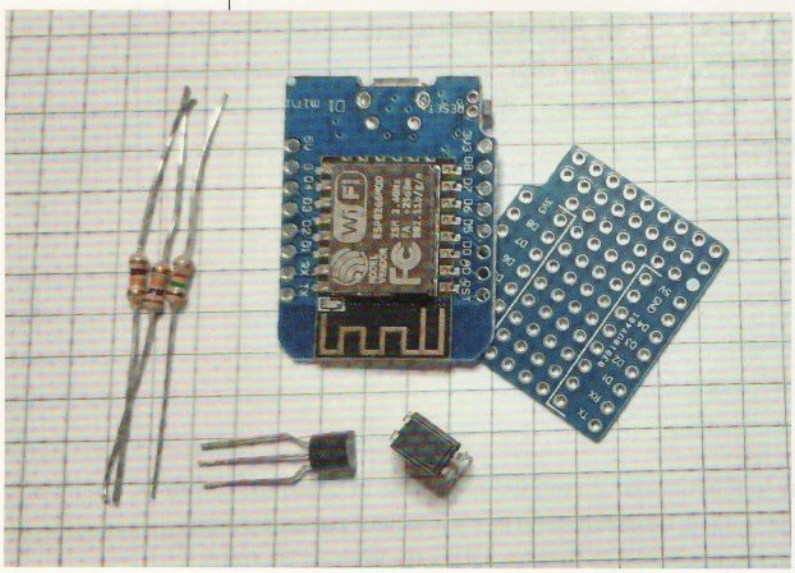
La TIC de mon Linky est configurée par le fournisseur en mode historique, la communication se fait donc en 1200 bauds. Pour le mode standard, ce serait 9600 et c'est le seul paramètre qui change. La réception se fera sur GPIO3, correspondant à la broche « RX » du Wemos D1 mini.

La transmission (« TX ») est également configurée mais rien n'est connecté sur cette broche. Notez le `historical_mode` à `true`, qui devient naturellement `false` pour le mode standard.

En dehors de la vitesse de transmission des données, les autres éléments qui changent entre le mode historique et standard sont les étiquettes. Voici ce que j'utilise :

```
# Liste des entités
# Une par étiquette TIC
sensor:
  - platform: teleinfo
    tag_name: "BASE"
    name: "base"
    unit_of_measurement: "Wh"
    teleinfo_id: mateleinfo
    state_class: total_increasing
    device_class: energy
  - platform: teleinfo
    tag_name: "IINST"
    name: "iinst"
    unit_of_measurement: "A"
    teleinfo_id: mateleinfo
  - platform: teleinfo
    tag_name: "PAPP"
    name: "papp"
    unit_of_measurement: "VA"
    teleinfo_id: mateleinfo
```

Si vous utilisez le mode standard, vous pouvez vous référer au début d'article ou directement à la documentation Enedis pour spécifier les étiquettes correspondantes. Il en va de même pour les informations relatives aux relevés heures pleines/creuses, ainsi que pour les déclinaisons en jours Tempo (bleu, blanc, rouge). L'élément **absolument capital** ici est la présence des lignes `device_class` et `state_class` pour l'entité `base` (l'index général du compteur en kWh depuis son installation). La première précise le type d'entité que verra HA et la seconde le type de données, qui **DOIT** être `total_increasing` (ou `total`) pour que la partie dédiée à la gestion d'énergie de HA puisse « voir » l'entité. Si vous ne spécifiez pas ces deux informations, l'entité `base` sera bien visible dans HA, mais ignorée dans la partie (tableau ou *dashboard*) ***Énergie***.



Le montage permettant de relier la TIC à une installation domotique se résume à peu de choses : un module ESP8266, une plaque pastillée, trois résistances, un MOSFET et un optocoupleur. Budget total : moins de 8 €.

Une fois ces lignes enregistrées dans un fichier au nom sans importance, mais ici judicieusement appelé `d1linky.yaml`, une simple ligne de commande invoquant le conteneur Docker sera suffisante pour transformer un simple module ESP8266 en ERL pour le Linky :

```
$ docker run --network=host \
  --rm -v "${PWD}":/config \
  --device=/dev/ttyUSB0
  -it esphome/esphome run d1linky.yaml
[...]
INFO Successfully compiled program.
Found multiple options for uploading, please choose one:
  [1] /dev/ttyUSB0 (USB2.0-Serial)
  [2] Over The Air (wemoslinky.local)
```

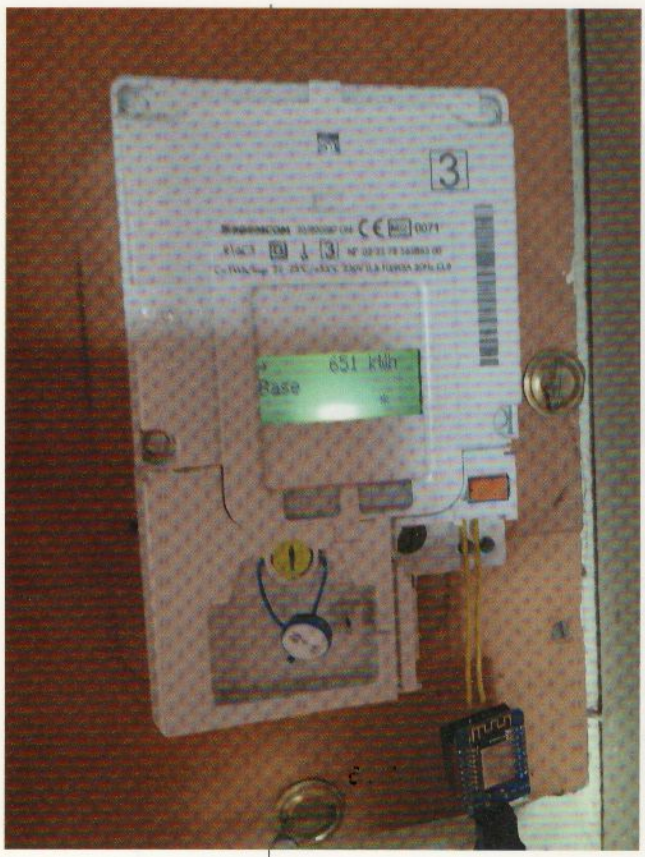
ESPHome vous demande, après construction, comment flasher le *firmware* en mémoire, via le port série spécifié dans les options (`/dev/ttyUSB0`) ou en OTA (*Over The Air*) via Wi-Fi en utilisant le nom du capteur comme nom d'hôte (complété de « .local » pour la résolution mDNS). Comme il s'agit de la première écriture, l'ESP8266 n'est pas connecté au réseau et vous êtes obligé d'utiliser le port série. Si vous faites des modifications du YAML par la suite, vous pourrez choisir l'OTA ou tout simplement supprimer l'option `--device=/dev/ttyUSB0` de la ligne de commande pour que la mise à jour se fasse automatiquement en OTA.

Il ne vous reste plus maintenant qu'à brancher le montage au module Wemos (RX, 3,3 V et masse) et le tout à L1/L2 sur le Linky. Puis à alimenter le tout, bien entendu...

3.1 Configurer le capteur dans HA

Dès que le Wemos D1 mini aura accès au réseau, il entrera en contact avec HA et l'interface web de ce dernier vous affichera une notification vous signalant l'arrivée d'un nouvel équipement. Suivez alors les indications pour intégrer ce périphérique et spécifiez la clé à utiliser (celle précisée avec `api_key` dans `secrets.yaml`). Un message devrait rapidement vous signaler que tout s'est passé correctement.

Vous retrouverez votre capteur dans ***Paramètres***, ***Appareils et Services***, ***ESPHome***, ***Wemos Linky*** et devriez, en plus des autres blocs, voir quelque chose comme ceci :



Une chose importante à savoir sur le Linky : la façade en plastique, dans le vert le plus affreux de l'Univers, est amovible et se retire pour accéder aux connecteurs TIC... Et rien ni personne ne vous oblige à la remettre en place ensuite.

Attendez un peu si les données ne sont pas présentées de suite, il faut un peu de temps (théoriquement 30 s) pour que le Wemos lise des données et les remontent à HA. Si rien ne s'affiche, vérifier tout d'abord dans quel mode est effectivement la TIC de votre compteur en utilisant l'interface en façade du compteur. Plusieurs appuis successifs sur l'un ou l'autre des boutons vous permettent de basculer l'afficheur sur l'information que vous cherchez. Si vous êtes bien dans le bon mode, vérifiez votre script YAML, vous avez peut-être fait une faute de frappe dans un nom d'étiquette. Et enfin, si ces points sont corrects, débrancher votre montage et vérifiez chaque connexion avec un testeur. Il est peu probable que l'interface TIC du compteur ait un problème mais si vous voulez vérifier, réalisez un petit montage avec deux LED en parallèle (une avec une polarité inverse de l'autre) et une résistance de 1,5 kΩ en série. Connectez le tout à la sortie TIC et vous devriez voir les LED faiblement clignoter, prouvant la présence d'un trafic. Un analyseur logique portable (type LA104) peut également être utile, mais le coût n'est, à mon sens, pas justifié (sauf comme excuse pour se donner bonne conscience).

Quoi qu'il en soit, si les informations sont correctement remontées dans HA, les entités subissent le traitement standard et les informations sont automatiquement stockées. Il vous est donc possible de les placer sur votre tableau de bord, que ce soit sous la forme du dernier état détecté ou d'un historique des états précédents. HA propose toute une sélection de « cartes » permettant de présenter les données sous forme de texte, de valeur, de jauge, de graphique, etc.

Mais la présentation la plus utile dans ce cas précis passe par un tableau de bord dédié que vous pourrez configurer via ***Paramètres***, ***Tableaux de bord*** et ***Énergie***. Là, vous trouverez une section ***Réseau électrique*** vous permettant de spécifier l'entité à utiliser qui dans mon cas (mode historique avec étiquette « BASE ») se nomme « base ». Le tableau de bord ***Énergie*** présente la consommation électrique (et éventuellement celle de l'eau et du gaz) sous la forme d'un graphique horaire, journalier, hebdomadaire, mensuel et/ou annuel, mais surtout, il peut y associer un coût.

Ainsi, avant de configurer le tableau ***Énergie***, il peut être intéressant de spécifier ce ou ces tarifs dans la configuration sous la forme d'entités particulières appelées « Entrées ». Rendez vous dans ***Paramètres***, ***Appareils et services***,

onglet ***Entrées*** et utilisez le bouton en bas à droite + ***Créer une entrée***. Là, sélectionnez le type ***Nombre*** et, dans le formulaire qui apparaît, donnez un nom (« tarifvialis » dans mon cas) et spécifiez l'unité de mesure « EUR/MWh » (euro par mégawatt-heure). Enregistrez les changements et cliquez sur l'entité dans la liste pour spécifier votre tarif (227,6 ici). Si vous utilisez un contrat heure pleine/creuse ou Tempo, créez simplement autant d'entrées qu'il y a de tarifs. Dans la configuration du tableau ***Énergie***, vous pourrez associer chaque relevé de consommation, décliné par type (heures creuses/pleines pour les jours bleu, blanc et rouge), à chacun des tarifs et donc avoir une vision globale et détaillée des mesures fournies par le compteur.

Ceci fait, vous pourrez aller dans la configuration du tableau de bord, à la section ***Réseau électrique*** et ajoutez l'entité (ou les entités) en cochant ***Utiliser une entité avec le prix actuel*** pour spécifier l'entrée avec le bon tarif. Dans mon cas, avec un contrat le plus basique possible, il n'y a que « base » et « tarifvialis », mais même avec un contrat Tempo EDF, vous pourrez constater que tout est également parfaitement pris en charge.

Une fois la ou les entités configurées, vous devrez patienter un peu (jusqu'à deux heures d'après le message de HS), le temps que les mesures soient collectées et traitées. Ceci peut prendre un certain temps et l'approche la moins frustrante, si vous êtes impatient, consiste à simplement oublier tout cela et y revenir plus tard. À terme, le tableau ***Énergie*** vous présentera un magnifique graphique de consommation accompagné d'un tarif (estimé, puisqu'il y a des coûts fixes supplémentaires sur vos factures).

Cette installation de base, réalisable à moindre coût (Raspberry Pi, Wemos D1 mini et quelques composants) et très rapidement, offre des avantages évidents. On garde un œil sur sa consommation en temps réel et étalée dans le temps. C'est tout simplement la base pour commencer à faire des économies et un excellent point de départ pour commencer à « domotiser » son lieu de vie ou un local professionnel.

4. INTÉGRER LA NOUVELLE DONNÉE À UNE INSTALLATION EXISTANTE

Dans le numéro 49, nous avons exploré une approche différente pour surveiller sa consommation électrique et les limites étaient claires. Non seulement un certain nombre d'équipements ne peuvent être surveillés facilement en raison de leur puissance (comme les plaques vitrocéramiques dépassant la capacité d'un capteur Sonoff POWR320D de 20 A), mais il n'est raisonnablement pas faisable d'équiper chaque prise et chaque circuit d'un capteur, ne serait-ce que pour des raisons de coût. Installer des capteurs partout est amusant (voir addictif), mais rappelez-vous que l'investissement doit être remboursé par les économies générées par l'installation.

En l'absence d'une mesure globale pour toute l'habitation, la mesure et la surveillance sont donc imparfaites et la facture du fournisseur d'énergie toujours supérieure à notre estimation. L'arrivée du Linky dans ma maison a changé totalement la donne, mais a surtout provoqué un énorme problème : le tableau de bord ***Énergie*** était utilisé de manière effective, mais incompatible avec la TIC d'un compteur.

Je m'explique. Nous avions déployé une flottille de capteurs Sonoff reflashés avec ESPHome et remontant des mesures de consommation. Celles-ci étaient ajoutées dans la partie ***Réseau électrique*** du tableau ***Énergie*** puisque c'était la seule solution disponible. Mais, et c'est un gros « mais », cette section est normalement destinée à regrouper les mesures provenant du « réseau » et non des points de consommation. C'est d'ailleurs pour cette raison qu'il est possible d'intégrer non seulement consommation électrique, mais également la production provenant de panneaux photovoltaïques, par exemple. De plus, il est également parfaitement envisageable de disposer de plusieurs sources, sur plusieurs bâtiments par exemple, et d'avoir ainsi une vision globale pour un parc de compteurs.

Le problème qui se pose dans une configuration où vous avez configuré les points de consommation comme des sources de données du réseau est que l'arrivée d'un compteur disposant d'une TIC vous ramène à une configuration classique. La conséquence de cela ne va pas vous plaire, car si vous êtes dans ma situation, intégrer proprement l'information provenant du compteur Linky est soit synonyme de cauchemar masochiste, soit de perte de son historique. Par « cauchemar », j'entends la tentative d'intégrer les données du Linky au même niveau que celles des Sonoff POWR en procédant à un ensemble de calculs (cf. plus loin). La procédure est théoriquement simple mais la mise en pratique excessivement difficile : utiliser l'index de consommation du compteur en watt-heure pour en déduire une consommation quotidienne manuellement, puis utiliser cette valeur pour en déduire la somme des relevés de la flottille de Sonoff et obtenir une différence représentant le nombre de watts-heures non collecté par les points de mesure individuels. Cette différence, quotidienne, peut alors théoriquement être utilisée comme source de données complémentaire dans le tableau ***Énergie***, comme s'il s'agissait d'un nouveau capteur.

Le problème qui se pose avec cette approche, en dehors du fait devoir dériver des données de tout un groupe d'entités via maints calculs, est celui de la synchronisation. Le moindre écart dans le *timing* des mesures ou des calculs provoque des aberrations dans les relevés du tableau ***Énergie*** (valeurs négatives, quantités astronomiques, etc.). Le tout, sachant que vous utilisez toujours le tableau en question d'une manière pour laquelle il n'est absolument pas prévu...

Au final, il est non seulement plus simple, mais aussi plus « propre » de tout bonnement se faire une raison et d'accepter de perdre les mesures précédentes, qui n'étaient de toute façon qu'une approximation, pour repartir sur des bases saines. Les capteurs Sonoff que nous avions déployés n'en deviennent pas inutiles pour autant, mais passent de la section ***Réseau électrique*** (où nous avons l'entité « base » de la TIC) à ***Appareils individuels***, dans la configuration du tableau. Cet emplacement, jusqu'à présent inutilisé, est là pour répartir la consommation globale entre différents points de consommation. Ils n'entrent donc pas dans le calcul du coût, car ils sont placés après la relève du compteur.

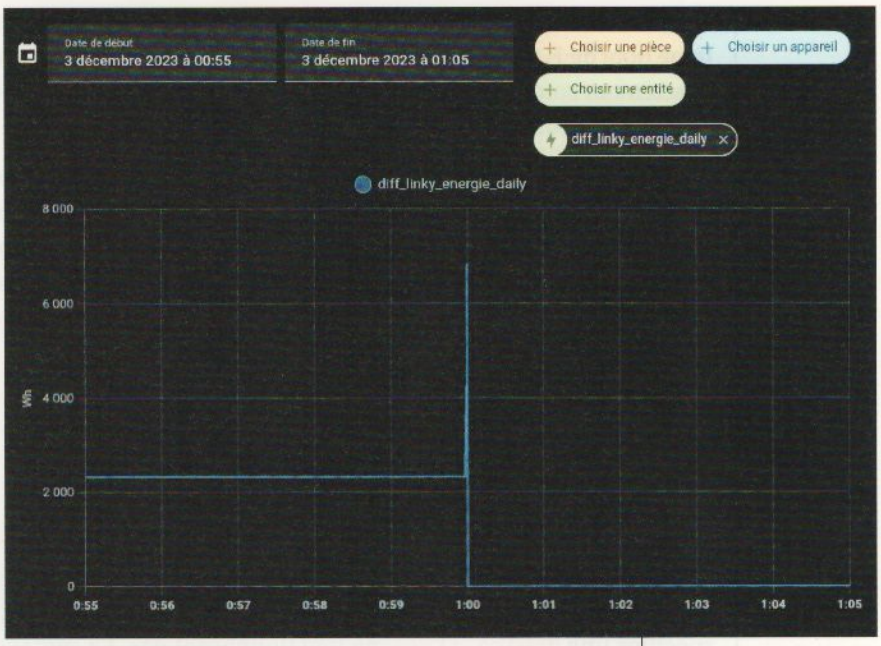
La configuration en tant que telle est donc très simple : on supprime les entités de la section ***Réseau électrique*** et on les ajoute dans ***Appareils individuels*** et ça s'arrête là. Immédiatement, le graphique général change et devient vide, le temps que les données soient collectées et, dessous, nous trouvons une partie ***Surveiller les appareils individuels*** avec un graphique horizontal, avec les mesures pour la période, classées par valeur décroissante en kWh. Ce graphique et ses données sont

présents si vous remontez dans le temps, mais pas la « consommation d'énergie » globale et le coût.

5. UN SUIVI EN TEMPS RÉEL ET PLEIN DE CALCULS

Le tableau de bord ***Énergie*** est très pratique, mais il reste très spécialisé. Il ne vous montrera pas, par exemple, la puissance utilisée en temps réel (en W), que ce soit celle des équipements surveillés avec les Sonoff POWR ou la différence avec ce que « voit » le Linky (en VA). Il ne vous montrera pas non plus la fameuse différence en énergie utilisée (Wh) entre la somme des mesures des POWR et l'index de consommation du Linky. Pour arriver à obtenir ces informations, il faut faire quelques efforts et créer de nouvelles entités, via le mécanisme de *template*. Un *template* est une entité présentant un état découlant de l'état d'une ou de plusieurs autres entités. Vous pouvez voir cela comme de « fausses » entités sans existence physique.

Le point le plus simple à régler ici est celui concernant la puissance utilisée.



Voici exactement le problème qu'on rencontre lorsqu'on essaie de déterminer la part non mesurée localement de sa consommation électrique, via des calculs impliquant de nombreuses entités. Au moment de la réinitialisation des index, dans les Sonoff et dans les templates HA, on se retrouve avec des données non fiables.

L'ensemble des capteurs (Sonoff ou Shelly reflashés avec ESPHome) fournissent directement une puissance en watts et il en va de même pour la TIC du Linky, mais en VA puisqu'il s'agit de puissance apparente (étiquette « PAPP » en mode historique). Pour connaître « ce qu'on rate » avec nos mesures individuelles, il suffit de prendre la puissance apparente et de lui déduire la somme des mesures de puissances actives de nos capteurs.

Créer un *template* nécessite une édition de fichier et plus particulièrement du fichier `configuration.yaml` se trouvant dans le répertoire `config/` de l'administrateur (*root*). Il existe plusieurs façons d'y accéder et de modifier ce fichier. La documentation recommande d'installer un module permettant de l'éditer directement depuis l'interface web (Lovelace) de HA. Vous les trouverez et pourrez les installer via ***Paramètres***, ***Modules Complémentaires*** et ***Boutique des modules complémentaires***. Le premier est « *Studio Code Server* » qui n'est autre qu'un IDE *Visual Studio Code* fonctionnant directement dans HA et le second, plus sobre, est « *File editor* ». Une troisième option, que je préfère largement, consiste à ajouter le module « *Terminal & SSH* » vous permettant d'avoir un accès au shell via l'interface web, mais aussi, et surtout, de

vous connecter en SSH à l'hôte où fonctionne HA et de transférer des fichiers avec `scp`. Le système hôte de HA sur Pi (HAOS) intègre l'éditeur de fichier Vim vous permettant donc de procéder exactement comme vous le faites sans doute déjà avec Raspberry Pi OS (oui, `nano` est aussi présent, mais soyons sérieux **[6]**).

Le support des *templates* dans HA repose sur le moteur de *templating* Jinja qui utilise une syntaxe particulière, mais qui n'est pas sans rappeler Python, ou même le shell Bash. Notez tout d'abord qu'il existe deux façons de créer des *templates*, mais seulement l'une d'elles devrait être utilisée. Par le passé, un *template* se déclarait avec quelque chose comme :

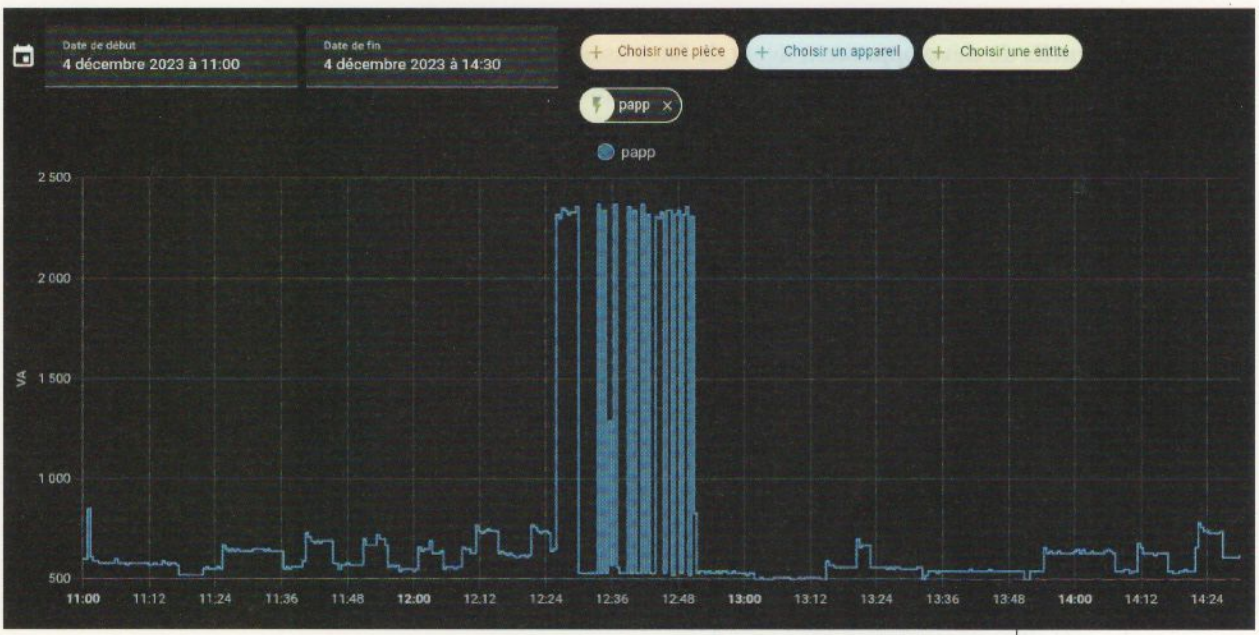
```
sensor:
- platform: template
  sensors:
    mon_template:
      friendly_name: 'Coucou le template'
[...]
```

Ceci est la syntaxe *legacy*, toujours supportée, mais qui n'est plus activement maintenue. Ceci signifie donc qu'un certain nombre de fonctionnalités, comme, par exemple, le fait de pouvoir spécifier une `device_class`, ne fonctionneront pas. La syntaxe *legacy* ne devrait plus être utilisée même si un grand nombre d'exemples de ce type peuplent le Web et les forums.

La nouvelle syntaxe utilise ceci :

```
template:
  - sensor:
    - name: "Total Power SonOff"
      state: >
        {{ expand('sensor.sonoffpowr316_1_power',
        'sensor.sonoffpowr316_2_power',
        'sensor.sonoffpowr316_3_power',
        'sensor.sonoffpowr316_4_power',
        'sensor.sonoffpowr316_5_power',
        'sensor.sonoffpowr316_6_power',
        'sensor.sonoffpowr316_7_power',
        'sensor.sonoffpowr320d_1_power',
        'sensor.sonoffpowr320d_2_power')
        | map(attribute='state') | map('float', 0)
        | sum | round(0) }}
      unit_of_measurement: W
```

Cet exemple est précisément notre fameuse somme de mesures de puissance provenant d'une tripotée de capteurs Sonoff POWR. Ce qui se trouve entre `{{` et `}}` est notre *template* en langage Jinja. Pour le comprendre, nous allons détailler l'utilité de chaque fonction (`expand()`, `map()`, etc.), mais si vous êtes coutumier de la ligne de commande, vous avez sans doute déjà remarqué l'utilisation du `|` permettant de *piper* ou connecter les commandes entre elles. La sortie de `expand()` est dirigée vers l'entrée de `map()`, et ainsi de suite. Suivons donc le cheminement dans l'ordre en commençant par `expand()`. Cette fonction prend en argument une liste d'entités et produit une autre liste, classée et développée (si une des entités en argument est un groupe).



Ce graphique est une représentation de la puissance nécessaire à la cuisson de quatre crépinettes accompagnées de haricots verts, avec une plaque vitrocéramique. Ce n'est pas très bien discernable sur la courbe, mais c'était un régal !

`map()` est ensuite utilisé deux fois pour extraire de cette liste chaque attribut d'état et sa valeur en un nombre à virgule, c'est la liste des mesures. Le tout est additionné avec `sum` puis la valeur obtenue est arrondie à l'entier le plus proche avec `round(0)`.

Avec ce *template*, nous créons une nouvelle entité appelée `sensor.total_power_sonoff` dont l'état est la somme des valeurs rapportées par tous nos capteurs concernant la puissance. Nous pouvons directement l'utiliser sur un tableau de bord, par exemple. Mais ceci constitue surtout le premier calcul nous rapprochant de l'objectif, que nous atteignons avec un autre *template* :

```
- sensor:
  - name: "diff_linky"
    state: >
      {{ ( states('sensor.papp') | float) -
      ( states('sensor.total_power_sonoff')
      | float  ) | round(0) }}
    unit_of_measurement: W
    device_class: power
```

Là, les choses sont bien plus simples, à condition que l'on n'oublie pas le `float` pour transformer l'état de `sensor.papp` et `sensor.total_power_sonoff`, obtenus avec la fonction `states()`, en valeur numérique (par défaut, les états sont des données textuelles). Cette simple soustraction nous donne l'entité `sensor.diff_linky` qui peut venir compléter la liste des puissances relevées par les Sonoff sur un tableau de bord dédié. Je vous fais grâce ici de la création du *template* `sensor.total_power` qui reprend exactement la même chose que `sensor.total_power_sonoff`, mais ajoute également `sensor.diff_linky` pour arriver à un total identique à `sensor.papp`, permettant donc de vérifier que tout fonctionne (et d'avoir une belle somme en bas de la liste).

À propos de vérification justement, pensez à toujours utiliser ***Vérifier la configuration*** dans ***Outils de développement*** avant de faire usage de ***Redémarrer***, pour vous assurer que la syntaxe de votre `configuration.yaml` ne risque pas de bloquer le démarrage de HA. Autre point intéressant, l'onglet ***Modèle*** (traduction assez malheureuse de « template ») de ce même écran vous permet de tester vos *templates* et donc d'en vérifier la syntaxe et la sortie.

Passons maintenant au plus gros problème. Les capteurs Sonoff proposent une entité appelée, par exemple, `sensor.sonoffpowr316_1_total_daily_energy` fournissant la quantité d'énergie (en Wh) mesurée pour la journée. Cette valeur augmente au fur et à mesure et est réinitialisée à minuit UTC (le terme « GMT » est normalement déconseillé aujourd'hui). C'est cette valeur qui était utilisée précédemment par le tableau ***Énergie*** dans mon installation, car contrairement au Linky, l'index global de mesure n'est pas conservé en cas de rupture d'alimentation du Sonoff, ce qui peut provoquer des problèmes. Et, justement, le Linky, lui, n'a pas de « total_daily_energy », uniquement la valeur en watt-heure qui ne fait qu'augmenter. La question est donc : comment obtenir une quantité d'énergie quotidienne à la manière des Sonoff à partir de l'index du compteur ?

Si vous avez envie de répondre « il suffit de prendre la valeur actuelle et lui soustraire celle que la TIC nous donnait à minuit », vous avez parfaitement raison. Le raisonnement est évident, mais l'implémentation plus complexe.

Pour réaliser cela, nous avons besoin de stocker l'index du compteur à minuit (UTC, j'insiste) et donc d'un endroit pour le faire. Ceci s'appelle une entrée de type nombre ou `input_number`. Vous pouvez la créer facilement via ***Paramètres***, ***Appareils et services***, onglet ***Entrées*** et bouton ***Créer une entrée***. Dans le formulaire qui apparaît, spécifiez un nom (« linky_conso_minuit »), une valeur minimum (0), une valeur maximum (999999999) et une unité de mesure (« Wh »). Pour initialiser ce compteur manuellement et pouvoir tester sans attendre le milieu de la nuit, affichez le graphique de l'entité liée à l'étiquette « BASE » (mode historique) et relevez la mesure vers le minuit précédent. Pas besoin d'une valeur exacte, il faut juste être approximativement dans les bonnes valeurs pour avoir quelque chose de cohérent de suite et pour le reste de la journée courante.

À ce stade, nous avons une entité qui possède comme état une valeur d'index à un instant « t ». Nous pouvons donc créer un nouveau *template* pour faire le calcul, exactement comme pour la puissance :

```
- sensor:
  - name: "linky_total_daily_energy"
    state: >
      {{ ( states('sensor.base') | float) -
      ( states('input_number.linky_conso_minuit') |
      float  ) | round(0) }}
    unit_of_measurement: Wh
    device_class: energy
```

Ceci fonctionnera immédiatement et nous pouvons constater que `sensor.linky_total_daily_energy` mesurera la quantité d'énergie utilisée (ou fournie par le compteur) depuis notre minuit arbitraire. Nous pouvons même pousser un peu plus loin en calculant, de suite, la différence avec la somme de l'énergie quotidienne mesurée par les Sonoff :

```
- sensor:
  - name: "Total Energie SonOff"
    state: >
      {{ expand('sensor.sonoffpowr316_1_total_daily_energy',
      'sensor.sonoffpowr316_2_total_daily_energy',
      'sensor.sonoffpowr316_3_total_daily_energy',
      'sensor.sonoffpowr316_4_total_daily_energy',
      'sensor.sonoffpowr316_5_total_daily_energy',
      'sensor.sonoffpowr316_6_total_daily_energy',
      'sensor.sonoffpowr316_7_total_daily_energy',
      'sensor.sonoffpowr320d_1_total_daily_energy',
      'sensor.sonoffpowr320d_2_total_daily_energy')
      | map(attribute='state') | map('float', 0)
      | sum | round(0) }}
    unit_of_measurement: "Wh"
- sensor:
  - name: "diff_linky_energie_daily"
    state: >
      {{ [0, ( states('sensor.linky_total_daily_energy')
      | float) - ( states('sensor.total_energie_sonoff')
      | float  ) | round(0)] | max }}
    unit_of_measurement: Wh
    state_class: total
    device_class: energy
```

Notez la syntaxe légèrement différente d'avec la puissance, car nous avons ici une donnée cumulative qui est réinitialisée toutes les 24 heures. Ceci signifie qu'une fraction de seconde de décalage peut conduire au calcul d'une valeur négative. Pour un simple affichage sur le tableau de bord, ce n'est pas un problème, ni même avec un graphique historique, mais si vous avez l'intention d'utiliser cela comme une source de données pour le tableau ***Énergie***, c'est un énorme problème. Notez que ceci ne règle pas réellement le problème et nous avons toujours un souci au moment de la réinitialisation, découlant sur un pic aberrant dans les valeurs. C'est exactement ce qui m'a fait abandonner l'idée de procéder à ces calculs pour ajouter le différentiel Linky/Sonoff dans le tableau ***Énergie***. C'est une mauvaise idée et ça ne fonctionne pas.

Il ne nous reste plus maintenant qu'à nous occuper de la mise à jour de notre index nocturne, mais pour cela, nous avons besoin d'une intégration supplémentaire : `time_date`. Celle-ci ne peut être ajoutée via l'interface web, un message est très clair à ce sujet en tentant l'opération (mais pourquoi alors la proposer dans la liste ?) et nous devons le faire dans notre `configuration.yaml` :

```
sensor:
  - platform: time_date
    display_options:
      - 'time'
      - 'date'
      - 'time_utc'
```

Ici, un redémarrage complet de HA sera nécessaire, puisque vous n'aurez pas accès à l'information dans le cas contraire. Remarquez également l'indentation, nous ne sommes pas dans la section `template:` mais ajoutons un nouveau capteur. Cette intégration met à disposition un certain nombre d'entités **[7]** mais nous n'en considérons ici que trois, respectivement, l'heure, la date et l'heure UTC.

Ceci fait, nous pouvons créer l'automatisation qui sera chargée, à l'heure donnée, de faire exactement ce que les Sonoff font en interne :

```
alias: Reset Linky daily energie
description: ''
trigger:
  platform: template
  value_template: "{{ is_state('sensor.time_utc', '00:00') }}"
action:
  - service: input_number.set_value
    data:
      value: "{{ states('sensor.base') | float(0) }}"
    target:
      entity_id: input_number.linky_conso_minuit
```

Le déclencheur est ici l'état de `sensor.time_utc` et l'action est une copie de l'état de `sensor.base`, la valeur courante de l'index du Linky remonté via la TIC, dans `input_number.linky_conso_minuit`, notre entrée stockant l'index à minuit. Ceci revient donc à faire ce que nous avons fait manuellement après avoir créé l'entrée, mais automatiquement, et toutes les nuits. Inutile de redémarrer pour appliquer le changement, il suffit de passer par ***Outils de développement*** et rafraîchir les automatisations en cliquant sur ***Automatisations*** dans la liste. L'état de notre entité `sensor.diff_linky_energie_daily` sera donc continuellement à jour, en temps réel, et nous pouvons l'ajouter au côté des `sensor.sonoff*_total_daily_energy` des Sonoff sur un tableau de bord. Bien entendu, on pourra compléter le tout en créant un autre *template* pour faire la somme de toutes ces valeurs et pour obtenir un total, normalement égal à `sensor.linky_total_daily_energy`.

6. POUR FINIR

Relier son compteur électrique, qu'il s'agisse du Linky ou d'un CBE, est un avantage certain lorsqu'il s'agit d'une installation domotique existante, mais plus intéressant encore, et surtout plus simplement, c'est un excellent premier pas pour débuter en domotique. En effet, l'investissement de départ, loin de l'achat de prises connectées et de capteurs en tous genres, est minimal : un SBC comme une Pi, un Wemos D1 mini, quelques composants et le tour est joué. Bien sûr, compléter le tout avec des capteurs placés à des points stratégiques est un plus et réserve parfois des surprises.

Ainsi, j'ai fait fonctionner mon installation durant plusieurs mois avec près d'une dizaine de Sonoff POWR, pensant réellement couvrir la majorité de la consommation électrique de l'habitation. Tout ce qui n'était pas surveillé était les luminaires aux plafonds et la plaque vitrocéramique hors limite de puissance. Je pensais réellement être relativement proche des chiffres figurant sur mes factures d'électricité. C'était une grossière erreur !

Jongler avec les valeurs et procéder aux calculs comme nous venons de le voir n'est pas anodin et m'a permis de constater que 150 à 300 watts étaient « oubliés » en permanence, représentant quelque chose comme 2 à 3 kWh quotidiennement et donc entre 45 et 70 centimes

d'euro TTC par jour (160 à 250 euros par an). Certes, on peut supposer que l'éclairage non surveillé, l'usage ponctuel de la plaque vitrocéramique ou encore l'utilisation d'appareils (aspirateur, scie à onglet, ponceuse, etc.) sur des circuits dénués de capteurs (typiquement au sous-sol), peut expliquer cet écart. Mais la réalité est plus surprenante et troublante au moment où, tôt le matin, alors que la maison est encore en sommeil, quelque 200 watts disparaissent on ne sait où. Aurais-je résolu le mystère cosmologique de l'énergie sombre **[8]** et découvert la force qui tend à accélérer l'expansion de l'Univers ?

Non, pas vraiment. Vous l'avez peut-être deviné, mais je ne le savais pas. La coupable est... ma chaudière fioul !

Hé oui. Même si instinctivement, on pense que ce genre d'installation consomme, par définition, du fioul, un volume non négligeable d'énergie électrique est englouti par ce genre de dispositif. Ce n'est certes pas comparable à un ballon d'eau chaude électrique, mais en additionnant la puissance des pompes, de l'électronique, du transformateur THT permettant de générer un arc électrique, de l'électroaimant de la valve de la pompe et surtout du circulateur (externe à la chaudière mais faisant partie intégrante du système), on arrive à un total non négligeable de l'ordre de 100 à 300 watts, selon le mode de fonctionnement et l'âge de la chaudière. Ajoutez à cela le fait que mon installation combine chauffage (radiateurs fonte) et l'eau sanitaire et vous avez l'explication concernant cette mystérieuse énergie fantôme...

Cette découverte, que je trouve relativement surprenante, je ne l'aurais certainement pas faite si on ne m'avait pas changé mon compteur et si je n'avais pas révisé mon installation domotique pour l'occasion. Il va de soi qu'un nouveau capteur Sonoff POWR sera installé sous peu, pour surveiller la chaudière et le circulateur et, en fonction des données collectées, des changements seront effectués après maints tests. La question étant de savoir si couper totalement la chaudière la nuit (durant environ 5 h) est ou non un avantage financier (puisqu'il faut remonter en température plus durement le matin, surtout en hiver).

Je ne peux donc conclure cet article qu'en vous recommandant, si vous le pouvez, d'utiliser la TIC de votre compteur. Avec ou sans Sonoff POWR, ça ne peut être qu'une bonne idée pour vos finances. **DB**

RÉFÉRENCES

[1] **https://connect.ed-diamond.com/hackable/hk-049/surveillez-votre-consommation-electrique-et-faites-des-economies**

[2] **https://www.enedis.fr/media/2035/download**

[3] **https://www.kelwatt.fr/guide/compteur/electricite/linky-prise-secrete**

[4] **https://www.home-assistant.io/installation/**

[5] **https://tasmota.github.io/docs/Teleinfo/**

[6] **https://twitter.com/cyberbl0g/status/1278413186748887040**

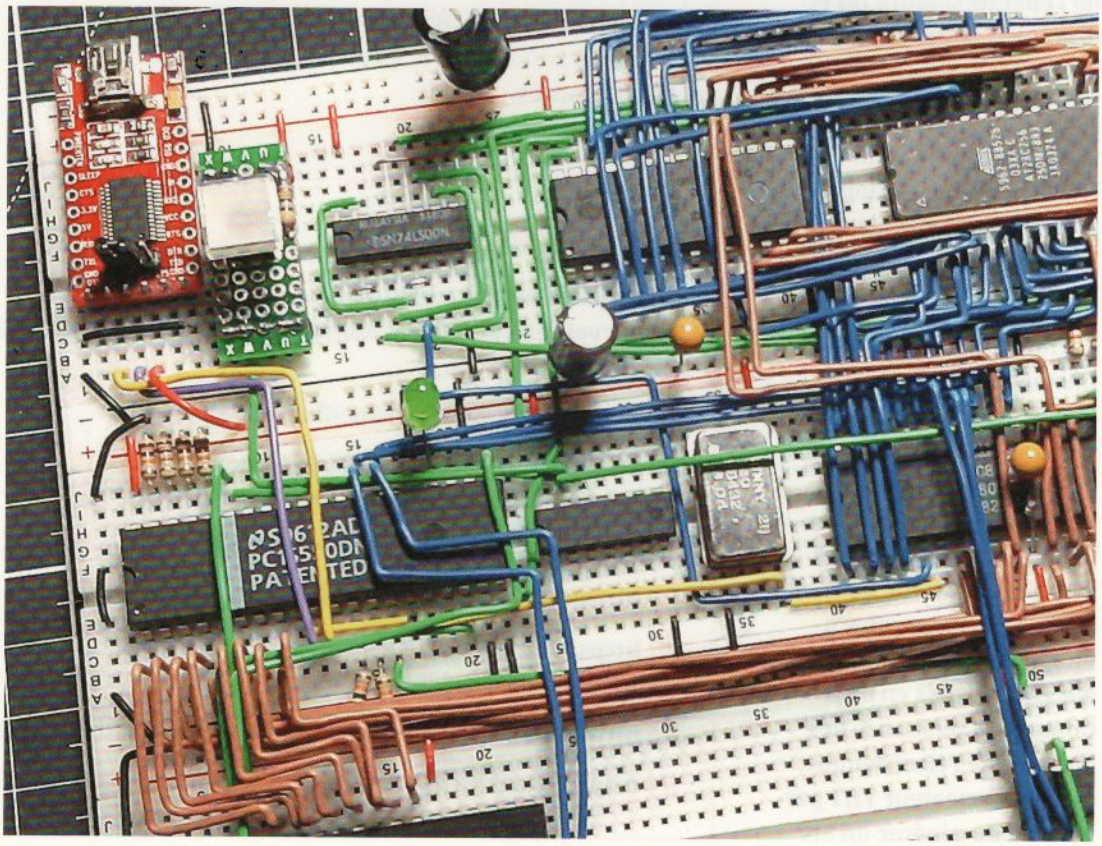
[7] **https://www.home-assistant.io/integrations/time_date/**

[8] **https://fr.wikipedia.org/wiki/%C3%89nergie_sombre**

Z80 SUR PLATINE À ESSAIS : LE C, ENFIN !

Denis BODOR

Nous avons mis un certain temps à atteindre notre objectif, mais nous y sommes. Nous voici fin prêts à programmer en C pour notre Z80 et c'est précisément ce que nous allons faire aujourd'hui. Très peu d'assembleur sera encore nécessaire à présent, et nous disposons désormais d'un crt0 préparant parfaitement le terrain et faisant ce pour quoi il est conçu : jp _main !



Ne vous inquiétez pas, il nous reste du travail à faire. Au stade où nous en sommes, nous disposons d'un *runtime* C (*crt0*) permettant de préparer l'exécution du code issu de la compilation de notre (ou nos) future(s) source(s) C. Pour rappel, ce code en assembleur, stocké dans le fichier `mycrt0.s` est le premier qui sera exécuté après un *reset* du Z80. Il sera placé à l'adresse `0x0000` et son travail est d'initialiser à zéro toutes les futures variables globales (ou plus exactement « statiques ») et d'initialiser avec leur valeur celles déclarées avec une affectation. Tout cela après avoir initialisé le registre de la pile (SP pour *Stack Pointer*) avec l'adresse mémoire la plus haute disponible (`0xFFFF`) et avant de finalement passer la main à la routine identifiée par le label `_main`, qui est en réalité la fonction `main()` de notre source C.

Pourquoi donc cette différence de nom, puisque notre fonction s'appelle `main()` et non `_main()` ? Ceci est un mécanisme historique toujours utilisé aujourd'hui : les noms de fonctions C, une fois traduits en assembleur, sont automatiquement préfixés d'un trait de soulignement (*undescore*), car ils deviennent des labels, et ceux-ci ne peuvent en aucun cas correspondre à des directives ou des mots-clés de l'assembleur. `main` n'est pas réellement un problème, mais imaginez simplement que vous ayez besoin d'une fonction C `ret()`. Si ce nom est simplement réutilisé à l'identique, il correspond à l'*opcode* `RET` du Z80 (qui prend la valeur au sommet de la pile et la place dans le registre PC) et ne peut donc pas être utilisé comme label. Notre `ret()` devient donc tout simplement `_ret`, et notre `main()` sera donc `_main`. Ceci sera important par la suite...

1. ALLONS-Y ! DES FICHIERS, PLEIN DE FICHIERS !

Plutôt que de détailler l'architecture du code et donc de me paraphraser sur des points que nous avons vus sous un autre angle dans les précédents articles, faisons le tour des différents fichiers qui composent ce projet à l'heure actuelle.

1.1 util.s et util.h

Notre *crt0* fait le travail de base, mais rien de plus, et en particulier, il ne se charge pas de la gestion des ports. Comme nous l'avons vu dans le numéro 50, notre UART doit être configuré pour supporter la vitesse de communication de notre choix, et ce, avec un format de données bien spécifique (9600 8O2). L'émission et la réception de données passent également par l'accès à ces ports, directement accessibles via les bus d'adresse et de données lorsque la broche /IORQ du Z80 est mise à la masse. Deux instructions assembleur, `in` et `out` font cela automatiquement, mais nous devons les rendre accessibles au code C. Pour ce faire, nous créons `util.s` :

```
        .module util
        .area   _CODE

; void outb( BYTE port, BYTE byte );
_outb::
        push bc
        ld c,a
        out (c),l
        pop bc
        ret
```

```
; BYTE inb( BYTE port );
_inb::
        push bc
        ld c,a
        in a,(c)
        pop bc
        ret
```

Ceci ne devrait pas être nouveau pour vous, car il s'agit ni plus ni moins que de la directe mise en œuvre des principes détaillés dans le précédent numéro : la *calling convention* de SDCC ou, en d'autres termes, comment passer des arguments à des routines et fonctions. Notez les `::` en suffixe des labels, permettant de rendre ces symboles visibles globalement et donc accessibles à travers tout le code du projet. Ainsi que les `_` au début des labels en question, comme expliqués précédemment.

Bien sûr, l'assembleur est la base, mais nous devons également détailler au compilateur C comment utiliser ces routines. Pour cela, nous devons en spécifier les prototypes dans le fichier `util.h` que nous « inclurons » (`#include`) dans notre fichier source principal (`prem.c`) :

```
#ifndef _UTIL_H_
#define _UTIL_H_

#define lowByte(w) ((uint8_t) ((w) & 0xff))
#define highByte(w) ((uint8_t) ((w) >> 8))

extern void outb(const char port, const char byte);
extern unsigned char inb(const char port);

#endif /* _UTIL_H_ */
```

En plus des deux prototypes pour `outb()` et `inb()`, nous définissons également deux macros nous permettant d'extraire un octet d'un mot de 16 bits. Ceci nous simplifiera la vie, plus loin, lorsqu'il s'agira de spécifier le diviseur pour configurer la vitesse de communication de l'UART.

1.2 uart.h et config.h

À propos d'UART justement, devoir connaître par cœur les adresses des différents registres n'est pas pratique, pas plus que d'être obligé de mémoriser l'utilité de chaque bit dans certains d'entre eux. Ceci n'est ni agréable ni pertinent en termes de maintenance du code. Nous pouvons donc réunir les informations pertinentes dans un fichier `uart.h`, sous la forme d'une série de macros aux noms intelligibles et explicites :

```
#ifndef _UART_H_
#define _UART_H_

/* UART_ADDR in config.h */
// RX Buffer/TX Holding Register
```

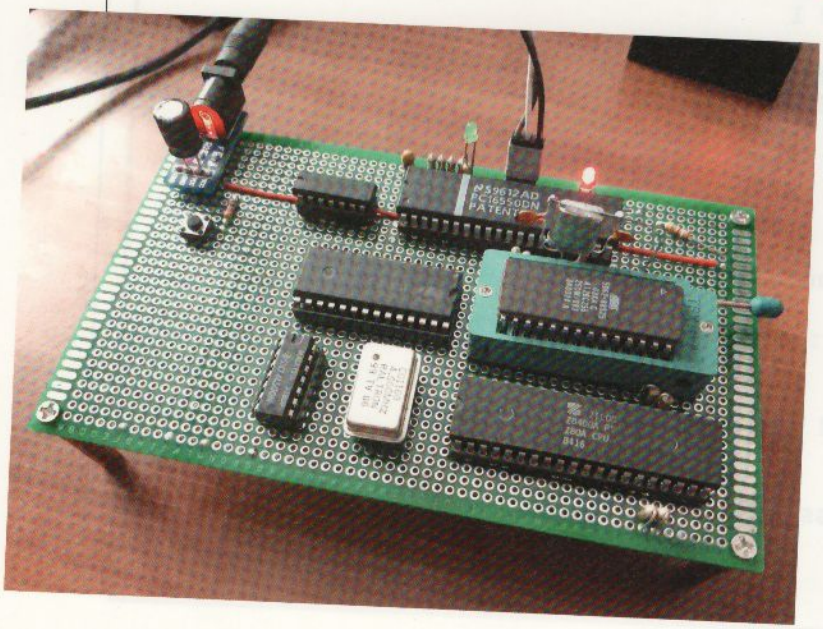
```
#define UART_RBR (UART_ADDR)
// Interrupt Enable Register
#define UART_IER ((UART_ADDR) + 0x01)
// Interrupt Identifier Register
#define UART_IIR ((UART_ADDR) + 0x02)
// FIFO Control Register
#define UART_FCR ((UART_ADDR) + 0x02)
// Line Control Register
#define UART_LCR ((UART_ADDR) + 0x03)
// Modem Control Register
#define UART_MCR ((UART_ADDR) + 0x04)
// Line Status Register
#define UART_LSR ((UART_ADDR) + 0x05)
// Modem Status Register
#define UART_MSR ((UART_ADDR) + 0x06)
// Scratch Register
#define UART_SCR ((UART_ADDR) + 0x07)
// Divisor Latch LSB (when DLAB=1 in LCR)
#define UART_DLL (UART_ADDR)
// Divisor Latch MSB (when DLAB=1 in LCR)
#define UART_DLM ((UART_ADDR) + 0x01)

// Baud divisor with 1.8432 MHz Xtal
// Xtal_Hz / (baudrate * 16) -> DLH:DLL
#define UART_BPS_1200   96
#define UART_BPS_2400   48
#define UART_BPS_4800   24
#define UART_BPS_9600   12
#define UART_BPS_19200  6
#define UART_BPS_38400  3
#define UART_BPS_56000  2
#define UART_BPS_115200 1

// Line Control Register bits
// Word Length Select Bit 0 & Bit 1
// 00=5bits, 01=6bits, 10=7bits, 11=8bits
#define UART_LCR_WLS0   0x01
#define UART_LCR_WLS1   0x02
// Number of Stop Bits : 0=1bit, 1=2bits
#define UART_LCR_STB    0x04
// Parity Enable : 1=enable
#define UART_LCR_PEN    0x08
// Even Parity Select : if parity on -> 0=odd, 1=even
#define UART_LCR_EPS    0x10
// Stick Parity
#define UART_LCR_STICK  0x20
// Set Break
#define UART_LCR_SBRK   0x40
// Divisor Latch Access Bit
#define UART_LCR_DLAB   0x80

#endif /* _UART_H_ */
```

Tous ces registres et bits ont été vus dans le numéro 50 et n'ont donc rien de neuf ici. Ce qui l'est, en revanche, c'est la facilité que cette approche (assez classique) nous offre, en particulier sur l'aspect vitesse de communication. Ici, tous les calculs sont faits pour différentes vitesses en fonction de la fréquence d'horloge utilisée par l'UART. Passer de 1200 b/s à 9600 b/s, par exemple, est l'affaire d'un simple choix de macro à présent. Et, oui, mes commentaires sont en anglais, car c'est un fichier que je pourrai utiliser pour un projet public (un jour).

Le fichier `uart.h` précise en commentaire (sinon, j'oublie) que `UART_ADDR`, l'adresse de base du PC16550DN dans notre montage, provient de `config.h`. Ce fichier contient les informations « globales » sur la plateforme. Ce sont celles que nous changerons si, d'aventure, notre montage est modifié physiquement, en termes de répartition ROM/RAM par exemple, ou d'ajout d'autres périphériques, utilisant donc d'autres ports. À ce stade, cette configuration ressemble à ceci :

```
#ifndef _CONFIG_H_
#define _CONFIG_H_

/* 32kB ROM at 0x0000 */
#define ROM_START 0x0000
#define ROM_SIZE 0x8000
#define ROM_END  ((ROM_START)+(ROM_SIZE)-1)

/* 32kB RAM at 0x8000 */
#define RAM_SIZE 0x8000
#define RAM_START 0x8000
#define RAM_END  ((RAM_START)+(RAM_SIZE)-1)

#define MEM_SIZE 0xffff

/* Base addresses of devices */
#define UART_ADDR 0x00

#endif /* _CONFIG_H_ */
```

La plupart de ces informations ne nous seront pas utiles immédiatement. Seul `UART_ADDR` importe pour l'instant.



1.3 prem.c

Et nous voici arrivés à la pièce de résistance qui, comme promise, est en C et fait la quasi-totalité du travail. Pour cela, nous devons configurer l'UART comme bon nous semble, créer quelques fonctions de base nous permettant d'utiliser `printf()` et faire quelque chose de vaguement intéressant et démonstratif dans notre boucle principale. Commençons donc par le plus important, la configuration de la communication :

```
#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include "config.h"
#include "uart.h"
#include "util.h"

// Configuration UART
void uart_init(void) {
    // Désactive les interruptions
    outb(UART_IER, 0);
    // Active DLAB pour la configuration du diviseur
    outb(UART_LCR, UART_LCR_DLAB);
    // Diviseur octet bas
    outb(UART_DLL, lowByte(UART_BPS_9600));
    // Diviseur octet haut
    outb(UART_DLM, highByte(UART_BPS_9600));
    // Conf. 8 bits de données, 2 de stop, parité impaire et reset DLAB
    outb(UART_LCR, UART_LCR_WLS0 | UART_LCR_WLS1 | UART_LCR_STB | UART_LCR_PEN);
}
```

Ceci est littéralement la traduction en C de ce que nous avons vu la dernière fois. Nous ajustons le contenu des différents registres pour obtenir une communication 9600 8O2. Après l'appel à cette fonction, notre port série est prêt à être utilisé en écrivant dans le registre servant de *buffer* RX/TX (`RBR`), après avoir vérifié que l'emplacement en question était disponible via `LSR`, le *Line Status Register*. Ceci nous permet de créer une fonction `putchar()` utilisée par le `printf()` fourni par la LibC de SDCC :

```
// Écrire un caractère
int putchar(int c) {
    // Attendre que l'UART soit prêt
    while( !(inb(UART_LSR) & 0x20) ) { ; }
    // Envoi du caractère
    outb(UART_RBR, c);
    return(0);
}

// Recevoir un caractère
unsigned char getc(void) {
    // Attendre l'arrivée d'un caractère
    while( !(inb(UART_LSR) & 0x01) );
    // Lire le caractère dans le registre
    return inb(UART_RBR);
}
```

Comme vous pouvez le voir, nous en profitons également pour créer `getc()`, permettant de recevoir un caractère en attendant que celui-ci apparaisse dans le registre `RBR`. Cependant, recevoir un caractère après l'autre n'est pas très pratique pour traiter les saisies d'un utilisateur et nous pouvons construire par-dessus cette fonction simpliste pour créer quelque chose de plus étoffé :

```
// Recevoir une ligne
size_t getline(char *buf, size_t size)
{
    size_t count;
    int c = 0;

    if (size == 0)
        return 0;

    for (count = 0; count < size - 1; count++) {
        c = getc();
        if (c == 0x0d ) { // '\n'
            if (count == 0)
                return 0;
            break;
        } else {
            buf[count] = (char) c;
        }
    }

    buf[count] = '\0';
    return (size_t) count;
}
```

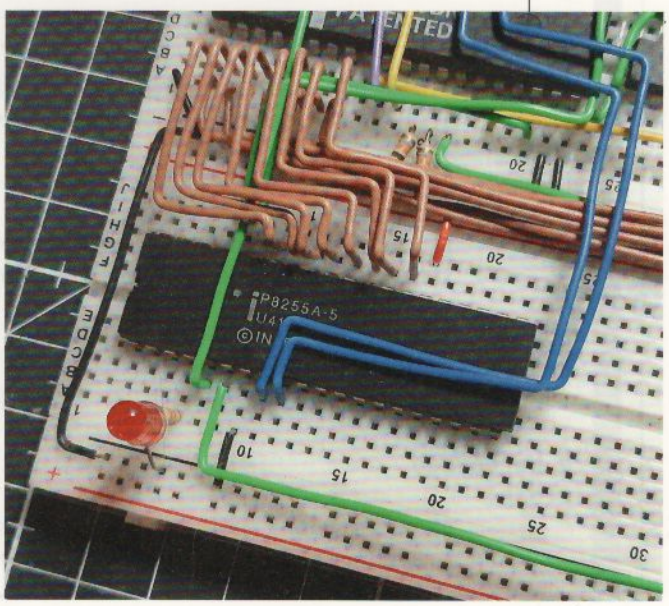
Cette fonction attend un certain nombre (`size`) de caractères qui est équivalent à l'espace disponible pour les stocker (`*buf`). Nous bouclons ensuite sur les appels à `getc()` jusqu'à soit atteindre le nombre maximum de caractères moins 1, soit tomber sur un caractère CR (*Carriage Return*, un retour chariot). Ceci nous permet donc, via un émulateur de terminal (Minicom, par exemple) connecté à notre UART, d'envoyer des lignes de texte, avec un nombre maximum arbitraire de caractères. Et c'est précisément notre intention dans la fonction `main()` :

```
void main(void)
{
    char *buffer;
    size_t bufsize = 32;
    size_t characters;
    int i;

    uart_init();

    buffer = (char *)malloc(bufsize * sizeof(char));

    if(buffer == NULL)
        while(1)
            printf("Erreur malloc!\r\n");

    printf("Z80 monitor v0.0.1 pre-alpha\r\n");
    printf("\r\n\r\n");
```

```
    while(1) {
        printf("Tapez un texte: ");
        characters = getline(buffer, bufsize);

        if (characters > 0) {
            printf("\r\n%zu car. réceptionné(s).\r\n", characters);
            printf("Vous avez envoyé: '%s'\r\n", buffer);
            for (i = 0; i < characters; i++) {
                printf("[%02x]", buffer[i]);
            }
            printf("\r\n");
        } else {
            printf("Rien à afficher.\r\n");
        }

        printf("\r\n");
    }

    // Inutile, on arrive jamais ici
    free(buffer);
}
```

Ce code est relativement simple puisqu'il se contente d'initialiser l'UART, d'allouer un *buffer* pour les caractères reçus et d'afficher un petit message avant de partir dans une boucle infinie. Celle-ci affiche simplement un *prompt* demandant la saisie de texte et retourne à l'utilisateur la représentation hexadécimale de ce qui a été traité.

Notez l'utilisation de `malloc()` qui n'est aucunement obligatoire ici, nous aurions tout aussi bien pu déclarer un tableau, mais je trouve tout bonnement fantastique d'utiliser une allocation sur le tas (dans la *heap*) dans un tel contexte.
Nous avons littéralement un groupe de composants, sauvagement assemblés sur platine à essais ou plaque pastillée qui, pourtant, forment un système fonctionnel avec une pile et un tas, permettant d'utiliser `malloc()` et `printf()`. C'est magique !



1.4 Makefile

Assembler et compiler les différents éléments de notre projet à la main n'est clairement pas amusant. Nous devons automatiser cette procédure pour pouvoir nous concentrer sur l'aspect le plus important : l'évolution du code. Pour éviter de passer notre temps à taper des commandes, nous pouvons faire appel à `make` et donc créer un `Makefile` qui trouvera place au milieu de tous les fichiers sources :

```
# Définition de quelques variables
MEMSIZE  := 0x8000
TARGET   := prem
CC       := ${SDCCPATH}sdcc
AS       := ${SDCCPATH}sdasz80
OBJCPY   := ${SDCCPATH}sdobjcopy
CFLAGS   :=
LFLAGS   := -Wl-u
MFLAGS   := -mz80 --no-std-crt0 --code-loc 0x0100 --data-loc 0x8000 --verbose

ROMTYPE := AT28C256
ROMSIZE := 32k
DD      := dcfldd

# Tout construire c'est construire prem.bin et prem_padded.bin
all: ${TARGET}.bin ${TARGET}_padded.bin

mycrt0.rel: mycrt0.s
        $(AS) -plosgffw mycrt0.rel mycrt0.s

util.rel: util.s
        $(AS) -plosgffw util.rel util.s

${TARGET}.rel: ${TARGET}.c
        $(CC) -mz80 --verbose -c -o ${TARGET}.rel ${TARGET}.c

${TARGET}.ihx: mycrt0.rel util.rel ${TARGET}.rel
        $(CC) $(MFLAGS) $(LFLAGS) -o ${TARGET}.ihx mycrt0.rel ${TARGET}.rel
util.rel

# Conversion de l'Intel Hex en binaire
${TARGET}.bin: ${TARGET}.ihx
        # makebin -p < ${TARGET}.ihx > ${TARGET}.bin
        $(OBJCPY) -Iihex -Obinary ${TARGET}.ihx ${TARGET}.bin

# Image à la taille de l'EEPROM
${TARGET}_padded.bin: ${TARGET}.ihx
        $(OBJCPY) -Iihex -Obinary --gap-fill 0x00 --pad-to \
        $(MEMSIZE) ${TARGET}.ihx ${TARGET}_padded.bin

# Programmation de l'EEPROM (TL866XX)
prog: ${TARGET}_padded.bin
        minipro -p ${ROMTYPE} -w ${TARGET}_padded.bin

# Nettoyage
clean:
        rm -f *.cdb *.rel *.hex *.ihx *.lst *.map *.o *.rst \
        *.sym *.lnk *.lib *.bin *.mem *.lk *.noi *.asm *.dis
```

Ceci fait, il nous suffira d'utiliser `make`, `make prog` ou encore `make clean` pour, respectivement, compiler le code, enregistrer le résultat dans une EEPROM avec `minipro` et faire le ménage dans les fichiers objets et binaires générés automatiquement. Bien sûr, comme il s'agit d'un système de construction dédié à un tel travail et reposant sur un mécanisme de dépendances, un `make prog` sur un projet « vierge » provoquera automatiquement l'assemblage, la compilation et l'édition de liens, si ce n'est pas déjà fait.

CONCLUSION

Nous avons à présent la base d'un système de développement pour notre ordinateur 8 bits et n'avons que très peu d'intérêt à utiliser l'assembleur. La structure même du projet est prévue dans ce sens, en mettant à disposition le strict minimum au plus bas niveau pour développer tout le reste en C. C'est un choix et vous pouvez, si vous le souhaitez, adopter une autre approche comme, par exemple, développer les routines de configuration, d'envoi et de réception série en assembleur.

La suite de cette série d'articles sera intermittente et dépendante des fonctionnalités que nous pourrons ajouter à cette plateforme. Dans l'état, nous avons un système relativement minimal, même en comparaison avec quelque chose comme une carte Arduino. Nous avons RAM, ROM et port série, mais aucun autre périphérique : pas de *timer*, pas de GPIO, pas de DMA, pas de contrôleur d'interruptions, etc. Faire évoluer le projet consistera donc à l'agrémenter de nouveaux composants pour ajouter de plus en plus de périphériques et de fonctionnalités. Je pense en particulier à l'intégration d'un Intel 8255 (alias PPI pour « *Programmable Peripheral Interface* ») qui pourrait nous fournir 24 GPIO, à condition de gérer le fait que les deux périphériques (UART et PPI) partagent le bus d'adresse lorsqu'il s'agit d'accès aux ports. Ceci peut être une problématique intéressante, impliquant l'utilisation de circuits logiques discrets supplémentaires (série 74xx) ou, pourquoi pas, de PAL et de GAL (pour être plus « moderne »).

À présent que notre escapade a atteint un point décisif, libre à vous d'explorer plus loin sur cette base, mais vous pouvez également basculer sur quelque chose de plus évolué comme le « Zeal 8-bit Computer » **[1]**, également construit sur une base de Z80. C'est moins « bricolo » que l'option « platine à essais », mais vous serez contraint par les choix du créateur de la plateforme.

Ou alors, comme le dirait Karadoc, « du passé faisons table en marbre ». Vous pouvez parfaitement oublier le Z80 et réutiliser tous ces principes pour construire une nouvelle machine, basée sur un MOS 6502, un Motorola MC68008 ou, pourquoi pas, un Intel 8088. La structure générale sera la même et les outils logiciels sont disponibles en *open source*, mais le jeu consistera à vous adapter aux spécificités du nouveau processeur. Certains font usage de modes particuliers (comme le mécanisme de *zero-page* du 6502) ou encore utilisent certaines broches à la fois pour le bus de données et le bus d'adresse (8088/8086), ce qui soulève toujours des problématiques très intéressantes.

En vérité, la limite est votre imagination et le temps que vous pouvez investir, car nous vivons une époque formidable où tout est à portée de main pour qui veut réellement comprendre comment fonctionnent toutes ces technologies, et ça aussi, c'est magique ! **DB**

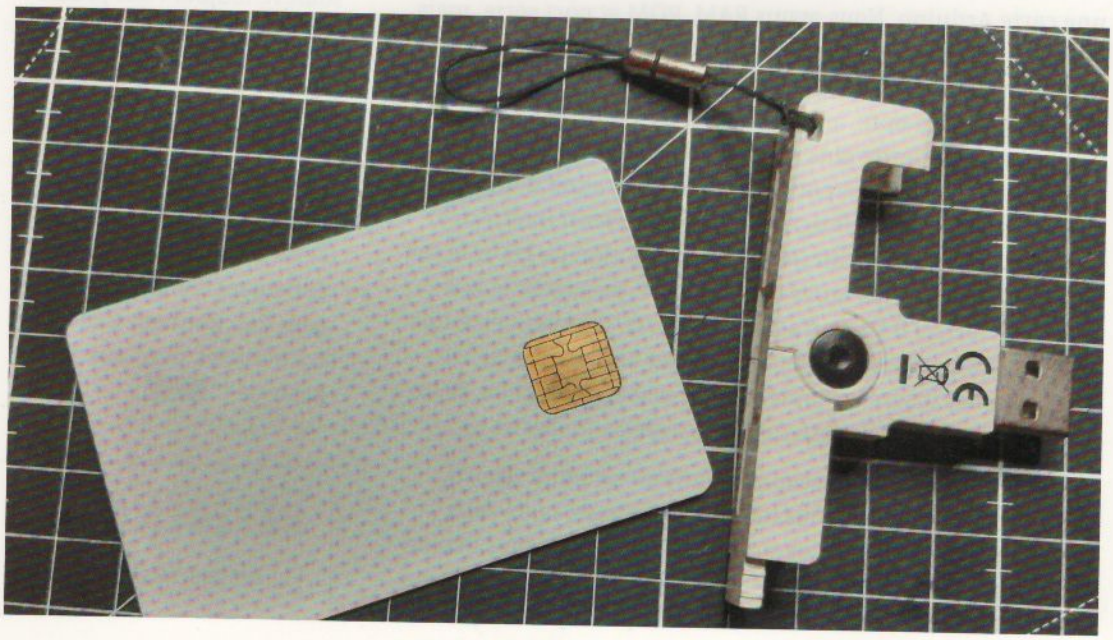
RÉFÉRENCE

[1] **https://zeal8bit.com/**

CRÉONS UNE APPLICATION POUR UNE CARTE À PUCE : HELLO WORLD !

Denis BODOR

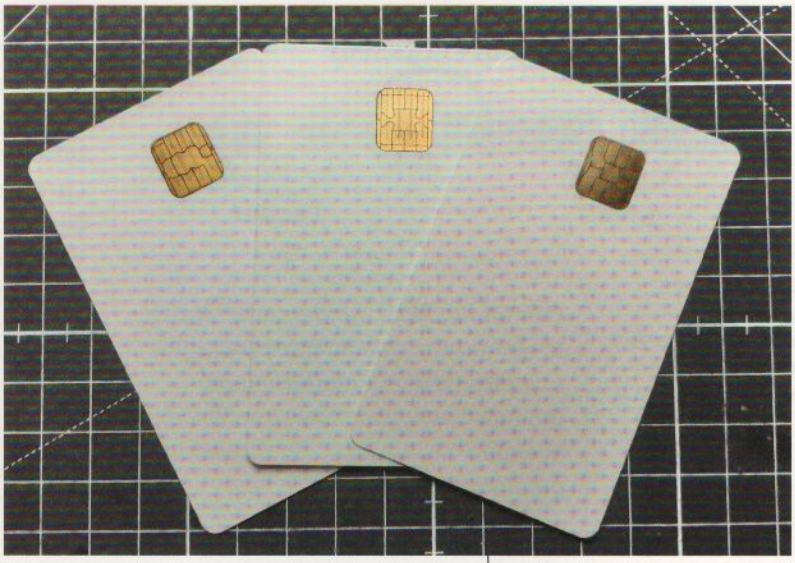
En regardant une carte à puce comme sa carte bancaire ou la SIM d'un smartphone, on ne se doute pas un instant de ce qui se cache derrière ce petit rectangle de contacts métalliques. Il s'agit ni plus ni moins que d'un microcontrôleur avec sa flash, sa RAM et une tripotée de périphériques et modules intégrés. Pour prendre en main cette technologie, tout ce qu'il vous faut, c'est un lecteur, une carte à puce adaptée, un environnement de développement dédié et, ici, une certaine tolérance vis-à-vis du langage Java.



– Créons une application pour une carte à puce : Hello World ! –

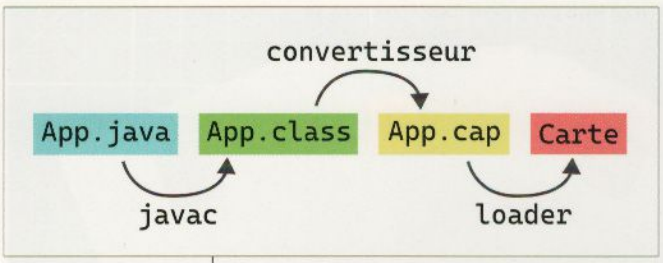
Il existe toute une gamme de cartes à puce (ou *smartcard*) puisque ces termes ne désignent rien de plus qu'un support en plastique d'une taille standard équipé d'un circuit intégré (mémoire et/ou processeur) avec des contacts apparents à un emplacement tout aussi standardisé. L'invention, brevetée par le français Roland Moreno en 1974, a grandement évolué au cours de ces 50 dernières années et les gammes et types de cartes sont tout aussi diversifiés que les *tags* RFID/NFC, qui sont parfois techniquement également des cartes à puce. La proximité entre les deux technologies ne s'arrête pas là, car comme pour les *tags*, les cartes à puce reposent sur des standards et des mécanismes souvent similaires. La façon de communiquer et d'échanger des données repose (dans de très nombreux cas) sur un protocole normalisé (ISO 7816 section 4) utilisant des *Application Protocol Data Unit* ou APDU. C'est le format standard de données utilisé pour dialoguer avec le composant et celui-ci repose sur une série de champs organisée d'une certaine façon (classe, instruction, paramètres, taille, données, états, etc.). Si vous êtes coutumier des technologies NFC, vous retrouverez rapidement vos petits avec une *smartcard*.

Et c'est là d'ailleurs précisément la raison d'être de cet article, car RFID et NFC composent un domaine dont je suis de plus en plus friand. Or, justement, dans ma collection de *tags* se trouvait une quantité non négligeable de *tags* combinant RFID



Un désagrément insoupçonné lorsqu'on commence à s'intéresser au domaine des smartcards est la difficulté à différencier les cartes les unes des autres dans sa collection. Ici, on a, de gauche à droite, une Cyberflex Access e-gate 32K (Java Card 2.1.1), une NXP J2A081 JCOP 2.4.1 R3 (Java Card 2.2.2) et une plus moderne ACS ACOSJ-GJ1AACSA (Java Card 3.0.4). La solution tient en un mot : de l'ordre.

13,56 MHz (MIFARE DESFire MF3ICD40, arrêté depuis 2010), piste magnétique HiCo (*High Coercivity*) et Java Card 2.2.2 sous la forme d'une puce NXP J2A081 JCOP 2.4.1 R3. Tout cela acquis il y a fort longtemps, sur eBay (si j'ai bonne mémoire) sous la forme d'un lot à un prix dérisoire. Comprenez bien que ces *tags*/cartes sont définitivement obsolètes, aussi bien sur la partie RFID que *smartcard* et ceci va être un problème. NXP fabrique toujours ce type de produits, mais avec une gamme bien plus moderne et des prix qui sont loin de concurrencer celui de la vieillerie qui nous occupe ici, même via AliExpress (par exemple **[1]**). Remarquez qu'on trouve également, sur eBay, des vendeurs qui bradent (~20 € pour 5 pièces) des cartes similaires utilisant une puce J2A040 d'exactement la même famille que la J2A081. Vous pouvez aussi tenter votre chance en optant pour des annonces encore plus attractives, en cherchant « *chip card rfid* », par exemple. On tombe tantôt sur des affaires (10 exemplaires à moins de 9 €) où le vendeur propose ce qu'il pense



Processus de construction d'une application ou applet Java Card.

être un *tag* RFID ou une carte à piste magnétique, mais intègre en réalité une Java Card et non une simple mémoire de type SLE4442 ou 24C16 (à éviter, ce *chip* n'est définitivement pas l'objet de l'article).

Ce qui va suivre devrait s'appliquer exactement de la même manière quelle que soit la Java Card (J2A081, J2A040, J2E145, J2A080n, etc.) et pourra également être transposé à des cartes/puces plus récentes comme la J3R150 (ou J2H145, J3H082n, etc.). Comme vous allez le constater, tout se joue sur la version du SDK à utiliser qui, lui-même, est lié avec une version donnée du JDK.

1. JAVA ?

Vous l'aurez compris, les *smartcards* sont littéralement de microscopiques ordinateurs sans interfaces pour les humains. Qui dit « ordinateur » dit forcément « programme » et donc « langage de programmation ». Certaines cartes sont programmables en C, en Java ou même en BASIC, selon le fabricant et le modèle. Ici, nous avons affaire à une Java Card, un terme décrivant un jeu de spécifications définies par Sun Microsystems (maintenant Oracle) et respectées par un certain nombre de fabricants (NXP, Gemalto, etc.) pour leurs produits. Contrairement à ce qu'indique la page Wikipédia française **[2]**, Java Card n'est pas un système d'exploitation, mais une technologie intégrant, entre autres, l'utilisation d'un système d'exploitation appelé JCOP pour *Java Card OpenPlatform* (dont le nom combine Java Card et les spécifications GlobalPlatform de Visa). Voilà qui explique donc immédiatement la désignation du produit et précise également les versions des spécifications utilisées : Java Card 2.2.2 et JCOP 2.4.1. Notez que ces informations sont capitales pour identifier votre ou vos *smartcards* et qu'on voit également tantôt des mentions comme « *compliant to JC3.0.4 and to GP 2.2.1* » faisant référence respectivement à JCOP (« JC ») et *GlobalPlatform* (« GP »), qui est un standard décrivant la manière de gérer le contenu des *smartcards* en question.

Comme on peut s'y attendre, le système JCOP intègre une machine virtuelle Java ou JCVM (*Java Card Virtual Machine*) permettant d'exécuter du *bytecode* Java. Si vous vous êtes déjà frotté à ce langage, ceci peut paraître surprenant, car il n'est pas spécialement réputé pour sa sobriété en termes d'espace disque ou d'utilisation mémoire. Il faut bien comprendre cependant que nous avons affaire à quelque chose de très différent d'une machine virtuelle Java standard (voir la présentation de D. Donsez page 6 à 10 **[3]**), mais qui présente un avantage évident ici : la portabilité du code entre les différents modèles de cartes compatibles avec les spécifications, quel que soit le fabricant. Un programme, ou plus exactement une application (ou *applet*), écrite pour une Java Card répondant à une certaine version des spécifications s'exécutera de la même manière sur une autre utilisant le même standard (à quelques nuances près, puisque certaines fonctionnalités cryptographiques sont optionnelles dans les spécifications).

Comme pour le développement de n'importe quel code Java, vous aurez donc besoin d'un JDK (kit de développement Java) en plus d'un JRE

(l'environnement d'exécution Java), avec une petite particularité ici : notre Java Card est ancienne et repose sur des spécifications (2.2.2) qui correspondent avec une certaine version du kit de développement Java Card (Java Card SDK ou « JC Kit ») qui n'est pas compatible avec un JDK récent. Mais, d'un autre côté, l'outil *open source* que nous allons utiliser pour gérer le contenu de la carte et installer notre application, *GlobalPlatformPro*, demande un JDK récent (> 11). De la même manière, un problème de compatibilité se pose avec la conversion du format de *bytecode* utilisé par la carte.

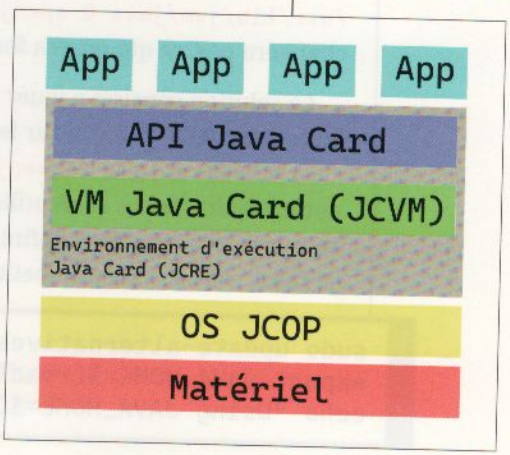
En effet, ce ne sont ni des `.class`, ni des `.jar` standard qui sont utilisés pour placer l'application sur la carte, mais un « JAR » spécifique via un fichier utilisant l'extension `.cap`. En fonction de la version du Java Card SDK utilisé, l'outil permettant de créer ce fichier ne sait prendre en entrée que des fichiers de classe Java à hauteur d'une certaine version. Dans le cas du SDK en version 2.2.2 que nous devons utiliser, la version maximum du fichier de classe Java en entrée est 49 (correspondant Java 5, alias 1.5, voir **[4]**) et celui-ci ne pourra pas être produit avec un compilateur et un JDK récent.

Le créateur de GlobalPlatformPro, Martin Paljak, résume les correspondances via un tableau présent sur une page wiki GitHub de l'un de ses autres projets **[5]** que nous utiliserons également. On constate alors que nous avons besoin d'un JDK en version 8 en plus du JDK récent mis à disposition via le système de gestion de paquets de notre distribution GNU/Linux. J'utilise Debian GNU/Linux 12 et le JDK par défaut est OpenJDK 17, soit l'implémentation *open source* de la plateforme Java SE en version 17. Ceci conviendra parfaitement pour compiler et exécuter GlobalPlatformPro, mais pas pour l'application Java Card elle-même. Nous allons devoir « bricoler » un peu...

2. JONGLER AVEC LES VERSIONS DE JAVA

Fort heureusement, JDK 8 n'est censé être en fin de vie qu'en 2030 (version LTS) et de ce fait, on peut raisonnablement penser qu'installer une telle version sur un système récent est possible. Et effectivement, en fouillant un peu, on constate qu'il existe une solution qui ne cassera pas le système de gestion de paquets de la distribution. Notez au passage qu'on parle ici de l'implémentation *open source* du JDK et donc d'OpenJDK par opposition au JDK Oracle qui est propriétaire. Je n'ai tenté l'expérience ni avec la version Oracle ni avec un autre système que Debian 12, mais ceci doit également être possible.

Pour installer OpenJDK 17 sur Debian, rien de plus simple puisqu'il suffit d'utiliser le paquet `openjdk-17-jdk-headless`, « *headless* » faisant référence au fait que cette version ne produit pas d'applications capables d'afficher et de gérer des interfaces graphiques (fenêtre, *widget*, composants AWT, etc.). La version 8 n'est, bien entendu, pas disponible par défaut,



Architecture logicielle interne d'une Java Card.

mais plutôt que de triturer le contenu de son `/etc/apt/sources.list`, il suffira de pointer un navigateur sur **https://packages.debian.org/fr/sid/openjdk-8-jre-headless**. Là, on suivra l'un des liens en bas de page, à la section « *Télécharger openjdk-8-jre-headless* », « amd64 » pour x86 64 bits (ou « arm64 », ou « armhf », pour une Pi respectivement 64 ou 32 bits), pour être redirigé vers une page de sélection de miroirs permettant de directement récupérer le fichier `.deb` correspondant. Le JDK ayant une dépendance vers le JRE, il sera également nécessaire de faire de même avec le paquet `openjdk-8-jre-headless` (listé dans les dépendances sur la page `openjdk-17-jdk-headless`).

Une fois ces deux fichiers `openjdk-8-jdk-headless_8u382-ga-2_amd64.deb` et `openjdk-8-jre-headless_8u382-ga-2_amd64.deb` obtenus, il suffira de procéder à l'installation avec `sudo dpkg -i` suivi du nom des fichiers. Pour vérifier que tout est correctement installé, listez simplement les paquets avec quelque chose comme :

```
$ dpkg -l | grep jdk
ii  openjdk-17-jdk-headless:amd64   17.0.8+7-1~deb12u1[...]
ii  openjdk-17-jre:amd64            17.0.8+7-1~deb12u1[...]
ii  openjdk-17-jre-headless:amd64   17.0.8+7-1~deb12u1[...]
ii  openjdk-8-jdk-headless:amd64    8u382-ga-2[...]
ii  openjdk-8-jre-headless:amd64    8u382-ga-2[...]
```

Vous voici donc avec deux JDK installés, mais comment choisir l'un ou l'autre ? Lancer un `javac -version` vous affichera simplement la version du compilateur Java par défaut. Heureusement, Debian utilise un mécanisme spécifique pour ce genre de situation. La commande `update-alternatives` vous permet de basculer entre plusieurs versions d'un programme si plusieurs paquets le fournissent. Votre `javac`, placé dans `/usr/bin`, est en réalité un lien symbolique pointant vers `/etc/alternatives/javac` qui est également un lien symbolique, qui à ce moment précis, pointe vers le vrai binaire : `/usr/lib/jvm/java-17-openjdk-amd64/bin/javac`.

En utilisant `sudo update-alternatives --config javac`, un menu s'affiche vous permettant de choisir, via un numéro, quelle version de `javac` vous souhaitez utiliser par défaut. Mais ceci ne changera que le lien symbolique de cette commande précise, qui deviendra alors `/usr/lib/jvm/java-8-openjdk-amd64/bin/javac`. Le reste de l'environnement Java ne changera pas, ce qui posera forcément problème pour la suite des opérations.

La solution consiste à jouer sur la variable d'environnement `JAVA_HOME`, désignant l'endroit où est installé le JDK/JRE sur le disque, soit `/usr/lib/jvm/java-17-openjdk-amd64/`, soit `/usr/lib/jvm/java-8-openjdk-amd64/`. Ça tombe bien, c'est précisément là où se trouve notre binaire `javac` et en utilisant la commande `readlink` ainsi qu'une substitution de chaîne avec `sed`, nous pouvons définir cette variable automatiquement. Tout ce que nous avons besoin de faire, c'est de créer un petit script shell appelé `switchJDK.sh` :

```
sudo update-alternatives --config javac
export JAVA_HOME=$(readlink -f /usr/bin/javac | sed "s/bin\/javac//")
echo "Using JAVA_HOME=$JAVA_HOME"
```

Celui-ci pourra alors être invoqué avec `. switchJDK.sh` ou `source switchJDK.sh`. `update-alternatives` changera le lien symbolique en fonction de la version souhaitée et `$JAVA_HOME` pointera au bon endroit. Notez que nous utilisons un point (ou `source`) pour invoquer le script afin d'utiliser l'environnement du shell courant et non un shell créé pour l'occasion (comme avec `sh switchJDK.sh` ou une ligne `#!/usr/bin/env bash` en début de script), qui ne nous permettrait pas de définir correctement `JAVA_HOME` dans le shell actuel. Cette technique est perfectible puisque, si le `javac` est déjà le bon, la variable d'environnement n'est pas forcément définie. Vous devez donc obligatoirement sourcer ce script à chaque ouverture d'un terminal (ajouter l'export de `JAVA_HOME` dans votre `.bashrc` peut être une solution).

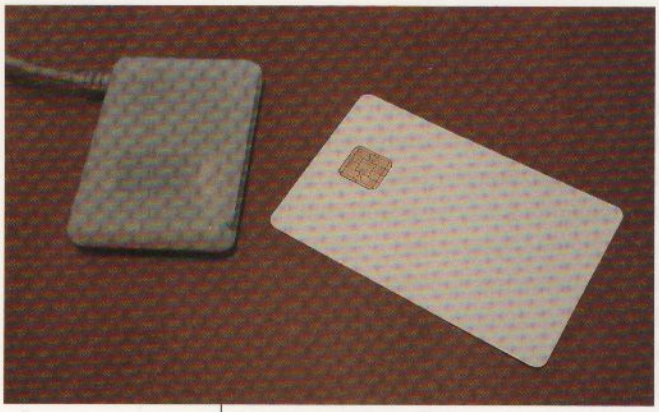
3. « VOUS N'AVEZ PAS DE COMPTE ORACLE ? »

Nous avons deux environnements JDK prêts à l'emploi sur notre système, mais nous ne sommes pas au bout de nos peines. Un environnement de développement est une chose, un SDK adapté à la Java Card en est une autre. Récupérer le *Java Card SDK* est, en principe, assez simple, puisqu'il suffit de pointer son navigateur sur **https://www.oracle.com/java/technologies/javacard-sdk-downloads.html**. Vous trouverez là différentes versions disponibles (incluant la 2.2.2) pour différents systèmes, sous la forme de fichiers ZIP téléchargeables… après création d'un compte utilisateur et connexion. Certains ne se soucient pas de ce genre de choses, mais au-delà des préférences personnelles, je n'ai pas la moindre idée de comment structurer mon environnement pour intégrer un tel SDK, ni même quel système de construction choisir. De plus, je possède une forte allergie aux IDE poussifs consommant une quantité titanesque de mémoire (Eclipse), ce qui est malheureusement un thème récurrent dans le monde Java.

Il existe une solution pour nous simplifier la vie et faire d'une pierre plusieurs coups. Vous souvenez-vous de la page wiki résumant les correspondances JC Kit / JDK ? Celle-ci fait partie du projet/dépôt appelé *ant-javacard* **[6]**, permettant de produire des fichiers `.cap` avec Apache Ant (un système de *build* comme Make pour les projets Java). *ant-javacard* s'intègre à un projet Ant et se chargera même de télécharger les SDK (sous forme de sous-module **https://github.com/martinpaljak/oracle_javacard_sdks**) tout en vérifiant la compatibilité avec le JDK utilisé.

Malheureusement, mes années Java sont loin derrière moi et les choses ont de toute façon grandement évolué. Je n'ai pas réellement envie, de suite, de mettre mon nez dans un système de *build*, mais simplement envie d'écrire rapidement du code pour la Java Card et communiquer avec cette dernière. Si seulement il existait un projet démo comme un « Hello World » pour créer une application Java Card…

J'ai cherché, je n'ai pas trouvé. Mais ce sur quoi j'ai mis la main, en revanche, est un projet relativement conséquent faisant précisément usage de *ant-javacard* : *IsoApplet* **[7]**. Il s'agit d'une application Java Card compatible avec OpenSC, écrite par Philip Wendland et permettant tout un lot de fonctions cryptographiques adaptées pour une infrastructure à clés publiques (PKI), comme la génération de secrets, la signature, le stockage PKCS#15, le déchiffrement, etc. Le code est relativement dense puisqu'on parle ici de quelque 4000 lignes de codes distribuées dans 14 fichiers, ce qui est plus que conséquent pour quelqu'un n'ayant pas touché à Java depuis plus de 15 ans. Mais la structure du projet, elle, est très simple.



Fut un temps, les lecteurs de smartcards étaient difficiles à trouver et relativement coûteux. Aujourd'hui, on en trouve pour une poignée d'euros sur des sites aussi grand public qu'Amazon.

Donc, pour créer notre « *Hello World* » Java Card, tout ce que nous avons à faire est de cloner ce projet, remplacer les sources par quelque chose de simple et modifier sensiblement la configuration (`build.xml`) pour utiliser le bon SDK en version 2.2.2. Comprenez bien que *IsoApplet*, dans l'état, ne pourra pas être compilé avec un SDK 2.2.2 (il existe une branche Git spécifique « main-javacard-v2.2.2 »), nous avons juste l'intention de vampiriser la structure. Obtenir les sources de l'application se fera très simplement avec un `git clone --recursive https://github.com/philipWendland/IsoApplet.git` permettant de récupérer automatiquement le sous-module contenant les SDK (cf. plus haut).

Ceci fait, supprimez purement et simplement tout ce que contient le répertoire `src/`, puis créez une nouvelle arborescence correspondant au nom complet de votre future classe Java. En principe, vous pouvez utiliser n'importe quoi, à partir du moment où vous ne comptez pas distribuer le *package*, sinon vous devrez respecter les conventions Java et utiliser un nom globalement unique, ce qui sous-entend une association à un nom de domaine puisqu'il s'agit d'une notation en *reverse DNS* (si vous avez « exemple.com » alors vous pouvez par exemple utiliser « com.exemple.javacard.Hello » et créer `com/exemple/javacard/HelloWorld/` pour y mettre vos sources Java). Notez qu'il est d'usage de structurer ses sources Java en utilisant le nom des classes implémentées, d'où la correspondance classe/chemin.

Il faut ensuite se pencher sur le fichier `build.xml` qui, en grande partie, est déjà utilisable et ne nécessite que d'adapter la section (ou tâche) `javacard` qui ressemble pour l'instant à ceci :

```
<javacard jckit="ext/sdks/jc310r20210706_kit">
  <cap targetsdk="3.0.4" aid="f2:76:a2:88:bc:fb:a6:9d:34:f3:10"
       output="IsoApplet.cap" sources="src" version="1.0">
    <applet class="xyz.wendland.javacard.pki.isoapplet.IsoApplet"
       aid="f2:76:a2:88:bc:fb:a6:9d:34:f3:10:01"/>
  </cap>
</javacard>
```

`jckit` précise le chemin vers le SDK à utiliser, pour nous ce sera `ext/sdks/jc222_kit`. Le *tag* `cap` permet de spécifier des informations pour chaque fichier `.cap` produit avec `targetsdk` afin éventuellement de spécifier un SDK différent. Avec le fichier d'origine, le SDK 3.1.0 supporte plusieurs versions et 3.0.4 est donc précisé explicitement. Dans notre cas, avec 2.2.2, c'est le même chemin lui-même qui sera utilisé et nous n'avons rien à spécifier, ce paramètre disparaît.

Vient ensuite l'AID ou *Application IDentifier* qui, comme l'acronyme l'indique, est un identifiant d'application. Celui-ci se compose de 5 à 16 octets répartis en un identifiant fournisseur (RID pour *Resource IDentifier*) sur 5 octets, suivi d'une extension d'identification

d'application (PIX pour *Proprietary Identifier eXtension*) de 0 à 11 octets. Encore une fois, si vous ne distribuez/ne commercialisez pas le résultat de votre travail, cet identifiant importe peu. Dans le cas contraire, en revanche, c'est l'ISO qui assigne un RID spécifique à chaque entreprise, qui elle gère ensuite en interne l'attribution des PIX à ses applications.

Ce qui est important ici, c'est la différentiation entre l'AID du *package* contenant l'application et l'AID de l'*applet* elle-même, puisqu'un package peut contenir plusieurs applications. Généralement et comme avec *IsoApplet*, les deux AID partagent forcément le même RID et le PIX diffère par le dernier octet (qu'il soit ajouté ou changé). Je n'ai pas trouvé de registre public des RID, ou éventuellement un RID standard pour les phases de test, et j'ai donc décidé de réutiliser celui de l'application *IsoApplet* (`f2:76:a2:88:bc`) en complétant avec le PIX `de:ad:be:ef` pour le package et `:de:ad:be:ef:01` pour l'application.

`output`, `sources` et `version` désignent respectivement le nom de fichier à créer (`Hello.cap`), le répertoire source et la version du package. On arrive ensuite au *tag* `applet`, en précisant le nom de la classe où la méthode `install()` est définie (cf. le code ci-après) avec `class` et l'AID de l'application avec `aid`).

Cette partie de notre `build.xml` ainsi modifiée ressemble maintenant à ceci :

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

Tout est prêt, nous pouvons enfin faire un peu de code !

4. ÉCRIVONS NOTRE APPLICATION

Avant de nous lancer à corps perdu dans le code, nous devons savoir ce qu'il nous faut implémenter et donc établir un mini cahier des charges pour notre application. Un exemple simpliste consistera à attendre une commande reçue sous la forme d'un APDU et, si c'est bien celle espérée, retourner une série de données formant notre *Hello World* (« Coucou Monde ! » plus exactement).

Un APDU (ou peut-être « une » ?) est un message dont le format est normalisé (ISO 7816-4) constituant le dialecte standard entre une carte (ou un *tag*) et une application cliente utilisant le lecteur de carte. Il est constitué d'octets spécifiant respectivement une classe de commandes (CLA), une instruction (INS), deux paramètres (P1 et P2), 0, 1 ou 3 octets pour la taille des données (Lc) accompagnant éventuellement la commande, les données elles-mêmes et, finalement 0, 1, 2 ou 3 octets spécifiant la taille des données attendues dans la réponse (Le). Ce type d'APDU, envoyé de l'application client vers la carte est désignée par le terme APDU-C (pour *Command*).

Certains lecteurs sont parfaitement adaptés pour un usage nomade. Une fois plié, ce modèle SCR3500 ne prendra pas plus de place qu'une clé USB, tout en offrant les mêmes fonctionnalités qu'un modèle standard.

La réponse de la carte prend également la forme d'un APDU. C'est l'APDU-R (*Response*), avec une structure plus simple, puisqu'elle se compose de 0 à *n* octets de données, suivi d'un code d'état sur deux octets, nommés SW1 et SW2. Un certain nombre de ces codes sont normalisés, `0x90 0x00` indiquant par exemple une opération réussie, ou encore `0x6D 0x00` pour une instruction non valide ou non supportée.

Un *Hello World* simple consisterait donc à réagir à l'arrivée d'un APDU-C comme `D040000` (en hexadécimal ici et dans la suite), classe `D0`, instruction `40`, les deux paramètres à zéro et aucune donnée (donc aucune taille Lc). Ce faisant, l'application pourra retourner les octets composant la chaîne `"Coucou Monde!"` suivie de `9000`.

Nous allons faire cela, mais nous allons également ajouter trois autres instructions, `50`, `51` et `52` permettant respectivement de lire la valeur d'une donnée de 8 bits stockée dans la carte, d'incrémenter cette valeur et de la décrémenter. Imaginez cela comme un système de points de fidélité simpliste et pas du tout sécurisé. Ces nouvelles instructions, aux valeurs choisies tout aussi arbitrairement que la précédente, font également partie de la classe `D0`, définie tout aussi arbitrairement.

Il est temps de parler de code. Pour écrire une application pour la Java Card, nous utiliserons, bien entendu, Java, qui est un langage orienté objet reposant sur l'utilisation de classes et de méthodes. Dans ce contexte, une application Java Card, ou *applet*, étend la superclasse `javacard.framework.Applet` sur laquelle repose toutes les applications de la carte. La structure de base du code, stockée dans `src/dev/drrb/javacard/Hello/Hello.java`, sera donc la suivante et tout le reste se trouvera dans sa portée :

```
package dev.drrb.javacard.Hello;

import javacard.framework.APDU;
import javacard.framework.Applet;
import javacard.framework.ISO7816;
import javacard.framework.ISOException;
import javacard.framework.Util;

public class Hello extends Applet {
}
```

Les `import` que vous voyez là sont des *packages* Java et fonctionnent un peu comme des modules Python ou des `include` du C. Ils permettent de rendre disponibles des fonctionnalités pour notre code, qui lui-même est un *package* tel que défini à la première ligne. Comme expliqué, notre classe `Hello` est un `Applet` et nous devons maintenant implémenter un certain nombre de méthodes (fonctions) qui seront utilisées par l'environnement d'exécution. Mais, avant cela, nous avons besoin de déclarer quelques variables pour nous simplifier la vie (et les évolutions futures du code) :

```
// "Coucou Monde!"//
private static final byte[] helloWorld = {
    (byte)'C',(byte)'o',(byte)'u',(byte)'c',(byte)'o',(byte)'u',
    (byte)' ',(byte)'M',(byte)'o',(byte)'n',(byte)'d',(byte)'e',(byte)'!'
};

// Notre classe APDU-C
private static final byte HELLO_CLA = (byte)0xd0;

// Instruction APDU-C
private static final byte HELLO_INS = (byte)0x40;
private static final byte HELLOREAD_INS = (byte)0x50;
private static final byte HELLOINC_INS = (byte)0x51;
private static final byte HELLODEC_INS = (byte)0x52;

// La variable pour stocker le compteur
private byte valeur;

// Constructeur
private Hello() {
    valeur = 0;
    register();
}
```

Notez premièrement que pour éviter des erreurs quant à l'interprétation des valeurs littérales, il est nécessaire de *caster* ces dernières. À l'exception de `char` (qui n'est pas disponible ici), il n'existe pas de type non signé en Java. Ainsi, `0xf6` peut être soit une valeur négative sur 8 bits soit une valeur positive sur 16, et le compilateur n'a aucune idée de ce que nous voulons réellement. Le `(byte)` placé devant chaque valeur hexadécimale est précisément là pour expliciter la chose. Ensuite, si vous n'êtes pas coutumier de programmation orientée objet, les dernières lignes doivent vous paraître bien étranges. Ceci est un constructeur, appelé automatiquement lorsque la classe est instanciée ou, en termes plus clairs pour un amateur de programmation procédurale, lorsque la classe, qui n'est rien d'autre qu'un « moule », est utilisée pour créer un objet (avec `new`), ce code spécifique est invoqué pour initialiser la variable `valeur` et appeler la méthode `register()`. Cette dernière sert à enregistrer l'instance de l'application auprès de l'environnement d'exécution (JCRE) qui lui appliquera son AID par défaut.

En parlant d'instanciation justement, une méthode spécifique doit être implémentée pour que celle-ci puisse être invoquée par JCRE au moment de l'installation de l'application :

```
public static void install(byte[] bArray, short bOffset, byte bLength) {
    Hello h = new Hello();
}
```

On retrouve ici le new classique de Java qui déclenchera l'invocation du constructeur. Ce faisant, valeur sera donc mis à zéro juste après l'installation de l'application sur la carte.

Vient ensuite le traitement des APDU-C avec l'implémentation de la méthode process() invoqué par le JCRE lors de l'arrivée d'APDU de commandes :

```
public void process(APDU apdu) {

    if (selectingApplet()) {
        return;
    }

    byte[] buffer = apdu.getBuffer();
    byte CLA = (byte)(buffer[ISO7816.OFFSET_CLA] & 0xFF);
    byte INS = (byte)(buffer[ISO7816.OFFSET_INS] & 0xFF);

    if (CLA != HELLO_CLA) {
        ISOException.throwIt(ISO7816.SW_CLA_NOT_SUPPORTED);
    }

    switch (INS) {
        case HELLO_INS:
            sendHelloWorld(apdu);
            break;
        case HELLOREAD_INS:
            buffer[0] = valeur;
            apdu.setOutgoingAndSend((short)0, (short)1);
            break;
        case HELLOINC_INS:
            incrementer(apdu);
            break;
        case HELLODEC_INS:
            decrementer(apdu);
            break;
        default:
            ISOException.throwIt(ISO7816.SW_INS_NOT_SUPPORTED);
    }
}
```

Le switch/case est relativement simple et parle de lui-même, je pense, mais un certain nombre de points nécessitent des explications. Tout d'abord, l'utilisation de selectingApplet() est une facilité nous permettant de réagir lorsque l'application est sélectionnée. En effet, une Java

Card peut contenir plus d'une application et le client, côté PC par exemple, utilise un APDU spécifique (`00a40400`) permettant de désigner à laquelle il souhaite parler. Cette fonction retourne *VRAI* si nous venons de recevoir un tel APDU. Le simple `return` qui s'en suit provoque, en réponse, un APDU sans donnée avec l'état `9000` signifiant « j'accepte d'être sélectionné ».

Nous avons ensuite `apdu.getBuffer()` qui nous permet d'obtenir un tableau d'octets correspondant au contenu du *buffer* APDU, c'est-à-dire les octets qui le composent. Nous pouvons alors récupérer les octets de classe et d'instruction, respectivement dans `CLA` et `INS`. `ISO7816.OFFSET_CLA` et `ISO7816.OFFSET_INS` désignent les emplacements de ces octets dans le *buffer* et nous évitent de spécifier manuellement le décalage (*offset*) par rapport au début du tableau.

Et enfin, nous devons parler de `ISOException.throwIt()`. Cette méthode permet de déclencher une exception afin que le JCRE fournisse automatiquement un APDU de réponse avec l'état spécifié en argument. Ici, nous avons respectivement la valeur d'état pour une erreur correspondant à une classe ou une instruction non supportée par notre application.

Comme vous pouvez le voir, le `switch`/`case` sert à dispatcher les demandes et nous devons maintenant implémenter les fonctions appelées. Commençons donc par notre *Hello World* de base :

Il est parfois possible d'obtenir un lecteur à prix réduit, comme ici avec l'abonnement TER régional, afin de recharger facilement son titre de transport. Bien entendu, le matériel étant parfaitement standard (CCID), il fonctionnera sans le moindre problème avec une Java Card.

```
private void sendHelloWorld(APDU apdu) {
    byte[] buffer = apdu.getBuffer();
    short length = (short)helloWorld.length;

    Util.arrayCopyNonAtomic(helloWorld, (short)0, buffer, (short)0, length);
    apdu.setOutgoingAndSend((short)0, length);
}
```

Rien de bien extraordinaire, nous récupérons le *buffer* APDU et nous y copions simplement le contenu du tableau `helloWorld[]` avec `arrayCopyNonAtomic()`, avant d'envoyer la réponse avec `setOutgoingAndSend()`. Notez que cette méthode est une version « courte », mais que nous pourrions également utiliser :

```
apdu.setOutgoing();
apdu.setOutgoingLength(length);
apdu.sendBytes((short)0, length);
```

Les `(short)0` sont des valeurs de décalage inutilisées ici puisque notre réponse est constituée de l'ensemble des données brutes, mais ceci peut s'avérer utile, par exemple, pour envoyer des segments spécifiques d'une grosse masse de données. Le code d'état placé dans l'APDU-R est automatiquement ajouté pour nous. Une variation de cette méthode est celle concernant le solde de notre petit compteur, avec cette fois un unique octet en réponse :

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

Et enfin, nous avons les deux fonctions permettant d'incrémenter et de décrémenter notre compteur :

```
private void incrementer(APDU apdu) {
    if (valeur < 0xff)
        valeur++;
}

private void decrementer(APDU apdu) {
    if (valeur > 0)
        valeur--;
}
```

Ceci clôt le code de notre petite application et nous pouvons passer au plus amusant : l'installation dans la Java Card et le test. Compiler le code et obtenir l'application sous forme de fichier `.cap` se résume à utiliser Ant (paquet `ant` sous Debian) avec :

```
$ ant
Buildfile: /home/denis/HUB/javacard222_hello/build.xml
      [get] Destination already exists (skipping): /home/denis/HUB/javacard222_hello/
ant-javacard.jar
dist:
      [cap] INFO: using JavaCard 2.2.2 SDK in /home/denis/HUB/javacard222_hello/ext/
sdks/jc222_kit with JDK 8
      [cap] INFO: Setting package name to dev.drrb.javacard.Hello
      [cap] Building CAP with 1 applet from package dev.drrb.javacard.Hello (AID:
F276A288BCDEADBEEF)
      [cap] dev.drrb.javacard.Hello.Hello F276A288BCDEADBEEF01
  [compile] Compiling files from /home/denis/HUB/javacard222_hello/src
  [compile] Compiling 1 source file to /tmp/jccpro6752309262451817856
  [compile] /home/denis/HUB/javacard222_hello/src/dev/drrb/javacard/Hello/Hello.java
   [verify] Verification passed
      [cap] CAP saved to /home/denis/HUB/javacard222_hello/Hello.cap

BUILD SUCCESSFUL
Total time: 0 seconds
```

5. INSTALLONS ET TESTONS AVEC GLOBALPLATFORMPRO

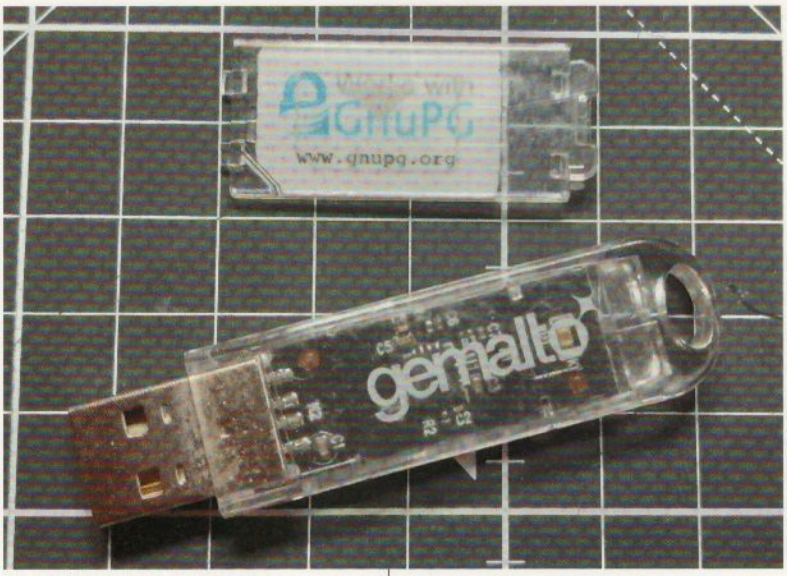
Certaines technologies de smartcard sortent de l'ordinaire et peuvent poser problème. Ici, ce lecteur est totalement passif et se contente d'établir une simple connexion entre la carte au format SIM et le port USB. Ce matériel sera totalement inutilisable avec une Java Card puisqu'il ne s'agit pas réellement d'un lecteur, mais plutôt d'un adaptateur, l'USB étant géré directement par la puce de la carte.

Pour accéder à notre carte et la programmer, nous avons besoin d'un périphérique dédié. Ce type de produit est relativement facile à trouver et ne vous coûtera qu'entre 12 € et 20 €. Il s'agit généralement de périphériques USB relativement simples, mais on trouve également des lecteurs directement intégrés à certains claviers. Il est même possible parfois d'en obtenir à prix réduit avec des abonnements de train ou d'autres services similaires (pour pouvoir « recharger » son titre de transport depuis chez soi via le Web).

Ce type de matériel est généralement automatiquement pris en charge par n'importe quel système d'exploitation, mais nécessitera l'utilisation de ce qu'on appelle un *middleware* dont le rôle est de fournir aux applications de la machine (PC ou Mac) une interface unique pour communiquer avec tout type de lecteurs. Sous GNU/Linux, ce *middleware* est PC/SC et prend la forme d'un service `pcscd`, accompagné d'un outil de test fort pratique : `pcsc_scan` (paquet `pcsc-tools` sous Debian et consorts). Celui-ci vous montrera immédiatement si votre lecteur fonctionne correctement :

```
$ pcsc_scan
PC/SC device scanner
V 1.6.2 (c) 2001-2022, Ludovic Rousseau <ludovic.rousseau@free.fr>
Using reader plug'n play mechanism
Scanning present readers...
0: SCM Microsystems Inc. SCR 335 [CCID Interface] (21120617208509) 00 00

Fri Oct 20 18:28:15 2023
Reader 0: SCM Microsystems Inc. SCR 335 [CCID Interface] (21120617208509) 00 00
  Event number: 0
  Card state: Card removed,
```

L'insertion d'une carte, et pas nécessairement d'une Java Card, vous affichera différentes informations intéressantes et même une estimation de son modèle et de son identité/utilité. Si tel est le cas, tout va bien, et vous pouvez passer à l'étape suivante. Dans le cas contraire, soit le matériel n'est pas reconnu, soit vous devrez installer un pilote pour PC/SC. Recherchez les paquets contenant « PC/SC driver » dans



Le format de la smartcard n'a pas réellement d'importance pour le lecteur, ceci est une simple limitation mécanique. Ce lecteur Gemalto USB Shell Token V2 accepte des cartes au format mini-SIM 2FF (ce qu'on appelle généralement une « carte SIM »), mais la connectique de la puce est strictement identique à celle qui est présente sur une carte (1FF) au format 85,6 mm par 53,98 mm.

leur descriptif et installez-les (les lecteurs GemPlus nécessitent généralement ces pilotes). Mais normalement, vous ne devriez pas rencontrer de problème puisque la plupart du matériel neuf qu'on trouve actuellement utilise le protocole standard CCID (*Chip Card Interface Device*) parfaitement pris en charge par défaut par PC/SC.

L'utilisation d'un *middleware* simplifie grandement les choses puisque toutes les applications capables de l'utiliser sont automatiquement compatibles avec votre lecteur, comme elles ne dialoguent pas directement avec le matériel. Il en existe plusieurs permettant de gérer les applications d'une Java Card, mais la plus utilisée est sans le moindre doute GlobalPlatformPro de Martin Paljak (oui, le même que celui de *ant-javacard*). Il s'agit d'une application Java et deux options s'offrent à vous, soit vous téléchargez le JAR (`gp.jar`) depuis le dépôt GitHub de Martin **[8]**, soit vous recompilez le tout après avoir cloné le dépôt. La procédure est simple et décrite dans le `README.md` (n'oubliez pas de rebasculer sur le JDK 17).

On pourra invoquer directement l'application avec `java -jar gp.jar` ou copier le JAR quelque part et créer un alias `gp` directement depuis votre `.bashrc` (`alias gp='java -jar /home/utilisateur/chemin/gp.jar'`). Pour tester, rien de plus simple, il suffit de lister le contenu de la Java Card dans le lecteur :

```
$ gp -l
# Warning: no keys given, defaulting to 404142434445464748494A4B4C4D4E4F
ISD: A000000003000000 (OP_READY)
  Privs: SecurityDomain, CardLock, CardTerminate, CardReset,
CVMManagement

PKG: A0000000035350 (LOADED)
  Applet:   A000000003535041
```

L'avertissement affiché précise que la clé par défaut est utilisée, c'est celle configurée dans une carte « vierge » et qui permet éventuellement de restreindre l'accès en gestion de la carte si elle est modifiée. En la changeant et en la perdant, vous pourrez toujours utiliser les applications qui se trouvent sur la carte, mais vous n'aurez plus aucun moyen

de gérer son contenu. Il faut donc faire très attention ! Par ailleurs, si l'avertissement concernant la clé vous perturbe, vous pouvez spécifier cette dernière explicitement en argument, en ajoutant l'option `-k 404142434445464748494A4B4C4D4E4F`, ce qui supprime le message.

En parlant de faire attention justement, vous remarquerez que la carte n'est pas si vierge que ça. Un package `A0000000035350` est installé, fournissant l'application `A000000003535041`. **IL NE FAUT PAS SUPPRIMER CE PACKAGE !** Contrairement à d'autres modèles de cartes où le coprocesseur cryptographique gère le stockage des clés en NVRAM, celle-ci est livrée avec une application qui implémente le stockage sécurisé. Si vous supprimez cette application, vous perdrez une partie des fonctionnalités de la carte et il ne semble pas y avoir de fichier `.cap` téléchargeable permettant de restaurer l'état initial (ne me demandez pas comment je le sais... deux fois de suite). Faites donc excessivement attention aux AID que vous utilisez ou copiez/collez, une erreur de manipulation peut gâcher significativement votre journée.

La première ligne qui apparaît dans la sortie fait mention de l'ISD ou *Issuer Security Domain*. C'est une application spécifique disposant de privilèges particuliers gérant les clés de sécurité ainsi que les opérations de gestion des applications. C'est avec cette application que GlobalPlatformPro communique pour installer ou supprimer des applications et *packages* sur la carte.

Pour installer notre application toute neuve, il suffit d'utiliser `--install` accompagné du fichier `.cap` à utiliser :

```
$ gp -k 404142434445464748494A4B4C4D4E4F --install Hello.cap
Hello.cap loaded: dev.drrb.javacard.Hello F276A288BCDEADBEEF
```

Une nouvelle utilisation de `-l` nous confirme l'installation de notre bébé :

```
$ gp -k 404142434445464748494A4B4C4D4E4F -l
ISD: A000000003000000 (OP_READY)
     Privs:  SecurityDomain, CardLock, CardTerminate,
             CardReset, CVMManagement

APP: F276A288BCDEADBEEF01 (SELECTABLE)

PKG: A0000000035350 (LOADED)
     Applet:   A000000003535041

PKG: F276A288BCDEADBEEF (LOADED)
     Applet:   F276A288BCDEADBEEF01
```

Il ne nous reste plus qu'à tester notre code et voir si notre application répond comme il se doit à nos APDU. Pour ce faire, nous pourrions utiliser GlobalPlatformPro, mais une solution plus interactive existe via le script Perl `scriptor` fourni par `pcsc-tools`. Tout ce que nous avons à faire est d'exécuter le script et saisir nos APDU-C sous la forme de valeurs hexadécimales, à commencer par la sélection de l'application :

```
$ scriptor
No reader given: using SCM Microsystems Inc. SCR 335
  [CCID Interface] (21120617208509) 00 00
Using T=1 protocol
Reading commands from STDIN
00a404000af276a288bcdeadbeef01
> 00 a4 04 00 0a f2 76 a2 88 bc de ad be ef 01
< 90 00 : Normal processing.
```

L'APDU `00a404000af276a288bcdeadbeef01` se décompose ainsi :

- `00a4` : classe `00`, instruction `a4` ;
- `0400` : P1 à `04` et P2 à `00` ;
- `0a` : la taille des données envoyées, 10 octets d'AID ;
- `f276a288bcdeadbeef01` : les données, ici constituées de l'AID de l'application à sélectionner.

La réponse `90 00` confirme le bon fonctionnement de l'opération et nous sommes maintenant en mesure de dialoguer avec notre application. Demandons l'état du compteur, incrémentons trois fois, décrémentons et consultons à nouveau le compteur :

```
D0500000
> D0 50 00 00
< 00 90 00 : Normal processing.
D0510000
> D0 51 00 00
< 90 00 : Normal processing.
D0510000
> D0 51 00 00
< 90 00 : Normal processing.
D0510000
> D0 51 00 00
< 90 00 : Normal processing.
D0520000
> D0 52 00 00
< 90 00 : Normal processing.
D0500000
> D0 50 00 00
< 02 90 00 : Normal processing.
```

Tout fonctionne à la perfection. Enfin, utilisons l'APDU-C *Hello World* en guise de dernière confirmation :

```
D0400000
> D0 40 00 00
< 43 6F 75 63 6F 75 20 4D 6F 6E 64 65 21 90 00 : Normal processing.
```

La réponse obtenue, amputée du `90 00` de fin, peut être convertie avec `xxd` pour obtenir une chaîne ASCII :

```
$ echo 436F75636F75204D6F6E646521 | xxd -r -p
Coucou Monde!
```

Victoire ! Notre application est pleinement fonctionnelle. Bravo, vous venez d'écrire votre première application Java Card ! Le même test avec GlobalPlatformPro peut être effectué et nous pouvons même combiner le tout avec la conversion hexa/ASCII ainsi :

```
$ gp -k 404142434445464748494A4B4C4D4E4F \
--applet F276A288BCDEADBEEF01 -a D0400000 | xxd -r -p
Coucou Monde!
```

Pour continuer le développement de notre application, il suffit de faire évoluer le code, le recompiler et de mettre à jour l'application en supprimant l'application, puis le *package*, de la carte (`gp --delete`) pour ensuite installer la nouvelle version (`gp --install`). En guise de bonus et pour écarter tout risque d'effacer le *package* de gestion de stockage sécurisé pré-installé par le constructeur, nous pouvons modifier le fichier `build.xml` de Ant en ajoutant deux cibles :

```
<target name="loadcard" description="load applet" depends="dist">
  <exec executable="./loadcard.sh"/>
</target>
<target name="updatecard" description="update applet" depends="dist">
  <exec executable="./updatecard.sh"/>
</target>
```

Les scripts shell en question sont respectivement :

```
#!/usr/bin/env sh
KEY="404142434445464748494A4B4C4D4E4F"
echo ">> Install Applet"
java -jar /home/denis/bin/gp.jar -k $KEY --install Hello.cap
echo ">> List"
java -jar /home/denis/bin/gp.jar -k $KEY -l
```

et :

```
#!/usr/bin/env sh
KEY="404142434445464748494A4B4C4D4E4F"
echo ">> Delete Applet"
java -jar /home/denis/bin/gp.jar -k $KEY --delete F276A288BCDEADBEEF01
echo ">> Delete Package"
```

```
java -jar /home/denis/bin/gp.jar -k $KEY --delete F276A288BCDEADBEEF
echo ">> List"
java -jar /home/denis/bin/gp.jar -k $KEY -l
echo ">> Install Applet"
java -jar /home/denis/bin/gp.jar -k $KEY --install Hello.cap
echo ">> List"
java -jar /home/denis/bin/gp.jar -k $KEY -l
```

Il suffira alors d'utiliser `ant loadcard` ou `ant updatecard` pour automatiquement compiler, convertir et « flasher » l'application sur la carte, sans risquer une mauvaise manipulation impliquant la suppression malheureuse de `A0000000035350`.

6. POUR CONCLURE

Je n'ai jamais vraiment aimé Java (ou la POO en général), mais je dois dire que le fait d'écrire un programme faisant exactement ce que je veux et fonctionnant à l'intérieur d'une carte à puce est une excellente motivation. Il y a quelque chose de magique dans le fait de voir la carte répondre à nos APDU et de pouvoir se dire que c'est notre code, exécuté dans ce minuscule composant, qui en est responsable. Quelque part, cela va au-delà de la programmation sur microcontrôleur, sans doute parce que le monde des cartes à puce semble, de but en blanc, mystérieux et inaccessible.

Ceci n'est qu'un petit avant-goût de ce qu'il est possible de faire avec ce type de plateforme, et ceci en utilisant une technologie relativement ancienne impliquant l'utilisation d'un JDK et d'un SDK en fin de vie. Cependant, pour un budget relativement modeste, il est possible de s'initier sans trop de difficulté à cet univers et d'obtenir rapidement des résultats satisfaisants. Pour continuer l'aventure, on pourra s'inspirer des créations d'autres développeurs, impliquant l'utilisation de fonctionnalités plus avancées comme toute la partie cryptographique intégrée à ces plateformes. Et pour cela, nous avons à disposition la ressource idéale : une sélection d'applications Java Card maintenue par le CRoCS (*Centre for Research on Cryptography and Security*). Là **[9]**, vous trouverez une longue liste d'applications, classée et triée par thème, où chaque projet est décrit et accompagné d'un lien vers ses sources.

Comme pour les lecteurs de cartes à puce au format standard ISO/IEC 7810 ID-1, les produits acceptant des mini-SIM existent en différents modèles de différents constructeurs. La seule chose importante à l'achat est de toujours s'assurer que le matériel est compatible avec le protocole CCID.

Applet e-ID, *Token* FIDO U2F, *crypto-wallet*, système d'authentification, signature électronique... tout est listé et accessible. De quoi s'inspirer pour ses codes et gagner en expérience, dans la plus pure tradition *open source*.

Pour ma part, prenant goût au domaine et frustré des limitations cryptographiques de mes vieilles J2A081, en particulier au niveau des signatures et HMAC, j'ai fini par acheter trois exemplaires de la carte J3R150 dont je parlais en début d'article **[1]**. Non seulement celle-ci est bien plus capable que les antiquités en ma possession, mais elle me permettra également d'explorer pleinement le code de *IsoApplet* et d'autres projets incompatibles avec les spécifications Java Card 2.2.2. De plus, l'aspect *dual-interface*, avec d'une part l'utilisation d'un lecteur de cartes et de l'autre le NFC ISO/IEC 14443 Type A, attise grandement ma curiosité. Sans oublier, bien sûr, le fait de ne plus utiliser que le JDK 17 et donc ne plus avoir à basculer en permanence vers un JDK 8. Il est donc possible que nous reparlions de ce sujet dans un prochain article, avec un projet plus pratique qui, pour l'heure, reste encore à déterminer. Nous verrons...

Note de dernière minute : je viens de découvrir que la société française Hitools Access propose des Java Card 3.0.4 de type ACOSJ d'ACS via sa boutique en ligne **[10]**, et ce, à un prix très intéressant (entre 5,70 € et 7,14 € TTC) qu'il s'agisse de carte à contact, sans contact ou double interface. Je ne saurais me prononcer sur la qualité de ces produits (je viens juste de passer commande et n'ai pas expérimenté), mais avec un prix presque trois fois moindre que celui d'une J3R150 d'AliExpress, et avec une livraison nationale, je pense que c'est une information digne d'être partagée. Un autre revendeur potentiel est l'allemand *cardomatic.de* proposant des lots de 5 cartes ACOSJ entre 30 € **[11]** et 34 € **[12]** hors taxe (plus port). **DB**

RÉFÉRENCES

[1] https://fr.aliexpress.com/item/1005005364667733.html

[2] https://fr.wikipedia.org/wiki/Java_Card

[3] https://lig-membres.imag.fr/donsez/cours/javacard.pdfet

[4] https://javaalmanac.io/bytecode/versions/

[5] https://github.com/martinpaljak/ant-javacard/wiki/JavaCard-SDK-and-JDK-version-compatibility

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

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

[8] https://github.com/martinpaljak/GlobalPlatformPro/releases

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

[10] https://www.hitools-access.com

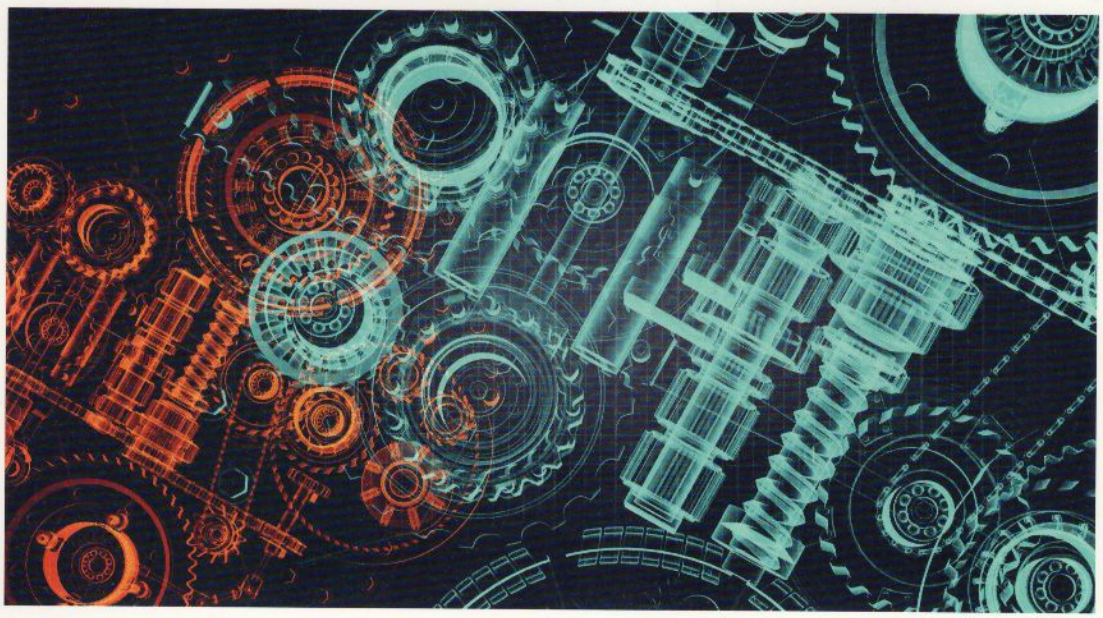
[11] https://www.cardomatic.de/en/p/acosj-di-java-card-pack-of-5

[12] https://www.cardomatic.de/en/p/acosj-di-95k-java-card-pack-of-5

MODÉLISEZ ET SIMULEZ TOUS VOS SYSTÈMES AVEC OPENMODELICA

Laurent DELMAS
Docteur ingénieur généraliste spécialisé en électronique embarquée et vision par ordinateur

Actuellement, les systèmes sont de plus en plus complexes dans le sens multiphysique et nécessitent donc des outils adaptés qui permettent de prendre en compte cet aspect multiphysique. De plus, l'approche systémique ou ingénierie des systèmes définit un système comme l'interaction de plusieurs composants. Pour répondre à ce besoin a été créé en 1995 le langage Modelica dont l'objectif est de modéliser et de simuler des systèmes dynamiques hybrides. Par hybride est entendue la combinaison de systèmes continus et discrets.



– Modélisez et simulez tous vos systèmes avec OpenModelica –

Plusieurs articles ayant abordé les thèmes de modélisation et simulation, tâches quotidiennes de nombreux ingénieurs et chercheurs, ont par le passé été publiés dans ces pages : simulation électronique, simulation mécanique... **[1-7]**. À chaque fois, il s'agissait d'un seul domaine physique : mécanique, électronique... Dans cet article, nous allons découvrir et mettre en œuvre OpenModelica, un outil *open source* développé par Open Source Modelica Consortium (OSMC). La première partie de cet article consiste en une prise en main au travers d'un exemple simple de suspension. La seconde partie, quant à elle, se focalisera essentiellement sur l'aspect multiphysique d'un problème, à savoir, la chaîne de traction d'un véhicule électrique.

1. PRÉSENTATION D'OPENMODELICA

Modelica est un langage de modélisation orienté objet, basé sur les équations mathématiques pour la modélisation de systèmes multiphysiques **[8]**. C'est un langage moderne qui s'appuie de facto sur une modélisation acausale et emploie un typage statique. Les outils qui intègrent ce langage permettent de construire des modèles graphiquement ou directement par code. Dans la première partie de cet article, nous décrirons nos modèles en mode texte directement à partir des équations du système et utiliserons OMShell pour simuler et exploiter les résultats. Dans la seconde partie, nous définirons un système complet avec l'outil graphique OMEdit et la librairie standard Modelica en version 4.0.0.

De nombreux logiciels commerciaux intègrent Modelica tels que AMESim, Dymola, MatLab, MapleSoft, etc., cependant il existe une alternative à ces derniers qu'est OpenModelica, un environnement de modélisation et de simulation basé sur le langage Modelica et qui s'appuie exclusivement sur des logiciels libres.

OpenModelica est disponible tant pour une utilisation commerciale que non-commerciale et est distribué sous licence OSMC Public Licence. L'Open Source Modelica Consortium (OSMC) est une organisation non

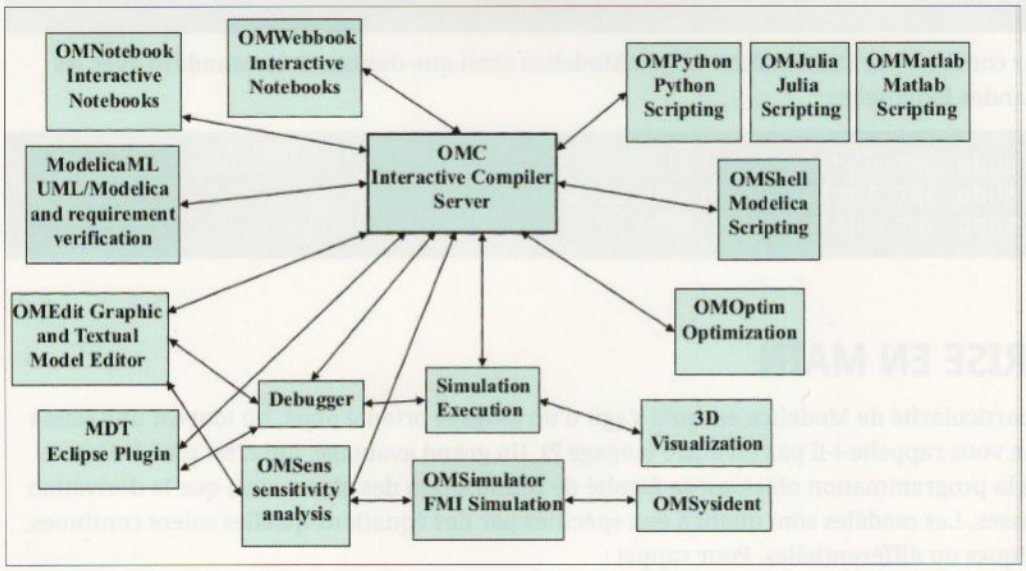


Figure 1 : Architecture de l'environnement OpenModelica **[9]**.

gouvernementale et à but non lucratif, créé en 2007 afin de promouvoir le langage Modelica. Actuellement, de nombreuses entreprises et universités font partie de cette organisation : Bosch Rexroth, ABB, VTI Institute, Politecnico di Milano...

OpenModelica intègre tout un ensemble d'outils : des éditeurs en versions console et graphique, respectivement les applications OMShell, OMEdit, un compilateur, un débogueur, un analyseur de performance, un optimiseur, un *notebook* électronique pour l'apprentissage (OMNotebook) et bien d'autres comme le montre la figure 1.

À cela s'ajoutent de nombreuses librairies de composants Modelica utilisables avec OpenModelica : Modelica Standard Library (3.2.3 et 4.0.0), OpenHydraulics, PhotoVoltaics, PlanardMechanics, SystemDynamics, VehicleInterface... Nous verrons comment construire une librairie dans la suite de l'article.

2. INSTALLATION

Plusieurs versions sont disponibles : *Official Release*, *Stable Development* et *Nightly Build*. Tout au long de cet article, nous allons utiliser la version *Stable*. Commençons par ajouter la clé du dépôt officiel :

```
$ sudo apt update
$ sudo apt install ca-certificates curl gnupg
$ curl -fsSL http://build.openmodelica.org/apt/openmodelica.asc | sudo gpg
--dearmor -o /usr/share/keyrings/openmodelica-keyring.gpg
```

Puis les dépôts eux-mêmes :

```
$ echo "deb [arch=amd64 signed-by=/usr/share/keyrings/openmodelica-keyring.
gpg] http://build.openmodelica.org/apt jammy stable" | sudo tee /etc/apt/
sources.list.d/openmodelica.list
```

Pour conclure par l'installation d'OpenModelica ainsi que des librairies standard avec les commandes habituelles :

```
$ sudo apt update
$ sudo apt install openmodelica
$ sudo apt install omlibrary
```

3. PRISE EN MAIN

La particularité de Modelica est qu'il s'agit d'un langage orienté objet, où tout est une classe (cela ne vous rappelle-t-il pas un autre langage ?). Un grand avantage, qui n'est plus à démontrer de la programmation objet, est sa faculté de réutilisation des objets ainsi que la dérivation des classes. Les modèles sont quant à eux spécifiés par des équations qu'elles soient continues, algébriques ou différentielles. Pour rappel :

- une équation différentielle représente l'évolution d'un système uniquement en fonction du temps et de ses dérivées, par exemple l'évolution de la tension aux bornes d'un condensateur (circuit RC)...
- une équation algébrique traduit une contrainte géométrique telle que l'équation d'un cercle : $x^2+y^2=L^2$.
- une équation aux dérivées partielles exprime un système dont les solutions dépendent des variables d'espace et du temps : équation de transport, équation de Laplace, équation de Poisson...

Nous allons découvrir et prendre en main Modelica au travers de quelques exercices très simples dans un premier temps, pour ensuite terminer par un système plus complexe tel qu'une chaîne de traction de véhicule électrique.

Comme mentionné précédemment, tout est objet avec Modelica. Depuis la version 3, un objet ou un modèle se déclare indifféremment avec le mot-clé `class` ou `model`. D'autres formes particulières de déclaration d'objets existent : `record`, `func`, `connector`, `package`... et apportent des fonctionnalités et/ou contraintes spécifiques. Par exemple, un objet `record` ne contient pas d'équation. Nous verrons cela un peu plus tard.

4. PREMIER EXEMPLE : SUSPENSION DE VÉHICULE

4.1 Modélisation de la suspension

Pour notre premier exemple, nous allons modéliser et simuler la suspension d'un véhicule. Nous aurions également pu étudier la tenue de route dudit véhicule, c'est-à-dire la répartition des efforts sur chaque roue en fonction de la trajectoire, le profil de vitesse, etc. Cela étant décrit par la deuxième loi de Newton, autrement dit le principe fondamental de la dynamique. Dans notre analyse de la suspension, nous nous focalisons uniquement sur une seule roue du véhicule. La figure 2 représente le système masse-ressort-amortisseur qui en résulte.

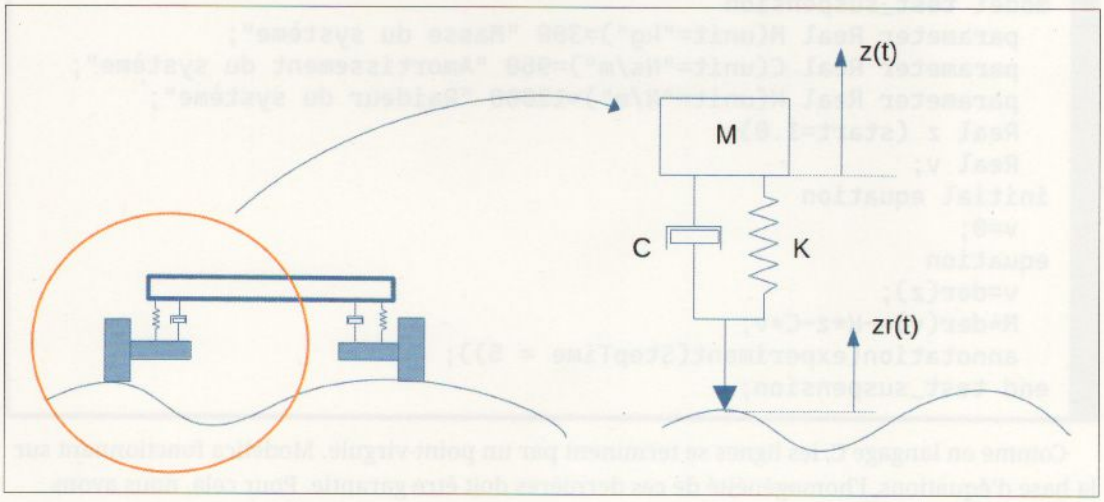


Figure 2 : Système de suspension.

Autour de la position de repos, le système de suspension est régi par l'équation suivante :

$$M\ddot{z} = -K(z - zr) - C(\dot{z} - \dot{zr})$$

soit :

$$M\ddot{z} + Kz + C\dot{z} = -Kzr - C\dot{zr}$$

Nous retrouvons une équation différentielle du second ordre avec second membre qu'il est possible de résoudre de manière analytique. Le second membre est lié à l'excitation produite par la route sur la roue.

Comme toute programmation objet, nous devons tout d'abord déclarer les variables également appelées attributs de notre modèle. La masse *M*, la raideur *K* et l'amortisseur *C* sont des variables réelles qui d'une simulation à l'une autre peuvent changer. Par conséquent, celles-ci seront donc définies avec le préfixe `parameter`. Quant au déplacement *z* et à la vitesse *v*, dérivée de la position *z*, ils sont les variables qui caractérisent le système et sont déclarés par leur type, en l'occurrence des réels. Nous devons également définir des conditions initiales aux problèmes telles que la position de départ ainsi que sa vitesse initiale.

Un modèle Modelica suit toujours la même structure. Définition avec un mot-clé associé au type d'objet que nous souhaitons définir, ici `model`, suivi du nom du modèle. Puis vient ensuite la déclaration de toutes les variables et de tous les paramètres. Et enfin, les équations qui décrivent le modèle lui-même. La syntaxe est celle représentée ci-dessous :

```
model nom_model
  // ensemble des variables utilisées dans le modèle
equation
  // liste des équations constitutives du modèle
end nom_model
```

Ouvrons donc un éditeur de texte et saisissons les lignes suivantes pour définir notre modèle de suspension :

```
model test_suspension
  parameter Real M(unit="kg")=300 "Masse du système";
  parameter Real C(unit="Ns/m")=950 "Amortissement du système";
  parameter Real K(unit="N/m")=12000 "Raideur du système";
  Real z (start=1.0);
  Real v;
initial equation
  v=0;
equation
  v=der(z);
  M*der(v)=-K*z-C*v;
  annotation(experiment(StopTime = 5));
end test_suspension;
```

Comme en langage C, les lignes se terminent par un point-virgule. Modelica fonctionnant sur la base d'équations, l'homogénéité de ces dernières doit être garantie. Pour cela, nous avons

pris soin de spécifier pour chaque paramètre, entre parenthèses avec le mot-clé `unit`, les unités employées. Modelica permet d'apporter des compléments d'information aux variables via l'utilisation de modificateurs tels que `unit` que nous venons d'utiliser. Les conditions initiales peuvent être définies de deux manières différentes. Soit au moment de la déclaration avec le modificateur `start`, soit dans la partie spécifique `initial equation`.

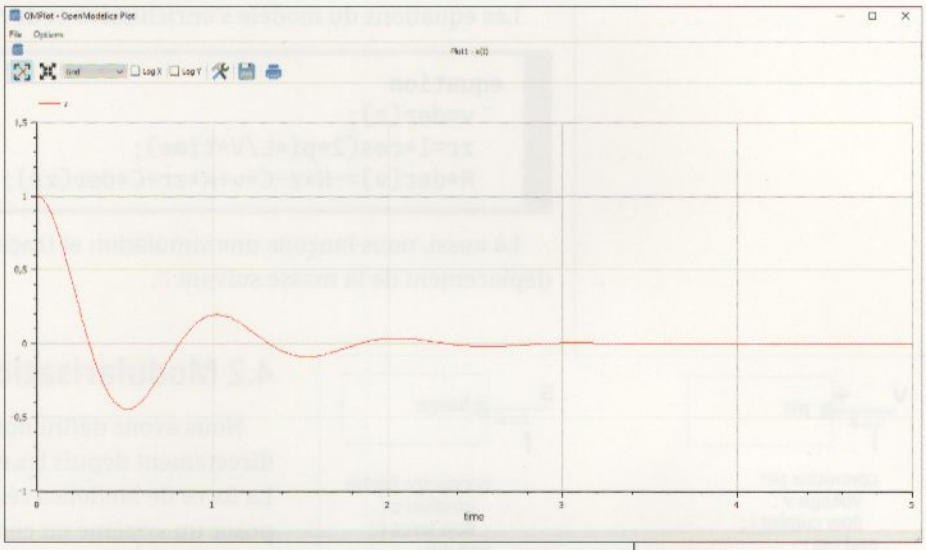


Figure 3 : Courbe de déplacement z.

Maintenant que notre système est défini, lançons le terminal OMShell. Chargeons le modèle avec la commande `LoadFile`. Instancions ensuite notre modèle puis démarrons la simulation avec la commande `simulate`.

```
LoadFile('/home/user/Tuto_Modelica/test_suspension.mo')
instanciate(test_suspension)
simulate(test_suspension)
```

Après un bref temps de calcul, Modelica nous informe de la réussite de la simulation, nous pouvons maintenant tracer le déplacement *z* en fonction du temps avec la fonction `plot`.

```
plot(z)
```

Par défaut, le temps de simulation est une seconde. Pour modifier ce paramètre, nous ajoutons une annotation afin de spécifier au compilateur le temps d'arrêt :

```
annotation(experiment(TimeStop=5.0))
```

Ajoutons à notre modèle un second membre correspondant à un déplacement sur une route sinusoïdale de période spatiale *L* et d'amplitude *A*. Le véhicule se déplace à une vitesse *V*, ce qui se traduit par un déplacement de la route :

$$zr(t)=Acos(wt) \text{ avec } w=2pi*L/V$$

Les équations du modèle s'enrichissent et deviennent :

```
equation
  v=der(z);
  zr=1*cos(2*pi*L/V*time);
  M*der(v)=-K*z-C*v+K*zr+C*der(zr);
```

Là aussi, nous lançons une simulation et traçons l'évolution du déplacement de la masse suivant `z`.

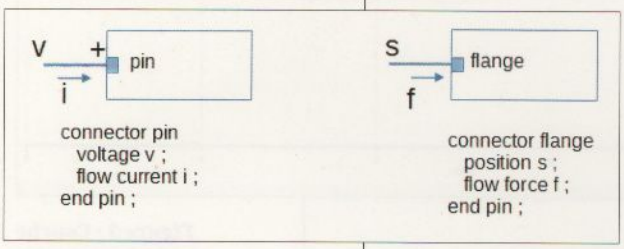


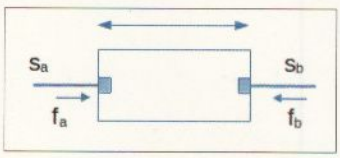
Figure 4 : Exemples d'interfaces.

4.2 Modularisation de la suspension

Nous avons défini notre système dans sa globalité directement depuis les équations qui le régissent. La force de Modelica réside dans le fait de décomposer un système en composants élémentaires, pour ensuite les connecter les uns aux autres. Afin de connecter des éléments ensemble, il faut donc définir des interfaces. Pour cela, Modelica dispose de la classe particulière `connector` dont les classes de la librairie standard *pin* et *flange* dérivent pour définir respectivement des interfaces électriques et mécaniques telles que représentées sur la figure 4.

La liaison entre les différents modules s'effectue via des interfaces. Les interfaces doivent être compatibles. Par exemple, une interface mécanique linéaire 1D nommée `Flange` comporte une position et une force. *A contrario*, une interface mécanique 3D de la librairie standard (*MultiBody*) nommée `Frame` comporte une position 3D et une force 3D qui ne sont autres que des vecteurs ainsi qu'un couple 3D, une position angulaire 3D et une orientation du repère. Pour relier des modules de ces deux librairies, il faut créer ou utiliser un modèle qui effectue la transformation d'un espace à l'autre (1D vers 3D et vice versa). La librairie standard met à disposition un modèle `LineForceWithMass` et `LineForceWithTwoMasses` qui effectue cette opération.

Figure 5 : Représentation d'un composant ayant deux points d'attache.



Notre système est composé d'un ressort, d'un amortisseur, d'un solide... Chacun de ces éléments peut être modélisé et implémenté indépendamment pour ensuite être connecté ensemble et ainsi former notre système de suspension, mais également être utilisé dans d'autres applications. Prenons le cas de notre ressort par exemple, il est physiquement constitué de deux points d'attache qui sont donc naturellement ses interfaces avec les éléments extérieurs, tout comme notre amortisseur. Nous pouvons donc définir un objet physique ayant deux points d'attache, dont la distance entre ces deux points est liée par une caractéristique du composant.

Notre ressort héritera ainsi de cet objet. La création de cet objet permet ainsi d'introduire le concept d'héritage mis en œuvre dans Modelica. Une analogie peut être faite avec les composants électriques : résistances, inductances, capacitances qui sont des dipôles passifs et peuvent également hériter de l'objet dipôle que l'on retrouve dans nombres d'exemples sous la classe `TwoPins`.

Notre objet ne sera que partiellement défini. La loi de comportement du composant telle que la raideur, l'amortissement, etc., sera définie dans le composant lui-même. L'implémentation d'un objet partiel, comme sa désignation le laisse penser, est faite en précédent la déclaration `model` par le mot-clé `partial`.

Bien que ces éléments soient présents dans la librairie standard Modelica, nous allons toutefois définir l'objet ressort et les éléments associés. L'ensemble des éléments que nous allons définir constituera donc une première librairie.

Modelica propose deux méthodes pour créer une librairie : soit un seul fichier où l'intégralité des composants sont implémentés, soit sous forme structurée de répertoires. Nous optons pour la seconde méthode. Afin de définir un répertoire comme librairie, il faut ajouter un fichier `package.mo` qui contient les définitions du `package`. La première ligne doit nécessairement comporter le mot-clé `within` suivi d'aucun paramètre. Ensuite la librairie est définie avec le mot-clé `package` comme nous l'avons fait précédemment pour le modèle. Le nom du *package* doit être celui du répertoire.

```
within;
package MonPremierPackage
end MonPremierPackage;
```

Afin d'apporter plus de clarté à une librairie, il est possible d'ajouter des sous-répertoires correspondant à des sous-librairies comme dans la librairie standard. Notre exemple étant relativement simple, nous allons rester dans le répertoire principal.

Commençons par créer un répertoire `MonPremierPackage` puis définissons quelques types avec les unités qui vont bien dans le fichier `test_composant.mo`.

```
type position=Real(unit="m",quantity="position");
type force=Real(unit="N",quantity="force");
type masse=Real(unit="kg",quantity="masse");
type vitesse=Real(unit="m/s",quantity="vitesse");
type acceleration=Real(unit="m/s2",quantity="acceleration");
```

Puis implémentons dans un autre fichier nommé pour l'occasion `interfmeca.mo` notre interface mécanique avec la classe `connector`. L'interface mécanique a pour objectif de spécifier le comportement de l'interface, c'est-à-dire lorsque plusieurs composants sont reliés entre eux sur une même interface, la relation entre les caractéristiques de l'interface. En l'occurrence, la somme des forces appliquée sur une interface est nulle donc la caractéristique sera représentée dans Modelica avec le modificateur `flow`. De plus, sur une même interface chaque position est identique entre tous les éléments.

```
connector interfmeca
   position s;
   flow force f;
end interfmeca;
```

Implémentons ensuite un objet partiel qui tient compte uniquement des interfaces physiques.

```
partial model ObjetPartiel
  interfmeca a;
  interfmeca b;
  position s_rel(start=0);
  force f;
equation
  s_rel=b.s-a.s;
  a.f=-f;
  b.f=f;
end ObjetPartiel;
```

Comme mentionné précédemment, notre ressort hérite de `ObjetPartiel` via le mot-clé `extends` auquel nous venons ajouter la loi de comportement du composant.

```
model ressort
  extends ObjetPartiel;
  parameter Real k(unit="N/m")=1;
  parameter position s_rel0=0;
equation
  f=k*(s_rel-s_rel0);
end ressort;
```

Définissons de la même manière notre amortisseur.

```
model amortisseur
  extends ObjetPartiel;
  parameter Real c(unit="Ns/m")=1;
 vitesse v(start=0.0);
equation
 v=der(s_rel);
  f=c*v;
end amortisseur;
```

Afin de pourvoir construire notre suspension, nous avons besoin de définir un objet masse ainsi qu'une base fixe sur lesquelles seront fixés le ressort et l'amortisseur.

```
model ground
  parameter position s0=0;
  interfmeca G;
```

```
equation
  G.s=s0;
end ground;

partial model ObjetRigide
  position s;
  parameter Real dim(unit="m",start=0);
  interfmeca a;
  interfmeca b;
equation
  a.s=s-dim/2;
  b.s=s+dim/2;

end ObjetRigide;

model body
  extends ObjetRigide;
  parameter masse M(start=1);
  vitesse v(start=0);
  acceleration acc(start=0);
equation
v=der(s);
acc=der(v);
M*acc=a.f+b.f;
end body;
```

Définissons notre système de suspension avec les modules que nous venons de créer. La connexion entre composants est réalisée dans la partie équation avec le mot-clé `connect`. Pour ceux qui connaissent *Ngspice*, vous pouvez remarquer de fortes similitudes.

```
model app_ressort
   ground fix(s0=0.0);
   ressort Res(k=12000,s_rel0=0.0);
  amortisseur Am(c=950);
   body obj(dim=1,s(start=1.0),M=300);
equation
  connect(Res.b,fix.G);
  connect(Res.a,obj.b);
  connect(Am.b,fix.G);
 connect(Am.a,obj.b);
end app_ressort;
```

Nous chargeons puis lançons notre simulation comme précédemment :

```
LoadFile('/home/user/Tuto_Modelica/test_modules.mo')
instanciate(test_modules.test_systeme)
simulate(test_modules.test_systeme,stopTime=5.0)
```

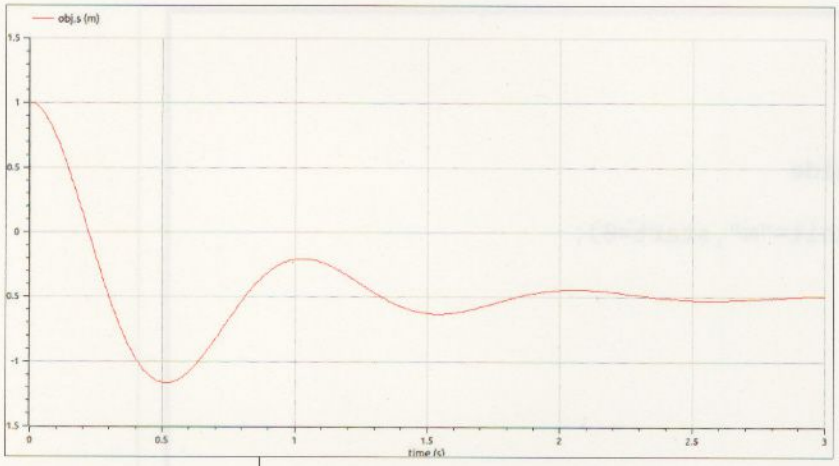


Figure 6 : Résultat de simulation du déplacement de la masse.

Enfin, nous pouvons visualiser l'évolution du déplacement de la masse au cours du temps avec la commande suivante :

```
plot(obj.s)
```

5. MODÉLISATION D'UNE CHAÎNE DE TRACTION DE VÉHICULE ÉLECTRIQUE

Après cette phase de découverte et prise en main sur un exemple tout simple, nous allons utiliser Modelica pour construire et étudier un modèle multiphysique plus complet. Nombre d'applications peuvent servir de support : par exemple en cette saison estivale, cela pourrait être le chauffage, traitement et maintien à niveau d'eau d'une piscine ou bien la chaîne de traction d'une trottinette électrique ou l'assistance d'un vélo électrique voire d'un véhicule électrique ou radiocommandé... C'est ce dernier cas que nous allons utiliser pour mettre en avant la facilité avec laquelle Modelica permet d'étudier un système multiphysique. Dans cette seconde partie, nous allons utiliser l'interface graphique OMEdit ainsi que les composants présents dans la librairie standard pour construire notre modèle. La figure 7 représente les différents éléments d'une chaîne de traction que nous allons modéliser.

Nous constatons que tous les éléments sont connectés entre eux. Les connexions du moteur sont de deux natures : l'une électrique, thermique ou hydraulique vers le système de commande et l'autre mécanique vers le réducteur. Le moteur, quelle que soit sa nature, électrique, thermique ou hydraulique, est l'élément par excellence de l'aspect multiphysique, en raison de sa fonction de convertisseur.

Nous pouvons étudier notre chaîne de traction en deux étapes : la première consistant à modéliser le moteur et sa commande, la seconde à modéliser la partie mécanique de la traction en remplaçant l'unité motrice par une commande en couple représentative du cycle que nous souhaitons étudier. L'ordre des étapes n'a que peu d'importance.

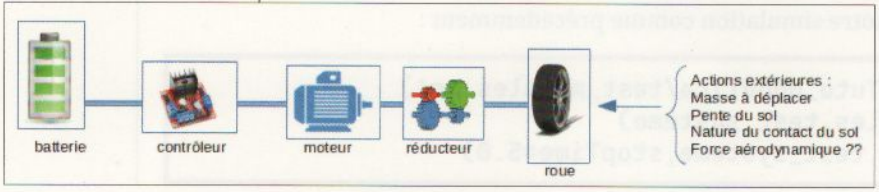


Figure 7 : Chaîne de traction.

5.1 Étape 1 : modélisation de l'unité motrice

Ouvrons l'éditeur OMEdit, puis chargeons la librairie standard : ***Fichier->Librairies système->Modelica->4.0.0***. Apparaît alors à gauche de l'éditeur une librairie nommée Modelica dans laquelle se trouvent différentes sous-librairies : ***Electrical***, ***Mechanics***, ***Thermal***, ***Magnetic***... que nous allons exploiter pour créer notre chaîne de traction.

Commençons par placer les éléments d'un moteur électrique continu, une résistance d'une valeur de 2,5 Ω, une inductance de 0,1 H et une force électromotrice 2,0 Nm/A comme nous le ferions avec n'importe quel logiciel de simulation électronique tel que KiCAD. Tous ces éléments se trouvent dans la librairie ***Modelica->Electrical->Analog***. Nous ajoutons également une inertie représentant celle du rotor du moteur de 0,005 kgm². La figure 8 représente notre modèle.

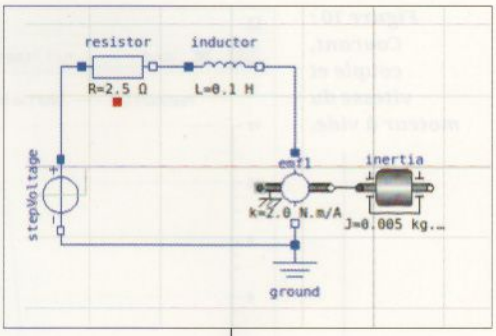


Figure 8 : Modèle du moteur électrique.

Afin de simuler l'ensemble, nous alimentons le moteur avec une source de tension constante de 12 V décalée au démarrage de 0,2 seconde. Dans la barre d'outils se trouvent plusieurs icônes vertes permettant de vérifier le modèle, de configurer et lancer la simulation avec ou sans débogueur. Par défaut, la durée de la simulation est d'une seconde. Cela convient tout à fait pour notre moteur seul. Nous ajusterons ce paramètre après avoir ajouté les autres éléments qui constituent une charge pour le moteur.

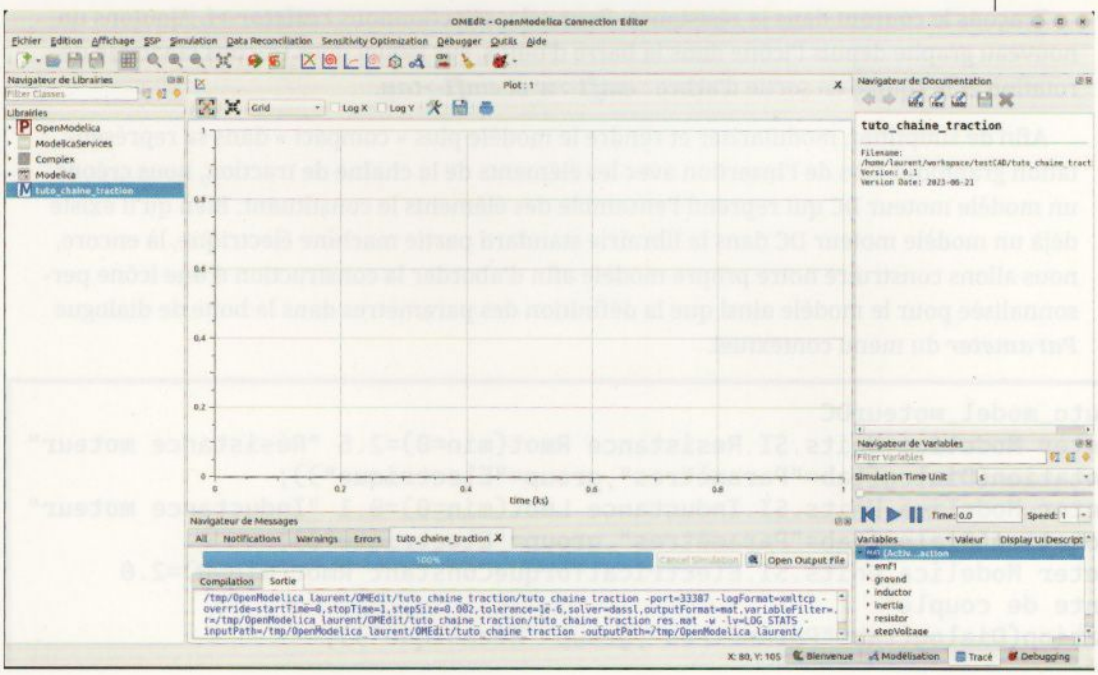


Figure 9 : Fenêtre d'exploitation des résultats.

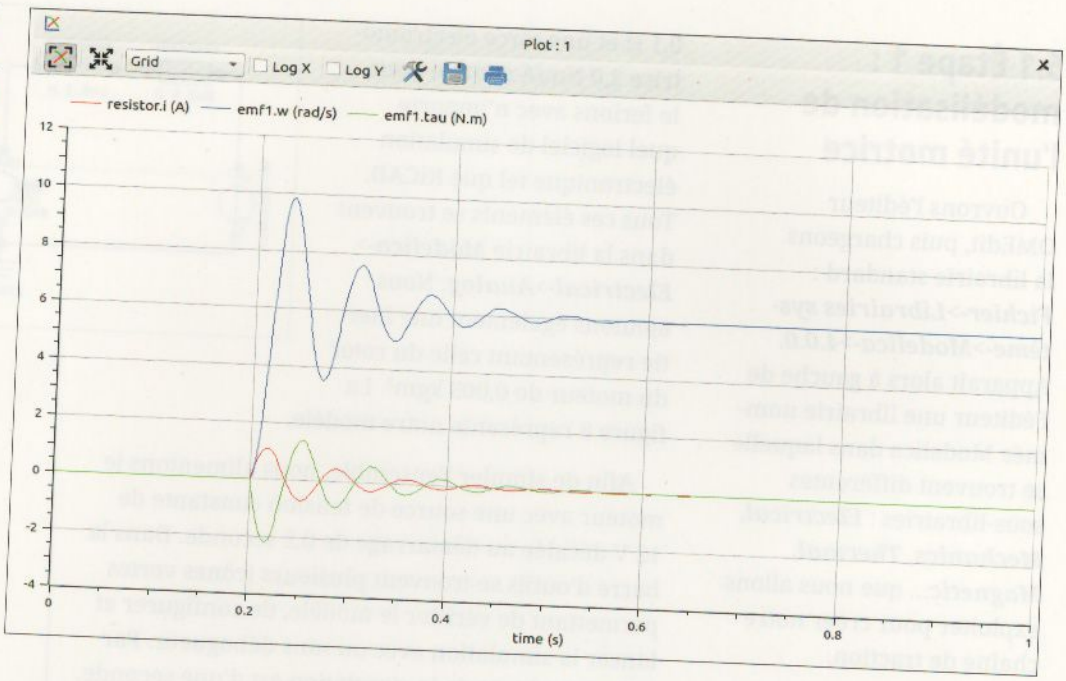


Figure 10 : Courant, couple et vitesse du moteur à vide.

La figure 9 représente la fenêtre de tracé permettant d'exploiter les résultats de simulation. En bas à droite se trouve une fenêtre avec l'ensemble de nos composants pour lesquels nous pouvons choisir les paramètres à tracer.

Au-dessus dans la barre des statuts, à droite se trouvent des onglets qui permettent de naviguer entre les différents modes : modélisation, tracé...

Traçons le courant dans la résistance. Pour cela, sélectionnons ***resistor->i***. Ajoutons un nouveau graphe depuis l'icône dans la barre d'outils, sur lequel nous traçons la vitesse de rotation et le couple en sortie d'arbre : ***emf1->w*** et ***emf1->tau***.

Afin de simplifier, modulariser et rendre le modèle plus « compact » dans sa représentation graphique lors de l'insertion avec les éléments de la chaîne de traction, nous créons un modèle moteur DC qui reprend l'ensemble des éléments le constituant. Bien qu'il existe déjà un modèle moteur DC dans la librairie standard partie machine électrique, là encore, nous allons construire notre propre modèle afin d'aborder la construction d'une icône personnalisée pour le modèle ainsi que la définition des paramètres dans la boîte de dialogue ***Parameter*** du menu contextuel.

```
model tuto_model_moteurDC
  parameter Modelica.Units.SI.Resistance Rmot(min=0)=2.5 "Résistance moteur"
    annotation(Dialog(tab="Paramètres",group="Electrique"));
  parameter Modelica.Units.SI.Inductance Lmot(min=0)=0.1 "Inductance moteur"
    annotation(Dialog(tab="Paramètres",group="Electrique"));
  parameter Modelica.Units.SI.ElectricalTorqueConstant kmot(min=0)=2.0
"Constante de couple"
  annotation(Dialog(tab="Paramètres",group="Mécanique"));
```

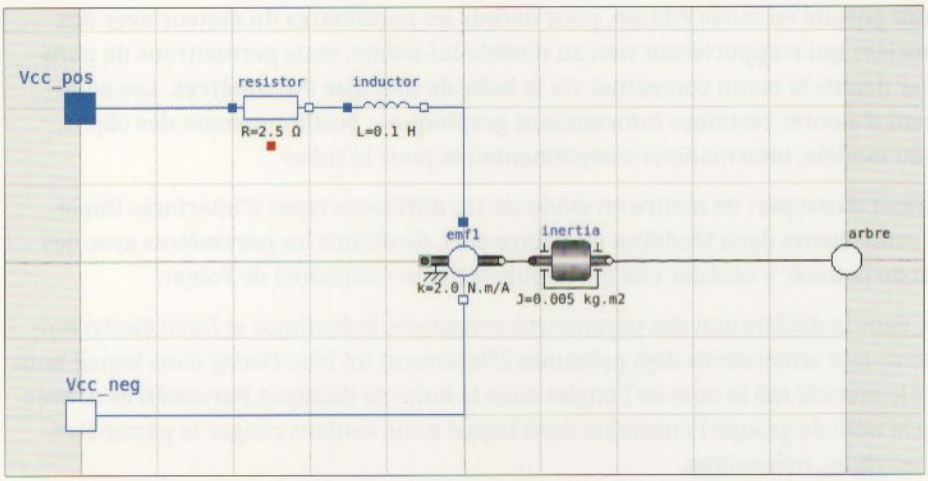


Figure 11 : Modèle du Moteur DC.

Nous dupliquons notre modèle `moteurDC` dans un autre modèle. Nous remplaçons la source de tension ainsi que la masse par deux connecteurs *Pin* dédiés à la connexion avec le contrôleur moteur. Nous ajoutons également un connecteur *Flange* pour la connexion mécanique de l'arbre moteur avec l'extérieur. L'ensemble est représenté sur la figure 11.

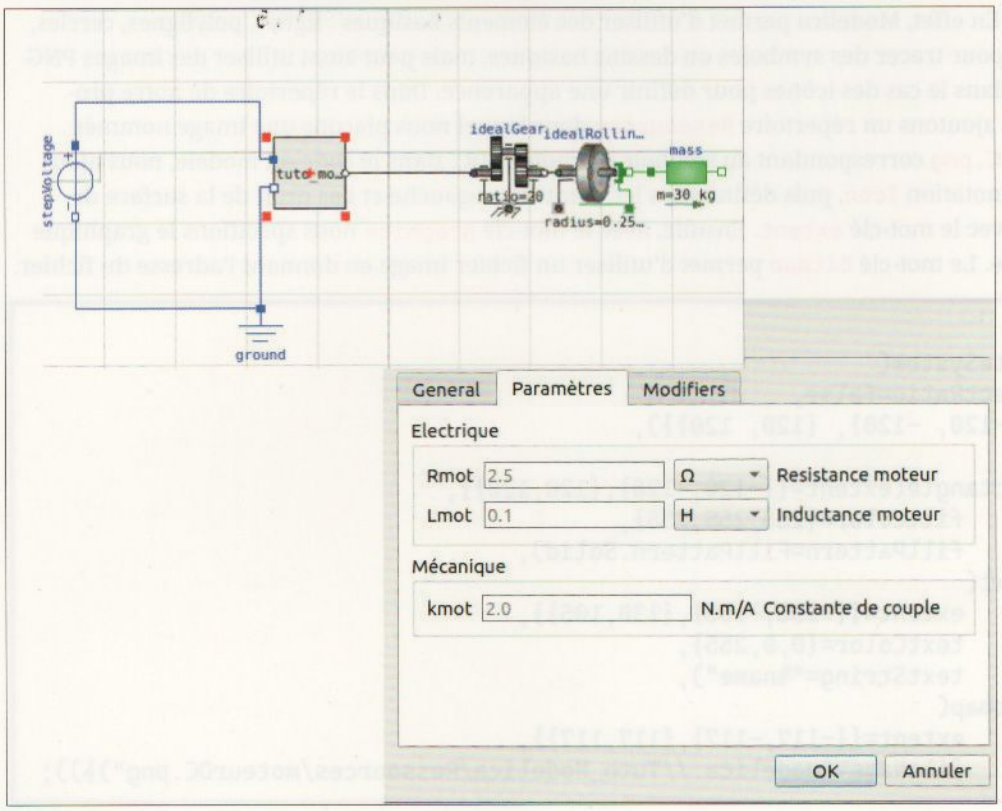


Figure 12 : Menu contextuel du modèle du moteur DC.

Nous basculons ensuite en mode édition, pour définir les paramètres du moteur avec des annotations associées qui n'apporteront rien au modèle lui-même, mais permettrons de paramétrer ce dernier depuis le menu contextuel via la boîte de dialogue ***Paramètres***. Les annotations permettent d'ajouter certaines informations graphiques : positionnement des objets, représentation du modèle, informations complémentaires pour le *solver*...

Cela nous permet d'une part de mettre en évidence les différents types d'interfaces implémentés par des connecteurs dans Modelica et d'autre part, de définir les paramètres avec des annotations afin de pouvoir y accéder ensuite depuis le menu contextuel de l'objet.

Pour ce faire, dans la déclaration des paramètres résistance, inductance et force électromotrice, nous ajoutons aux annotations déjà présentes (*Placement*) un bloc *Dialog* dans lequel nous définissons avec le mot-clé *tab* le nom de l'onglet dans la boîte de dialogue ***Paramètres*** et avec le mot-clé *group* le nom de groupe l'ensemble dans lequel nous voulons ranger le paramètre, par exemple : électrique, mécanique.

Pour tester cela, nous revenons à notre premier modèle dans lequel nous remplaçons les éléments du moteur par le modèle que nous venons de créer. Vous constatez un carré gris avec trois connecteurs. Nous modifierons son apparence dans un instant. Depuis le menu contextuel ***Paramètres*** du modèle, nous retrouvons sur la figure 12 l'onglet ***Paramètres*** dans lequel se trouvent la résistance, l'inductance et la force électromotrice.

Passons maintenant à la personnalisation de l'apparence de notre modèle. Pour ce faire, nous revenons au mode ***Edition*** de ce dernier et ajoutons une nouvelle annotation en fin de modèle. En effet, Modelica permet d'utiliser des éléments basiques : lignes, polylignes, cercles, textes... pour tracer des symboles ou dessins basiques, mais peut aussi utiliser des images PNG comme dans le cas des icônes pour définir une apparence. Dans le répertoire de notre projet, nous ajoutons un répertoire `Ressources` dans lequel nous plaçons une image nommée `moteurDC.png` correspondant au symbole du moteur DC. Dans le code du modèle, nous ajoutons l'annotation `Icon`, puis définissons les points haut gauche et bas droit de la surface de l'icône avec le mot-clé `extent`. Ensuite, avec le mot-clé `graphics` nous spécifions le graphique lui-même. Le mot-clé `Bitmap` permet d'utiliser un fichier image en donnant l'adresse du fichier.

```
annotation(
  Icon(coordinateSystem(
    preserveAspectRatio=false,
    extent = {{-120, -120}, {120, 120}}),
    graphics={
            Rectangle(extent={{-120,-120},{120,120}},
                 fillColor={255,255,255},
                 fillPattern=FillPattern.Solid),
            Text(
                 extent={{-130,-105},{130,105}},
                 textColor={0,0,255},
                 textString="%name"),
            Bitmap(
                 extent={{-117,-117},{117,117}},
                 fileName="modelica://Tuto_Modelica/Ressources/moteurDC.png")}));
```

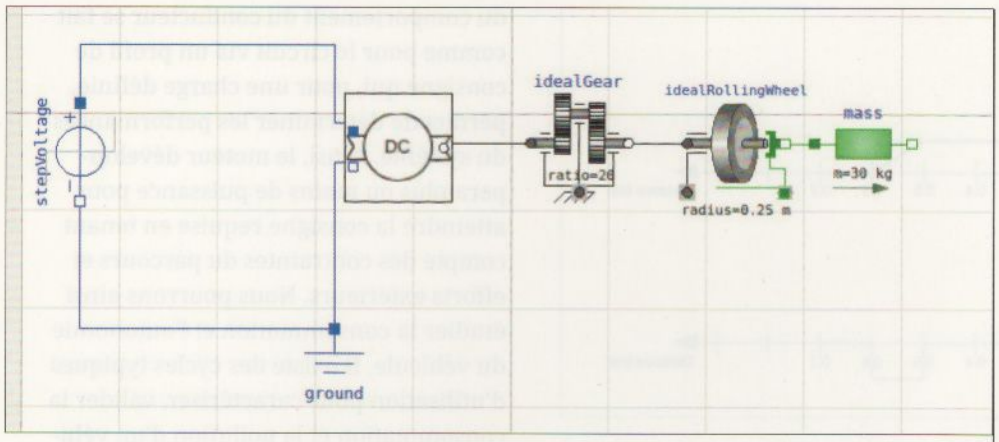


Figure 13 : Chaîne de traction complète hors efforts extérieurs.

Nous constatons dans le modèle que la surface grise est remplacée par le symbole du moteur DC.

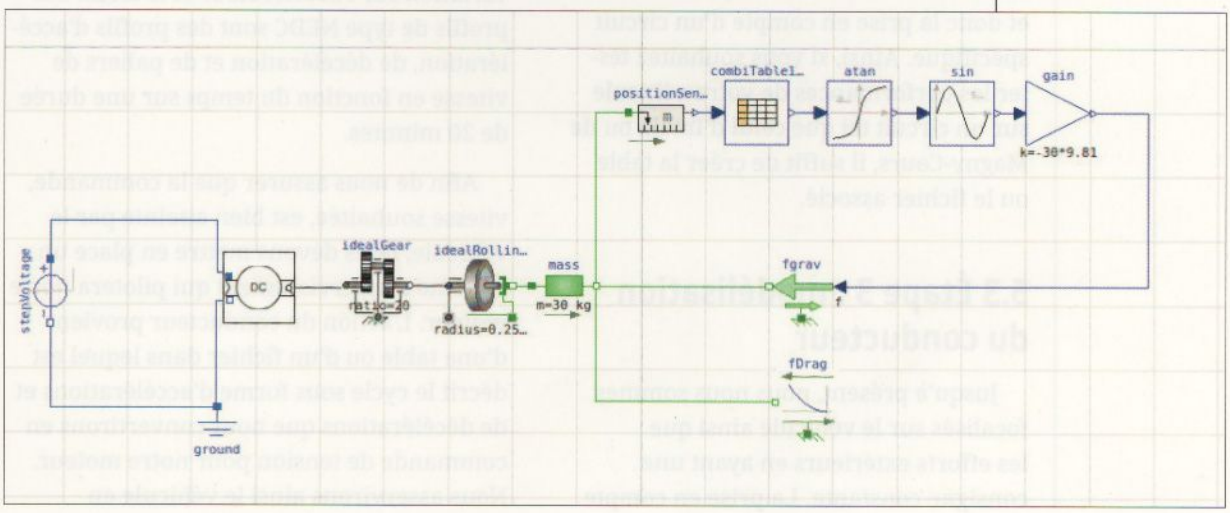
5.2 Étape 2 : modélisation de la partie mécanique

Nous complétons la chaîne de traction comme représentée dans la figure 7 avec les éléments mécaniques : réducteur, roue et masse du véhicule. La figure 13 montre la chaîne complète hors efforts extérieurs (aérodynamique, frottements...).

Après avoir paramétré chaque composant, nous lançons une première simulation sur une durée de 5 secondes. Nous traçons la distance parcourue par le véhicule via ***mass->Flange.b->s*** où la variable `s` correspond à la position.

Afin d'avoir un modèle complet, nous ajoutons les différents efforts qui s'exercent sur le véhicule, à savoir, les efforts de gravité qui dépendent de l'inclinaison de la route, les efforts aérodynamiques et les efforts de frottements de la roue au sol. La figure 14 représente la chaîne de traction

Figure 14 : Chaîne de traction complète avec les efforts extérieurs.



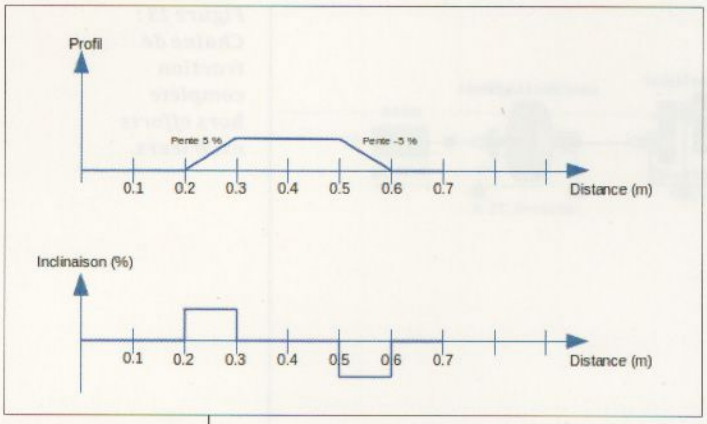


Figure 15 : Profil de roue.

complète avec les efforts extérieurs. L'effort aéronautique est modélisé par une force dont la dépendance est quadratique avec la vitesse.

Pour tenir compte de l'influence de la route, l'inclinaison en particulier, nous utilisons une table dont l'entrée correspond à la distance parcourue et la sortie à l'inclinaison de la route à ladite distance. La figure 15 représente le profil de route que nous allons tester.

La lecture de la distance parcourue est obtenue via un capteur de position. Nous pouvons ainsi implémenter aisément l'effort de gravité sur le véhicule en fonction de la distance parcourue et donc la prise en compte d'un circuit spécifique. Ainsi, si vous souhaitez tester les performances de votre véhicule sur un circuit tel que celui d'Imola ou de Magny-Cours, il suffit de créer la table ou le fichier associé.

5.3 Étape 3 : modélisation du conducteur

Jusqu'à présent, nous nous sommes focalisés sur le véhicule ainsi que les efforts extérieurs en ayant une consigne constante. La prise en compte du comportement du conducteur se fait comme pour le circuit via un profil de consigne qui, pour une charge définie, permet de déterminer les performances du système. Ainsi, le moteur développera plus ou moins de puissance pour atteindre la consigne requise en tenant compte des contraintes du parcours et efforts extérieurs. Nous pourrons ainsi étudier la consommation et l'autonomie du véhicule. Il existe des cycles typiques d'utilisation pour caractériser, valider la consommation et la pollution d'un véhicule. Par exemple, le NEDC (*New European Driving Cycle*) dont l'objectif est d'établir un scénario reproductible des conditions rencontrées sur les routes européennes. Ce dernier a évolué pour devenir le WLTP (*Worldwide harmonized Light vehicles Test Procedures*) afin d'être harmonisé au niveau mondial **[10]**.

Nous allons modéliser l'action du conducteur et implémenter celle-ci via une interface électrique associée. Dans un véhicule automatique, bien qu'il soit possible là aussi de mettre en œuvre une boîte de vitesse mécanique et de gérer les rapports de vitesse, nous nous contenterons de modéliser l'action du conducteur par une variation sur l'accélérateur et le frein. Les profils de type NEDC sont des profils d'accélération, de décélération et de paliers de vitesse en fonction du temps sur une durée de 20 minutes.

Afin de nous assurer que la commande, vitesse souhaitée, est bien atteinte par le véhicule, nous devons mettre en place un système d'asservissement qui pilotera notre moteur. L'action du conducteur provient d'une table ou d'un fichier dans lequel est décrit le cycle sous forme d'accélérations et de décélérations que nous convertirons en commande de tension pour notre moteur. Nous asservirons ainsi le véhicule en

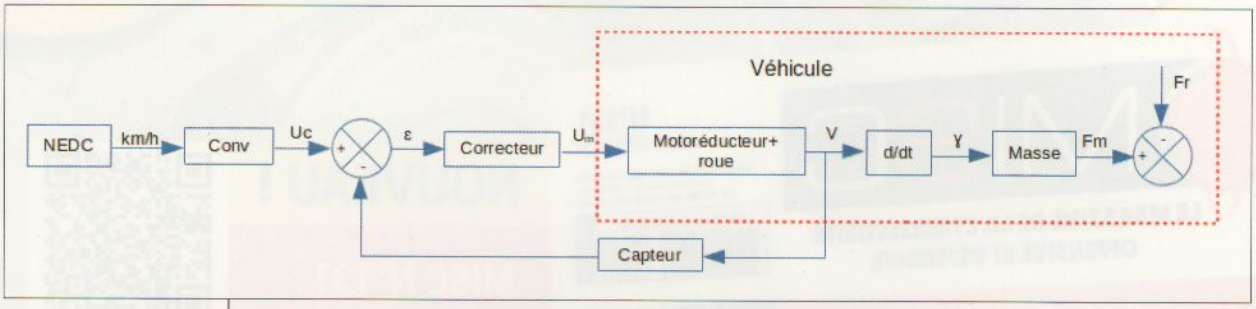


Figure 16 : Boucle d'asservissement en vitesse du véhicule.

vitesse. Nous ne détaillons pas ici le calcul de la boucle d'asservissement et la détermination des paramètres, mais rappelons uniquement les grandes lignes afin de pouvoir implémenter la commande de notre véhicule dans Modelica. La figure 16 représente la boucle d'asservissement en vitesse complète du véhicule.

Cette dernière peut se réduire à l'étude de la fonction de transfert représentée sur la figure 17 dans laquelle nous implémentons un correcteur C(p), un retour unitaire et la fonction de transfert du moteur. Dans la fonction de transfert du moteur, nous n'avons pas tenu compte des efforts aérodynamiques ainsi que des efforts de frottement, mais seulement de l'inertie.

$$H_m(p)=\frac{1/k}{1+\frac{RJ}{k^2}p+\frac{JL}{k^2}p^2}=\frac{1/k}{1+\tau_{em}p+\tau_{em}\tau_e p^2}$$

avec :

$$\tau_e=\frac{L}{R}$$

et :

$$\tau_{em}=\frac{JR}{k^2}$$

La fonction de transfert d'un correcteur proportionnel intégral est de la forme :

$$C(p)=G\left(1+\frac{1}{T_i p}\right)$$

La stabilité du système et la détermination des paramètres du correcteur sont obtenues en utilisant le critère de Routh-Hurwitz **[11]**. La fonction de transfert du système en boucle fermée est la suivante :

$$FTBF(p)=\frac{G(1+T_i p)(1+\tau_{em}p\,\tau_e\,\tau_{em}p^2)}{T_i p(1+\tau_{em}p+\tau_e\,\tau_{em}p^2)+\frac{G}{k}(1+T_i p)}$$

Nous analysons donc le polynôme du dénominateur :

$$\tau_e\,\tau_{em}T_i p^3+\tau_{em}T_i p^2+T_i\left(1+\frac{G}{k}\right)p+\frac{G}{k}$$

pour lequel tous les coefficients sont positifs, ce qui est une condition nécessaire pour utiliser le second critère de Routh. Second critère qui stipule qu'un système est stable lorsque tous les coefficients de la première colonne du tableau de Routh sont de même signe. Dans notre application, les coefficients de la première colonne sont :

$$\begin{cases} \tau_{em}\,\tau_e\,T_i > 0 \\ \tau_{em}\,T_i > 0 \\ -\tau_e\frac{G}{k} + T_i\left(1+\frac{G}{k}\right) > 0 \\ \frac{G}{k} > 0 \end{cases}$$

Le système est donc stable pour des valeurs de G>0 et :

$$T_i > \frac{\tau_e G}{k+G}$$

Maintenant que les paramètres du correcteur sont déterminés, nous pouvons reprendre la construction de notre modèle sous Modelica.

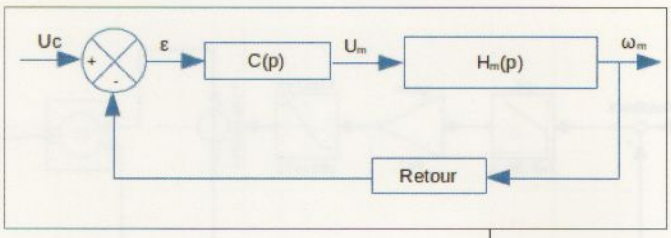
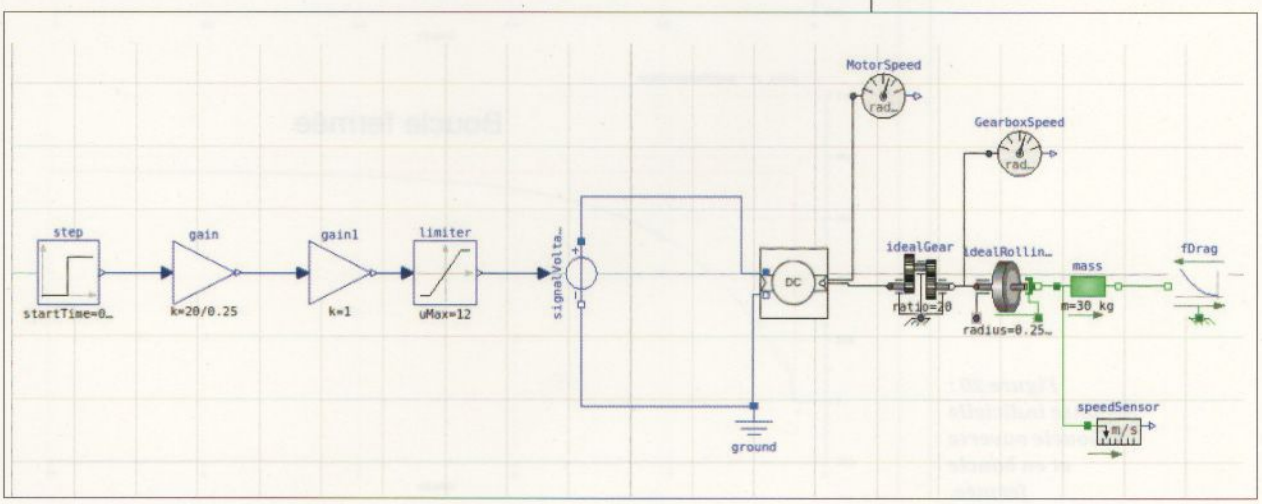


Figure 17 : Boucle d'asservissement en vitesse du véhicule réduite.

La consigne de notre système est le profil de vitesse du cycle étudié, nous devons donc convertir le profil de vitesse en profil de tension pour piloter notre moteur. Nous devons cependant nous assurer que la tension d'alimentation du moteur n'excédera pas la tension admissible. Deux possibilités, soit nous mettons en œuvre un limiteur de tension, soit nous nous assurons que le profil de vitesse est compatible avec le système, c'est-à-dire que les vitesses du cycle sont atteignables par le système. Ce qui n'est pas le cas dans la vraie vie, car cela revient à dire que la vitesse maximale du véhicule est supérieure ou égale à la vitesse maximale du profil. Nous mettrons donc en place

Figure 18 : Asservissement en vitesse du véhicule en boucle ouverte.



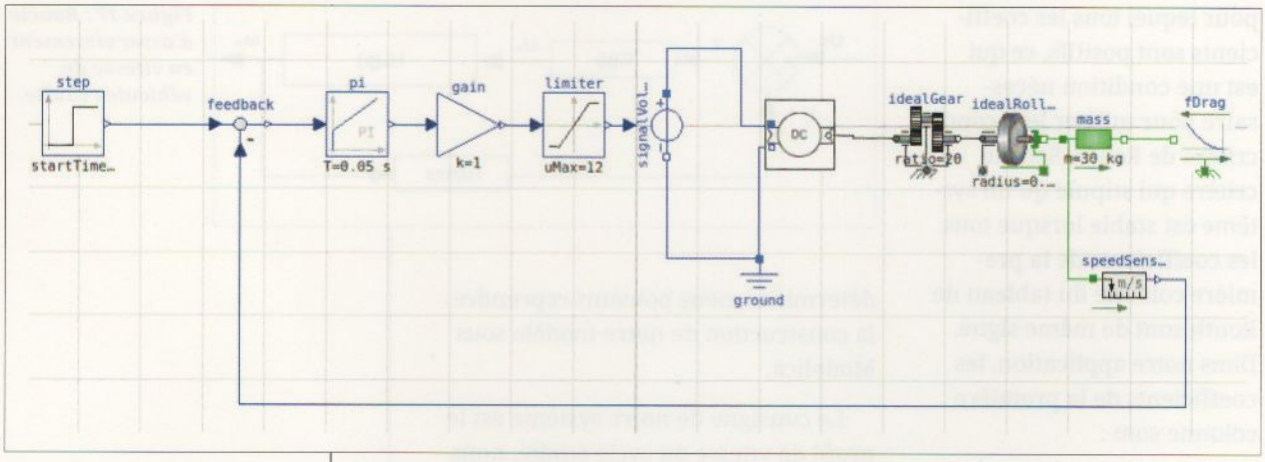


Figure 19 : Asservissement en vitesse du véhicule en boucle fermée.

un limiteur de tension dans notre modèle. De plus, nous utilisons le composant *SignalVoltage* pour générer la tension à partir d'une commande sous forme de valeur réelle.

Afin de valider le calcul du correcteur, nous allons simuler notre système à une réponse indicielle, en

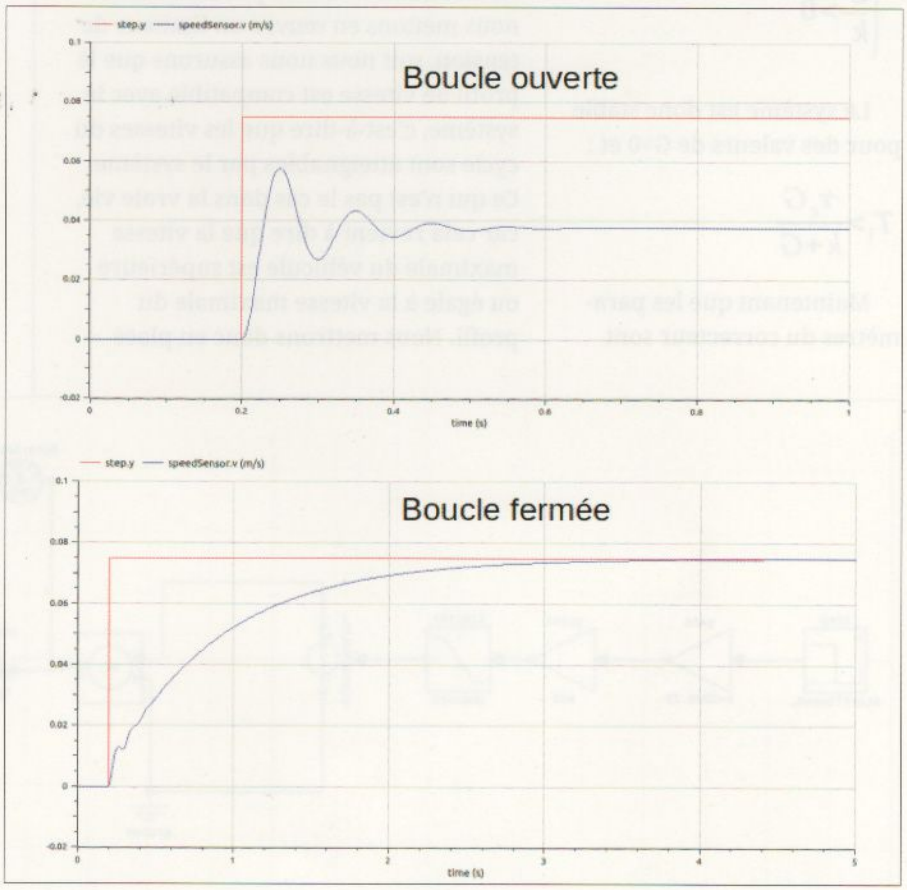


Figure 20 : Réponse indicielle en boucle ouverte et en boucle fermée.

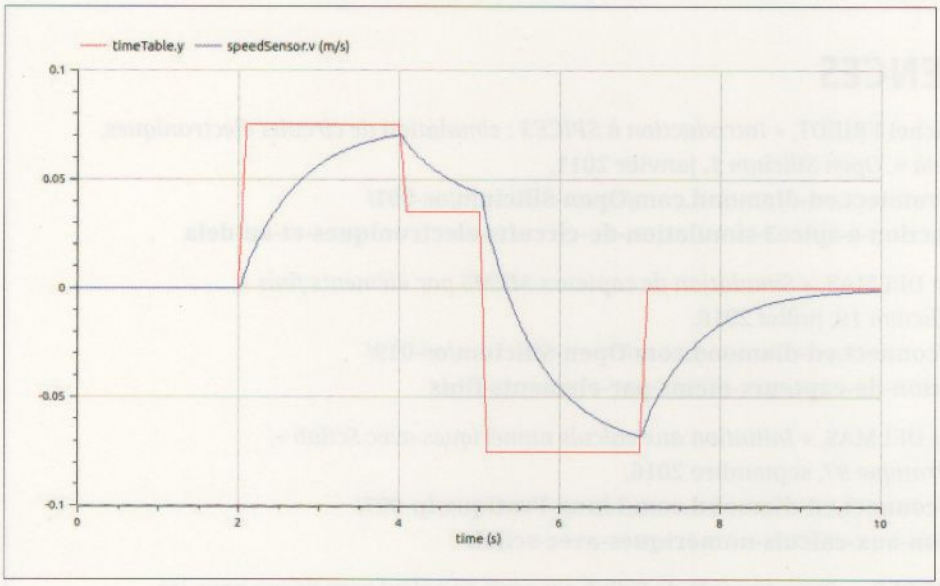


Figure 21 : Simulation d'un profil de vitesse complet.

l'occurrence, une consigne de vitesse du véhicule. La figure 18 montre le système en boucle ouverte et la figure 19, le même système en boucle fermée dans lequel nous avons implémenté un bloc *feedback* et un bloc *PI* de la librairie standard Modelica. Les paramètres du correcteur sont déterminés à partir des conditions énumérées précédemment soit pour un G=12, Ti=0,05 (>0,034). Le *limiter*, comme nous en avons parlé précédemment, a pour but de limiter la tension d'alimentation à la tension nominale du moteur, soit 12 V.

La figure 20 montre la réponse du système en boucle ouverte et en boucle fermée. Nous remarquons qu'en boucle ouverte, le système oscille avant de se stabiliser à une valeur en deçà de la consigne. L'application du correcteur permet l'atteinte de l'objectif sans oscillation.

Nous pouvons donc définir un profil un peu plus complet avec des valeurs de vitesse positives et négatives et analyser le comportement de notre véhicule. La figure 21 représente en rouge la consigne en vitesse et en bleu la vitesse du véhicule.

CONCLUSION

Dans cet article, nous avons tout d'abord découvert et pris en main le langage Modelica avec un exemple simple de suspension. Ensuite, nous avons modélisé et simulé un système multiphysique par excellence qu'est la chaîne de traction d'un véhicule électrique. Nous avons implémenté un système d'asservissement en vitesse du véhicule et simulé l'ensemble sur un profil de vitesse test de pilotage. Certes, nous n'avons pas modélisé la batterie et le convertisseur électrique dans le détail, mais uniquement simplifié leur fonctionnement, laissant ainsi aux lecteurs le soin de s'exercer en appliquant les connaissances acquises dans cet article. **LD**

RÉFÉRENCES

[1] Jean-Michel FRIEDT, *« Introduction à SPICE3 : simulation de circuits électroniques, et au-delà »*, *Open Silicium 1*, janvier 2011, **https://connect.ed-diamond.com/Open-Silicium/os-001/introduction-a-spice3-simulation-de-circuits-electroniques-et-au-dela**

[2] Laurent DELMAS, *« Simulation de capteurs MEMS par éléments finis »*, *Open Silicium 19*, juillet 2016, **https://connect.ed-diamond.com/Open-Silicium/os-019/simulation-de-capteurs-mems-par-elements-finis**

[3] Laurent DELMAS, *« Initiation aux calculs numériques avec Scilab »*, *Linux Pratique 97*, septembre 2016, **https://connect.ed-diamond.com/Linux-Pratique/lp-097/initiation-aux-calculs-numeriques-avec-scilab**

[4] Yann MORER, *« Prise en main de Scilab, un outil de calcul numérique pour les scientifiques »*, *Linux Pratique 112*, mars 2019, **https://connect.ed-diamond.com/Linux-Pratique/lp-112/prise-en-main-de-scilab-un-outil-de-calcul-numerique-pour-les-scientifiques**

[5] Laurent DELMAS, *« Simulez vos circuits électroniques avant de réaliser vos cartes électroniques avec KiCAD »*, *Hackable 32*, janvier 2020, **https://connect.ed-diamond.com/Hackable/hk-032/simulez-vos-circuits-electroniques-avant-de-realiser-vos-cartes-electroniques-avec-kicad**

[6] Yann GUIDON, *« Circuitjs simule des circuits électroniques dans votre navigateur »*, *Hackable 33*, avril 2020, **https://connect.ed-diamond.com/Hackable/hk-033/circuitjs-simule-des-circuits-electroniques-dans-votre-navigateur**

[7] Jean-Michel FRIEDT, *« Quel temps fera-t-il la semaine prochaine ? Évolution d'un système chaotique simulé en virgule fixe »*, *Hackable 47*, mars 2023, **https://connect.ed-diamond.com/hackable/hk-047/quel-temps-fera-t-il-la-semaine-prochaine-evolution-d-un-systeme-chaotique-simule-en-virgule-fixe**

[8] Modelica Language Specification version 3.5, **https://specification.modelica.org/maint/3.5/MLS.html**

[9] Modeling, Simulation, and Development of Cyber-Physical Systems with OpenModelica and FMI, Peter FRITZSON, **https://openmodelica.org/images/M_images/Modelica-OpenModelica-slides.pdf**

[10] NEDC, WLTP, EPA : que signifient ces normes dans le secteur auto ? **https://www.autojournal.fr/environnement/nedc-wltp-epa-normes-secteur-auto-300341.html#item=3**

[11] Polynôme de Hurwitz, **https://fr.wikipedia.org/wiki/Polynôme_de_Hurwitz**

EUROPEAN CYBER CUP

L'EC2 est la première compétition d'eSport dédiée au hacking éthique !

25 équipes

250 joueurs

6 épreuves

RENFORCEZ VOTRE MARQUE EMPLOYEUR, RECRUTEZ VOS TALENTS DE DEMAIN