ÉLECTRONIQUE | EMBARQUÉ | RADIO | IOT

HACKABLE

L'EMBARQUÉ À SA SOURCE

N° 58
JANVIER / FÉVRIER 2025

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 - 58 - F: 14,90 € - RD

CPPAP : K92470

USB / CY7C68013A / C

Créez vos périphériques USB basés sur le microcontrôleur EZ-USB FX2LP avec SDCC et LibUSB p.62

FPGA / FRAMEWORK

Plongeons au cœur de LiteX et créons deux systèmes RISC-V complets fonctionnant sous Linux p.100

RASPBERRY PI / CAMERA / MOTIONEYE

0% Cloud !

Comment garder un œil sur votre habitat ET votre vie privée ?

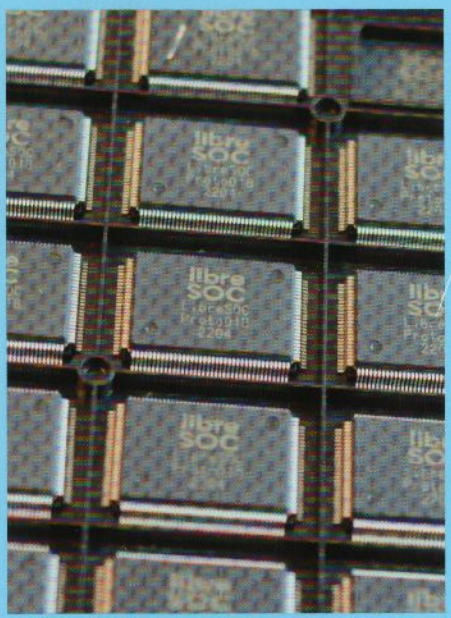
DOMOTIQUE & VIDÉOSURVEILLANCE

p.38

- Transformez vos Pi en IPcam
- Détectez les mouvements
- Déclenchez des actions
- Intégrez le tout à Home Assistant

ACTU / CPU / ASIC

FSiC 2024 : État de l'art, rétrospective et futurs défis du silicium libre et de la production de circuits intégrés open hardware p.04



SECURITÉ / CRYPTO

Ajoutez des signatures cryptographiques ECDSA miniatures à n'importe quels textes et QRcodes p.80

RPI / GPU / CALCUL

Initiez-vous facilement à la programmation parallèle sur GPU avec OpenCL et Raspberry Pi p.50

connect.ed-diamond.com

EUROPEAN CYBER CUP

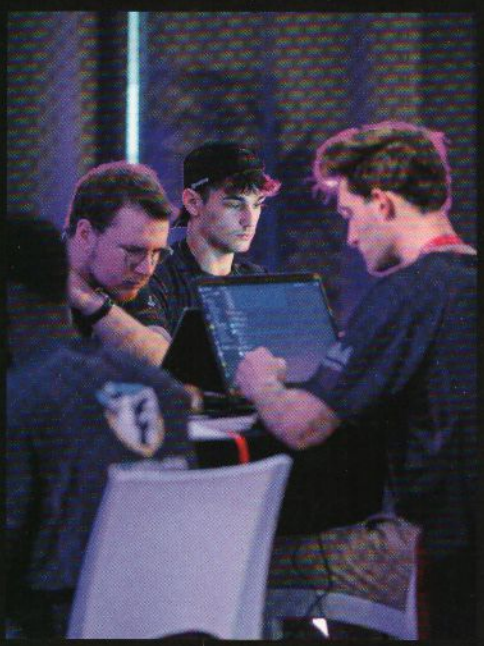
LA PREMIÈRE COMPÉTITION EUROPÉENNE DE CYBERSÉCURITÉ !

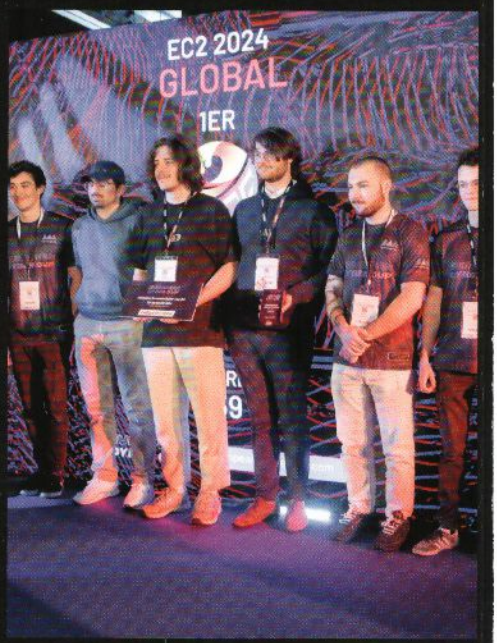
L'EC2 est le rendez vous à ne pas manquer, permettant de mêler team building de vos équipes sécurité, marque employeur et communication auprès de l'écosystème cybersécurité français et européen !

Vous serez nombreux mais à la fin il n'en restera qu'un !
L'équipe vainqueur recevra un chèque d'un montant de 5 000 €.

1-2 AVRIL 2025

LILLE GRAND PALAIS
FRANCE





ÉPREUVES 2025

.01/ **CTF**

.02/ **OSINT**

.03/ **FORENSIC**

.04/ **OT**

.05/ **RÉCUPÉRATION DE DONNÉES**

.06/ **BATTLE ROYALE**

.07/ **AD/WINDOWS**

↗ **european-cybercup.com**

ÉDITO

Une vidéo récente sur la chaîne secondaire de Matthias Wandel **[1]** m'a récemment fait vérifier presque l'ensemble des câbles que j'utilise pour mes montages et autres expériences. En effet, Matthias, qui généralement diffuse du contenu autour du travail du bois, a constaté que ses câbles munis de pinces, achetés très récemment, présentaient une résistance relativement surprenante, quelque 300 mΩ pour 15 cm, ce qui n'est peut-être pas un problème pour des signaux (selon la fréquence), mais en est clairement un pour une alimentation.

Habituellement, il s'agit généralement d'un problème de fabrication au niveau des connexions (en particulier pour les pinces) et ajouter un point de soudure règle le problème. Mais ici, le souci était différent : il s'est finalement avéré que non seulement le nombre de brins était très faible, mais qu'en plus, il ne s'agissait pas du tout de cuivre ou d'aluminium, seulement de fer plaqué cuivre, expliquant totalement les surprenantes mesures. Depuis peu donc, les câbles et clips « chinois », tels qu'on en trouve un peu partout pour peu cher (typiquement AliExpress et consorts), viennent tout juste de drastiquement chuter en qualité...

L'astuce accompagnant la vidéo consiste à tout simplement utiliser un aimant néodyme et voir si effectivement nous avons affaire à un alliage ferromagnétique ou non. Et effectivement, j'en avais quelques-uns dans ma collection (dont un clip de test SIOC-8 !). Notez au passage que la vidéo teste ainsi également un câble de souris, mais là, en vérité, ceci est parfaitement normal, c'est le blindage tressé protégeant le câble, pas de quoi s'inquiéter.

Ce qui inquiétant, en revanche, et bien plus que la méfiance d'usage dont il faut toujours faire preuve en achetant à certains endroits du Net, est le fait que ce type de production n'est absolument pas réservé au marché « extérieur ». Les câbles, composants et matériaux bas de gamme, copiés, plagiés à bas coût... infestent également le marché chinois lui-même, se retrouvent « sourcés » pour des produits plus légitimes et impactent également les progrès techniques de ce pays, qui prend déjà ses aises côté sécurité sur d'autres plans (comme les *boosters* de fusées qui retombent de-ci de-là).

C'est une contamination à la source de la chaîne de production, tel qu'on le voit déjà pour les circuits intégrés contrefaits. Tout ceci va, à un moment, tourner très mal...

Denis Bodor

[1] https://www.youtube.com/watch?v=15sMogK3vTI

Hackable Magazine
est édité par Les Éditions Diamond

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 :
Valérie Fréchard - 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 €

POUR DEVENIR AUTEUR
Contactez :
contrib@hackable.fr
Consultez :

Suivez-nous sur Twitter
@hackablemag

Alsace

SOMMAIRE

À PROPOS DE HACKABLE...

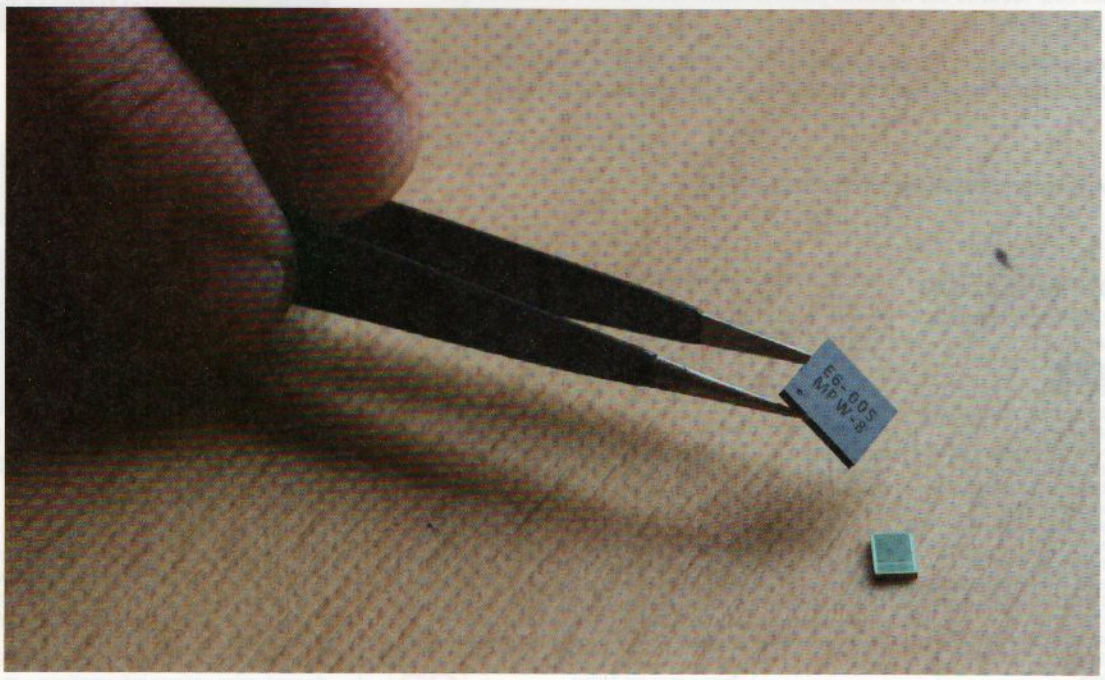
HACKS, HACKERS & HACKABLE

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.

FSIC2024 : L'OPEN SILICIUM DÉCOLLE À PARIS !

Yann Guidon
[whygee@f-cpu.org]

Du 19 au 21 juin a eu lieu la FSiC2024 [1], ou « Free Silicon Conference » sur le campus de Jussieu à Paris : trois intenses journées organisées par le projet « Go IT ! » (financé par l'Union européenne [2]) et hébergées par le département LIP6 de la Sorbonne/Paris 6. On y a croisé des académiques, bien sûr, mais aussi des bricoleurs de différents niveaux et d'horizons variés, des industriels et même des institutionnels ! Voici un petit résumé de ce que vous avez manqué et pourquoi vous auriez dû venir.



– FSiC2024 : L'Open Silicium décolle à Paris ! –

Quelle époque incroyable nous vivons ! Il y a quatre ans, je vous expliquais **[3]** l'importance de l'accord historique entre Google, eFabless et Skywater. Le (pro)moteur principal de cette initiative, Tim « mithro » Ansell, n'est plus aussi impliqué, mais la masse critique a été indéniablement atteinte : le premier PDK (*Process Design Kit* **[4]**) a été libéré et le flambeau a été repris par les autres acteurs. Résultat : aujourd'hui, nous assistons à une multiplication des utilisateurs et deux autres PDK **[5]** sont maintenant librement disponibles à tous !

Les débutants peuvent aujourd'hui facilement créer un tout petit circuit, comme celui de la figure 1, intégré par le projet Tiny Tapeout **[6]**. Ce galop d'essai à quelques centaines de dollars se transforme parfois en une aventure à dix mille dollars pour occuper toute la surface de la puce, en utilisant librement les dix millimètres carrés (en passant directement par eFabless **[7]**). Mille dollars par millimètre carré peut sembler délirant (comparé au marché immobilier), mais ce forfait comprend typiquement une centaine d'exemplaires. Et c'est un bond considérable, d'autant plus que tout le *workflow* est intégré dans un serveur `git` public. Dire qu'en 2001, je me plaignais qu'il n'était pas possible de concevoir un circuit intégré avec des outils libres **[8]** alors qu'aujourd'hui, ils sont plébiscités par les gouvernements !

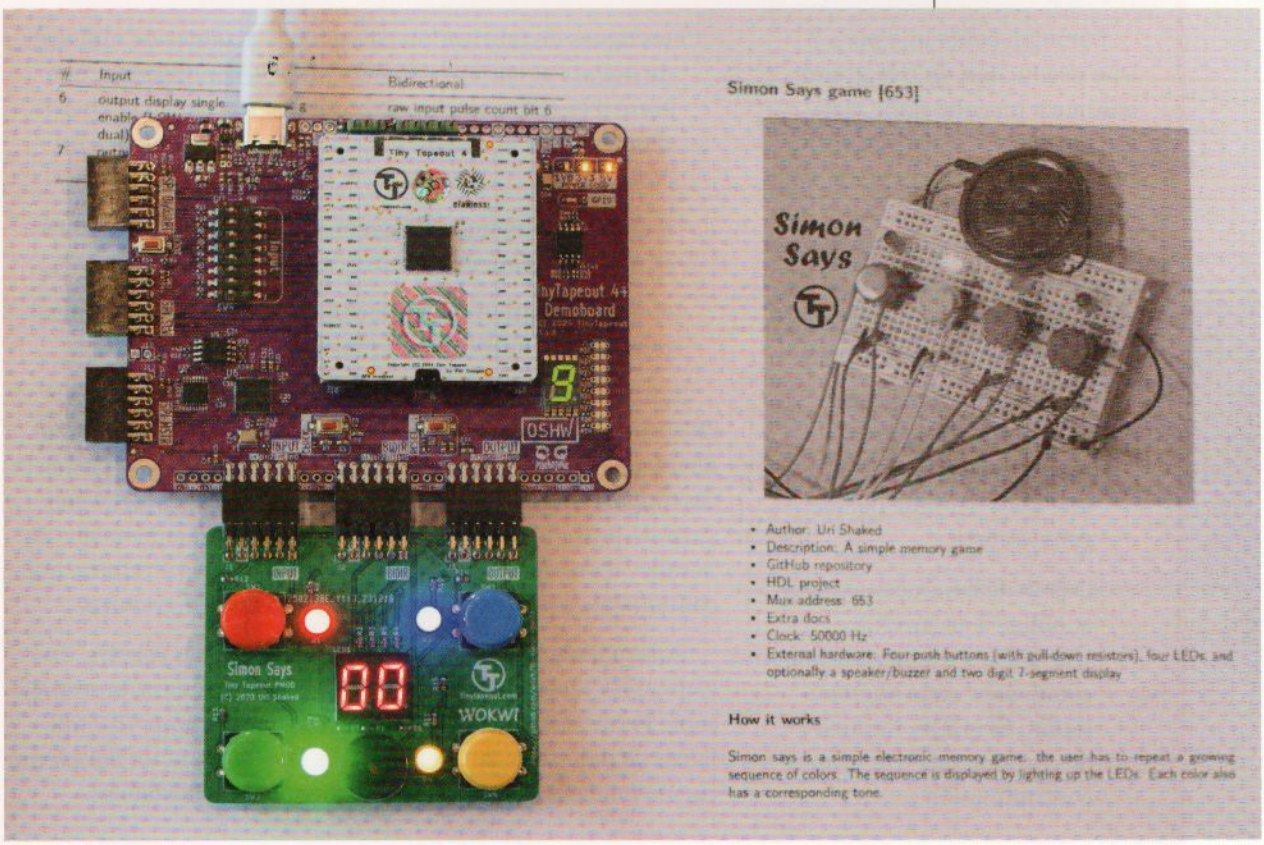


Figure 1 : Vous aussi, vous pouvez faire votre propre puce ! Le projet TinyTapeout permet de concevoir des circuits électroniques de quelques centaines ou milliers de portes, puis de les faire réaliser sur un circuit intégré partagé avec une centaine d'autres projets (pour quelques centaines d'euros). Ici, l'un de ces projets est un « Simon » doté de sa carte d'extension spéciale (pour les LED, les boutons et le buzzer), présenté sur le livret qui détaille les dizaines de projets hébergés par la puce en question.

1. LA NOUVELLE RÈGLE DU JEU

La pandémie et les guerres économiques ont électrisé tous les États du globe durant ces dernières années ; chaque pays y va de son programme de promotion des circuits intégrés locaux, de la conception à la fabrication, dans une optique de « souveraineté » puisqu'ils ont été reconnus comme essentiels à notre société moderne :

- La Chine (lire : le Parti communiste chinois) avait déjà plus ou moins son « plan » pour rattraper son retard sur les toutes dernières technologies, subventionnant les usines à tour de bras. Les lois étaient déjà conçues pour s'approprier les technologies étrangères en obligeant des « transferts de compétences » pour toute entreprise non chinoise désirant s'établir sur le territoire. Cela existait déjà au Japon, mais demandez à ARM comment ils se sont fait voler leurs brevets **[9]**. Cependant, l'injection de budgets faramineux (financés à l'échelle nationale et régionale) n'a pas eu tous les effets escomptés, car cela a surtout attiré les monteurs de projets financiers (aux motivations et méthodes douteuses, en pleine bulle spéculative), au lieu d'industriels compétents. Les pressions politiques et la corruption poussent de nombreuses usines à la faillite alors qu'elles sortent à peine de terre **[10]**.

« LE JUSTE PRIX »

Combien coûte une puce ? Une production en très grand volume a un coût, donc logiquement un prix, *relativement raisonnable* et dégressif. Ce qui a changé depuis l'an 2000, c'est l'investissement exponentiel pour son développement, atteignant des sommes délirantes, ce qui a plusieurs conséquences :

- de moins en moins de sociétés ont le budget requis pour développer une puce de toute dernière génération ;
- le prix de ces puces explose et oblige à en produire considérablement plus pour compenser cette augmentation, ce qui les limite à des produits de très grande consommation ;
- les lignes de fabrication coûtent des dizaines de milliards de dollars et ont une capacité limitée, conduisant à une sorte d'enchère secrète pour réserver la capacité de production pour un client, parfois même pour bloquer un concurrent ;
- il en résulte un appauvrissement de la diversité de l'offre et une réduction du nombre des plateformes majeures ;
- et une stagnation sur les technologies « déjà suffisantes » pour les applications peu sensibles ou matures.

Mais de combien parle-t-on ? Les coûts sont difficiles à chiffrer, car cela dépend de la complexité donc du temps de développement, du nombre d'ingénieurs assignés au projet, du support informatique (acheter et maintenir les serveurs surpuissants et leurs logiciels tout aussi chers) et bien sûr, du nombre d'erreurs donc du nombre de masques à refaire, qui coûtent chacun une fortune, selon la finesse de gravure.

Puisqu'il n'y a pas d'absolu, parlons en termes relatifs. Une estimation récente a été partagée par Sandra Rivera, la responsable FPGA chez Intel (suite au rachat d'Altera pour 16,7 milliards de dollars en 2015). Selon elle, les FPGA ont un bel avenir, car les fonderies comme TSMC deviennent inabordables.

L'explosion de la bulle immobilière (avec la chute d'Evergrande) et les réactions à la pandémie ont semé le doute dans l'industrie. En plus, les USA ont été échaudés par l'espionnage industriel éhonté et ont décidé d'un embargo sur toutes les technologies de pointe. ASML, NVIDIA et TSMC sont parmi les plus touchés par l'interdiction de vendre leurs joyaux. La guerre était déclarée et l'orgueil chinois piqué au vif : ils veulent « leurs puces » !

- Le gouvernement américain prend conscience de sa fragilité, suite à la crise industrielle provoquée par la COVID, les retards causés par l'administration Trump, et surtout les tensions politiques en mer de Chine qui font craindre de ne plus disposer des précieuses puces dernier cri fabriquées en majorité par TSMC à Taïwan (sans oublier le Japon et la Corée du Sud juste à côté). Le « *CHIPS and Science Act* » est voté en 2022 **[11]** et alloue 39 milliards de dollars pour inciter les industriels à installer des usines de dernière génération sur le territoire des USA. Des subventions énormes sont accordées entre autres à Intel (qui se met à ouvrir ses usines à des « partenaires » pour concurrencer TSMC, comme Samsung l'avait fait plus tôt, dans l'espoir de mieux rentabiliser ses usines) et TSMC (menacé par lesdites prétentions territoriales chinoises).

Les clients FPGA d'Intel partagent leurs difficultés, et Intel est déjà au courant des tarifs (qui sont jalousement gardés par des NDA), puisqu'il est lui aussi client. Une fourchette de budget a fini par percoler dans un article de Timothy Prickett Morgan :

> The thing that is in the favor of FPGAs and the[sic] mitigates against the massive migration to ASICs is the increasing cost of developing for each process node, and she rattled them off using Taiwan Semiconductor Manufacturing Co as the example, which was fun:
> - $40 million for a 28 nanometer device,
> - $100 million for a 14 nanometer device,
> - $300 million to $400 million for an N7 device,
> - $600 million for an N3 device, and
> - $800 million for an N2 device.

Évidemment, ces nombres sont à considérer avec prudence, car Intel veut concurrencer TSMC en se séparant de ses propres fonderies.

On peut quand même estimer que pour le prix d'une puce de toute dernière génération, on peut s'en acheter deux en technologie plus ancienne, ou même s'offrir une usine âgée de trente ans. Et il s'agit du développement uniquement, car chaque puce produite ajoute ses propres coûts unitaires, pour le silicium, la gravure, l'encapsulation et tous les tests ! C'est donc logique que TinyTapeout et d'autres restent sur des technologies anciennes, bien plus économiques et « suffisamment bonnes ». Il y a déjà tellement à faire à 130 nanomètres !

Source : Timothy Prickett Morgan, *« Altera Is Being Realistic About FPGA Compute In The Datacenter »* **https://www.nextplatform.com/2024/09/26/altera-is-being-realistic-about-fpga-compute-in-the-datacenter/** 26 septembre 2024.

- L'Union européenne emboîte le pas et crée son propre programme similaire : l'« *European Chips Act* » adopté en 2023 **[12]**. Les programmes existants passent à la vitesse supérieure et Intel est courtisé pour installer une de ses dernières usines sur notre continent. C'est Magdebourg qui est choisi, entre autres grâce à dix milliards d'euros d'aide du gouvernement allemand. Intel prévoit aussi de s'étendre en Pologne et en Irlande, mais ses finances ne suivent pas et TSMC est sur ses talons : ESMC s'installe à Dresde **[13]** grâce à un partenariat avec Bosch, Infineon et NXP (qui financent à hauteur de 10 % chacun). Pour aider à former des concepteurs microélectroniques européens, l'Union européenne a aussi lancé *Edu4chip* **[14]** : c'est un programme qui associe des instituts et des sociétés de France (Mines-Telecom Saint-Étienne en fait partie), Allemagne, Danemark, Suède, Finlande, Bulgarie (et j'ai entendu Roumanie et Norvège, sans trouver de confirmation).
- Au Japon se crée l'« *Open Source Utilized Silicon Initiatives* » (Open-SUSI) **[15]**.
- La Suisse aussi se réveille et lance son « *SwissChips* » en février 2024 **[16]**.
- Le Royaume-Uni tente de se réorganiser après le Brexit, cherchant à reconstituer un rôle de prestige, mais n'en a pas les moyens. De plus, le Brexit a coupé l'île de précieux liens avec l'Europe, l'allié étasunien pose un risque de souveraineté (suite à l'embargo imposé à ASML en Chine) et les Chinois investissent massivement dans l'ombre **[17]**. Heureusement qu'il leur reste ARM, aujourd'hui coté en bourse.
- Les pays du Golfe veulent être de la partie pour diversifier leurs investissements de pétrodollars : l'Arabie Saoudite lance son « *National Semiconductor Hub* » **[18]** dans l'espoir d'attirer une cinquantaine de sociétés de conception, ainsi que des « talents », en mettant sur la table un quart de milliard de dollars de fonds d'investissement. Ce sujet fait aussi l'objet d'un récapitulatif par Asianometry **[19]**.
- La Russie continue de prétendre qu'*elle aussi y arrivera bientôt, promis*. D'ici 2030, la « Mère Patrie » pourra graver toute seule comme une grande en 28 nm **[20]**, puisque TSMC applique les sanctions internationales. Pour l'instant, ce sont les gentils voisins chinois qui assurent l'intérim, mais la gravure n'est rien sans une technologie d'encapsulation appropriée **[21]**. Pendant ce temps, le marché noir reprend « comme au bon vieux temps » et il se dit que des composants de chars sont obtenus en désossant des équipements électroménagers.
- Neuf pays d'Amérique du Sud resserrent leurs liens au sein de « *Latin Practice* » **[22]**.
- Sans oublier la *Chip Alliance* gérée depuis 2019 par la Linux Foundation **[23]**.

Ce sont les initiatives les plus médiatisées, mais comme Andrew Kahng le remarque dans sa présentation **[24]**, *« il n'y a pas d'open source chinois/américain/européen »*. Par contre, il y a des différences de soutien selon les régions. Et on peut presque parler d'une nouvelle révolution industrielle, car les règles ont changé brusquement, toutes les cartes ont été redistribuées, l'intégration verticale comme la consolidation industrielle et financière sont contournées (pour l'instant) grâce à l'*open source*. Tout le monde est sur le pont, car chacun a de nouveau sa chance et veut sa part, comme dans un nouveau Far West.

Simultanément, depuis la révolution ChatGPT, tous les investissements ont basculé de la crypto/*blockchain* à l'« intelligence artificielle » (quoi que

cela signifie) qui a l'heureuse propriété de recycler beaucoup de technologies et d'infrastructures mises en place par les cryptomineurs. Les actions NVIDIA se sont encore plus envolées (atteignant une capitalisation comparable à Apple) grâce à des prévisions radieuses sur l'explosion de la puissance de calcul nécessaire pour « soutenir l'IA » : McKinsey prévoit au moins une multiplication par 100 d'ici cinq ans. Avec de telles perspectives, l'Union européenne songe à détourner son financement des projets *open source* **[25]**.

Toutes les mégaconstructions mentionnées plus haut font courir le risque d'une nouvelle crise de surproduction d'ici quelques années, que ni Apple ni Nvidia ne pourront probablement absorber. L'espoir est que le marché devienne plus compétitif, mais toutes les entreprises seront-elles rentables ? Le prochain cycle industriel pourrait se terminer bientôt, puisque les voix s'élèvent depuis juillet 2024 pour prévenir les investisseurs sur l'exagération des promesses de l'IA : quand la bulle éclatera-t-elle ?

À l'heure où j'écris ces lignes, Intel vient justement d'annoncer une « mesure de réduction des coûts » qui licenciera un employé sur sept d'ici novembre **[26]**. Mécaniquement, son action a baissé d'un tiers en quelques heures, retournant à sa valeur de 2013. La direction est en crise*⁰. Même si la cause officielle est d'avoir « raté le coche de l'IA », d'autres événements simultanés (comme Apple trahi par B-H, Google coupable de monopole ou les soucis au Japon) ont poussé les places boursières de trois continents à chuter significativement, comme en écho à 2001. Pas de krach boursier pour l'instant, Intel n'a pas coulé, mais cela compromet ses plans d'expansion et sa capacité à recevoir des subventions. Parallèlement, on s'inquiète sur la surévaluation de NVIDIA, société *fabless* donc qui n'a même pas ses propres usines, contrairement à Intel.

Toujours du côté de la rentabilité, les usines établies proposent plus longtemps des technologies « mûres », moins performantes mais beaucoup moins chères, à plus de clients grâce à la libération de leur PDK : c'est le pari de GlobalFoundries, qui a suivi l'exemple de SkyWater.

- En regardant plus attentivement, même si TSMC est actuellement à la pointe sur les toutes dernières technologies de gravure (au point de laisser Intel dans son sillage), une partie significative de son chiffre d'affaires est toujours réalisée avec des géométries plus anciennes comme 45 nm ou 28 nm, déjà rentabilisées et encore appropriées pour beaucoup d'applications modernes. Cela permet de financer ses indispensables recherches (un budget de 5,5 milliards de dollars en 2023), mais même TSMC ne peut pas se permettre de construire de nouveaux bâtiments pour chaque nouvelle génération : ses clients sont poussés à migrer vers des technologies plus récentes en prévision de l'arrêt des plus vieilles lignes de fabrication (moins rentables à cause de la concurrence plus forte). Grâce à sa domination, TSMC n'a aucun intérêt direct à jouer le jeu de l'*open source*.
- Par contre, de nombreux fondeurs plus modestes ne peuvent plus suivre le rythme ahurissant des investissements : ils font donc durer leurs équipements existants au maximum. L'ouverture de leur PDK devient nécessaire pour se démarquer et attirer de nouveaux clients, surtout si les autres fondeurs font de même. C'est pour cela que l'audace de SkyWater a été déterminante : le premier à libérer son PDK a une avance commerciale de quelques années sur les autres, et redore son blason. Ce fut flagrant durant la conférence !

*⁰ Juste avant d'imprimer ce magazine, le président d'Intel Pat Gelsinger vientd'annoncer son « départ à la retraite » alors qu'une partie des subventions allouées à Intel par le CHIPS Act est remise en cause parce que les projets de nouvelles usines sont repoussés.

Sachant qu'une nouvelle usine de pointe coûte aujourd'hui environ *vingt milliards de dollars* avant même de livrer ses premières gaufres, il faut beaucoup de clients pour garder les lignes complètement occupées. De plus, les généreuses subventions des États hébergeurs doivent s'accompagner d'économies pour eux, en particulier sur les licences des logiciels. Heureusement que les universités et les amateurs en développaient depuis des décennies, sans le coût prohibitif des systèmes propriétaires : Cadence et Synopsys facturent très cher, en plus des budgets pour les formations et surtout les plateformes logicielles et matérielles. Bien que leurs résultats soient (pour l'instant) moins optimisés, les logiciels *open source* sont bien plus faciles à bricoler, l'industrie pense souvent « on verra plus tard pour contribuer des améliorations en retour ».

De plus, une expansion industrielle si brusque crée des tensions sur le marché du travail : les entreprises peinent à recruter*1 ! C'est particulièrement problématique dans un monde tellement sous l'emprise du secret industriel (les *Non-Disclosure Agreements* sont la règle) qu'un employé ne peut pas vraiment parler de son travail sur son CV, par exemple. Les outils à la mode peuvent changer d'un coup et il faut alors former des dizaines de milliers de nouveaux techniciens, sur des technologies qui changent tous les cinq ans environ. Les programmes comme Edu4chip ne suffiront pas. La continuité inhérente des projets *open source* réduit la déprécation des compétences et permet d'envisager des projets à plus long terme, sans crainte que les outils nous trahissent ou nous prennent en otage.

Mais les logiciels d'EDA ne sont rien sans les PDK. L'industrie microélectronique est tellement prisonnière des NDA qu'il est aujourd'hui impossible de comparer quoi que ce soit, ou de reproduire des résultats, publiés tronqués dans les journaux académiques. Comme dans plusieurs domaines scientifiques, on craint une « crise de reproductibilité » qui rendrait toute recherche caduque ou vaine. Un exemple typique nous en a été offert par l'ETH de Zurich lors de la période de questions-réponses, après avoir présenté un de leurs super-projets :

Public : Quel outil commercial utilisez-vous ?
ETH : je ne crois pas que je peux le dire.

Les projets libres n'ont pas tous ces inconvénients et ont même d'autres avantages, ce qui les rend enfin légitimes auprès des « décideurs ». Il n'est plus mal vu d'être un *hacker* !

2. PETITE CONFÉRENCE, GRANDS ENJEUX

C'est dans cette conjoncture que s'ouvre FSiC 2024, le 19 juin 2024. Le calendrier de bouclage de Diamond n'a pas permis de publier cet article plus tôt, mais le délai a permis de prendre un peu de recul et de relativiser l'optimisme.

L'amphithéâtre 43 a accueilli environ 120 personnes (dont la photo de groupe est sur la figure 2), françaises bien sûr, mais aussi venues des USA, Inde, Allemagne, Pays-Bas, Grèce, Égypte, Finlande, Belgique, Chili, et probablement d'autres. Étrangement, le Royaume-Uni ne semble pas représenté, probablement à cause des conséquences du Brexit.

L'inscription est gratuite par e-mail, sans autre formalité : c'est une rencontre très pointue, mais sans barrière d'accès. Les débutants

*1 Alors même qu'Intel et d'autres « dégraissent ».

Figure 2 : La plupart des 120 participants à la conférence ont posé pour une photo de groupe, lors de la pause méridienne du premier jour. Merci à eux et à tous les autres !

et les vétérans échangent librement, comme le montre la figure 3. Il manque seulement Tim Ansell pour que la galerie soit complète.

On note aussi qu'aucun « acteur majeur » de l'industrie n'est présent, donc pas d'Intel, d'AMD, de TSMC ou de Samsung, mais Skywater et Global Foundries sont souvent mentionnés puisque eFabless (présent) peut dorénavant les utiliser. Les nombreuses présentations s'enchaînent avec un rythme assez soutenu, une toutes les 20 à 30 minutes environ (au début). Forcément, des vignettes autocollantes circulent, dont certaines sont présentées sur la figure 4, page suivante.

On y parle beaucoup d'EDA, sigle de « *Electronic Design Automation* » qui signifie plus ou moins « automatisation de la conception électronique » et désigne le monde des outils de

Figure 3 : À gauche, Jérémy Alcim est un jeune ingénieur de talent, adepte de parallélisme, d'IA, de HDL et de C++, qui explore l'EDA Libre depuis un certain article dans Hackable [3]. Il rencontre le vétéran Tim Edwards (à droite) : il est (entre autres) le mainteneur actuel de l'outil historique MAGIC (http://opencircuitdesign.com/) ainsi que « Senior Vice President of Analog and Design » chez eFabless, où il a contribué à l'ouverture du PDK SkyWater et travaille sur le Caravel Panamax (voir la figure 12).

Figure 4 : Pas de conférence sans autocollants ! C'est toujours mieux d'en avoir un peu d'avance, pour échanger plus tard.

conception des circuits microélectroniques. Si l'EDA *open source* hérite directement de la philosophie du monde des logiciels libres, nous avons déjà dit que l'EDA commerciale interdit le partage des fichiers (scripts, paramètres, sources ou objets) et même le *benchmarking* ! Et jusqu'à très récemment, tous les PDK étaient strictement secrets, ce qui empêchait de reproduire ou de comparer des résultats publiés dans les journaux académiques. L'ouverture du PDK SkyWater a donc eu un impact profond sur l'industrie et certaines présentations nous en montrent les effets tangibles.

Pour comprendre l'intérêt de cette rencontre internationale, rien ne vaut la toute première présentation **[27]** : Matt Venn y résume l'état général des développements ouverts/libres et leurs enjeux. Je ne vais pas trop la paraphraser puisque la conférence a été filmée, n'hésitez pas à aller examiner sa présentation en ligne !

Matt a une perspective très optimiste de l'évolution récente et des perspectives pour « nous les bricoleurs », grâce à sa position stratégique puisqu'il organise le projet Tiny Tapeout (qu'il nous présente fièrement dans l'amphithéâtre sur la figure 5). Selon lui, les coûts de conception des circuits intégrés vont continuer de chuter, les outils seront de plus en plus faciles à obtenir et à installer, et les ateliers d'initiation se multiplient : plus de 280 kits Tiny Tapeout ont déjà été livrés !

Je retiens en particulier qu'à la vingtième page de sa présentation, Matt évoque la « métrique PPA » qui est typiquement utilisée dans l'industrie pour caractériser et comparer des technologies, selon leur **P**uissance, leur **P**erformance et l'**A**ire requises pour réaliser un circuit donné. À sa place, il propose une autre métrique qu'il appelle RED : **R**eproductibilité, **E**fficacité (du développement : installation aisée des outils et partage du code source) et **D**ocumentation (et en plus, cela peut devenir un acronyme, et non plus un sigle). Sous cet angle, les outils *open source* et la communauté sont loin devant l'industrie, puisque cette dernière s'est enfermée dans la culture du NDA. Tout change selon la perspective !

- Compte tenu du choix considérable offert par les fonderies aujourd'hui, allant du micromètre au nanomètre, la performance absolue d'un circuit ne compte plus autant qu'avant, car ce n'est alors qu'une question de budget. Mais sans reproductibilité, il est impossible d'itérer le développement d'un circuit pour l'améliorer.
- La gestion fluide des logiciels, non contraints par des serveurs de licences ou d'autres obligations contractuelles contre-productives, libère les développeurs qui peuvent se concentrer sur leur travail et moins lutter contre leurs propres outils (ou du moins, avoir plus de chances de gagner).
- Quant à la documentation, même si c'est souvent une considération « après coup » pour la plupart des développeurs, sa disponibilité pour tous sur Internet favorise encore plus la dissémination, la familiarité et l'évolution, alors qu'un produit propriétaire sous NDA est figé dans le temps.

3. RÉINVENTONS LA ROUE !

C'est un air connu pour les lecteurs assidus, mais le refrain a encore été chanté dans cette conférence.

Comme lors de toute « explosion cambrienne », l'augmentation de la variété des espèces (ou des projets) entraîne aussi de la redondance. C'est inhérent à l'*open source* et à la liberté de développer, car chacun peut fixer ou choisir ses propres besoins, au lieu de se plier à un outil qui impose son univers avec ses idiosyncrasies. Forcément, lorsque des équipes séparées ont besoin de la même

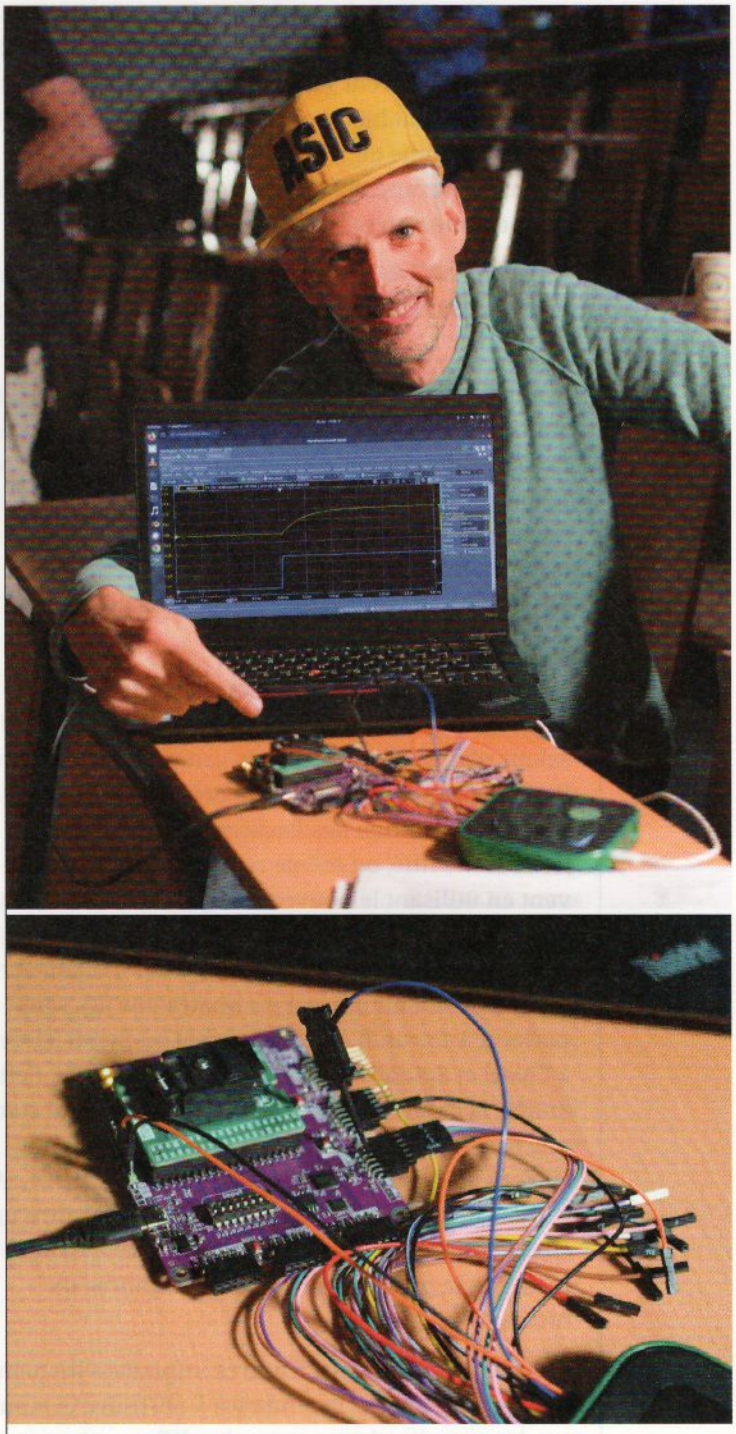


Figure 5 : Matt Venn est toujours prêt à faire une démonstration de la dernière fournée de TinyTapeout, avec son ordinateur, son analyseur logique, et les cartes d'adaptation, dont une dotée d'un support ZIF pour tester les puces sans les souder.

chose, elles développent des solutions similaires : c'est la *convergence*. Le travail est dupliqué, mais les différences peuvent grandir au point que les résultats deviennent incompatibles ou incomparables (comme Gnome ou KDE). À ce sujet, Andrew B. Kahng nous rappelle à la fin de sa présentation que ce phénomène n'est pas nouveau **[24]** :

« The first SHARE Design Automation Workshop was held in 1964 (SHARE = Society to Help Avoid Redundant Effort). »

Une bonne partie des duplications est due à l'ignorance : si l'on ignore que la roue existe, on finit par la réinventer. D'où l'importance de cette conférence : ignoriez-vous qu'elle avait eu lieu ? Pour vous rattraper, allez explorer le site web qui héberge toutes les présentations **[1]**, vous pourriez y découvrir des outils que vous auriez été tenté de développer vous-même.

Une autre raison de dupliquer un outil est le « syndrome NIH » (*Not Invented Here*) : soit un utilisateur n'aime pas, soit il ne le comprend pas ou bien il n'accepte pas les choix d'un développeur. Dans la plus pure tradition des *hackers*, le code source est *forké* et modifié, ou même repris depuis zéro, souvent en utilisant le nouveau langage à la mode, saupoudré de concepts équivalents ou de néologismes. C'est ce qui s'est produit avec Migen et nMigen écrits en Python, mentionnés il y a quatre ans **[3]**, sans parler d'Amaranth, magma, myhdl ou pymtl. Quant à Chisel et SpinalHDL, ils sont basés sur Scala. Cette année, la conférence a présenté Clash **[28]** qui compile et synthétise des portes logiques à partir de Haskell cette fois, et exporte en Verilog ou VHDL. Au rythme actuel, nous risquons de voir apparaître un HDL en Rust dans peu de temps, mais cela montre juste que la puissance de VHDL est ignorée ou incomprise.

Nous y voyons aussi d'autres influences du monde des logiciels libres : la tendance à abstraire et ajouter des complications « pour simplifier », alors que l'EDA traite des systèmes très tangibles qu'il faut pouvoir contrôler de A à Z, à tous les niveaux de description. Trop d'abstractions tuent l'abstraction, trop de langages tuent les langages : j'ai déjà expliqué cela dans le chapitre 6 d'un article précédent **[29]**.

Sinon, comme dans le cas du noyau Linux, un outil dupliqué a pu commencer comme un bricolage sur le coin d'une table « juste pour voir » puis a attiré des contributeurs et utilisateurs inattendus. Ce sont typiquement les projets les plus simples qui ont le plus de succès et de croissance… C'est moins probable dans le microcosme de l'EDA, mais nous voyons déjà des « masses critiques » se constituer.

Par exemple, alors que dans le monde des logiciels libres, nous avons vu les débats « Vim contre Emacs » ou « LLVM contre GCC », l'EDA libre balance entre OpenROAD et Coriolis.

- OpenROAD signifie « *Foundations and* **R**ealization *of* **O**pen, **A**ccessible **D**esign ». C'est la solution la plus adoptée et développée actuellement ; son historique est expliqué par son responsable dans une présentation passionnante **[24]**. Pour résumer, c'est issu d'un appel à projets DARPA en 2018 et après 17 millions de dollars de subventions, la fondation est créée depuis 2023. Le support commercial est fourni par **https://precisioninno.com** en partenariat avec Intel, Global Foundries, TSMC, SkyWater et Google. Côté logiciel, c'est une sorte de *patchwork* très flexible formé de nombreux outils existants (dont Yosys) ou expérimentaux ; d'ailleurs, certaines présentations décrivaient des modules supplémentaires pour

ajouter des fonctions (comme le *Design For Test*). Grâce à cela, selon A. B. Kahng, ce n'est plus Cadence, mais l'*open source* qui fait avancer le monde de l'EDA. Cela augure d'un « *brain drain* » comme cela s'est produit au début du millénaire, lorsque Linux a attiré progressivement beaucoup de développeurs contraints jusque-là par Windows.

- Coriolis est le nouveau placeur et routeur d'Alliance, la suite d'outils développés depuis 1990 à Jussieu au LIP6. Certains outils historiques et idiosyncratiques sont remplacés par de nouvelles alternatives *conformes aux standards*, comme GHDL et Yosys. Coriolis est peu représenté, mais il est très capable, comme l'a montré la fabrication du prototype de Libre-SOC (voir à la partie 7) rendue possible grâce au soutien de NLnet entre 2019 et 2022 **[30]**. Coriolis n'est pas encore validé pour des géométries inférieures à 130 nm. Et comme l'équipe de développement et de maintenance est minimale, Coriolis ne peut pas supporter toutes les distributions GNU/Linux (un grief qui a fait l'objet d'une des présentations **[55]**).

Du côté des langages HDL (« de description de matériel »), c'est toujours Verilog qui est utilisé en général (« parce que c'est plus simple » et comme langage intermédiaire avant synthèse) mais VHDL résiste, en particulier grâce à GHDL autour duquel tous les « VHDListes » se rallient, à la manière du ralliement autour de GCC il y a trente ans, délaissant les solutions commerciales plus rigides. GNU/Linux Magazine a publié un entretien avec son auteur, Tristan Gingold **[31]** il y a déjà 14 ans. Après une belle carrière chez AdaCore, il travaille depuis 2017 pour le CERN. Il nous a présenté **[32]** l'utilisation de GHDL dans un projet du CERN à base de Xilinx UltraScale. Le mélange de code *open source* avec des outils propriétaires cause encore et toujours des problèmes ubuesques. C'est là que les nombreuses qualités de GHDL brillent : la facilité d'intégration dans un *workflow*, la co-simulation, l'adhérence stricte aux standards, l'absence de toute limite arbitraire (comme le nombre d'instances exécutées simultanément) et VHDL est un « vrai langage »...

4. TRY, FAIL, ITERATE, AND BLINK !

La différence entre amateurs et professionnels est de plus en plus difficile à cerner, comme vous aurez pu le remarquer ces dernières années en lecteur assidu. Le critère essentiel est l'ampleur des moyens financiers disponibles, alors que les technologies sont plus abordables et accessibles, et les entreprises utilisent plus d'« outils amateurs ». Vous avez certainement remarqué la facilité déconcertante à réaliser des prototypes ou des petites séries de circuits imprimés, impensable il y a vingt ans. Non seulement les logiciels se sont démocratisés (entre autres grâce aux logiciels libres et aux versions gratuites de certains logiciels métiers), mais les industriels (surtout chinois) ont réduit encore plus la barrière d'entrée et les prix (vive le *dumping*) pour réaliser nos propres projets. Les PCB sont devenus une « commodité » et les composants basiques ont presque autant de valeur que des grains de sable !*2

*2 Et même moins, puisque le temps de se baisser pour chercher une résistance au format 01005 (soit 0,4 mm×0,2 mm) tombée par terre coûte plus cher à l'employeur que le composant lui-même, ce qui lui vaut d'être qualifié de « poussière » dans le métier.

Figure 6 : Le processeur de Tobias n'est pas intégré, mais à base de portes « discrètes », en boîtier en montage en surface. Le circuit imprimé est très large et pour économiser du budget, les portes sont soudées à la main, une à une, et chaque sortie est reliée à une LED pour vérifier son fonctionnement. Un analyseur logique injecte des données dans la chaîne de test pour automatiser la détection des fautes, c'est ce que nous voyons allumé sur cette image. D'autres diront que c'est une installation d'art abstrait, ne les écoutez pas.

Cela a inspiré certains, dont Tim Böscke[*3] en 2022, à détourner OpenROAD pour réaliser des circuits imprimés presque automatiquement à partir d'une description HDL **[33]**. Lorsque l'on peut générer son propre fichier Liberty (qui décrit les portes logiques proposées par une technologie particulière), rien n'oblige à ce que les portes soient destinées à un circuit intégré (qu'il soit bipolaire, CMOS, ECL ou autre). On a donc vu apparaître des prototypes de processeurs minimalistes « discrets », façon *MOnSter6502* **[34]** mais générés automatiquement, comme des ASIC ! Ils sont réalisés à base de transistors (MOSFET ou bipolaires), de résistances, de diodes, de LED (*LED-Transistor-Logic*, LTL **[35]**) et en poussant encore un peu, on peut même utiliser le NE555 comme porte logique **[36]** !

Pour couronner le tout, les fabricants de PCB (qui s'amalgament naturellement avec les distributeurs de composants) sont heureux de placer et souder à la machine des centaines ou milliers de composants CMS à votre place, pour un surcoût *presque marginal* ! C'est un terrain de jeux fabuleux, presque moins cher et tellement plus impressionnant que Tiny Tapeout. Le résultat, bien que peu performant, est palpable, facile à déverminer ou réparer, et décoratif : il ne demande qu'à être encadré et accroché à un mur.

*[3] Encore un autre Tim, mais celui-ci est connu sous le pseudonyme CPLDCPU.

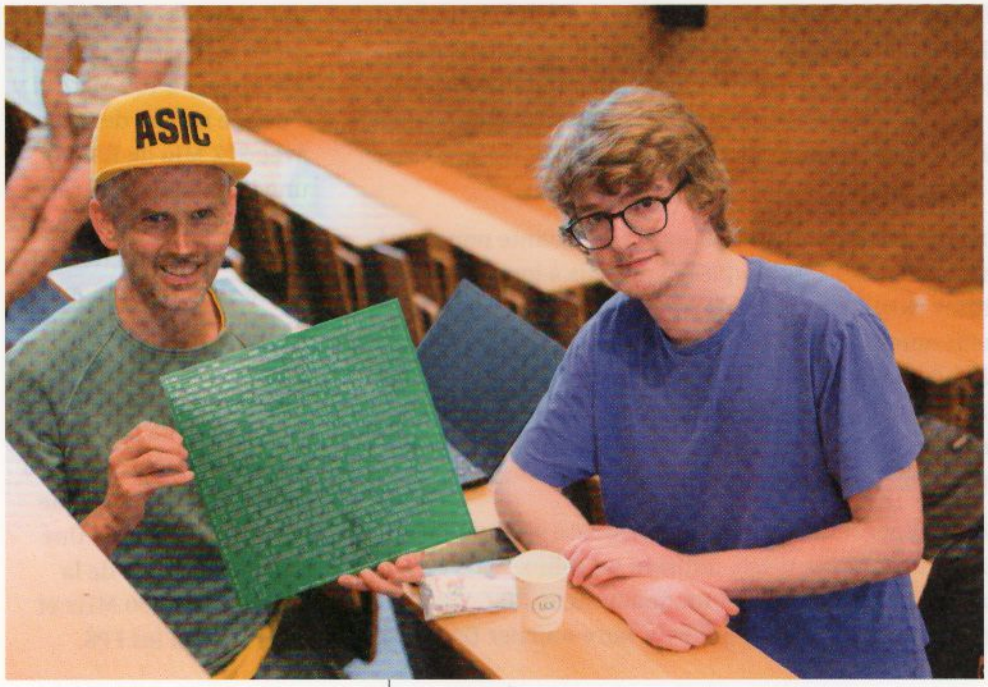


Figure 7 : Tobias Senti discute avec Matt Venn, à qui il offre un exemplaire du circuit imprimé de son « processeur désintégré ». Ensuite, à Matt de souder les milliers de composants, s'il veut que ça fonctionne !

C'est ce que Tobias Senti, étudiant à l'ETH de Zurich, a voulu essayer par lui-même **[37]**. Il a compilé un microprocesseur en bricolant Yosys, Verilator, OpenROAD et KiCad. Au lieu d'utiliser des transistors, il a créé une bibliothèque de portes logiques (ledit fichier Liberty) calquant cette fois-ci les fonctions des membres de la famille 74LVC1Gxx, qui sont des circuits intégrés en boîtier SOT23 ou SC70 contenant une seule porte logique. Le résultat est impressionnant (figures 6 et 7), même si le montage manuel n'était pas terminé. Tobias a aussi eu la bonne idée d'insérer une *scanchain* pour tester les circuits au fur et à mesure de l'assemblage.

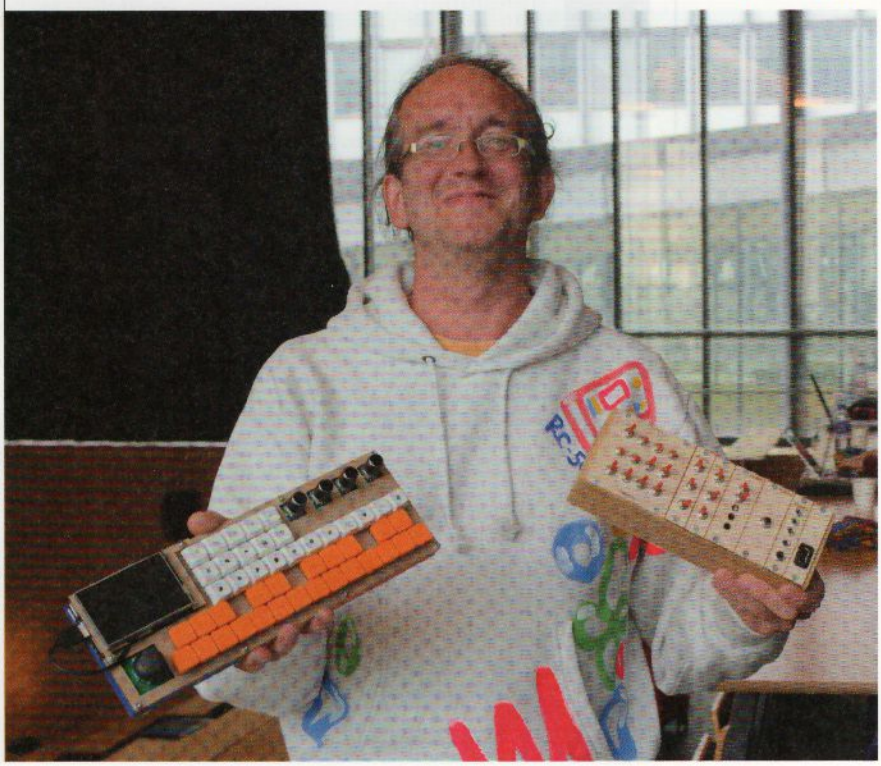


Figure 8 : Thorsten Knoll crée des instruments de musique électronique. La filière d'EDA open source lui permet d'intégrer ses circuits dans des puces au lieu de rester tributaire des fabricants de FPGA. Ces derniers ne servent plus qu'au prototypage.

Un autre bricoleur invétéré est venu d'Allemagne, bien qu'il n'ait pas fait de présentation. Parmi ses nombreux projets, Thorsten Knoll a développé un circuit de génération de musique dans un FPGA (figure 8, page précédente), qu'il va bientôt immortaliser dans un circuit intégré grâce aux nouvelles méthodes.

Encore une démonstration impressionnante nous a été offerte par Charles Papon **[38]**. Il a remis à jour le *softcore* VexRiscv (NDLR : voir article sur Linux et LiteX dans le présent numéro) en utilisant SpinalHDL : ce dernier est un *fork* (spirituel) de Chisel depuis fin 2014. Il est écrit en Scala, qui est un langage statiquement typé reposant sur la *Java Virtual Machine*, et génère du VHDL ou du Verilog (c'est ce dernier que Charles utilise).

Le résultat est un système complet basé sur une carte FPGA *Digilent Nexys Video*, utilisant un Xilinx Artix7 (figure 9). Le processeur est un quadricœur RISC-V (configurable en superscalaire) qui peut faire tourner Debian à une vitesse correcte malgré la modeste fréquence d'horloge de 100 MHz. Il y a peu de risques de *swapper* avec 512 Mio de RAM et une installation dégraissée. Le vidéoprojecteur est connecté (indirectement) sur la sortie graphique de la carte, démontrant qu'elle était totalement fonctionnelle en affichant les *slides* de la présentation. Nous avons aussi eu droit à une petite partie de DOOM à 25 fps ! Une version suivante de la carte atteint 200 MHz et double ainsi les FPS.

Figure 9 : Capture d'écran (au travers d'un dongle HDMI-USB) du système Debian tournant sur un processeur quadricœur RISC-V 100 MHz implémenté dans un FPGA Artix7. Il est largement assez puissant pour afficher la présentation avec xpdf, ou jouer à DOOM sans latence. Presque tout est libre, on n'attend plus que le FPGA ! Source : Charles Papon, domaine public.

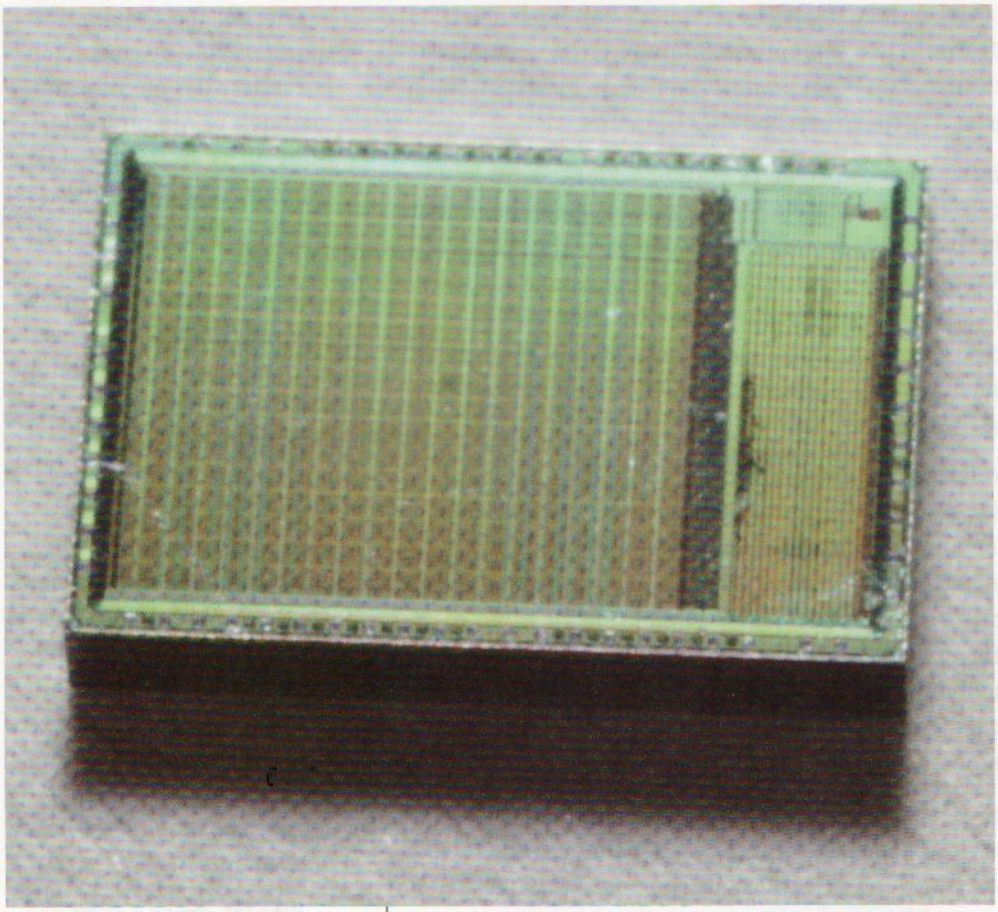


Figure 10 : Macrophotographie d'une puce TinyTapeout nue, où l'on devine le fin maillage sous lequel plus d'une centaine de projets occupent chacun quelques milliers de portes logiques.

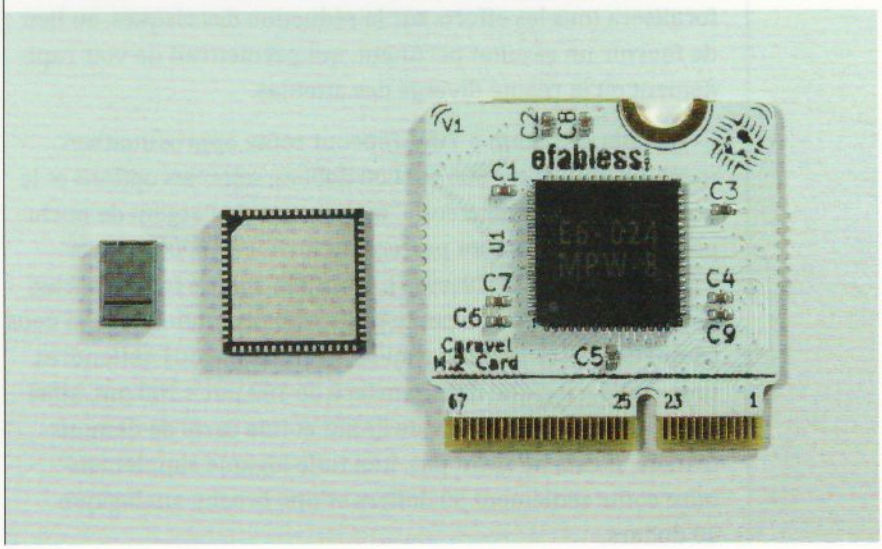


Figure 11 : Après fabrication, la puce TinyTapeout (à gauche) est encapsulée dans un boîtier plastique QFN (au milieu) à peine plus gros que la puce elle-même ! Ce boîtier est ensuite soudé (à l'air chaud) sur un petit circuit imprimé (au format M2, à droite) pour faciliter la connexion des signaux avec un connecteur standardisé.

5. TINY TAPEOUT DEVIENDRA GRAND

Vous avez compris qu'il y avait du Tiny Tapeout partout. Merci encore à Thorsten de m'avoir fourni un échantillon d'une puce nue (figure 10) et une version encapsulée (figure 11), afin que je puisse les photographier correctement à la maison !

Lors de la conférence, la quatrième fournée venait d'être livrée et la huitième

bouclera (*tape out* en anglais : « envoi des bandes magnétiques à l'usine ») le 6 septembre. Il faut environ six mois entre la clôture d'un *circuit multiprojets* et sa livraison, selon le calendrier à **https://tinytapeout.com/runs**.

Le cafouillage de la première édition est largement loin derrière : les logiciels ont été reconfigurés correctement pour fournir des puces qui fonctionnent, cette fois. Mais ce faux départ n'était pas un désastre, car même si la puce arrivait seulement à faire clignoter une LED, la fierté de posséder un circuit intégré qu'on a conçu soi-même compensait largement ! Cela augure de porte-clés en série limitée, et les éditions suivantes ont livré les circuits promis au départ.

Aujourd'hui, la machine est rodée, la file d'attente des projets est bien remplie et les utilisateurs expérimentent tous azimuts, encouragés par les limites techniques qui sont progressivement repoussées par les organisateurs. Par exemple, aujourd'hui, un utilisateur peut réclamer plusieurs blocs consécutifs au lieu d'un, en payant un surcoût proportionnel. Ils disposent ainsi de plus de place pour réaliser des fonctions plus complexes comme un microprocesseur, un accélérateur de Mandelbrot ou un bloc cryptologique.

Les technologies utilisées par Tiny Tapeout et eFabless ne sont pas les plus à la pointe, mais elles sont rendues très abordables, puisque l'objectif est de pouvoir expérimenter, de constituer une bibliothèque de circuits *libres de NDA* et surtout de **se permettre de faire des erreurs**. Si un essai coûte trop cher, la peur d'échouer (et donc de perdre de l'argent) focalisera tous les efforts sur la réduction des risques, au lieu de fournir un résultat pertinent, qui permettrait de voir rapidement où la réalité diverge des attentes.

Une participation à Tiny Tapeout coûte approximativement, en tout, entre 500 et 1000 dollars, selon les options et le nombre de puces désirées. C'est presque de l'argent de poche pour certains, grâce au partage du forfait à 10000 dollars « ChipIgnite » de eFabless **[7]**. Pour 300 dollars (plus les frais de port, mais avec une petite réduction promotionnelle si vous soumettez votre projet parmi les premiers), vous obtiendrez une « tuile » logique d'une surface de 160 µm × 100 µm, ainsi qu'un unique exemplaire de l'ASIC et une carte de démonstration. Si cela ne suffit pas, une tuile logique supplémentaire coûte seulement 50 dollars et une broche analogique 40 dollars.

Oui, car il est maintenant possible de concevoir et réaliser des circuits analogiques, ce qui ouvre de nouvelles perspectives ! Le concours Chipalooza 2024 **[39]** a permis de *crowdsourcer* une bibliothèque de modules analogiques indispensables même pour les circuits numériques, comme des oscillateurs, des PLL, des convertisseurs A/N et N/A, ou des détecteurs de perte d'alimentation. D'autres fonctions sont clonées ou explorées, comme le fameux *timer* 555, des amplificateurs opérationnels, des pédales de distorsion... Aussi bien Tiny Tapeout que eFabless bénéficient de ces développements, qui doivent être soumis sur un `git` public en licence libre.

Bientôt, grâce à cette réutilisation et à la modularité, des circuits analogiques et mixtes plus avancés seront aussi faciles à réaliser qu'avec des portes logiques booléennes. La septième fournée (TT07 pour les intimes) contient déjà, parmi une centaine d'autres projets, un RISC-V32I, un amplificateur pour sonde d'électrocardiographie et même des émulateurs de générateurs de sons analogiques vintage d'ordinateurs 8 bits (style AY-3-8913 des CPC, MSX, Atmos, Vectrex...). Les amoureux de *chiptunes*

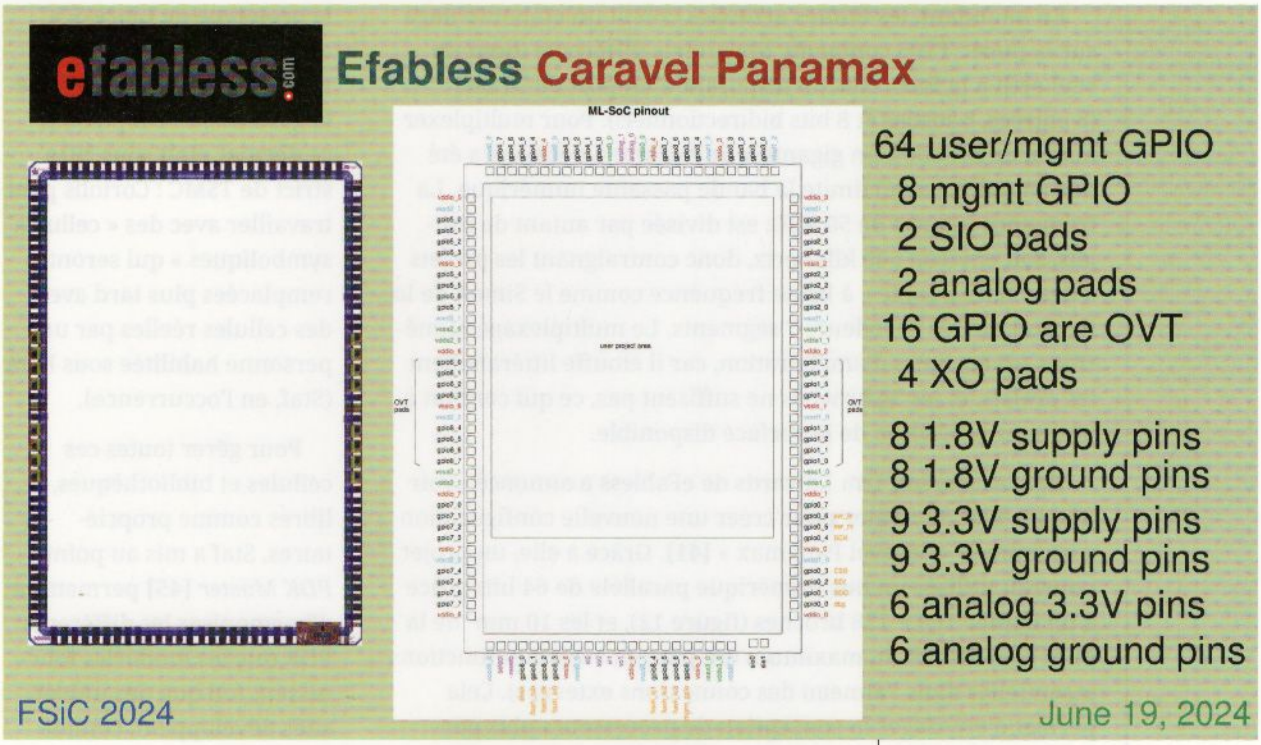


Figure 12 : La configuration « Caravel Panamax » de eFabless permettra d'utiliser au mieux la surface et les broches permises par la puce en technologie 130 nm de SkyWater. Par rapport aux 34 broches de la configuration actuelle, c'est un bond important pour le prototypage de circuits plus complexes et nécessitant beaucoup de bande passante ! D'autant plus que le boîtier TQFP128, bien que plus cher, est plus facile à souder à la main. Source : [41].

et de *sasfépus* réfléchissent déjà à comment remplacer leurs puces d'origine qui vieillissent mal.

N'oublions pas la gestion de la puissance. Une équipe chilienne a montré l'utilisation du PDK SkyWater130 et de Caravel pour réaliser des régulateurs de tension DC/DC totalement intégrés sur une puce. Sur un exemple, la moitié de la surface (environ 5 mm²) est couverte par les condensateurs flottants d'une pompe de charges **[40]** : c'est la preuve du potentiel et de la flexibilité de la technologie de SkyWater. Le but est de fournir des blocs configurables de gestion d'alimentation (PMIC) intégrables dans des puces plus grosses, pour faciliter la vie des concepteurs amateurs. Les circuits intégrés « ouverts » pourront donc optimiser leur performance *et* leur consommation.

Un autre domaine analogique est l'électronique radiofréquence, en particulier pour les communications (radio ou filaire), mais la bande passante des interfaces disponibles sur la puce actuelle ne le permet pas encore. La fournée TT07 (qui sera livrée début 2025) et les suivantes permettront probablement de repousser cette limite.

Après l'électronique de puissance et radiofréquence, il ne restera plus que les MEMS, mais il n'en a pas été question. Je crois que ce sera pour beaucoup plus tard, avec une autre fonderie !

En attendant, les limites actuelles créent un embouteillage sur la puce : TT03 contenait 249 projets différents (dont un seul actif à la fois), chacun disposant d'un port de 24 bits (8 entrées, 8 sorties et 8 bits bidirectionnels). Pour multiplexer autant de broches, un gigantesque registre à décalage a été mis au point, ce qui limite la bande passante numérique. La fréquence interne de 50 MHz est divisée par autant de projets, soit environ 500 kilohertz, donc contraignant les projets à des « bacs à sable » à basse fréquence comme le Simon de la figure 1 ou des décodeurs 7 segments. Le multiplexage numérique est en cours d'amélioration, car il étouffe littéralement les projets, et les 34 broches ne suffisent pas, ce qui conduit à une sous-utilisation de la surface disponible.

Heureusement, Tim Edwards de eFabless a annoncé avoir discuté avec SkyWater pour créer une nouvelle configuration surnommée « Caravel Panamax » **[41]**. Grâce à elle, un projet pourrait utiliser un bus numérique parallèle de 64 bits grâce à un boîtier TQFP 128 broches (figure 12), et les 10 mm^2 de la puce sont libérés au maximum (en plaçant certaines fonctions essentielles dans l'anneau des connexions externes). Cela permet d'envisager la conception de processeurs plus performants ou d'allouer plus de broches pour des projets Tiny Tapeout. Le prix hors Tiny Tapeout (si vous voulez disposer de tout pour vous tout seul) n'est pas encore annoncé, mais probablement supérieur aux 10000 dollars habituels, en raison du boîtier plus grand. Mais cela vaut la peine pour les projets qui ont besoin de beaucoup de bande passante, ou si vous préférez souder à la main !

6. L'AMBITION ALLEMANDE

Une autre présentation fascinante nous a été offerte par Staf Verhaegen **[42]**. Après des années à travailler pour IMEC (l'institut interuniversitaire de recherche microélectronique à Louvain en Belgique), il a fondé **https://chipflow.io**, qui est similaire à eFabless, mais s'occupe aussi de la sous-traitance de la conception des circuits. Pour cela, il a développé ses propres outils en licence libre en Python à partir d'Amaranth (basé sur nMigen, suite à des controverses de droit des marques **[43]**).

Les autres projets de Staf (dont **https://chips4makers.io/**) l'ont amené à collaborer avec les suspects habituels comme Matt Venn (**[44]** pour son cours « *Zero to ASIC* ») ou LIP6 (pour le prototype de Libre-SOC avec Coriolis). Dans ce dernier cas, Staf s'est chargé de la gestion du PDK puisque ce dernier était sous NDA strict de TSMC : Coriolis peut travailler avec des « cellules symboliques » qui seront remplacées plus tard avec des cellules réelles par une personne habilitée sous NDA (Staf, en l'occurrence).

Pour gérer toutes ces cellules et bibliothèques, libres comme propriétaires, Staf a mis au point *PDK Master* **[45]** permettant d'harmoniser les différents PDK que les fonderies fournissent (ou que des universités développent, comme FreePDK), souvent en formats incompatibles entre eux. Il s'est logiquement retrouvé impliqué dans la « libération » de l'OpenPDK de l'iHP **[46]**, fourni cette fois par le « *Leibniz Institute for High Performance Microelectronics* » près de Francfort (voir **https://www.ihp-microelectronics.com/**).

Ce troisième PDK libre permet dorénavant d'accéder à la technologie BiCMOS 130 nm (appelée SG13G2) développée à l'institut, pour les recherches (académiques ou non) et les prototypes. Sur la base d'une technologie CMOS 130 nm, des transistors HBT (*High Performance Heterojunction Bipolar*

Transistors ultrarapide à base de silicium-germanium) sont ajoutés, rendant possibles des circuits intégrés « mixtes » qui combinent de la logique booléenne dense et des traitements des signaux radiofréquences à plusieurs dizaines de gigahertz. Pour résumer, c'est ce qu'on trouve de mieux avant de faire appel à l'arséniure de gallium (AsGa) ou au phosphure d'indium (InP) qui ne sont toujours pas disponibles en fabrication de masse, en particulier parce que les circuits bipolaires ne peuvent pas rétrécir autant que les circuits MOS.

Cette technologie mixte pourrait accélérer certains traitements logiques avec les HBT, gourmands en énergie, et garder les portes CMOS pour les fonctions lentes ou basses consommations (comme les mémoires). Cela permet d'envisager des microprocesseurs plus rapides (Intel avait fait appel au BiCMOS pour le premier Pentium) ou des modems 5G/Wi-Fi/Bluetooth par exemple. On atteint un niveau de sophistication bien supérieur à celui de Tiny Tapeout !

Et puisque nous vivons à l'aube de l'ère de l'« IA », ou du moins des LLM, l'iHP a introduit un *chatbot* (basé sur ChatGPT) dans son PDK pour faciliter la recherche dans la documentation. Ce type d'innovation (ou de bricolage) est un autre avantage de l'ouverture !

Bien que l'OpenPDK de l'iHP ne soit pas le plus complet ou avancé, il rend accessible une technologie de pointe, ce qui devrait bénéficier à tout le monde, et surtout inciter d'autres fonderies commerciales à libérer leurs propres PDK. Mais ce n'est pas le plus surprenant : en collaboration avec Europractice, les universités peuvent utiliser le *process* de l'iHP gratuitement, à condition de rendre leurs fichiers publics sous licence libre. La soumission des projets se fait déjà avec un `git` public, comme Tiny Tapeout/eFabless. L'*originalité* ici est que la « propriété intellectuelle » (IP) est reversée à l'iHP, qui **reste aussi propriétaire des puces** (puisqu'il les subventionne). Un système appelé « *OpenSamples StoreHouse* » a aussi mis en place, pour « prêter » les « échantillons gratuits » des puces fondues, puisqu'il faut bien vérifier qu'elles fonctionnent. C'est une nouvelle organisation étrange, concoctée par les institutionnels pour remplir un objectif évident de formation et de *crowdsourcing*, tout comme le Chipalooza d'eFabless ou même l'initiative originale de Google. Nous verrons dans les années à venir comment cela évolue et si elle fait des émules.

Dans le cas de l'iHP, les projets commerciaux sont majorés pour sponsoriser les développements libres. Donc plus l'iHP a de clients, moins chers seront les projets, mais pour qu'il y ait plus de clients, il leur faut plus de ressources gratuites pour concevoir leurs circuits. C'est une problématique courante d'*amorçage*. Déjà, il se dit que eFabless commence à collaborer avec l'iHP, laissant présager son implantation en Europe, ainsi que des projets « signaux mixtes » encore plus perfectionnés et utiles, peut-être au travers de Tiny Tapeout (donc sans passer par *OpenSamples*).

Il a été peu question des licences *open source* pour le code source du matériel, la licence CERN a été évoquée, mais ce n'est apparemment plus un débat comme il y a vingt ans, grâce à l'expérience acquise. On a surtout discuté de facilité d'utilisation, de coût, d'exploration, de NIH, de diversité... Et tout se fait par des logiciels (donc avec des licences existantes), ce qui court-circuite la question du « copyright » sur les fichiers générés ou sur la puce elle-même. Aucun brevet n'a été aperçu.

J'ai remarqué que les Allemands étaient venus déterminés, avec plusieurs présentations ainsi que des institutionnels dans le public, prenant des notes méticuleuses ; c'était probablement des membres du ministère de la Recherche allemand qui subventionne l'iHP, ce dernier distribuant des dépliants sur une table. Entre TSMC et Intel qui vont chacun ouvrir une méga-usine sur leur sol, grâce à des subventions inégalées en Europe, j'imagine qu'ils veulent être sûrs d'avoir les moyens de leurs ambitions. Cela préfigure aussi l'événement évoqué en conclusion.

En parallèle du pragmatisme allemand, l'européanisation des projets se poursuit, mais la préférence nationale persiste : par exemple, l'argent allemand finance uniquement les entités allemandes. Si on vit en dehors de l'Allemagne, il faut demander de l'aide à son pays de résidence ou à l'Union européenne. Ce qui pose l'autre question du financement des individus : « il est plus facile d'établir des contrats avec des sociétés », dit-on. Comment soutenir les communautés, alors ? Comment définir, officialiser et légaliser les communautés et leurs projets, sans les entraver par des obligations et de la paperasse ?

Puisqu'ils sont des auteurs importants des outils libres et *open source*, les développeurs indépendants, qui ne sont pas dans l'industrie ou le circuit académique, doivent aussi être soutenus. Ce qui est un problème majeur, pas seulement dans le cadre spécifique de la microélectronique : on entend ici et là les plaintes que les FOSS sont « libres à télécharger et utiliser, mais pas à développer » ! Et l'Union européenne est connue pour focaliser des millions d'euros d'aide sur des sociétés déjà implantées.

C'est là qu'intervient la Fondation NLnet qui « finance les 500 millions d'Européens » en redistribuant des fonds venus de différents programmes comme *Next Generation Internet* (**https://ngi.eu**). NLnet lance des appels à projets tous les deux mois et la présentation de Michiel Leenaars **[47]** est simple, claire et devrait vous donner toutes les informations pour soumettre votre projet. Mais faites vite, car l'Europe semble reconsidérer ses priorités **[25]** en cédant aux sirènes de l'IA.

7. ADIEU, LIBRE-SOC

Pour terminer en beauté ces trois jours, j'ai profité des circonstances favorables pour rendre visite à ce qu'aujourd'hui je qualifierais d'« artefacts historiques ». Cette conférence étant hébergée à Jussieu, où le LIP6 a ses bureaux et salles de cours, l'occasion était trop belle.

Comme mentionné précédemment, le LIP6 développe la suite Alliance/Coriolis et a participé (entre autres) à la fabrication du prototype de Libre-SOC, annoncé il y a quatre ans **[3]**. En suivant les conversations sur la liste de diffusion **[48]** je savais que des puces avaient été fondues en CMOS 180 nm par TSMC en passant par l'IMEC **[49]** puis livrées au LIP6, mais leur fonctionnalité n'avait pas encore été testée au-delà de la chaîne JTAG et de la PLL.

Au moins, *les puces étaient là* et vingt-cinq ans après le début du projet F-CPU, il était important d'aller immortaliser ce qui était (d'une certaine façon) son aboutissement : le premier prototype d'un microprocesseur réellement innovant, conçu avec des logiciels libres et (presque) totalement exempt de NDA. J'ai pu approcher un responsable du département qui m'a gracieusement ouvert le local des archives, où s'est déroulée une petite séance photo improvisée (figures 13 et 14).

Il faut rappeler que le *tapeout* est arrivé à un moment, on va dire, peu propice, en pleine crise mondiale de production assortie d'une *petite*

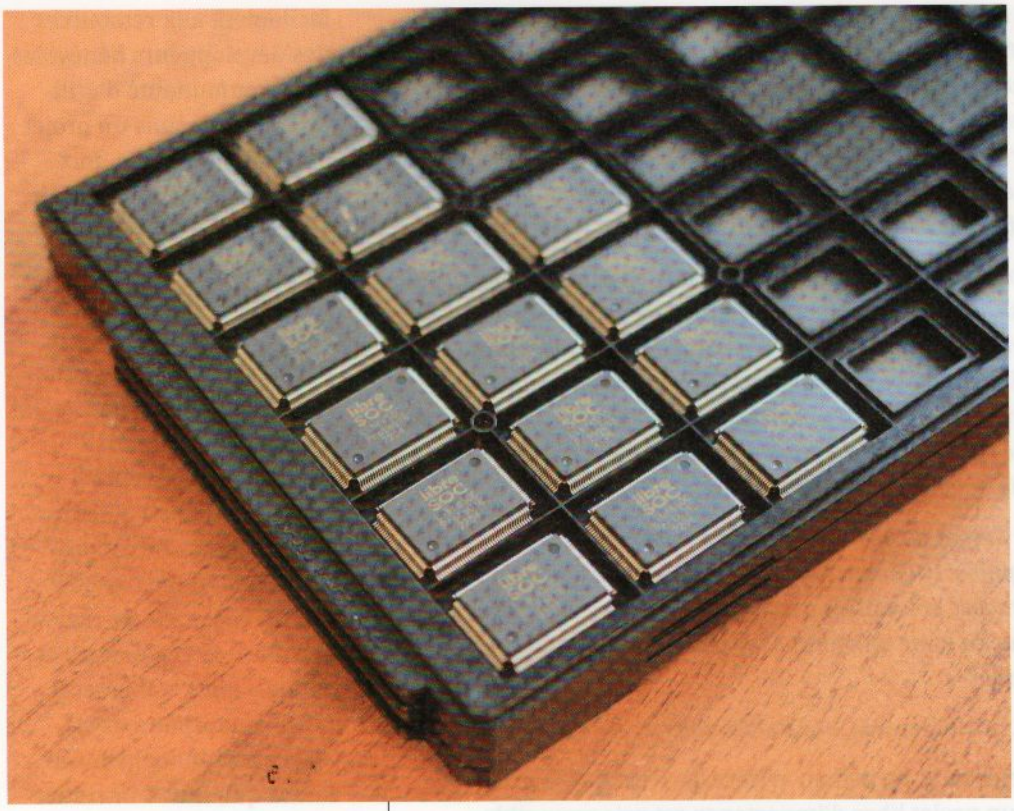


Figure 13 : La centaine de prototypes du Libre-SOC est stockée sur trois niveaux de plateaux. Certains ont été partiellement testés, mais leur destin s'arrête là.

Figure 14 : Le prototype du Libre-SOC contient un cœur POWER simplifié, encapsulé dans un boîtier TQFP128 tel que pourraient l'être de futures versions de TinyTapeout utilisant le Caravel Panamax.

pandémie. Depuis, la situation a encore empiré. Selon Luke Leighton (*lkcl*), le principal développeur :

« Libre-SOC est un projet incroyablement complexe avec deux cents pages de spécifications, développées durant six ans. L'objectif a toujours été de créer un processeur central, que l'on pourra intégrer dans des produits informatiques comme des smartphones, des ordinateurs portables et d'autres appareils grand public, procurant un grand rapport efficacité/consommation sans compromettre la performance ni la confiance des utilisateurs.

Cela implique d'abord un énorme engagement dans la conception d'un jeu d'instructions (ISA), puis des spécifications, un simulateur, une réalisation matérielle et finalement une commercialisation, s'appuyant sur le droit des marques pour assurer l'interopérabilité. Cette commercialisation nécessite un budget à sept chiffres, ce qui est considérablement au-delà d'un projet de logiciel libre usuel.

Malheureusement, deux partenariats pour la commercialisation se sont soldés par des tentatives de prise de contrôle des recherches. La dernière a eu des effets dévastateurs sur le projet : intimidation de contributeurs, tentatives de récupération de l'infrastructure et des serveurs, et bien plus, ce qui a choqué toutes les personnes suivant le projet et indirectement mis en jeu la vie du développeur principal.

Avec autant de menaces pesant sur un projet libre, totalement public (y compris de la surveillance de l'activité sur le bugtracker et les listes publiques), il devient impossible de travailler en toute quiétude, sans peur d'intimidations ou de représailles. »

Et là, normalement, vous allez vous exclamer **« Quoi ? »**

En regardant un peu en arrière, ce n'est pas la première fois qu'un projet libre d'importance est pris pour cible. Souvenez-vous du « recadrage » de Linus Torvalds, dont la communication était jugée « toxique », ou de la mise sur la touche de Richard Stallman, suite à des allégations calomnieuses. Brendan Eich (le papa de JavaScript) a été éclaboussé et évincé de Mozilla pour des histoires d'opinions politiques (comme s'il n'avait pas le droit à la vie privée hors du travail), et récemment Tim Peters (un pilier de Python) s'est fait suspendre par le comité de direction pour de prétendues violations du code de conduite **[50]**. Il y a encore d'autres exemples, que je vous épargne, mais cela se résume souvent à « il n'est pas assez ceci » ou « il est trop cela », en dehors de toute considération technique.

Dans tous ces cas, ces personnes gèrent des projets qui sont des briques essentielles, indispensables au paysage technologique d'aujourd'hui. Les prises de position par les acteurs industriels (IBM/Red Hat ou Microsoft) suffisent pour comprendre à qui profite le jeu des chaises musicales.

Le mode opératoire est souvent basé sur une prise de pouvoir progressive en exploitant les structures politiques en place, afin d'affaiblir les protections et accéder un jour plus facilement aux ressources (les développeurs bénévoles et la communauté d'utilisateurs). Prenons un projet libre au hasard qui, fort de son succès, doit devenir « sérieux » donc créer une fondation par exemple, pour pérenniser les travaux et structurer le processus de développement. Alors que le projet était jusque-là relativement horizontal, le « dictateur bienveillant à vie » (BDFL) ou autre figure fondatrice est progressivement mis de côté par un ensemble de règles, de codes, de règlements et comités qui s'établissent grâce à des sponsors et autres personnes bienveillantes qui, sans contribuer une seule ligne de code, expliquent « comment on fait ». Une fois la machine mise en place, il ne reste plus qu'à tourner la manivelle et lâcher les chiens dans les forums de discussion.

Pour Libre-SOC, c'est un peu différent puisqu'il s'agit au final de concevoir du matériel. *Du peu que j'ai compris*, le fondateur avait cocréé une société privée (RED Semiconductor Ltd **[51]**) destinée à la commercialisation du projet, séparée soigneusement du développement public, qui reste libre et subventionné

par NLnet. Cette dernière est une fondation charitable, pas une entreprise, et ne fonctionne pas sur la base de contrats, mais de *mémorandums d'entente* (MoU), une forme de convention sans force exécutoire. C'est donc très flexible, mais au bon vouloir de chacun, donc sans aucune garantie. *Normalement, tout va très bien.*

Luke Leighton s'est retrouvé chassé de sa propre entreprise, privé ainsi de son investissement personnel, puis la société a tenté de recruter les développeurs de Libre-SOC, d'interférer avec une conférence tout en essayant de bénéficier des subventions. Le chaos n'était pas immédiatement palpable, mais les accusations qui ont fusé de part et d'autre n'ont pas plu à la fondation néerlandaise chargée de gérer et distribuer les aides. Acculé par les pressions, Luke a vu sa santé chuter catastrophiquement, ce qui l'a encore plus isolé. Je me demande par quel miracle il est encore en vie après les nombreux passages aux urgences médicales, conduisant à des maltraitances supplémentaires.

Au moment de boucler cet article, la situation du projet Libre-SOC est encore trop confuse pour déduire ou conclure à quoi que ce soit. On sait juste que NLnet a définitivement suspendu toute collaboration (après avoir payé le mauvais contributeur) et Luke est totalement démuni et en état de choc permanent. Le projet est mort.

En six ans, Libre-SOC a reçu 800000 euros **[52]** de financement, ce qui met des institutionnels très mal à l'aise puisqu'il n'a pas abouti. À cause des interférences et ententes privées, de la fuite des contributeurs, des réactions épidermiques et d'une obscure zizanie, tout le projet était en phase terminale depuis de nombreux mois. J'espère que certaines parties, comme les unités de calcul en virgule flottante de Jacob, seront réutilisées par d'autres projets. Mais il y a très peu de chances qu'une des puces des figures 13 et 14 soient un jour remises sous tension, même si leur documentation est en ligne. Qu'en ferait-on et où irait le projet sans son créateur ?

Nous pouvons en tirer (ou confirmer) certaines leçons :

- Passé un certain degré de popularité, il est quasiment impossible d'empêcher un projet technique de devenir politique. Aujourd'hui, *tout est politique* et les techniciens, surtout ceux qui excellent dans la conception, sont rarement dotés de sens social et de psychologie. D'ailleurs, Luke Leighton comme Richard Stallman sont notoirement autistes, donc des cibles privilégiées pour les attaques personnelles, car il est facile de les faire passer pour abuseurs.
- Tout projet prenant de l'ampleur financière attire les profiteurs. Le pouvoir attire le pouvoir. Il suffit de voir ce qu'est devenue la Linux Foundation **[53]** : on dirait un cartel de l'*openwashing* qui en plus sponsorise le développement du *kernel*[*4].
- Les codes de conduite ne protégeront personne puisqu'ils sont conçus pour être interprétés à loisir dans le sens qui convient à qui les utilise. Les bonnes intentions et les documents savamment rédigés seront toujours faciles à détourner, les situations retournées. Luke pensait s'en prémunir en créant une charte **[54]**, un code d'honneur très simple auquel chaque participant doit se conformer, qui se résume par :

*[4] Tout comme un smartphone est un terminal intelligent qui est en mesure de passer des appels téléphoniques.

« Always do good
Never do harm
The Code applies 100% of the time
Everyone knows the Code »

NLnet a contré en sortant de sa manche un code de conduite de l'IEEE et les querelles restent cachées « pour protéger les personnes concernées ». Contrairement à du code informatique, la communication entre humains est naturellement ambiguë et les humains ont fait de la distorsion sémantique un sport depuis l'Antiquité : le sophisme.

Comment empêcher la mauvaise foi ? Faut-il éviter les projets qui ont un « code de conduite » ? Faut-il fuir les projets nécessitant des financements ? Comment empêcher un projet de devenir aussi toxique que les réseaux sociaux ?

Je n'en sais rien, mais Libre-SOC a essayé sans réussir, et j'ai ressenti une gêne et une tristesse voilées durant la conférence. Souhaitons à Luke de se rétablir.

CONCLUSION

Je ne peux malheureusement pas parler des années précédentes (pour cela, voir les pages correspondantes sur leur wiki **[1]**), mais j'ai pu enfin assister à l'édition 2024 : c'était résolument un grand cru ! Il y avait tellement d'autres choses à mentionner et expliquer ici, mais je n'ai ni le temps ni la place, malgré les aléas du calendrier de publication.

Durant ces trois jours, j'ai fait de nombreuses rencontres et même des retrouvailles très surprenantes. Et j'ai constaté avec joie que le monde des circuits intégrés a commencé sa période de *tsunami*, un peu comme en l'an 2000 où Linux commençait à être pris au sérieux par l'industrie. Tout le travail de fond avait déjà été réalisé par le projet GNU depuis sa fondation et il aura fallu une décennie avant que le projet « amateur » d'un jeune Finlandais devienne la clé de voûte de sociétés milliardaires.

Comparé à l'industrie des logiciels, le monde des circuits intégrés est d'une nature très différente, mais les parallèles sont toujours là, avec un décalage d'un quart de siècle. C'est émouvant de se dire que dès 1999, des groupes de *geeks* (F-CPU, OpenRISC et d'autres) ont osé y *croire*. Ils ont partagé une vision commune (ou du moins, semblable) et ont œuvré sans relâche à construire toutes les briques qui manquaient. On peut mentionner beaucoup de jalons (OR1K, LEON, GHDL, RISC-V, OpenROAD, Tiny Tapeout et j'en passe), mais aujourd'hui, nous pouvons contempler tout le travail accompli, célébrer les résultats tangibles et nous réjouir du changement d'attitude des industriels et des politiques.

Nous avons pu discuter des défis qui pointent à l'horizon, ainsi que des orientations futures : par exemple, la coopération entre le monde propriétaire et l'*open source* va encore s'accélérer. Comme le disait si bien Andrew Kahng, *« open source is not a goal but a way »*. Le fait que GNU/Linux ait pu « trouver sa place » sans tout remettre en question rassure aujourd'hui de nombreuses entreprises et leur donne un mode opératoire pour profiter sans risque de la nouvelle vague. Le *FUD* ne disparaît pas, mais il aura moins d'emprise.

Il a été aussi mentionné plusieurs fois que les outils freinent l'adoption par

les débutants. Les logiciels EDA classiques sont lourds, complexes, abscons, souvent incompatibles, ce qui rappelle à beaucoup ce qu'il s'est passé avec les logiciels pour programmer les microcontrôleurs (avant la vague Arduino), dessiner les circuits imprimés (avant KiCAD) ou configurer les FPGA (*soupir*). Aujourd'hui, **https://wokwi.com** reprend ces nouveaux *workflows* libres pour faciliter l'accès du silicium aux néophytes, en réutilisant des outils et concepts familiers, mais la multiplicité des distributions GNU/Linux freine clairement beaucoup d'initiatives. On a évidemment parlé du cauchemar des dépendances entre paquets **[55]**, de l'enfer des versions d'OS, de scripts spécifiques aux distributions… Et installer les outils n'est pas tout : il faut les prendre en main et savoir s'en servir de bout en bout !

Ces difficultés ne sont pas encore bloquantes. La démocratisation de la conception de circuits intégrés avance plus vite, plus fort, et l'argent commence à couler : les vannes des financements sont ouvertes ! Cela attire forcément les nouveaux acteurs, les *startups* commencent à pointer leur nez, ou à « pivoter » vers celle-ci (la vague de l'IA finira par retomber, mais il faut toujours des puces pour la faire tourner).

Même si le marché est relativement réduit (et il a surtout été question d'eFabless), il s'agrandit : grâce à la réduction des coûts de fabrication, de plus en plus de personnes et hobbyistes vont sauter du FPGA à l'ASIC

RÉFÉRENCES

[1] FSiC 2024 : **https://wiki.f-si.org/index.php/FSiC2024**
« The Free Silicon Foundation (F-Si) is a nonprofit organization with the scope of promoting:
1) Free and Open Source (FOS) CAD tools for designing integrated circuits,
2) the sharing of hardware designs and libraries,
3) common standards,
4) the freedom of users in the context of silicon integrated circuits. »

[2] Go IT! : **https://wiki.goit-project.eu/**
The Go IT! is an EU Coordination and Support Action (CSA) about Open Source Hardware for ultra-low-power, secure processors funded under cluster 4 (Digital, Industry and Space), of Horizon Europe. The project was submitted on October 21 2021, was awarded on March 18 2022 and started on September 1 2022. Its planned duration is three years.

[3] Guidon, Yann : *« Une brève histoire des ASIC libres » Hackable* n°36, janvier 2021, pp. 42-66 **https://connect.ed-diamond.com/Hackable/hk-036/une-breve-histoire-des-asic-libres** (Creative Commons).

[4] **https://en.wikipedia.org/wiki/Process_design_kit**

[5] **https://open-source-silicon.dev/** liste et documente les trois PDK disponibles actuellement : SkyWater Sky130, GlobalFoundries GF180MCU et l'IHP SG13G2 Open Source PDK, présenté à FSiC2024.

pour réaliser leurs projets, comme Thorsten ! C'est un rêve que d'innombrables électroniciens ont gardé enfoui longtemps dans leur tête, et il se réalise enfin. Les choses sérieuses commencent donc vraiment : maintenant que tout le monde se réveille, il faut se retrousser les manches.

Une autre chose se profile au loin : l'EDA *open source* commence à rattraper les outils commerciaux sur certains points. Grâce à sa flexibilité, les coûts d'exploration de l'espace des paramètres ont chuté radicalement : j'en parlais dans le cadre de GHDL (qui peut être lancé autant de fois que l'on souhaite), mais d'autres outils bénéficient d'une accélération par GPU ou IA (et pas juste pour la documentation). Ce n'est qu'une question de décennies avant que les outils propriétaires soient relégués à une niche particulière.

J'adresse mes chaleureux remerciements à chacun des participants et organisateurs, trop nombreux à lister individuellement ! J'aurai plaisir à vous retrouver, mais l'édition 2025 se déroulera probablement en Allemagne, ce qui est un peu loin pour moi (même si j'ai déjà participé à ce genre de rencontres outre-Rhin **[56]**). Je ne sais pas ce qui a motivé l'annonce des organisateurs après la dernière présentation, concernant le déplacement de l'événement, mais gardez un œil sur le site de F-SI pour obtenir les dernières informations à propos de la prochaine conférence ! **YG**

[6] Tiny Tapeout : **https://tinytapeout.com/**
Voir entre autres la vidéo de Robert Feranec *« How To Design and Manufacture Your Own Chip »* (14 juin 2024) à **https://www.youtube.com/watch?v=caXwuuXSB-A**

[7] **https://efabless.com/chipignite** : 10 mm² pour 10000 dollars.

[8] Yann Guidon : *« Développer en VHDL sous Linux » GLMF* n°36, 2001, **http://f-cpu.seul.org/new/lm02.tgz**
(état de l'art de la programmation avec Vanilla VHDL et Simili, GHDL n'existait même pas encore).

[9] Moss, Sebastian : *« Ousted Arm China CEO refuses to go after being fired, again - It's not clear if SoftBank knows how to get rid of him »* 5 mai 2022, **https://www.datacenterdynamics.com/en/news/ousted-arm-china-ceo-refuses-to-go-after-being-fired-again/** *« Allen Wu was also fired in 2020 amid claims of conflicts of interest, but simply did not leave, essentially operating the division as an independent company. »* Une saga épique qui rappelle que les entreprises chinoises ne fonctionnent pas du tout comme en Occident.

[10] Asianometry : *« How China Built a Semiconductor Industry »* 28 juin 2024, **https://www.youtube.com/watch?v=LSR4IxpRvQs**

[11] *« H.R.4346 - Chips and Science Act »* 117th Congress (2021-2022), **https://www.congress.gov/bill/117th-congress/house-bill/4346**

[12] **https://commission.europa.eu/strategy-and-policy/priorities-2019-2024/europe-fit-digital-age/european-chips-act_fr** *« Le règlement européen sur les semi-conducteurs renforcera la compétitivité et la résilience de l'Europe dans les applications et les technologies des semi-conducteurs, et contribuera à réaliser les transitions numérique et écologique, en accentuant l'avance technologique de l'Europe dans ce domaine. Après son adoption par le Parlement et le Conseil, le règlement est entré en vigueur le 21 septembre 2023. »* L'objectif est de doubler la part de marché de l'Europe, de 10 % à 20 % d'ici quelques années.

[13] Connatser, Matthew : *« TSMC confirms it'll dig into Dresden for chip giant's first fab on Euro soil - Partnership with NXP, Infineon, and Bosch finally gets under way »* 30 juillet 2024, **https://www.theregister.com/2024/07/30/tsmc_breaks_ground_dresden/**

[14] **https://digital-skills-jobs.europa.eu/en/edu4chip-joint-education-advanced-chip-design-europe**

[15] **https://www.opensusi.org/** (je n'ai toujours pas compris ce qu'ils font vraiment).

[16] **https://ggba.swiss/fr/la-suisse-lance-linitiative-swisschips-pour-soutenir-lindustrie-nationale-des-semi-conducteurs/**

[17] Robinson, Dan : *« UK semi industry exposed to supply chain risk, China state ownership - Report suggests govt get cracking on a proper ownership structure survey and ... hang on, did they forget the Midlands? »* 13 août 2024, **https://www.theregister.com/2024/08/13/uk_semi_industry_exposed_to/**

[18] Al-Barakati, Manal : *« Saudi Arabia launches 'National Semiconductor Hub' to drive industry localization »* 5 juin 2024, **https://www.arabnews.com/node/2524301/business-economy**

[19] Asianometry : *« Why GlobalFoundries Couldn't Give Abu Dhabi a Semiconductor Fab »* 5 août 2024, **https://www.youtube.com/watch?v=d7USgBB7sAQ**

[20] Cibean, Tudor : *« Russia plans to manufacture chips locally on a 28 nm node by 2030 »* 16 avril 2022, **https://www.techspot.com/news/94233-russia-plans-manufacture-chips-locally-28-nm-node.html** *« The Russian government has devised a preliminary plan to tackle the issue. It involves investing around 3.19 trillion rubles ($38.3 billion) in developing the local microelectronics industry. »* Mais le marché est surtout celui de l'armement, pas le grand public, ce qui limite la capacité à rembourser un tel investissement.

[21] Morphy, Erika : *« Russia's primary chipmaker is struggling with a defect rate of about 50 percent - Sanctions have crippled Baikal's production and packaging capabilities »* 30 mars 2024, **https://www.techspot.com/news/102453-russia-primary-chip-producer-struggles-meet-demand-defect.html**

[22] **https://latinpractice.com**

[23] Linux Foundation : *« CHIPS (Common Hardware for Interfaces, Processors and Systems) Alliance harnesses the energy of open source collaboration to accelerate hardware development. »* **https://www.chipsalliance.org/** (un jour, la Linux Foundation finira par s'occuper des croquettes pour chiens et de la fabrication des sofas).

[24] Kahng, Andrew B. : *« OpenROAD and The OpenROAD Initiative: Foundations for Open Innovation »* **https://wiki.f-si.org/index.php?title=OpenROAD_and_The_OpenROAD_Initiative:_Foundations_for_Open_Innovation**

[25] Vigliarolo, Brandon : *« FOSS funding vanishes from EU's 2025 Horizon program plans - Elimination of most Next Generation Internet funding 'incomprehensible,' says OW2 CEO Pierre-Yves Gibello »* 17 juillet 2024, **https://www.theregister.com/2024/07/17/foss_funding_vanishes_from_eus/**

[26] Rogoway, Mike : *« Intel execs admit they were caught off guard by business downturn, triggering historic cuts »* 2 août 2024, **https://www.oregonlive.com/silicon-forest/2024/08/intel-execs-admit-they-were-caught-off-guard-by-business-downturn-triggering-historic-cuts.html**

[27] Venn, Matt (Tiny Tapeout / YosysHQ) *« The long tail of semiconductors - Education, Tools and Artisanal ASICs »* 19 juin 2024, vidéo : **https://peertube6.f-si.org/w/nLjx998SVEuJPQRqbSVBbM**, slides : **https://docs.google.com/presentation/d/1mXy1HdWSIEvAgGoBLPZWd86KKDrZqtOyw6vtbseI8b4/**

[28] Martijn Bastiaan, Lucas Bollen (qbaylogic.com) : *« It's nice to have a choice: using Haskell for circuit design. »*
Présentation à FSiC2024 le 19 juin 2024 **https://wiki.f-si.org/index.php?title=It%27s_nice_to_have_a_choice:_using_Haskell_for_circuit_design**

[29] Guidon, Yann : *« Résistons à l'informatisation galopante de l'électronique ! » Hackable* n°50, septembre 2023, **https://connect.ed-diamond.com/hackable/hk-050/resistons-a-l-informatisation-galopante-de-l-electronique**

[30] **https://nlnet.nl/project/Coriolis2/**

[31] Guidon, Yann : *« Rencontre avec Tristan Gingold, l'auteur de GHDL » GLMF* n°127, mai 2010, **https://connect.ed-diamond.com/GNU-Linux-Magazine/glmf-127/rencontre-avec-tristan-gingold-l-auteur-de-ghdl**

[32] Gingold, Tristan : *« How to debug a simulation? »* **https://wiki.f-si.org/index.php?title=How_to_debug_a_simulation%3F**

[33] Böscke, Tim : *« VHDL/Verilog to Discrete Logic Flow - Work in progress: Flow to synthesize VHDL/Verilog code into a PCB »* **https://hackaday.io/project/180839** et **https://github.com/cpldcpu/PCBFlow**

[34] **https://monster6502.com/** *« The MOnSter 6502 [is] A dis-integrated circuit project to make a complete, working transistor-scale replica of the classic MOS 6502 microprocessor. »*

[35] Böscke, Tim : *« LCPU - A CPU in LED-Transistor-Logic (LTL) »* **https://hackaday.io/project/169948-lcpu-a-cpu-in-led-transistor-logic-ltl**

[36] Böscke, Tim : *« 555ENabled Microprocessor - A Microprocesser made in a digital logic family based on the NE555 »* **https://hackaday.io/project/182915**

[37] Senti, Tobias : *« Liberty74: An Open-Source Verilog-to-PCB Flow »* **https://wiki.f-si.org/index.php?title=Liberty74:_An_Open-Source_Verilog-to-PCB_Flow**

[38] Papon, Charles : *« Moving toward VexiiRiscv »* **https://wiki.f-si.org/index.php?title=Moving_toward_VexiiRiscv**

[39] *« Efabless Chipalooza Analog and Mixed-Signal Design Challenge »* février-avril 2024, **https://efabless.com/analog-and-mixed-signal-design-challenge**

[40] Marin, Jorge & Rojas, Christian (AC3E Valparaiso) : *« Unlocking the power: energy management open-source analog building blocks from concept to silicon-proven IP »* **https://wiki.f-si.org/index.php?title=Unlocking_the_power:_energy_management_open-source_analog_building_blocks_from_concept_to_silicon-proven_IP**

[41] Edwards, Tim : *« Caravel Panamax: The Next Generation »* **https://wiki.f-si.org/index.php?title=Caravel_Panamax:_The_Next_Generation**
« The Efabless «Caravel» harness chip has been the foundation of enablement for open source silicon, introduced in 2020 for the first Google-sponsored Open MPW with SkyWater foundry. The Caravel chip aims to simplify full-chip design by providing a padframe, processor, and other infrastructure around an empty area to be filled with a custom layout. This original architecture carried through nine Open MPW shuttle runs and the Efabless chipIgnite program, and was adapted for the two GlobalFoundries Open MPWs. After four years, it's time for an upgrade! «Caravel Panamax» is intended to support machine learning edge applications, with ultra-low-power modes of operation, more SRAM, a Hazard-3 RISC-V processor, and an array of on-chip sensors and data converters with full access to the user project area. With 128 pins and numerous specialty I/Os, Panamax opens the door to a vast array of potential user designs. With a test version taped out in June, Caravel Panamax should be available for chipIgnite shuttle runs by early 2025. »

[42] Verhaegen, Staf : *« Project Arrakeen: a PDKMaster based framework for scalable and portable digital and analog circuits »* **https://wiki.f-si.org/index.php?title=Project_Arrakeen:_a_PDKMaster_based_framework_for_scalable_and_portable_digital_and_analog_circuits**

[43] Catherine/@whitequark *« nMigen is now called Amaranth HDL! »* 10 décembre 2021, **https://x.com/whitequark/status/1469267241355124737**
Voir aussi *« nMigen : See Amaranth HDL.*
There is/was a dispute between M-Labs and whitequark about the ownership of the brand nMigen. Most of the codebase of nMigen was written by whitequark and the upstream was nmigen/nmigen. However, there is a fork in m-labs/nmigen and M-Labs claims the brand as being based on Migen (see nmigen.org). » à **https://hdl.github.io/awesome/items/nmigen/**

[44] Venn, Matt et Verhaegen, Staf : *« Interview with Staf Verhaegen from Chips4Makers »* **https://www.youtube.com/watch?v=agXJeIpdU6I**

[45] Verhaegen, Staf : *« PDKMaster »* **https://gitlab.com/Chips4Makers/PDKMaster**
PDK Master is a tool to manage PDKs for ASIC design and a framework for designing circuits and layouts in those technologies. It is a Python framework under heavy development and with an unstable API.
Voir aussi *« PDKMaster and Standard cell generator for the open source IHP PDK »* (conférence du 29 août 2023) sur la playlist de l'iHP à **https://www.youtube.com/watch?v=udBYyrUz8_8&list=PLiwFNOhPOqMPCU0zZ2GTxuaEeH-G8kJYH&index=11**

[46] **https://ihp-open-ip.readthedocs.io/en/latest/** et **https://github.com/IHP-GmbH/IHP-Open-PDK**

[47] Leenaars, Michiel : *« Update on libre silicon and OSHW related efforts within NGI and NLnet »* **https://nlnet.nl/pres/20240620/FSiC/** et **https://wiki.f-si.org/index.php?title=2024-Talk-MichielLeenaars**

[48] **https://lists.libre-soc.org/pipermail/libre-soc-dev/**

[49] *« Libre-SOC's Open Hardware 180nm ASIC Submitted To IMEC for Fabrication »* 10 juillet 2021, **https://hardware.slashdot.org/story/21/07/10/0154220/libre-socs-open-hardware-180nm-asic-submitted-to-imec-for-fabrication**

[50] Wouters, Thomas : *« Three month suspension for a Core Developer »* 7 août 2024, **https://discuss.python.org/t/three-month-suspension-for-a-core-developer/60250/22**

[51] Clarke, Peter : *« UK processor startup appoints James Lewis as CEO »* 11 octobre 2022, **https://www.eenewseurope.com/en/uk-processor-startup-appoints-james-lewis-as-ceo/**

[52] *« Bug 938 - All NLnet and NGI Grant Milestones »* **https://bugs.libre-soc.org/show_bug.cgi?id=938**

[53] *« Greg Kroah-Hartman Chastises Critic, Says Linux Foundation Strongly Supports Kernel Developers »* 8 octobre 2023, **https://linux.slashdot.org/story/23/10/08/0131214/greg-kroah-hartman-chastises-critic-says-linux-foundation-strongly-supports-kernel-developers**

[54] Charte du projet Libre-SOC : **https://libre-soc.org/charter/**

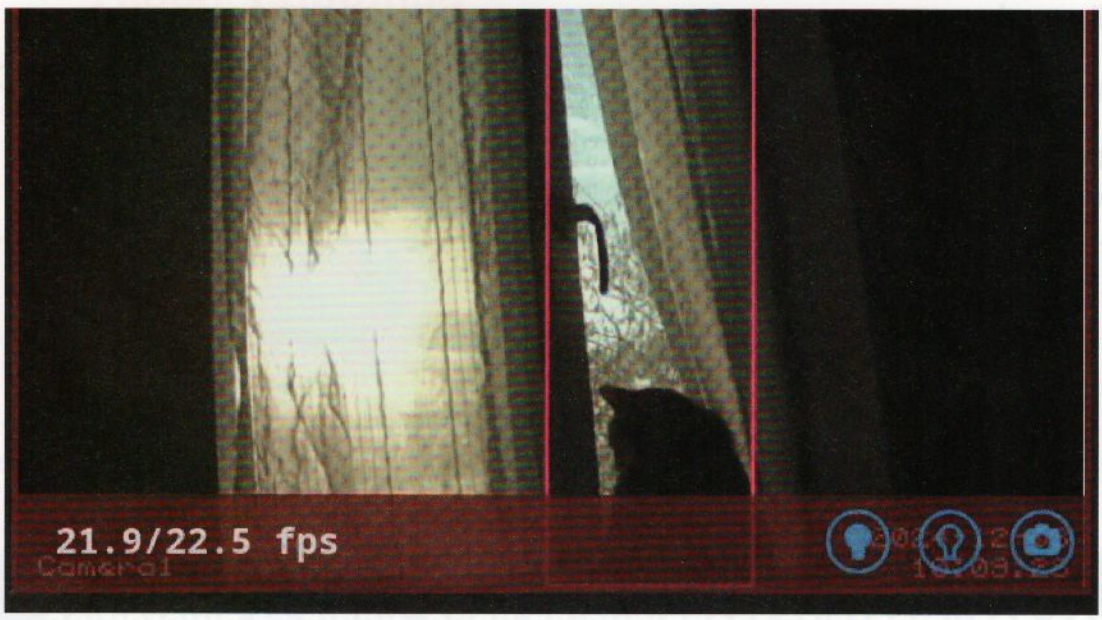
[55] Iqbal, Mazher (LIP6) : *« Accessibility and availability of Open EDA tools: the nightmare of distributions' dependencies »* **https://wiki.f-si.org/index.php?title=Accessibility_and_availability_of_Open_EDA_tools:_the_nightmare_of_distributions%E2%80%99_dependencies** (seule la vidéo est disponible).
« [...] This packaging could be a nightmare depending on OS distributions, PDK used, configurations needed etc.With the concrete case of packaging Coriolis, we will present various situations for the distribution of Open EDA tools, their benefits and drawbacks, the various suggested OS distributions, their installation process and a small tutorial. [...] »

[56] Guidon, Yann : *« EHSM : Une première réussie ! » GMLF* n°157, février 2013, **https://connect.ed-diamond.com/GNU-Linux-Magazine/glmf-157/ehsm-une-premiere-reussie**

METTRE EN PLACE UNE SURVEILLANCE DOMOTIQUE AVEC RASPBERRY PI

Denis Bodor

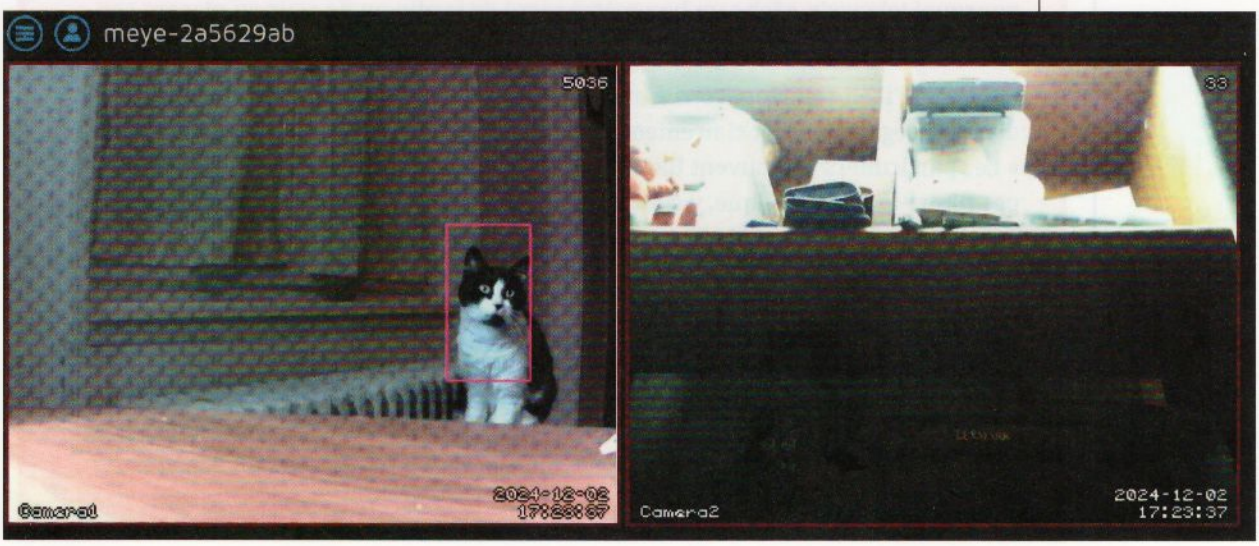
Lorsqu'on parle de domotique, on pense généralement aux capteurs et aux automatisations permettant de gérer facilement son habitation pour piloter son chauffage, simuler une présence, faire des économies d'énergie ou tout simplement améliorer son confort en se débarrassant des tâches fastidieuses et répétitives. Cependant, la sécurité est également un point important, qu'on soit sur place ou à distance, et dans ce cas précis, rien de tel que de mettre en place une vidéosurveillance.



– Mettre en place une surveillance domotique avec Raspberry Pi –

Surveiller son habitat avec une ou plusieurs caméras peut être fait pour plusieurs raisons et pas seulement pour se protéger des intrus et des voleurs, sachant que le terme de « protection » n'est pas forcément le bon, quand bien même les sociétés de service installant ce genre de dispositifs font miroiter monts et merveilles. En pratique, le fait de détecter une intrusion à distance à l'aide d'une caméra (ou d'un capteur quelconque) ne vous protège de rien du tout, cela ne fait que vous informer de l'incident. Même en cas de télésurveillance 24 h/24 h avec intervention sur place, le délai entre la détection et l'arrivée du personnel dédié (ou des forces de l'ordre), au mieux, réduira la durée de l'infraction, mais ne l'empêche pas. C'est en réalité le pouvoir de dissuasion et le facteur rassurant pour le souscripteur du service qui est vendu.

L'installation de caméras, cependant, a bien d'autres intérêts, à commencer par le fait d'avoir à éviter de se déplacer : « Mon enfant est-il bien rentré et son vélo est-il posé dans la cour ? », « Ai-je bien fermé la porte de la cabane dans le jardin ? », « Qui vient de sonner à la porte ? » (voir boîte), « Je dois déneiger ou ça fond ? », etc. De plus, comme nous allons le voir, une caméra permet non seulement d'obtenir un flux vidéo, mais également de capturer des images (fixes ou vidéos) lorsqu'un mouvement est détecté. Ceci, dans certaines situations, peut donc avantageusement remplacer ou compléter un autre système de détection (PIR, radar, etc.). Et enfin, il faut savoir que, selon le modèle de caméra utilisée, cette surveillance (active ou passive) peut être réalisée dans des conditions qui ne sont pas celles compatibles avec une simple webcam. Un excellent exemple de ce genre de choses est parfaitement illustré par les caméras infrarouges qui, en complétant l'installation d'une illumination adaptée, sont capables de « voir » dans ce qui nous parait être, à nous humains, une totale obscurité. Idéal pour faire tout ce qui vient d'être dit, mais sans déranger les voisins en inondant intempestivement le jardin à l'aide d'un projecteur à LED, par exemple.



L'interface de motionEye est sobre et claire, affichant par défaut le strict minimum pour laisser le maximum de place aux flux vidéos.

JE NE SUIS PAS JURISTE, MAIS...

Concernant l'usage de vidéosurveillance d'entrées, de portillons ou de sonnettes extérieurs, la loi est relativement claire et en particulier les articles L.223-1 et L.251-2 du Code de la sécurité intérieure (CSI) : « la transmission et l'enregistrement d'images prises sur la voie publique par le moyen de la vidéoprotection peuvent être mis en œuvre par les autorités publiques compétentes ». Vous n'êtes pas, et moi non plus, « les autorités publiques compétentes » et vous n'avez donc pas le droit de filmer et/ou transmettre des images prises sur la voie publique de cette manière.

Ceci a d'ailleurs été limpidement explicité au Sénat, grâce à une question posée par Jean Louis Masson en octobre 2017 et répondue par le ministère de l'Intérieur en août 2018 **[1]** (oui, les dates sont les bonnes) : « La mise en œuvre, par un particulier ou une copropriété, d'un dispositif de vidéoprotection filmant la voie publique, associé à une sonnette, aux fins de contrôler l'entrée dans un domicile ou dans un immeuble ne figure pas parmi les exceptions énumérées ci-dessus et ne peut donc être autorisée. En revanche, un particulier ou une copropriété peut installer un système de vidéosurveillance associé à une sonnette pour autoriser l'entrée d'un domicile ou d'un immeuble à condition que le dispositif ne filme que l'intérieur de la propriété privée. »

Pourquoi donc y a-t-il des sonnettes équipées d'une caméra qui donnent sur la rue ? À cela, deux réponses possibles : soit l'installation n'est pas légale, soit la voie publique est masquée. On peut en douter dans un certain nombre de cas, mais c'est aussi pour cela que vous avez des installations faites de manière très spécifique, avec l'objectif de la caméra perpendiculaire à la rue, généralement dans un renfoncement, et donc dans la propriété privée elle-même. Si vous vous êtes déjà demandé pourquoi tel voisin gâche un bon mètre carré de jardin ou de cour en plaçant son portillon en retrait de la rue, c'est probablement pour ça.

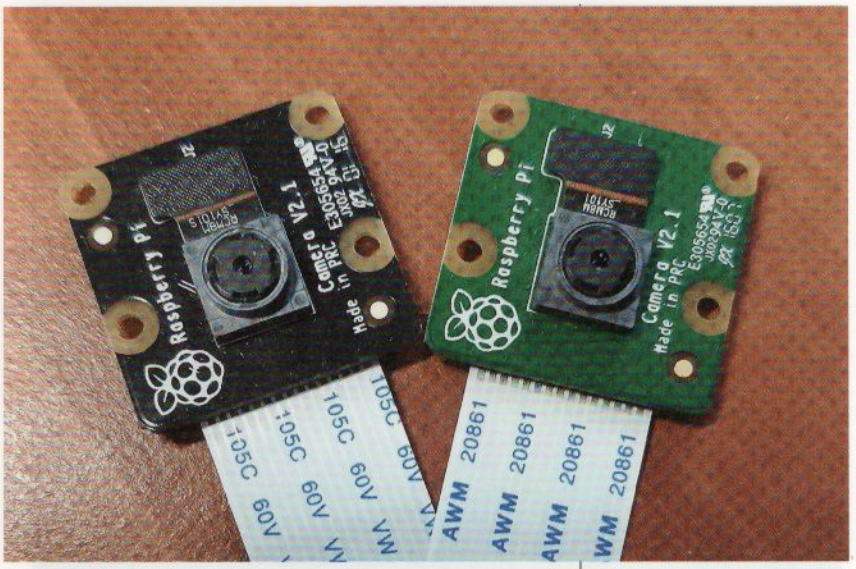
Quoi qu'il en soit, non, vous n'avez pas le droit de filmer la voie publique. Ceci est d'ailleurs également dit clairement dans un document d'information de la CNIL **[2]** : « Les particuliers ne peuvent filmer que l'intérieur de leur propriété. Ils ne peuvent pas filmer la voie publique, y compris pour assurer la sécurité de leur véhicule garé devant leur domicile. ». Vous voulez jouer au malin ? Les articles 226-1 du Code pénal et 9 du Code civil fixent le tarif : un an d'emprisonnement et 45 000 euros d'amende.

Et attention, si vous voulez déplacer votre portail/portillon, ceci peut nécessiter une déclaration préalable de travaux puisqu'il s'agit d'un élément de clôture. Effectivement, le Plan Local d'Urbanisme (PLU) peut imposer des restrictions, des caractéristiques et des obligations spécifiques... *Brazil* ? Vous avez dit *Brazil* ?

– Mettre en place une surveillance domotique avec Raspberry Pi –

L'autre point que je considère comme très important lorsqu'il s'agit de vidéosurveillance, mais également de domotique en général, est la sécurité des données. Il est capital, selon moi, de considérer l'installation comme faisant partie intégrante de votre habitat, et dans ce sens, de la même manière que vous ne laissez sans doute pas des inconnus ou même votre assureur, votre plombier ou votre électricien fouiller dans votre tiroir à chaussettes, pourquoi le feriez-vous avec vos données ? Les mesures de vos capteurs, ou les automatisations en place, ne regardent que vous et vous seul. Il en va de même pour les images provenant de votre lieu de vie. En cela, reposer sur un service de surveillance externe et non maîtrisé est, par définition, une bien mauvaise idée. Les fuites de données de grosses structures, comme on en voit régulièrement, ne sont pas imaginaires. Sans parler de la revente de données qui, fort heureusement, sont cadrées en Europe. Imaginez simplement combien vaudrait la cartographie de votre appartement, mesurée par votre aspirateur-robot, pour un assureur en quête de nouveaux clients...

Pire encore, vous pouvez avoir dans l'idée de tout simplement acheter un dispositif et le connecter à votre réseau,



Inutile de dépenser inutilement de l'argent dans de l'équipement pour mettre en place une vidéosurveillance efficace. Ces anciens modules en version 2 feront parfaitement l'affaire. À droite, la version standard et à gauche, la déclinaison « Noir » sans filtre infrarouge.

mais vous ne saurez jamais exactement ce qu'il fait et surtout, où vos images et vidéos seront effectivement transmises. Bon nombre de ces produits peu chers intègrent des *firmwares* dont le fonctionnement n'est pas audité et qui, effectivement pour certains, transmettent automatiquement les données en direction d'un *cloud* chinois sans en avertir l'utilisateur, comportent des *backdoors* ou, tout simplement, du fait de leur piètre qualité, offrent une cible de choix aux acteurs malveillants.

La solution, c'est donc de tout faire soi-même, avec du matériel connu et du logiciel *open source*. Et ce n'est pas difficile.

1. INSTALLATION DE MOTIONEYE ET CONFIGURATION

Notre intention est ici de transformer n'importe quelle carte Raspberry Pi en une caméra IP que nous pourrons utiliser directement, seule ou en compagnie d'une installation domotique sur base Home Assistant (voir le numéro 46 pour une installation en détail,

La bordure d'un flux vidéo dans l'interface motionEye (ici élargie via un clic) indique un déclenchement de détection de mouvement en devenant rouge. Notez que la zone de détection (cadre dans l'image) peut être présente sans pour autant déclencher un événement. Il faut un nombre minimum arbitraire d'images (frames) changeantes pour cela.

numéro accessible gratuitement via notre boutique en ligne **[3]**. Allez sur la boutique, connectez-vous ou créez un compte, allez dans votre compte, section « Kiosk Online », le numéro est là (pour les particuliers uniquement)). Et la façon la plus simple de le faire est d'utiliser la plus populaire des solutions existantes : motionEye **[4]**.

Ce projet est une interface web prenant en charge plusieurs types de sources vidéos, dont l'interface CSI de la Pi couplée à un module compatible officiel ou non (via l'API MMAL, *MultiMedia Abstraction Layer*), une webcam USB compatible UVC prise en charge par Linux via V4L2, une caméra réseau fournissant un flux RTSP/RTMP ou MJPEG, une simple source MJPEG ou une autre instance de motionEye. Bien sûr, la solution la plus aisée est de disposer d'un module caméra Raspberry Pi (dit RaspiCam) ou une ancienne webcam USB. Le tout, sachant que les deux peuvent servir en même temps, motionEye supportant sans problème plusieurs sources. Mais ce n'est pas tout, motionEye est également capable de gérer la détection de mouvements, prendre des instantanés et des séquences vidéos, et même de déclencher des actions en cas d'alerte. C'est une solution complète tout-en-un.

Si, comme moi, vous partez d'une Pi (ici, une Pi 2B v1.1) ou d'une carte supportée par le projet, inutile de prévoir une longue et pénible configuration, car le sous-projet motionEyeOS vous propose directement un système, basé sur Buildroot, intégrant motionEye par défaut. Pointez alors simplement votre navigateur sur **https://github.com/motioneye-project/motioneyeos/wiki/Supported-Devices** et trouvez votre bonheur dans la longue liste.

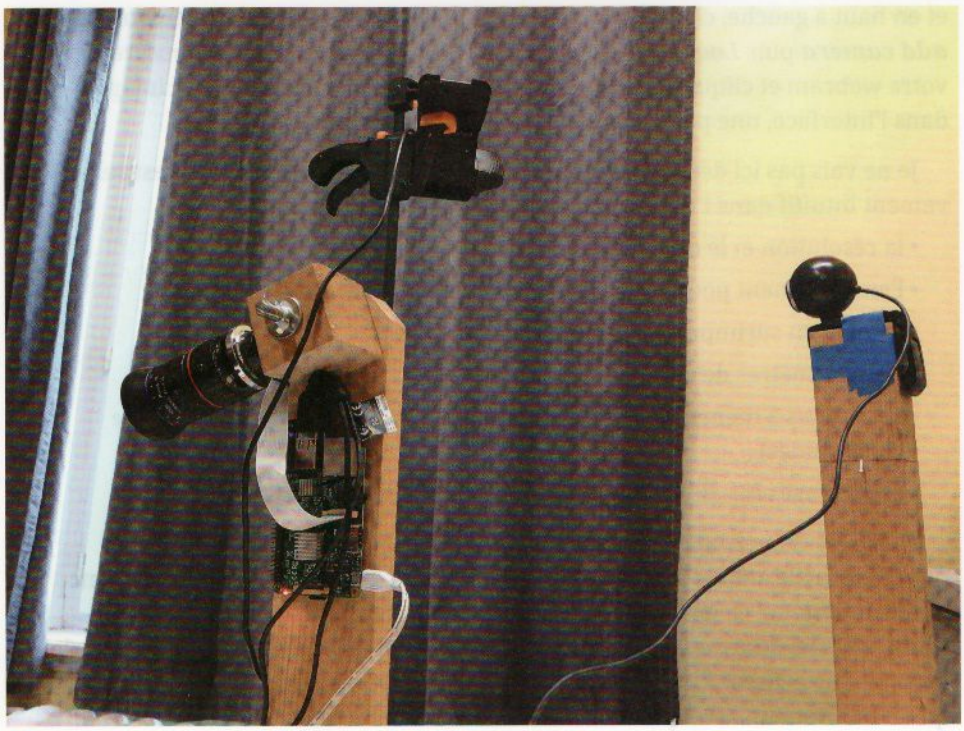
Une fois l'image téléchargée et décompressée, il vous suffira de l'inscrire sur une microSD avec dd, BalenaEtcher, Rufus (Windows uniquement) ou tout simplement Raspberry Pi Imager en spécifiant un fichier image local (motionEyeOS n'est pas dans la liste).

Placez ensuite la microSD dans votre SBC, connectez le réseau Ethernet et alimentez le tout. Au bout de quelques minutes (durant lesquels vous ne devez pas interrompre le premier démarrage), vous devrez trouver l'adresse IP de la carte. Il n'y a malheureusement pas de façon simple de le faire, si ce n'est vous tourner

vers votre routeur ou box internet pour chercher un hôte dont le nom commence par « meye- » suivi d'un numéro dépendant de l'adresse MAC de l'interface Ethernet. Une fois l'adresse trouvée, pointez votre navigateur vers celle-ci et une page d'authentification devrait apparaître. Il sera judicieux, par la suite, de configurer une adresse fixe pour cette interface, directement sur votre routeur ou dans la configuration de votre box (« Baux DHCP statiques » en spécifiant l'adresse MAC).

Par défaut, absolument rien n'est sécurisé et vous pouvez vous connecter avec l'utilisateur « admin » sans mot de passe. Précisons de suite que la connexion n'est pas chiffrée (simple HTTP). motionEye(OS) n'est pas quelque chose à exposer sur le Net ou, au pire, doit se trouver derrière un *reverse proxy* qui lui ajoutera une couche SSL/TLS (HTTPS donc). Les développeurs ont été très clairs sur le sujet **[5]**.

Si vous avez utilisé une caméra CSI, il y a de fortes chances que celle-ci ait été détectée automatiquement et que l'image s'affiche directement à l'écran. Il est relativement simple d'en ajouter une, mais avant cela, nous devons avoir un minimum d'hygiène informatique : définir un mot de passe. La configuration de base de motionEye se fait directement dans l'interface web, en cliquant en haut à gauche sur l'icône représentant trois barres horizontales. Dans le panneau qui apparaît, vous trouverez, à la section ***General Settings***, des zones de saisie pour le nom du compte administrateur, son mot de passe, ainsi que les mêmes informations pour un utilisateur « voyeur » (ou observateur). Vous pouvez en profiter pour changer le fuseau horaire qui est par défaut en UTC.



Le DIY, c'est bien, mais il faut partir du principe que cela finira plus ou moins comme ceci : un assemblage assez « bricolo » parfaitement fonctionnel (ici, avec trois caméras sur un seul SBC), mais pas vraiment « présentable »...

Les modules officiels Raspberry Pi ne sont, de loin, pas les seuls disponibles et utilisables, que ce soit avec Raspberry Pi ou tout autre SBC proposant une interface CSI.

Si vous comptez utiliser votre SBC en Wi-Fi (ce que je ne recommande pas), juste dessous vous trouverez la zone ***Network*** permettant d'activer ***Wireless Network*** et préciser le nom du point d'accès et le mot de passe associé. Tout changement à ce niveau nécessitera un redémarrage du système.

Juste dessous se trouve la zone ***Service*** qui, par défaut, active tout :

- serveur FTP (mon Dieu !) ;
- serveur Samba CIFS ;
- serveur SSH.

Les deux premiers servent à récupérer les images et vidéos capturées par la détection automatique, en plus de l'accès par l'interface web. Personnellement, je déteste ces protocoles, en particulier FTP qui date d'une autre époque, et je les désactive immédiatement. SSH seul, avec `scp`, permet le même type d'opérations via une connexion chiffrée, et c'est bien suffisant.

Les Raspberry Pi proposant plusieurs ports USB et les webcams ne valant plus grand-chose, il est très facile de compléter l'installation sur un seul et même SBC. Branchez la caméra USB, patientez un peu, puis allez dans le menu et en haut à gauche, cliquez sur ***Camera1***. Dans la liste qui apparaît, choisissez ***add camera*** puis ***Local V4L2 camera***. Dans le sélecteur juste dessous, repérez votre webcam et cliquez ***OK***. Vous devrez alors voir apparaître deux images dans l'interface, une pour chaque caméra. C'est aussi simple que ça.

Je ne vais pas ici détailler l'ensemble des paramètres puisque tout est relativement intuitif dans l'interface. On y trouve, pour chaque caméra :

- la résolution et le débit ;
- l'emplacement pour les images et vidéos enregistrées ;
- le texte en surimpression ;
- les paramètres de détection de mouvements ;
- les actions à mener en cas de détection (mail, appel web, exécution de commandes) ;
- et les plages horaires pour activer ou non la détection.

Plusieurs remarques, cependant. Régler la sensibilité de la détection n'est pas chose facile, mais s'améliore nettement lorsqu'on comprend comment fonctionne le système et surtout sa représentation à l'écran. Activer ***Show Frame Changes*** dans ***Motion Detection*** aide beaucoup en affichant un cadre autour de l'élément détecté sur l'image. Ensuite, ce n'est pas parce que ce cadre apparaît qu'un mouvement est effectivement « officiellement » détecté. Il faut un certain

nombre d'images (*frames*) pour que ce soit le cas, et là, la bordure complète passe du noir au rouge (cliquer sur l'image pour afficher les icônes aide) : c'est une détection. Inversement, lorsque plus aucun mouvement n'est capté, que plus aucun pixel ne change durant un certain temps, nous avons la fin de la détection et le cadre redevient noir. Ces deux délais sont réglés respectivement par ***Minimum Motion Frames*** et ***Motion Gap***, en nombre d'image ou « frame » (attention au *framerate* donc).

Autre point important : la notification par mail. L'authentification SMTP *over* TLS/SSL est supportée, ce qui est une bonne chose, mais ceci n'est malheureusement plus suffisant chez certains fournisseurs de messagerie (dont Google/Gmail) qui demandent d'utiliser un mécanisme OAuth 2.0. Ceci n'est pas pris en charge par motionEye et l'envoi de mail ne fonctionnera donc pas. Vous devez donc utiliser un fournisseur un peu moins... disons « exigeant » pour rester poli.

Et enfin, comme la détection de mouvement est basée sur le changement d'un certain nombre de pixels d'une *frame* à l'autre (réglé par ***Frame Change Threshold*** en pourcentage), un mouvement de la caméra elle-même sera problématique puisque tout change d'un coup. On peut cependant régler un nombre maximum de pixels changeants (***Maximum Change Threshold***) pour compenser cela, en s'aidant de la valeur affichée en haut à droite de l'image, comptant le nombre de pixels dans le cadre, mais l'ajustement reste délicat. Quoi qu'il en soit, procédez toujours à l'affinage de la détection **après** être décidé sur les caractéristiques du flux vidéo (résolution et *framerate*).

2. POUSSONS UN PEU PLUS LOIN

Vous l'aurez compris, l'ensemble est directement accessible en SSH et le système Buildroot qui se trouve derrière l'interface web est plutôt complet et bien configuré. Vous pouvez vous y connecter avec le même identifiant (« admin ») et mot de passe que pour l'interface web. Ceci peut être changé assez facilement en configurant une authentification par clé. Sur le système client, générez une paire de clés, disons EdDSA type ed25519, avec `ssh-keygen -t ed25519`, si ce n'est pas déjà fait. Connectez-vous ensuite au SBC par mot de passe, placez-vous dans « ssh/ » et créer un fichier `authorized_keys` pour y copier la clé contenue dans `~/ssh/id_ed25519.pub`. La prochaine connexion utilisera directement l'authentification par clé et pourra vous sortir une sacrée épine du pied en cas de problème.

Vous remarquerez peut-être que le `~/ssh` de motionEyeOS est en réalité un lien symbolique vers `/data/etc/ssh`. En effet, le système est assez drastiquement modifié et `/data/etc` contient non seulement la configuration de motionEye, mais également les scripts pouvant être lancés au démarrage, la configuration cliente de *WPA supplicant* (Wi-Fi) et même la *crontab*. Nous trouvons là bien plus d'options de configuration que n'en propose l'interface graphique.

Parmi ces options, nous avons des boutons qu'il est possible d'ajouter au bas du cadre sur chaque flux vidéo, les *action buttons*. Pour les utiliser, il suffit de placer, dans `/data/etc`, des scripts shell dont le nom se compose d'une action, définissant également l'aspect du bouton, suivi du caractère « _ » et de l'identifiant numérique de la caméra concernée. Les actions sont : « lock », « unlock », « light_on », « light_off », « alarm_on », « alarm_off », « up », « right », « down », « left », « zoom_in », « zoom_out », et « preset1 » à « preset9 ».

Ainsi, pour la caméra 1, nous pouvons créer `/data/etc/light_on_1` et `light_off_1`, qui seront appelés lorsque l'une ou l'autre icône sera cliquée, par exemple pour contrôler un système d'éclairage. Avec un SBC offrant un accès aux GPIO depuis le shell, comme c'est le cas ici, nous pouvons alors très simplement rédiger ces scripts (qu'on n'oubliera pas de rendre exécutables avec `chmod +x`) :

- `/data/etc/light_on_1` :

```
# !/usr/bin/env sh
/bin/echo 1 > /sys/class/gpio/gpio17/value
```

- `/data/etc/light_off_1` :

```
# !/usr/bin/env sh
/bin/echo 0 > /sys/class/gpio/gpio17/value
```

D'où sort le répertoire `gpio17/` spécifié ici ? Il est créé, toujours en shell, en influant sur le contenu de `/sys/class/gpio/export` et `/sys/class/gpio/gpio17/direction`. Chose que nous pouvons également faire dans un script qu'il sera alors nécessaire de lancer automatiquement au démarrage. Et comme Buildroot à l'excellente idée d'utiliser un système d'init digne de ce nom et non une horreur alambiquée dont je tairai le nom, un simple coup d'œil à `/etc/init.d/S98userinit` nous montre qu'un possible `/data/etc/userinit.sh` sera automatiquement utilisé s'il est présent. Il nous suffit alors de créer ce fichier (exécutable toujours), contenant :

```
#!/usr/bin/env sh
/bin/echo "17" > /sys/class/gpio/export
/bin/echo "out" > /sys/class/gpio/gpio17/direction
```

Dès le prochain redémarrage, le shell aura accès à GPIO17 sur la carte et nous pourrons, depuis l'interface web, changer l'état de cette broche. Il suffit alors d'ajouter un petit module avec un relais et un optocoupleur (pour une paire d'euros sur Amazon) pour piloter tout et n'importe quoi, y compris un projecteur LED ~230 V, une VMS ou un ventilateur, par exemple.

Comprenez bien que les actions elles-mêmes, en dehors d'être associées à un élément graphique, n'ont pas d'usage prédéterminé. `light_off_1` pourrait parfaitement faire autre chose, le choix est vôtre. De là à construire un système avec des moteurs permettant d'orienter la caméra, comme nous l'avons fait avec une antenne dans le numéro 35 **[6]**, il n'y a qu'un pas (« à pas », même).

3. LIAISON AVEC HOME ASSISTANT

Si vous possédez déjà une installation Home Assistant ou que vous comptez en mettre une en place, motionEye dispose d'une intégration très facile à ajouter. Il vous suffit de faire un tour dans ***Paramètres*** et ***Appareil et services***, puis de cliquer en bas à droite sur ***Ajouter une intégration***. Dans la liste, vous trouverez « motionEye » qui s'ajoutera alors aux intégrations existantes.

Si le module caméra utilisé ne dispose pas de filtre infrarouge, il est parfaitement possible d'illuminer la zone surveillée avec un projecteur comme celui-ci. Une longueur d'onde de quelque 800 nm devrait satisfaire plus ou moins tous les modules « IR » disponibles.

Dans celle-ci, vous pourrez alors utiliser ***Ajouter une entrée*** en spécifiant l'URL (la même que celle permettant d'accéder à l'interface), les noms d'utilisateurs (admin et observateur) ainsi que leurs mots de passe.

Ceci fait, chaque caméra apparaîtra dans HA comme un appareil, proposant un certain nombre d'entités permettant non seulement de remonter chaque flux vidéo directement dans un tableau HA, mais également de recevoir des événements liés aux détections, et donc de vous en servir pour vos automatisations. Il est alors possible de créer toute une infrastructure, aussi complexe que vous le souhaitez, pour gérer n'importe quel type de scénarios de surveillance et d'alertes. Ceci, complété d'autres systèmes de détection comme des modules PIR (infrarouge passif), ira bien au-delà de ce que n'importe quel fournisseur de service pourra vous proposer, et ce, en vous laissant une maîtrise totale du système.

4. POUR CONCLURE

Je trouve que motionEye est la solution idéale pour de la vidéosurveillance, avec ou sans Home Assistant. Bien sûr, rien n'est parfait et ce projet est perfectible sur bien des points. Fort heureusement, le développement est toujours très actif et, avec la dernière version (0.43) arrive également le support multilingue. La traduction française est terminée et vérifiée, mais ceci ne concerne, pour l'instant, que la version installable sur un système existant. Pour l'heure, motionEyeOS n'intègre que la version 0.42.1, mais on peut supposer qu'une nouvelle version ne saurait tarder.

Plus problématique pour un usage généralisé par le commun des mortels, la difficulté de trouver l'adresse IP du matériel peut être un problème pour certains. La majorité des utilisateurs ne dispose pas de routeur dédié et la configuration du serveur DHCP de leur box ne leur est pas nécessairement connue. L'intégration d'une résolution de nom mDNS pourrait être un plus, ou au minimum activer la console série

pour un usage *headless*. Bien sûr, il est toujours possible d'ajouter temporairement un écran et un clavier USB, mais ce n'est pas l'approche la plus aisée en fonction de l'installation. Ce n'est pas aussi simple qu'un produit du marché.

Ce qui nous amène alors précisément à parler du principal problème de ce type de mise en œuvre : l'aspect physique. En effet, à moins de disposer d'une imprimante 3D et surtout des compétences de modélisation, il est très peu probable d'arriver à installer une telle caméra pouvant être exposée aux éléments. Rendre le tout plaisant (ou acceptable) au regard est déjà difficile, mais faire en sorte que ceci soit réellement étanche et protégé est un défi qu'il est presque impossible de relever en partant de zéro et sans un nombre incalculable d'essais et d'échecs (coûteux). Cependant, si l'esthétique n'est pas une priorité pour vous, en intérieur, ceci ne posera aucun problème. Tout est une question de priorité... **DB**

RÉFÉRENCES

[1] **https://www.senat.fr/questions/base/2017/qSEQ171001534.html**

[2] **https://www.cnil.fr/sites/cnil/files/atoms/files/videosurveillance_voie_publique.pdf**

[3] **https://boutique.ed-diamond.com/**

[4] **https://github.com/motioneye-project/motioneye**

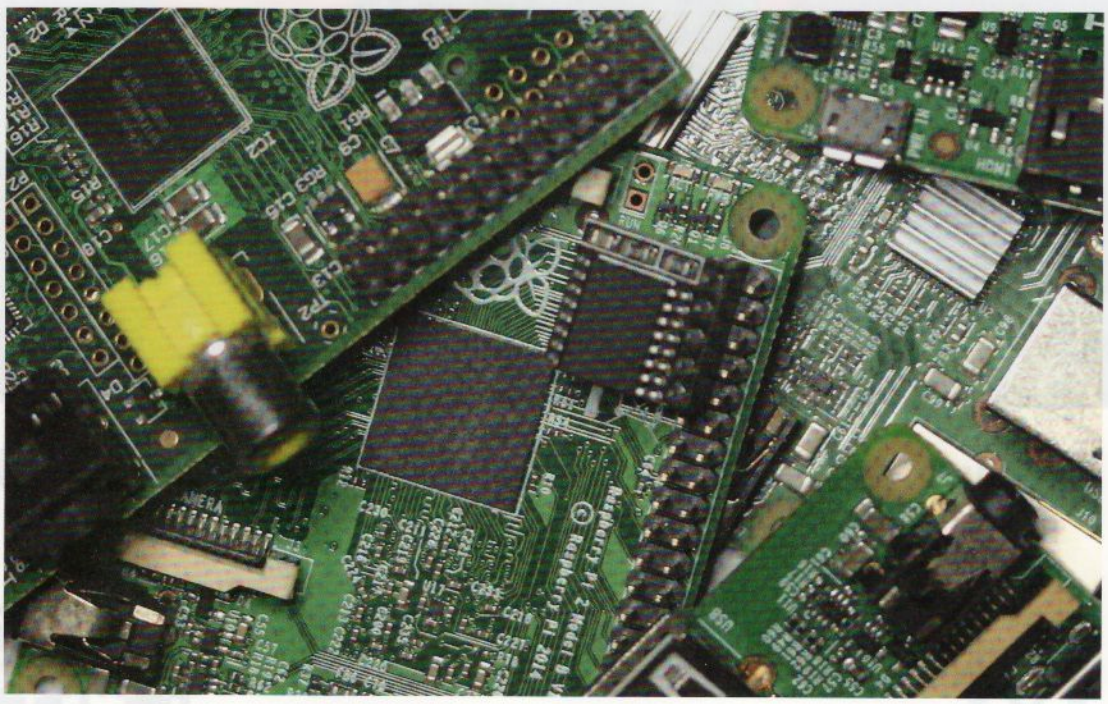
[5] **https://github.com/motioneye-project/motioneyeos/issues/49**

[6] **https://connect.ed-diamond.com/Hackable/hk-035/motoriser-une-antenne-directionnelle-avec-un-esp8266**

S'INITIER À OPENCL SUR RASPBERRY PI 3

Denis Bodor

Lorsqu'il s'agit de tirer le maximum des ressources à disposition d'une plateforme, en particulier dans l'embarqué, le processeur n'est généralement pas souvent la solution la plus adaptée. Du matériel dédié comme les accélérateurs matériels, en particulier pour les calculs matriciels ou la cryptographie, viennent souvent assister le processeur et l'alléger des tâches demandeuses en calcul. L'une des options possibles dans ces situations est l'utilisation du GPU, via des frameworks dédiés. Une humble Raspberry Pi peut être un excellent terrain de découverte pour prendre en main ce type de technologies.



– S'initier à OpenCL sur Raspberry Pi 3 –

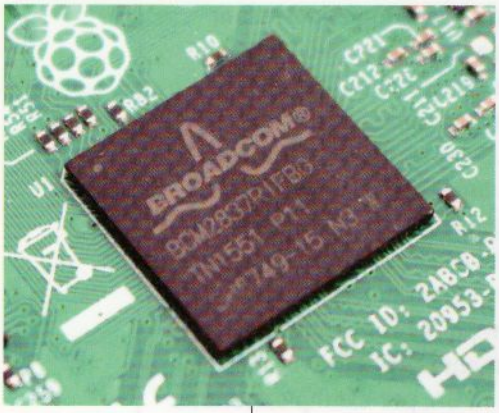
Avant toute chose, précisons que ce qui est décrit ici ne concerne **que** le SBC Raspberry Pi 3, offrant la bonne combinaison de fonctionnalités et de support pour la partie logicielle. Les Raspberry Pi 1 et 2 pourraient potentiellement servir, intégrant un SoC comprenant également un GPU Broadcom VideoCore IV (BCM2835 et BCM283696), mais une Pi 3B, et son BCM2837 à 1,2 GHz toujours en vente actuellement, est la bonne option. Les Raspberry Pi 4 et 5, respectivement BCM271197 et BCM2712, avec GPU VideoCore VI et VideoCore VII, **ne sont pas utilisables** (pour l'instant), car tout ceci repose sur le travail d'un seul développeur, *doe300* **[1]**, et non de la fondation Raspberry Pi elle-même, qui ne semble aucunement intéressée par tout ceci (contrairement à d'autres fabricants de cartes moins réputés).

1. OPENCL

L'idée derrière cette technologie est relativement simple : nous avons un processeur dédié aux tâches graphiques qui n'a, non seulement, rien à faire lorsqu'il n'y a rien à afficher, mais qui, de plus, est hautement spécialisé pour les calculs mathématiques et la parallélisation. Il est donc intéressant de s'en servir pour autre chose que du graphisme. C'est exactement la même logique que celle utilisée pour l'intelligence artificielle et les LLM (entraînement comme inférence) faisant massivement usage des GPU. Pour uniformiser ce type de technologie, des API communes à tous les constructeurs permettent de simplifier les développements. Ce petit monde se divise alors en deux, avec d'un côté CUDA, propriétaire et lié aux matériels NVIDIA, et de l'autre OpenCL (*Open Computing Language*), développé initialement par Khronos Group, mais qui est un standard ouvert, également utilisable avec des CPU et d'autres types de puces (DSP et FPGA, par exemple). À noter que, même si NVIDIA met en avant CUDA avec ses produits, ceux-ci restent compatibles OpenCL (via CUDA) au même titre que ceux des autres constructeurs (qui, actuellement, en termes de GPU se résument au duo AMD et Intel).

Dans la nomenclature OpenCL, l'architecture se divise en plusieurs éléments : l'hôte qui est la machine elle-même, le *Compute Device* qui peut être un GPU, un CPU (voir l'implémentation PoCL **[2]**), etc., les *Compute Units* qui sont les cœurs dans le silicium et enfin, les *Processing Elements* qui sont les unités arithmétiques et logiques utilisables dans les cœurs. L'API OpenCL donne accès à tout cela et permet donc de créer, compiler et exécuter des codes directement sur ce type de matériel. Ceci vous rappellera peut-être la notion de *shaders* qu'on trouve dans le monde de la programmation OpenGL et c'est bien normal, les deux sont liés. L'API elle-même est, à la base, dédiée au langage C, avec un *wrapper* pour C++ (très populaire dans le domaine), mais des *bindings* Python, Java, Go, Rust, et d'autres langages sont disponibles. Cette API vous permet de composer des programmes pour créer et compiler du code à destination du *Compute Device* supporté, mais également d'initialiser l'environnement et d'échanger des données vers et depuis l'hôte.

Les codes exécutés par le matériel compatible OpenCL sont appelés *kernels* et leurs sources sont écrits en OpenCL C, soit directement embarqués dans le programme rédigé pour l'hôte, soit enregistrés dans des fichiers utilisant l'extension `.cl`. Le nom du langage est trompeur, car même s'il est effectivement basé sur la syntaxe du C standard, il existe un grand nombre de différences : récursion interdite, existence d'un type de donnée vecteur, pas de pointeur de fonction, structures possibles mais à éviter, pas de communication inter-*kernels*, pas d'allocation dynamique (`malloc()`), etc.



Le SoC Broadcom BCM2837 équipant le SBC Raspberry Pi 3B intègre un GPU VideoCore IV qu'il est possible d'utiliser comme unité de calcul, en complément du CPU ARM, mais également seul. Un peu comme un coprocesseur mathématique d'antan...

Pour développer en utilisant OpenCL, la plate-forme doit disposer d'une implémentation spécifique fournissant l'API compatible. Ceci constitue donc le lien avec le matériel, permettant au programmeur de ne pas avoir à se soucier des spécifications exactes du périphérique utilisé (en dehors de la version de l'API supportée). Le code principal, pour l'hôte, est développé dans le langage de son choix (le C pour moi) et « embarque » le source OpenCL C pour produire un *kernel* exécuté par le périphérique OpenCL, soit sous la forme d'une chaîne de caractères, soit stocké dans un fichier distinct qui sera chargé en mémoire et traité comme un tableau de `char` (une chaîne aussi, donc). Dans les grandes lignes, c'est donc le code de l'hôte qui va, via les fonctions mises à disposition par l'API, compiler et charger le *kernel* dans la mémoire du périphérique, lui passer des données et déclencher son exécution.

Dans le cas qui nous intéresse ici, toute la partie prenant en charge le GPU VideoCore IV et fournissant une API compatible OpenCL 1.2 est gérée par le code développé par *doe300* **[1]** et que nous allons installer. Cependant, si vous souhaitez expérimenter sans avoir à toucher à un GPU, vous pouvez également opter pour PoCL **[2]**, une implémentation sous licence MIT du standard OpenCL reposant sur l'utilisation de CPU (amd64, ARM, RISC-V, etc.). Vous pouvez donc également installer PoCL aussi bien sur Raspberry Pi ou un autre SBC de votre choix, ou même un PC GNU/Linux, par exemple avec Debian/Devuan/Raspbian/etc., via le paquet `pocl-opencl-icd` (et ses dépendances). Ceci peut constituer une approche intéressante et à peu de frais (zéro, en fait) pour s'initier à OpenCL.

Mais nous ne sommes pas ici pour « tricher » en utilisant un CPU, toute la magie est justement d'utiliser le GPU, et il est temps d'installer la création de *doe300*.

2. INSTALLER VC4CLSTDLIB/ VC4C/VC4CL

Avant toute chose, précisons que ce qui va suivre a été fait sur une Pi 3B fraîchement (ré)installée avec Raspberry Pi OS Lite 32 bits. En effet, et même si *doe300* n'est pas explicite sur le sujet, l'OS en version 64 bits pose problème et bien que l'installation du support OpenCL passe sans souci, les plantages assez brutaux sont tout ce qu'on obtient à l'utilisation. Ceci ne me semble guère surprenant, étant donné que j'ai rencontré exactement les mêmes problèmes lors de développements OpenGL pour profiter de l'accélération graphique (via des *shaders*) en mode *headless* (rendu OpenGL sans affichage). Autre point important, clairement visible en installant également PoCL sur le SBC et en procédant à la comparaison avec `clpeak`, les performances ne sont pas le réel objectif de l'opération, en dehors de quelques cas particuliers où le support OpenCL/VideoCore se montre plus performant (*int16* et *float16* en particulier). Le domaine du calcul haute performance, que ce soit avec OpenCL ou CUDA, ne se résume pas à la simple installation d'une API et à

la rédaction d'un code de démonstration, il faut connaître son sujet, sa plateforme et les techniques d'optimisation adaptées. Ceci n'est qu'une très sommaire introduction, pas un tutoriel de programmation OpenCL.

Sans traîner davantage, installons les dépendances nécessaires ainsi que les sources des trois composants interdépendants depuis GitHub :

```
$ sudo apt-get install cmake git \
ocl-icd-opencl-dev ocl-icd-dev opencl-headers \
libraspberrypi-dev clang clang-format clang-tidy \
zlib1g-dev clinfo clpeak

$ mkdir OPENCL
$ cd OPENCL/
$ git clone https://github.com/doe300/VC4CLStdLib.git
$ git clone https://github.com/doe300/VC4C.git
$ git clone https://github.com/doe300/VC4CL.git
```

Nous avons là, sous forme de répertoires :

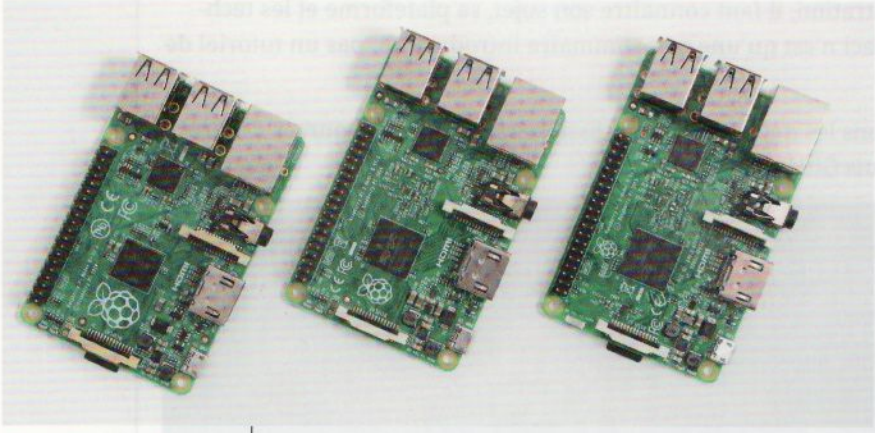
- `VC4CLStdLib/` : l'implémentation spécifique à la plateforme qui sera liée par VC4C aux *kernels* que nous allons produire ;
- `VC4C/` : c'est le compilateur qui transformera nos sources OpenCL C en binaires, et fournit également les fonctions standard OpenCL ;
- `VC4CL/` : la bibliothèque OpenCL destinée à l'hôte, chargée de compiler, exécuter et interagir avec les *kernels* que vous développez.

Ces trois éléments doivent être compilés dans un ordre bien précis et, disons-le directement, installés avec la méthode dite « à l'arrache ». Le contenu de ces sources semble disposer d'un support pour produire des paquets pour les distributions compatibles Debian, mais celui-ci est relativement sommaire et mes tentatives d'installer proprement tout cela, en accord avec le système de gestion de paquets, n'a pas porté ses fruits. Ne vous étonnez donc pas de l'utilisation de `sudo make install`, qui n'est pourtant pas dans mes habitudes, mais qui, tout de même, a le bon ton d'installer les éléments dans `/usr/local` par défaut.

Commençons par la base de l'ensemble :

```
$ cd VC4CLStdLib
$ mkdir build && cd build
$ cmake -DCROSS_COMPILE=OFF ../
$ make -j3
$ sudo make install
```

L'implémentation spécifique étant maintenant disponible, nous passons au compilateur, mais avec un petit ajustement. En effet, tenter de compiler VC4C tel quel découlera sur un message d'erreur impliquant `INT_MAX`. Le fichier `test/TestInstructions.cpp` déclare une variable `INT_MAX` en ligne 827 puis l'utilise à plusieurs reprises. Or, il s'agit du nom d'une macro déjà existante (`limits.h`) et nous nous retrouvons avec un conflit. Il sera donc nécessaire,



L'installation du support logiciel OpenCL développé par doe300 peut être faite sur n'importe quelle Pi 1 à 3. On préférera cependant utiliser le modèle le plus puissant, ne serait-ce que pour la (relative) rapidité de compilation du support OpenCL.

avant de lancer la construction du compilateur, d'éditer ce fichier et de remplacer toutes les occurrences de `INT_MAX` par un autre nom, `INTMAX` par exemple. Avec Vi, ceci se résumera à un simple `%s/INT_MAX/INTMAX/g`, mais vous pouvez bien sûr également le faire à la main (ou avec `sed`). Ceci fait, nous pouvons lancer la compilation et l'installation :

```
$ cd ../../VC4C
$ mkdir build && cd build
$ cmake ../
$ make
$ sudo make install
$ sudo ldconfig
```

Deux choses sont ici importantes. La première est l'absence de l'option `-j3` pour la compilation que nous avons utilisée précédemment et qui normalement accélère la construction en parallélisant les tâches (nous avons 4 cœurs avec un BCM2837). Le problème ici est la consommation de mémoire et le malheureux 1 Gio (non, je refuse d'utiliser un mot aussi niais que « gibioctet ») de RAM est bien insuffisant pour compiler trois sources C++ en même temps. La construction sera donc plus lente, trèèèèèèès lente même, mais arrivera à terme sans que tout s'effondre. Le second point concerne l'utilisation de `ldconfig` qui devrait rappeler des (mauvais) souvenirs aux utilisateurs GNU/Linux de longue date. Cette commande permet de rafraîchir les chemins de recherche des bibliothèques dynamiques pour que l'éditeur de liens, ou *loader* (`ld.so`) retrouve celle que nous venons d'installer (`libVC4CC.so`, en l'occurrence) et qui fournit le compilateur pour nos futurs programmes.

Il ne nous reste plus qu'à conclure l'installation avec la bibliothèque OpenCL, pour nous fournir l'API tant attendue :

```
$ cd ../../VC4CL
$ mkdir build && cd build
$ cmake -DBUILD_ICD=ON -DIMAGE_SUPPORT=ON -DBUILD_TESTING ../
$ make
$ sudo make install
```

Inutile de redémarrer ou de faire quoi que ce soit d'autre dans la configuration, tout ceci doit être immédiatement utilisable comme le montrera la sortie de la commande `clinfo` :

```
$ sudo clinfo
Number of platforms               1
  Platform Name                   OpenCL for the Raspberry Pi
                                  VideoCore IV GPU
  Platform Vendor                 doe300
  Platform Version                OpenCL 1.2 VC4CL 0.4.9999 (b5a6097)
  Platform Profile                EMBEDDED_PROFILE
  Platform Extensions             cl_khr_il_program cl_khr_spir
                                  cl_khr_create_command_queue
                                  cl_altera_device_temperature
                                  cl_altera_live_object_tracking cl_khr_icd
                                  cl_khr_extended_versioning
                                  cl_khr_spirv_no_integer_wrap_decoration
                                  cl_khr_suggested_local_work_size
                                  cl_vc4cl_performance_counters
  Platform Extensions
  function suffix                 VC4CL
  Platform Name                   OpenCL for the Raspberry Pi
                                  VideoCore IV GPU
Number of devices                 1
  Device Name                     VideoCore IV GPU
  Device Vendor                   Broadcom
  Device Vendor ID                0x14e4
  Device Version                  OpenCL 1.2 VC4CL 0.4.9999 (b5a6097)
  Device Numeric Version          0x402000 (1.2.0)
  Driver Version                  0.4.9999
  Device OpenCL C Version         OpenCL C 1.2
  Device OpenCL C Numeric
  Version                         0x402000 (1.2.0)
  Device Type                     GPU
  Device Profile                  EMBEDDED_PROFILE
  Device Available                Yes
  Compiler Available              Yes
  Linker Available                Yes
  Max compute units               1
  Available core IDs(ARM)         0
[...] PAUSE [...]
  Preferred / native vector sizes
    char                          16 / 16
    short                         16 / 16
    int                           16 / 16
    long                          0 / 0
    half                          0 / 0        (n/a)
    float                         16 / 16
    double                        0 / 0        (n/a)
[...]
```

La Raspberry Pi 4 utilise un GPU très différent de celui intégré aux BCM2837 et le support OpenCL décrit ici n'est malheureusement pas compatible. On pourra toujours se rabattre sur PoCL (support CPU), mais la magie n'est pas la même...

Nous avons là une plateforme OpenCL supportant un périphérique de type GPU, utilisant OpenCL 1.2 (profil « Embedded »), avec une seule et unique *compute unit*. À titre d'exemple, un GPU AMD Radeon R9 en fournit 44, un AMD Radeon HD 6700M en propose 8 et une absolument hors de prix NVIDIA GeForce RTX 4090 vous en donnera 128 (en échange de quelque 450 W de TDP) !

Un autre outil intéressant pour tester le bon fonctionnement de l'installation est `clpeak`, un outil de profilage du matériel permettant de repérer facilement les fonctionnalités, et plus exactement les types de données, offrant le maximum de performances. Le test dure un certain temps, mais les résultats sont intéressants :

```
$ sudo clpeak
Platform: OpenCL for the Raspberry Pi VideoCore IV GPU
  Device: VideoCore IV GPU
    Driver version   : 0.4.9999 (Linux ARM)
    Compute units    : 1
    Clock frequency : 300 MHz
[...]
    Single-precision compute (GFLOPS)
      float   : 0.61
      float2  : 1.19
      float4  : 2.29
      float8  : 4.04
      float16 : 6.38
    Integer compute Fast 24bit (GIOPS)
      int   : 0.60
      int2  : 1.17
      int4  : 2.13
      int8  : 3.51
      int16 : 5.74
[...]
    Kernel launch latency : 20.98 us
```

Nous pouvons comparer cela à la même opération, mais en utilisant PoCL sur la même plateforme :

```
$ sudo clpeak -p 1
Platform: Portable Computing Language
  Device: pthread-arm1156t2f-s-cortex-a53
    Driver version  : 3.1+debian (Linux ARM)
    Compute units   : 4
    Clock frequency : 1400 MHz
[...]
    Single-precision compute (GFLOPS)
      float   : 1.22
      float2  : 1.98
      float4  : 2.73
      float8  : 1.09
      float16 : 1.05
[...]
    Integer compute Fast 24bit (GIOPS)
      int   : 1.35
      int2  : 1.58
      int4  : 1.57
      int8  : 0.93
      int16 : 0.83
[...]
Kernel launch latency : 22.11 us
```

On peut clairement voir que le GPU, contrairement au CPU, affiche des performances très intéressantes pour les `float16` et les `int16`, mais pas nécessairement sur les autres tailles, alors que le CPU est relativement constant (bien que cadencé à une fréquence presque 5 fois supérieure). Ceci rejoint ce que montre la sortie de `clinfo` à la ligne « *native vector sizes* », qui nous indique explicitement que notre GPU est « doué » avec les valeurs 16 bits. Ce qui est parfaitement normal, c'est la base du calcul graphique en général.

Et enfin, nous avons `v3d_info`, installé avec VC4CL, qui vous donnera des informations plus génériques sur la plateforme, comme les diverses fréquences d'horloge utilisées par le GPU ou encore la quantité de mémoire à disposition de ce dernier ou la taille de la file d'attente pour l'exécution de commandes (*program queue*). À propos de RAM, il sera intéressant d'ajuster le volume utilisé par le GPU (et donc inutilisable par le CPU) en ajoutant une ligne dans votre `/boot/firmware/config.txt`, comme ceci :

```
#gpu_mem=128
gpu_mem=256
#gpu_mem=512
```

Un redémarrage sera nécessaire et, en plus de `v3d_info`, vous pourrez confirmer le changement avec :

```
$ vcgencmd get_mem gpu
gpu=256M
```

Si vous aussi avez accumulé les SBC au fil du temps, alors que les modèles récents sont généralement utilisés pour les nouveaux projets, il peut être amusant de leur donner une seconde vie pour tester de nouveaux domaines de développement ou même de nouveaux OS.

3. UN PETIT CODE DE TEST POUR FINIR

Lorsqu'on cherche « OpenCL Raspberry Pi » sur le Net, on retombe généralement sur les mêmes explications que celles données ici, le plus souvent non testées et sans les bons arguments de compilation. Mais l'étape suivante est systématiquement absente : la réponse à « *Et maintenant, je fais quoi ?* ». Je ne vais pas encombrer cet article d'un énorme code, mais souhaite tout de même vous donner du grain à moudre. Le code suivant définit un simple *kernel* OpenCL calculant la racine carrée de la somme d'une valeur arbitraire avec le numéro identifiant une tâche (*work-item*). Tout le reste du code C consiste à trouver le matériel disponible, initialiser l'environnement (*context*), compiler le code OpenCL, mettre en place des *buffers* pour passer les données et exécuter le code du *kernel* avec les bons arguments :

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

#define CL_TARGET_OPENCL_VERSION 120
#include <CL/cl.h>

#define PLATEFORMMAX    10
#define DEVICEMAX       10
#define NBRVAL          10

// Code OpenCL C
const char *kernelsource =
    "__kernel void test(\n"
    "                   __global int *indata,\n"
    "                   __global float *outdata)\n"
    "{\n"
    "   int gid = get_global_id(0);\n"
    "   outdata[gid] = sqrt(gid + (float)indata[gid]);\n"
    "}\n";

int main(int argc, char** argv) {
    cl_uint nplatforms;
    cl_platform_id platforms[PLATEFORMMAX];

    cl_uint ndevices;
    cl_device_id devices[DEVICEMAX];
```

```
cl_context context;
cl_command_queue command_queue;

int numvals = NBRVAL;
size_t global_work_size = NBRVAL;

const cl_int inmem[NBRVAL] =
  { 0, 10, 20, 30, 40, 50, 60, 70, 80, 90};
float outmem[NBRVAL] =  { 0.0 };

cl_mem inmemobj = NULL;
cl_mem outmemobj = NULL;

cl_program program = NULL;
cl_kernel kernel;

int i;

// Obtenir la liste de plateformes disponibles
clGetPlatformIDs(PLATEFORMMAX, platforms, &nplatforms);

// Obtenir la liste de périphériques pour la 1re plateforme
clGetDeviceIDs(platforms[0], CL_DEVICE_TYPE_DEFAULT,
        DEVICEMAX, devices, &ndevices);

// Créer un contexte / initialisation
context = clCreateContext(NULL, 1, devices, NULL, NULL, NULL);

// Créer une file de commandes pour le périphérique
command_queue = clCreateCommandQueue(context, devices[0],
        CL_QUEUE_OUT_OF_ORDER_EXEC_MODE_ENABLE, NULL);

// Créer un objet "programme" et charger le code source
program = clCreateProgramWithSource(context, 1,
        (const char **)&kernelsource, NULL, NULL);

// Construire le programme
clBuildProgram(program, 1, devices, NULL, NULL, NULL);

// Créer un objet kernel
kernel = clCreateKernel(program, "test", NULL);

// Affichage des données en entrée
printf("Data in:\n");
for (i = 0; i < numvals; i++)
    printf("  inmem[%d]: %d\n", i, inmem[i]);
```

```
    // Créer un buffer unidirectionnel pour les données hôte -> kernel
    inmemobj = clCreateBuffer(context, CL_MEM_READ_ONLY,
            numvals * sizeof(cl_int), NULL, NULL);

    // Créer un buffer unidirectionnel pour les données kernel -> hôte
    outmemobj = clCreateBuffer(context, CL_MEM_WRITE_ONLY,
            numvals * sizeof(float), NULL, NULL);

    // Ajouter la commande dans la file pour placer les données
    clEnqueueWriteBuffer(command_queue, inmemobj, CL_TRUE, 0,
            numvals * sizeof(cl_int), inmem, 0, NULL, NULL);

    // Régler les arguments à passer au kernel
    clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&inmemobj);
    clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&outmemobj);

    // Ajouter la commande dans la file pour exécuter le kernel
    clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
            &global_work_size, NULL, 0, NULL, NULL);

    // Attendre que les commandes soient traitées avant de lire les données
    clFinish(command_queue);

    // Ajouter la commande dans la file pour récupérer les données
    clEnqueueReadBuffer(command_queue, outmemobj, CL_TRUE, 0,
            numvals * sizeof(float), outmem, 0, NULL, NULL);

    // Affichage des données en sortie
    printf("Data out:\n");
    for (i = 0; i < numvals; i++)
        printf("  outmem[%d]: %f\n", i, outmem[i]);

    // Nettoyage
    clReleaseMemObject(inmemobj);
    clReleaseMemObject(outmemobj);
    clReleaseProgram(program);
    clReleaseKernel(kernel);
    clReleaseCommandQueue(command_queue);
    clReleaseContext(context);

    return (EXIT_SUCCESS);
}
```

Notez que j'ai condensé au maximum le code tout en le laissant lisible et facile à comprendre, mais il est **impératif** de tester les valeurs retournées (ou les codes de retour en argument) pour chaque fonction `cl*` utilisée, et stopper l'exécution en cas de problème. Vous devez obtenir `CL_SUCCESS` à chaque étape pour poursuivre. Ceci n'est donc pas un code « propre », mais vous trouverez une version plus pédagogiquement acceptable sur mon GitLab **[3]**.

Quoi qu'il en soit, vous pouvez compiler et tester avec :

```
$ cc -Wall -o opencltest main.c -lOpenCL
$ sudo ./opencltest
Data in:
  inmem[0]: 0
  inmem[1]: 10
  inmem[2]: 20
  inmem[3]: 30
  inmem[4]: 40
  inmem[5]: 50
  inmem[6]: 60
  inmem[7]: 70
  inmem[8]: 80
  inmem[9]: 90
Data out:
  outmem[0]: 0.000000
  outmem[1]: 3.316625
  outmem[2]: 4.690416
  outmem[3]: 5.744563
  outmem[4]: 6.633250
  outmem[5]: 7.416198
  outmem[6]: 8.124039
  outmem[7]: 8.774964
  outmem[8]: 9.380832
  outmem[9]: 9.949874
```

Je vous laisse le soin de vérifier les calculs à la main et même, éventuellement, de tester avec d'autres implémentations d'OpenCL sur Pi (PoCL) ou PC. Mais, surtout, de faire évoluer ce code « *hello world* » pour en faire quelque chose d'éventuellement utilisable (sans arriver pour autant à HashCat **[4]**, puisque HashCat existe déjà). Vous trouverez sans peine des exemples dans d'autres langages, surtout C++ et Python (C semble un peu délaissé dans le domaine) sur le Web et, si vous restez sur le C/C++, la documentation officielle d'OpenCL **[5]** **[6]** sera votre amie... DB

RÉFÉRENCES

[1] https://github.com/doe300

[2] https://portablecl.org/

[3] https://gitlab.com/0xDRRB/opencl_helloworld

[4] https://hashcat.net/hashcat/

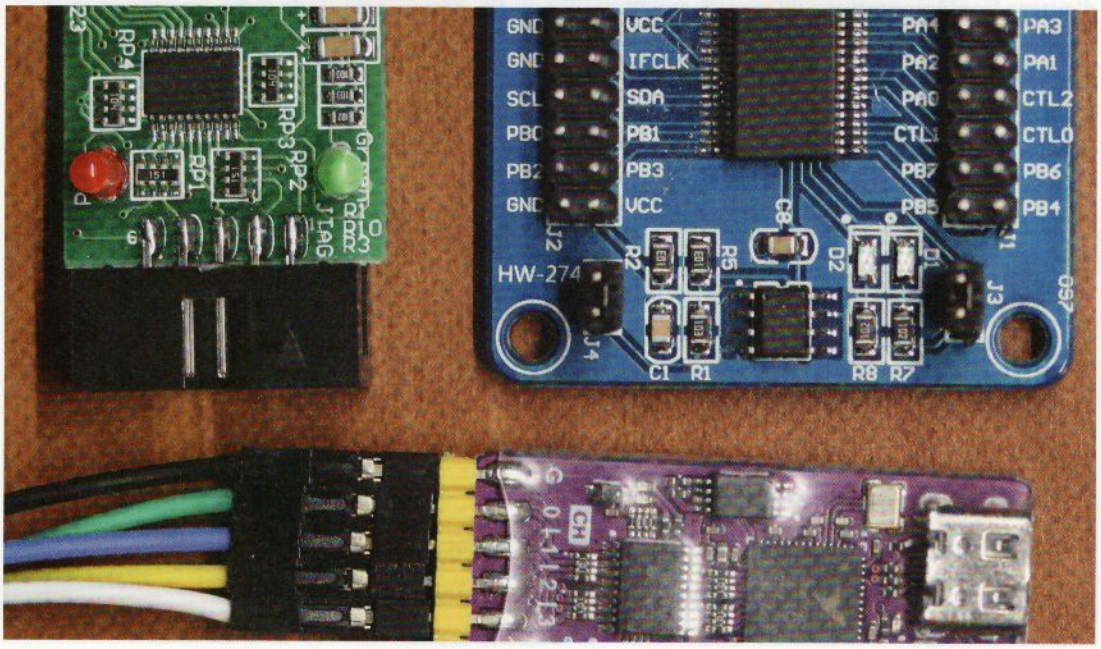
[5] https://registry.khronos.org/OpenCL/sdk/1.2/docs/man/xhtml/

[6] https://registry.khronos.org/OpenCL/sdk/3.0/docs/man/html/

FX2LP : UNE AUTRE SOLUTION POUR CRÉER DES PÉRIPHÉRIQUES USB

Denis Bodor

Dans un précédent article [1], nous avons vu comment il était possible, avec une Raspberry Pi Pico, de créer relativement facilement un périphérique USB au comportement arbitrairement défini. Mais bien avant l'arrivée du microcontrôleur RP2040, il était déjà possible de faire cela depuis bien longtemps, trèèèèèès longtemps même. Et je ne parle pas d'Arduino (Leonardo en l'occurrence et son ATmega32u4), mais d'un composant que vous possédez peut-être déjà, sans le savoir, dans votre boîte à outils : l'EZ-USB FX2LP.



– FX2LP : une autre solution pour créer des périphériques USB –

Si le nom Saleae Logic vous dit quelque chose, vous savez peut-être que la première itération de l'analyseur logique éponyme, abusivement cloné depuis par nombre de constructeurs chinois peu scrupuleux au point d'atteindre la pure contrefaçon, reposait sur un MCU très particulier de chez *Cypress Semiconductor* (maintenant absorbé par *Infineon Technologies*). Ce microcontrôleur, ou plus exactement cette famille de microcontrôleurs, est spécialisée dans les fonctionnalités de connexion USB et est d'ailleurs décrite dans sa documentation comme un « contrôleur USB périphérique Hi-Speed » (480 Mbit/s). C'est le composant principal (CY7C68013A, en particulier) de nombreux analyseurs logiques peu cher, mais c'est également un microcontrôleur qu'il est parfaitement possible de programmer, avec des logiciels libres, pour créer relativement facilement n'importe quel périphérique USB 2.0.

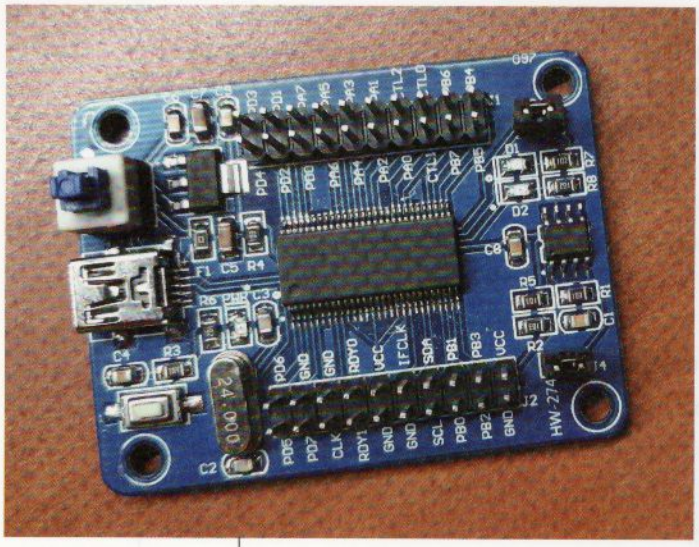
1. CYPRESS CY7C68013A ALIAS FX2LP

L'EZ-USB FX2LP, et plus précisément le CY7C68013A, est un microcontrôleur qui ne date pas d'hier et qui utilise à la fois une architecture et un processeur peu communs aujourd'hui (du moins pour les *devkits* « à la mode »). Point d'ARM Cortex-M ici, ce composant est construit autour d'un cœur 8051, un microcontrôleur initialement créé par Intel dans les années 80 (MCS-51) qu'on retrouve sous bien des formes et déclinaisons chez un grand nombre de fabricants (AMD, OKI, Fujitsu, Philips, Temic, etc.). Ce MCU Cypress intègre :

- un contrôleur USB 2.0 ;
- un cœur 8051 étendu, cadencé à 48 MHz, 24 MHz ou 12 MHz ;
- 16 Kio de SRAM ;
- deux USART ;
- trois compteurs/*timers* ;
- un contrôleur i2c (100 et 400 kHz) ;
- 24 GPIO (ou 40 selon le boîtier) ;
- etc.

Grande absente de cette liste, la flash. Le FX2LP n'en possède pas, mais propose en revanche trois solutions pour le stockage du code à exécuter : en RAM interne, téléchargé via USB, en RAM interne, chargé depuis une EEPROM i2c ou via un support mémoire externe (flash, EPROM, etc.), mais cette dernière option n'est disponible que pour les versions 128 broches, plus rares que le classique boîtier SSOP-56 qu'on trouve sur les *devkits* ou des analyseurs logiques.

En parlant des analyseurs logiques et en particulier ceux qui sont utilisables avec Sigrok/Pulseview (voir article sur la libsigrok dans le numéro 43 **[2]**) reposant sur un CY7C68013A, ceux-ci intègrent généralement une EEPROM i2c, mais reçoivent, en réalité, leur *firmware* directement par l'application cliente. Ce *firmware open source* **[3]** est chargé dynamiquement, transformant ce qui n'est qu'un simple circuit avec un MCU démarré en mode *bootloader* en un outil d'analyse directement utilisable. Cette approche n'est pas sans rappeler certains adaptateurs réseau et Wi-Fi nécessitant le chargement d'un BLOB (*Binary Large OBject*) par leur pilote pour pouvoir être utilisés et c'est là exactement le même principe de fonctionnement. L'intérêt, à condition que le *firmware* soit *open source*, est de pouvoir assurer une évolution du fonctionnement du périphérique dans le temps, et ce, sans avoir à mettre en



Ce genre de kit de développement se trouve sur les sites habituels (comprendre « AliExpress et consorts ») pour environ 6 euros. Notez que le connecteur mini USB témoigne clairement d'un matériel qui n'est pas ce qu'il y a de plus récent...

œuvre une procédure de flashage complexe. Ceci signifie également que le périphérique est impossible à « briquer »...

Les documentations **[4] [5] [6]** du microcontrôleur précisent clairement trois types de scénarios concernant l'utilisation de l'EEPROM, qui ne nous intéressent pas particulièrement ici, mais qui peuvent poser problème dans nos expérimentations. Pour les comprendre, voici comment démarre un FX2LP :

- au *reset*, le *bootloader* en ROM vérifie la présence d'une EEPROM sur le bus i2c ;
- si rien n'est détecté aux adresses 0x50 ou 0x51, le MCU est énuméré avec les identifiants VID/PID par défaut (04b4:8613) ;
- si une EEPROM est détectée à 0x50 (1010 et 000), le *bootloader* considère que c'est une EEPROM dite « de petite taille » (24C00 à 24C02, 16 à 256 octets) dont les données sont accédées via une adresse sur 8 bits ;
- si l'EEPROM est à 0x51 (1010 et 001), elle est considérée comme « de grande taille » (24C32, 24C64 ou 24C128, respectivement 4 Kio, 8 Kio et 16 Kio), avec un adressage sur 16 bits (« *double-byte-addressed* » dans la documentation) ;
- le premier octet de l'EEPROM est lu ;
- s'il s'agit de `0xc0`, le *bootloader* interprète le reste des données de l'EEPROM comme une configuration USB et s'énumère comme étant en mode *bootloader*, mais en substituant les VIP/PID par défaut par ceux stockés en EEPROM (c'est ce que fait généralement un analyseur logique) ;
- si le premier octet est `0xc2`, le reste du contenu de l'EEPROM est considéré comme un binaire de *firmware* (+ *metadata*) et est copié en RAM pour être exécuté. Le périphérique n'est alors plus accessible en mode *bootloader*.

Comme vous le voyez, une partie de l'adresse de l'EEPROM sur le bus i2c est utilisée pour déterminer sa taille (et implicitement son utilisation). Cet adressage se fait physiquement en mettant les broches A0, A1 et A2 du composant à la masse ou à VCC, comme c'est souvent le cas sur des puces i2c. La documentation n'impose pas directement l'utilisation d'une taille d'EEPROM pour un chargement dit *C0* ou *C2*, mais précise que généralement les EEPROM avec des adresses 16 bits sont utilisées pour *C2* et les autres pour *C0*.

Cette procédure nous indique également que c'est uniquement si le *bootloader* ne trouve pas d'EEPROM utilisable, si l'adresse i2c est différente de 0x50 ou 0x51, ou si le premier octet n'est pas `0xc2` que le chargement en RAM via USB est possible. Dans le cas de `0xc0`, c'est toujours possible, puisque le *bootloader* ne passe pas

la main à un *firmware* « maison », mais il faut alors utiliser les identifiants USB qui auront été lus par le *bootloader* durant l'énumération, qui ne seront donc probablement pas ceux par défaut (04b4:8613). C'est par exemple le cas avec un de mes clones Saleae Logic équipé d'une EEPROM 24C02 configuré avec les VID/PID 0925:3881.

À ce propos, ce qui va suivre est plus ou moins applicable à ces clones, à quelques nuances près. Cependant, il s'agit de périphériques dédiés à une tâche précise avec peu de broches/GPIO exposées directement et, lorsque c'est le cas, ce sera au travers d'un octuple *transceiver* 74HC245, ce qui peut être, ou non, un problème selon le projet que vous avez en tête. Mieux vaudra opter pour une carte de développement FX2LP qu'on trouvera au même prix qu'un analyseur bas de gamme, soit moins de 6 € **[7]**, proposant davantage de broches utilisables et une EEPROM de 16 Kio initialement vierge (si besoin).

2. DÉVELOPPEMENT AVEC SDCC ET FX2LIB

Cypress pour sa famille de composants FX2LP propose un SDK officiel, développé sur la base de la suite d'outils de développement de Keil (Keil PK51). Cet environnement (compilateur, assembleur, débogueur, IDE, etc.) est disponible gratuitement, mais il s'agit, bien entendu, d'un socle propriétaire. Une alternative relativement compatible et entièrement libre existe et c'est celle que nous allons utiliser ici. La chaîne de compilation, et ceci peut paraître étonnant aux lecteurs ayant suivi la série sur le Zilog Z80 sur platine, n'est autre que SDCC (*Small Device C Compiler*) qui sait parfaitement produire du binaire pour les microcontrôleurs à base 8051. N'importe quelle distribution GNU/Linux (x86 ou ARM) intégrera une version empaquetée du compilateur, et il en va de même pour [Free|Open|Net]BSD et consorts. La version utilisée ici est la 4.2.0 et elle s'installera donc directement via le système de gestion de paquets (ou ports).

Ceci devra être cependant complété d'une bibliothèque permettant d'exploiter réellement les fonctionnalités du microcontrôleur, et en particulier l'USB : FX2Lib **[8]** de Dennis Muhlestein. Ceci n'est pas aussi intégré que, par exemple, z88dk, mais nous pouvons très facilement régler le problème via le mécanisme de construction que nous allons mettre au point. Une fois SDCC installé sur votre système de développement, vous pouvez sans problème cloner le dépôt de Dennis et construire la bibliothèque :

```
$ cd kkpart
$ git clone https://github.com/djmuhlestein/fx2lib.git
$ cd fx2lib
$ make
make -C lib
make[1] : on entre dans le répertoire " /tmp/fx2lib/lib "
sdcc -mmcs51  -I../include -c serial.c -o serial.rel
sdcc -mmcs51  -I../include -c i2c.c -o i2c.rel
[...]
make[2] : on quitte le répertoire " /tmp/fx2lib/examples/usbmon_c "
make[1] : on quitte le répertoire " /tmp/fx2lib/examples "
```

Vous vous retrouverez alors avec un `lib/fx2.lib` qui est une archive `ar` directement utilisable avec les options `-L/chemin/fx2lib/lib -lfx2` de `sdcc` lors de l'édition de liens (`sdcc` appelle `sdld`). Cependant, plutôt que d'avoir FX2Lib dans un répertoire distinct, il est préférable, à mon sens, de directement intégrer cela à notre projet de démonstration. Ceci peut être fait via un simple sous-répertoire ou, si comme moi vous avez l'habitude d'utiliser systématiquement Git pour vos réalisations, sous la forme d'un sous-module ajouté avec `git submodule add` suivi du chemin vers le dépôt GitHub en question.

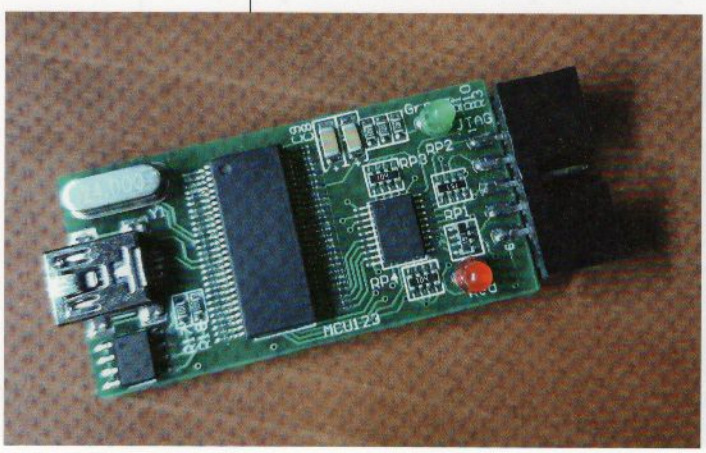
Ceci nous amène naturellement au système de construction lui-même et plus exactement au `Makefile` qui nous permettra de créer notre binaire, et plus encore. Notre objectif ici n'est pas juste le classique « *Hello World* » de la LED qui clignote, mais quelque chose de plus étoffé. Nous allons créer un *firmware* qui, certes, va piloter une LED, mais le fera en fonction des requêtes USB que nous lui enverrons. Comme pour l'exemple équivalent reposant sur le RP2040 de la Pico **[1]**, ceci se fera avec un petit code en C utilisant la LibUSB, mais ce n'est pas tout. Puisque le FX2LP peut parfaitement fonctionner sans EEPROM i2c et voir son *firmware* chargé via USB, notre client pourra donc utiliser n'importe quel *devkit* en mode *bootloader* en chargeant directement le code qu'il souhaite voir être exécuté. Ceci fait, il cherchera alors le périphérique possédant les identifiants USB spécifiés dans notre *firmware* et l'utilisera. Exactement comme Sigrok/Pulseview le fait.

En attendant de développer le client qui fait donc office de « chargeur », on pourra éventuellement reposer sur un outil tiers comme `cycfx2prog` **[9]** ou FXLoad **[10]** permettant, tous deux, de charger un code en SRAM et/ou de programmer l'EEPROM i2c. FXLoad est d'ailleurs également intégré à mon dépôt Git en sous-module, en cas de problème ou pour la mise au point du *firmware*. FX2Lib et FXLoad sont compilés directement dans leur répertoire respectif par le `Makefile` sous la forme d'une dépendance pour les cibles de construction du *firmware* binaire et/ou `load` (SRAM) et `flash` (EEPROM).

La création du binaire découlera de l'utilisation de `sdobjcopy` (le `objcopy` de SDCC), convertissant le fichier Intel Hex (`*.ihx`) produit par l'éditeur de liens automatiquement. Ce dernier aura la responsabilité de créer le fichier Hex à partir des objets provenant de la compilation (`main.rel` et `usb_descriptors.rel`) et de `fx2lib/lib/fx2.lib` préalablement obtenu avec `make -C fx2lib`.

Le client USB, que nous étudierons plus loin, est également construit ainsi, dans son propre sous-répertoire (`client/`) avec son propre `Makefile` et est une dépendance de compilation de la cible `runclient`. Bien entendu, comme le client charge également

Un analyseur logique comme celui-ci pourra également faire l'affaire pour s'initier au développement sur FX2LP, même s'il s'agit d'une solution moins pratique du fait de l'absence d'accès aisé à bon nombre de broches du microcontrôleur.



le *firmware* stocké dans le fichier binaire, sa compilation dépend de la construction de ce dernier. Au final, un simple `make runclient` fera tout le travail : il compilera FX2Lib, notre *firmware*, le client et lancera son exécution automatiquement, en une commande.

Ceci étant dit, il est temps de passer aux choses sérieuses et donc, au code.

3. FIRMWARE DE DÉMO, JUSTE UNE LED

Notre modeste projet nécessitera deux composants. Nous avons d'une part un code en C (`main.c`) très classiquement chargé de faire fonctionner la base de notre création (initialisation, configuration, boucle principale et fonction *callback* attendues par FX2Lib) et d'autre part, un morceau en assembleur (`usb_descriptors.s`) contenant les descripteurs USB caractérisant le périphérique que nous souhaitons créer. Je ne vais pas ici reprendre toute la litanie des explications sur la structure et le fonctionnement de l'USB, ceci est vastement couvert sur le Web et dans le précédent article concernant le RP2040 et TinyUSB. Sachez simplement qu'un périphérique USB est analysé (énumération) lors de sa connexion et qu'il décrit lui-même les fonctionnalités qu'il propose pour que le système puisse le prendre en charge.

Cette description prend la forme de « descripteurs » que nous devons rédiger et, dans le cas du FX2LP et de la FX2Lib, ceci prend la forme de données placées dans une zone précise de la mémoire. Comme il serait bien plus difficile de produire ces descriptions en binaire (ou devoir reposer sur un outil spécifique), l'alternative consiste donc à reposer sur un code assembleur composé de macros, de labels spécifiques et d'une tripotée d'instructions `db` plaçant simplement des octets dans le binaire résultant de l'assemblage (avec `sdas8051`). Fort heureusement pour nous, le dépôt Git FX2Lib regroupe une belle série d'exemples dont nous pouvons nous inspirer. Ceux comprenant un support USB sont facilement reconnaissables à la présence d'un fichier `dscr.a51`, correspondant à notre `usb_descriptors.s`. Je ne vais pas couvrir ici l'ensemble des quelque 250 lignes d'assembleur, mais simplement traiter quelques extraits, sachant que le projet lui-même (*firmware* et client) est disponible sur mon GitLab **[11]**.

Voici, par exemple, la partie décrivant le périphérique lui-même (*device descriptor*) :

```
_dev_dscr:
        .db     dev_dscr_end-_dev_dscr  ; taille
        .db     DSCR_DEVICE_TYPE        ; type
        .dw     0x1002                  ; usb 2.0
        .db     0xef                    ; classe (vendor specific)
        .db     0x02                    ; sous-classe (vendor specific)
        .db     0x01                    ; protocole (vendor specific)
        .db     64                      ; taille max paquets (ep0)
        .dw     0x0912                  ; ID vendeur
        .dw     0x0100                  ; ID produit
        .dw     0x0100                  ; ID version
        .db     1                       ; constructeur (chaîne)
        .db     2                       ; produit (chaîne)
        .db     4                       ; numéro de série (chaîne)
        .db     1                       ; nombre de configurations
dev_dscr_end:
```

Notez que notre périphérique sera identifié par les ID `1209:0001` et que le mot de 16 bits est ici spécifié en *little-endian* tel que le stipulent les spécifications USB. Le *little-endian* est omniprésent en USB du fait de son historique et des systèmes développés par les acteurs ayant participé à sa création (voir article sur la libiconv pour jouer avec les chaînes de caractères d'un module FTDI FT232R **[12]**).

On retrouve également, un peu plus loin, les descripteurs pour l'interface et les *endpoints* :

```
        .db     DSCR_INTERFACE_LEN
        .db     DSCR_INTERFACE_TYPE
        .db     0                       ; num interface
        .db     0                       ; alt setting
        .db     2                       ; nombre d'endpoints
        .db     0xff                    ; classe
        .db     0xff
        .db     0xff
        .db     3                       ; interface (chaîne)

; endpoint 1 out
        .db     DSCR_ENDPOINT_LEN
        .db     DSCR_ENDPOINT_TYPE
        .db     0x01                    ; ep1 dir=OUT
        .db     ENDPOINT_TYPE_BULK      ; type
        .db     0x20                    ; taille max paquets (LSB)
        .db     0x00                    ; taille max paquets (MSB)
        .db     0x00                    ; intervalle polling

; endpoint 1 in
        .db     DSCR_ENDPOINT_LEN
        .db     DSCR_ENDPOINT_TYPE
        .db     0x81                    ; ep1 dir=IN
        .db     ENDPOINT_TYPE_BULK      ; type
        .db     0x20                    ; taille max paquets (LSB)
        .db     0x00                    ; taille max paquets (MSB)
        .db     0x00                    ; intervalle polling
```

Classiquement, nous avons donc un périphérique, proposant une seule configuration, avec une interface et deux *endpoints*, avec 32 octets maximum pour chaque transaction vers et depuis ceux-ci.

Cette façon de procéder, en structurant des données binaires de façon assez spartiate, est très différente de ce qu'on trouve actuellement avec des MCU modernes et des choses comme TinyUSB, mais dans les grandes lignes, ceci est plus ou moins équivalent à composer un `usb_descriptors.c`. C'est juste un peu plus étrange. Ce qui l'est presque tout autant d'ailleurs, c'est la gestion de l'USB par le microcontrôleur ou plus exactement par la FX2Lib, qui se calque directement sur la solution proposée par le constructeur. En effet, ceci repose massivement sur la notion de fonctions *callback* qui doivent être présentes pour que le *framework* implémenté par la FX2Lib fonctionne correctement.

Ceci fait que la fonction `main()` est relativement simple et se contente de configurer le microcontrôleur (et la FX2Lib) avant d'entrer dans une boucle infinie simplissime :

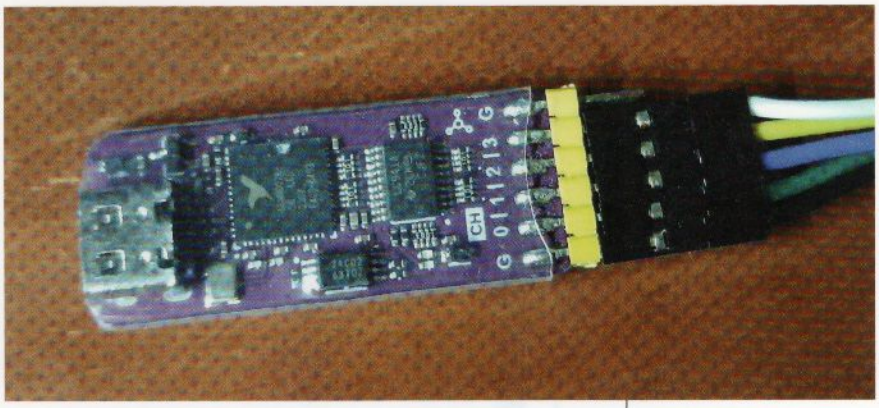
```
#include <string.h>
#include <stdio.h>
#include <fx2regs.h>
#include <fx2macros.h>
#include <autovector.h>
#include <setupdat.h>
#include <delay.h>
#include <eputils.h>

// setup data
volatile __bit got_sud;

void main(void) {
    // reénumération si changement d'ID USB
    RENUMERATE_UNCOND();

    // pas de gestion avancée des endpoints
    REVCTL = 0;
    // fréquence CPU
    SETCPUFREQ(CLK_48M);

    // gestion interruption
    USE_USB_INTS();
    ENABLE_SUDAV();
    ENABLE_HISPEED();
    ENABLE_USBRESET();

    // désactivation endpoint EP2, EP4, EP6 et EP8
    EP2CFG &= ~bmVALID;
    SYNCDELAY4;
    EP4CFG &= ~bmVALID;
    SYNCDELAY4;
    EP6CFG &= ~bmVALID;
    SYNCDELAY4;
    EP8CFG &= ~bmVALID;
    SYNCDELAY4;

    // configuration des GPIO leds
    OEA = (1 << 0) | (1 << 1);
    PA0 = 1; // led D1 off
    PA1 = 1; // led D2 off

    delay(10);

    // activation interruptions globales
    EA = 1;
```

```
    while(TRUE) {
        if (got_sud) {
            handle_setupdata();
            got_sud = FALSE;
        }
    }
}
```

Plusieurs choses sont intéressantes ici. La configuration est relativement simple et se limite à peu de choses, sachant que tout n'est pas forcément nécessaire, comme la désactivation des *endpoints* non utilisés. Notez à ce propos que les numéros des *endpoints* spécifiés ici ne sont pas leurs ID, mais ceux disponibles dans le MCU en nombre fini. La gestion des GPIO est également un point amusant, car alors que la direction de chaque ligne d'un port est configurée via un registre (ici `OEA`), impacter sur l'état d'une sortie se fait en changeant la valeur d'une variable qui la désigne directement (et non un bit dans un registre comme avec les AVR, par exemple). Et enfin, dernier point, notez que notre boucle `while` ne s'intéresse qu'au *flag* `got_sud`, indiquant la disponibilité de données de configuration, *SUD* signifiant *SetUp Data*. La gestion de ces données est faite directement par FX2Lib, ce qui simplifie grandement notre code.

De la même manière, la gestion des requêtes de type *vendor* (hors classe, donc) est gérée pour nous, mais nous devons implémenter une fonction *callback* pour les traiter. La nôtre sera la suivante :

```
#define CMD_SET_LED 0x15

BOOL handle_vendorcommand(BYTE cmd) {
    WORD value = SETUP_VALUE(); // wValue

    // équivalent aux macros LibUSB :
    //   LIBUSB_REQUEST_TYPE_VENDOR |
    //   LIBUSB_RECIPIENT_INTERFACE |
    //   LIBUSB_ENDPOINT_OUT
    if (SETUP_TYPE == 0x41) {
        switch (cmd) {
        case CMD_SET_LED:
            if (value != 0)
                PA0 = 0; // led on
            else
                PA0 = 1; // led off
            break;
        default:
            return FALSE; // bad cmd
        }
        return TRUE;
    }
    return FALSE;
}
```



Une déclinaison plus moderne des analyseurs logiques économiques, héritiers des clones des premiers Saleae Logic, repose sur exactement le même circuit. La connectivité USB-C et l'aspect bien plus compact en font cependant des outils parfaitement acceptables.

Contrairement à ce qu'on peut connaître avec TinyUSB sur RP2040, nous n'avons pas besoin de faire la distinction dans les types de requêtes puisque cette fonction se limite à ce qui nous intéresse. Nous pouvons donc très simplement implémenter notre petit protocole propriétaire simpliste consistant à espérer une requête `0x15` (macro `CMD_SET_LED`) et, le cas échéant, réagir en fonction de la valeur passée : si la requête s'accompagne d'une valeur nulle, nous éteignons PA0 et nous l'allumons dans le cas contraire.

D'autres fonctions *callback* doivent être implémentées, pour la gestion des configurations et la configuration des interfaces, mais se limitent à retourner un simple booléen (dans notre cas). Celles-ci ne nécessitent pas plus d'explications que celles présentes en commentaire dans le code de démonstration (et provenant des sources de FX2Lib) que je vous laisse consulter par vous-même.

Le dernier point très important du code, en général, concerne le *mapping* mémoire et l'éditeur de liens. En effet, les descripteurs USB doivent prendre place en RAM de la même manière que la table des vecteurs d'interruption, le code et la mémoire pour les données (tas, etc.). Tout ce petit monde doit se partager la RAM de 16 Kio puisqu'il n'y a pas, comme avec d'autres microcontrôleurs, de code exécuté en flash (notion de XIP pour *Execute In Place*). Nous devons donc indiquer à l'éditeur de liens l'emplacement des éléments clés ainsi que passer en argument à ce dernier les adresses de chaque zone (*area*). Tout ceci se fait directement via des options passées au compilateur qui utilisera `sdld` pour l'édition de liens à notre place. Comme nous utilisons un `Makefile`, nous pouvons intégrer ceci très facilement :

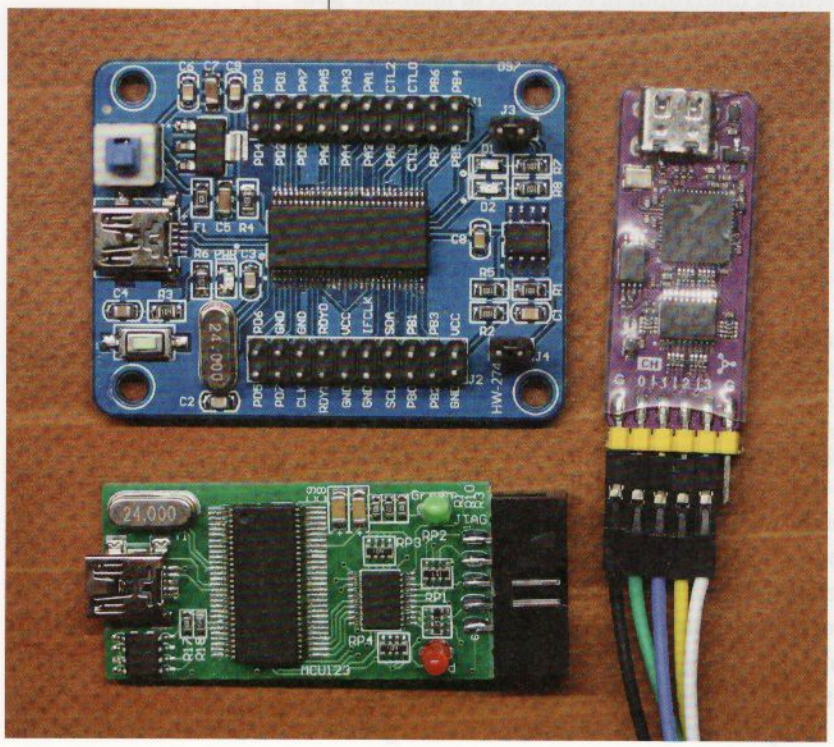
```
LDFLAGS := -mmcs51
LDFLAGS += --code-size 0x3c00
LDFLAGS += --xram-size 0x0200
LDFLAGS += --xram-loc 0x3c00
LDFLAGS += -L./$(FX2LIB)/lib -lfx2
LDFLAGS += -Wl"-b DSCR_AREA=0x3e00" -Wl"-b INT2JT=0x3f00"
```

Notre « RAM », du point de vue du code, débute donc en 0x3c00 et a une taille de seulement 512 octets (bien suffisante pour ce type de projet), et notre code débute en 0x0000 et s'étend sur quelque 15360 octets (0x3c00). Ceci laisse de la place pour les descripteurs USB de 0x3e00 à 0x3eff (256 octets) et pour la *jump table* des interruptions de 0x3f00 à 0x3fff (256 octets). Comme pour GCC/Clang, l'option `-Wl` permet de passer des options à l'éditeur de liens lui-même et nous spécifions ainsi les adresses de début de chaque zone. On retrouvera d'ailleurs ces symboles dans le fichier `main.map` généré :

```
[...]
C:   00003C00  s_XSEG
C:   00003E00  s_DSCR_AREA
C:   00003F00  s_INT2JT
[...]
User Base Address Definitions

HOME = 0x0000
XSEG = 0x3c00
PSEG = 0x3c00
ISEG = 0x0000
BSEG = 0x0000
DSCR_AREA=0x3e00
INT2JT=0x3f00
```

J'aborde ce point spécifique, car bon nombre de codes qu'on trouve en ligne utilisent un *mapping* pouvant être différent. Vous-même pouvez d'ailleurs souhaiter avoir davantage de mémoire « vive » pour votre projet et donc réduire celle dédiée au code. Il est amusant de relever que c'est là quelque chose qui n'est pas envisageable avec des MCU comme ceux des Arduino. Un ATmega328P, par exemple, dispose de 2 Kio de SRAM (et de 32 Kio de flash pour le code), s'il vous en faut plus, vous n'avez pas le choix, il faut passer à l'ATmega2560 de l'Arduino Mega 2560...

Tous ces périphériques coûtent, peu ou prou, la même chose et pourront servir pour apprendre à développer du code pour le MCU Cypress CY7C68013A. Le format « devkit » (en haut à gauche) est, à mon sens, à privilégier.



Ceci conclut la partie concernant le *firmware* et nous pouvons nous pencher brièvement sur le client USB.

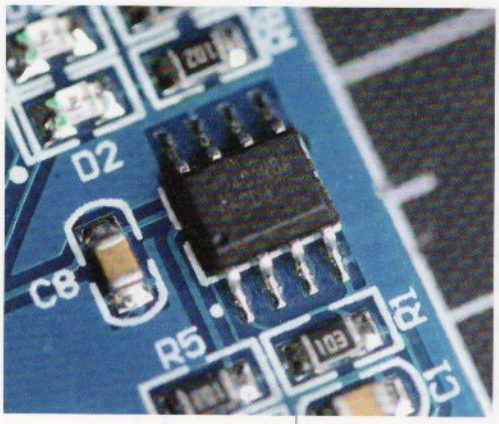
4. PARTIE CLIENTE ET CHARGEUR

Comme détaillé précédemment, notre client en C destiné à fonctionner sur n'importe quel système capable d'utiliser la libUSB (typiquement un Unix *open source*) aura deux tâches, charger notre *firmware* et, ceci fait, piloter la LED connectée à PA0 via des requêtes type *vendor*. La détection du périphérique est relativement simple puisque nous nous basons sur les ID USB :

si 1209:0001 n'est pas trouvé (notre périphérique), nous cherchons 04b4:8613 (le *bootloader*) et utilisons une requête *vendor* propre à ce dernier pour écrire dans la RAM du FX2LP, après avoir effectué un « reset+stop ». Nous relisons ensuite le contenu de la RAM pour vérification et si tout est en ordre, nous procédons à un « reset+continue » et attendons l'apparition du périphérique (énumération par l'OS) avec les bons identifiants, pour pouvoir l'utiliser.

Je vous ferai grâce ici des fonctions d'initialisation et autres procédures standard concernant la gestion des options de la ligne de commande et du chargement du binaire depuis un fichier avec `mmap()`. Tout est dans le dépôt GitLab et nous allons nous concentrer sur les échanges avec le microcontrôleur en commençant par la fonction permettant d'écrire en RAM, car celle-ci est également utilisée pour gérer les *resets* :

```
int fx2writeRAM(libusb_device_handle *handle,
    uint16_t addr, unsigned char *data, size_t len)
{
  int ret;
  int nbrerr = 0;
  unsigned char *pdata = data;
  unsigned char *end = data + len;

  if (len > RAMSIZE)
    return(-1);

  while (pdata < end) {
    size_t blksz = end - pdata;

    if (blksz > CHUNKSZ)
      blksz = CHUNKSZ;

    size_t loadaddr = addr + (pdata - data);

    if ((ret = libusb_control_transfer(
            handle,
            (LIBUSB_REQUEST_TYPE_VENDOR |
             LIBUSB_RECIPIENT_INTERFACE |
             LIBUSB_ENDPOINT_OUT), //bmRequestType
            CMD_RWDATA, // bRequest
            loadaddr, // wValue
            0, // wIndex
            pdata, // wData
            blksz, // wLength
            1000) // timeout
          ) < 0) {
      warnx("libusb_control_transfer() failed");
      nbrerr++;
    }
   pdata += blksz;
  }
  return(nbrerr);
}
```



Le devkit FX2LP intègre une EEPROM i2c 24C128 de 16 Kio permettant de stocker l'ensemble d'un firmware pouvant occuper la totalité de la SRAM du microcontrôleur.

L'écriture dans la RAM du CY7C68013A passe par l'utilisation de multiples requêtes USB `0xa0` à destination du *endpoint* OUT, accompagnées de l'adresse de départ sur 16 bits en guise de valeur et des données à inscrire par blocs de 16 octets (`CHUNKSZ`). Ceci explique la boucle `while` parcourant `data[]` et incrémentant `loadaddr`. La fonction `fx2readRAM()` permettant la lecture en RAM est quasiment identique, si ce n'est par le *endpoint* concerné (`LIBUSB_ENDPOINT_IN`) spécifié dans le type de requête (*bmRequestType* dans les spécifications). Notez au passage que ce type de requête est exactement le même que celui que nous utilisons pour notre *firmware* et que nous testons dans la fonction *callback* `handle_vendorcommand()`. Le contrôle de la LED sera donc très similaire au chargement du binaire.

Nous avons maintenant des fonctions de lecture et d'écriture en RAM du MCU et pouvons nous pencher sur les *resets*. Ceux-ci se font justement par écriture d'un unique octet à une adresse bien précise, et invalide, de la RAM (`0xe600` ou 58880 en décimal, au-delà des 16 Kio de RAM réelle). Cette opération est interceptée par le *bootloader* et interprétée comme une directive pour provoquer un *reset*. L'octet écrit détermine le type de *reset* avec 1 pour un *reset* avec arrêt et 0 pour un *reset* et exécution du *firmware*. Ceci se traduit respectivement par les macros `STATE_STOP` et `STATE_RUN` dans le nôtre, et la fonction ressemble à :

```
int fx2reset(libusb_device_handle *handle, unsigned char state)
{
  const uint16_t reset_addr = 0xe600;
  return(fx2writeRAM(handle, reset_addr, &state, 1));
}
```

Avec ces trois fonctions, nous pouvons implémenter celle permettant le chargement du *firmware* (`loadfirmware()`) qui repose sur la procédure suivante :

```
// reset + stop MCU
if (fx2reset(handle, STATE_STOP) != 0) {
  warnx("Reset error!");
  return(-1);
}

// écriture en RAM
if (fx2writeRAM(handle, 0x0000, fw, fwsize) != 0) {
  warnx("Write RAM error!");
  return(-1);
}
```

```
// lecture RAM
if (fx2readRAM(handle, 0x0000, buffer, fwsize) != 0) {
  warnx("Read RAM error!");
  return(-1);
}

// comparaison
if (memcmp(fw, buffer, fwsize) != 0) {
  warnx("RAM verify failed!");
  dumpbuf(buffer, fwsize);
  return(-1);
} else {
  printf("Firmware loaded and verified\n");
}

// reset + exec
if (fx2reset(handle, STATE_RUN) != 0) {
  warnx("Reset error!");
  return(-1);
}
```

Et il ne nous reste plus, alors, qu'à nous occuper de la logique de détection du périphérique, rendant conditionnel le chargement du *firmware*, dans `main()` :

```
if ((ret = libusb_init(&ctx)) < 0)
  err(ret, "LibUSB initialisation error");

if ((handle = libusb_open_device_with_vid_pid(
    ctx, VID, PID)) == NULL) {
  printf("Device not found. Load firmware...\n");
  if (loadfirmware(ctx) != 0) {
    libusb_exit(ctx);
    errx(EXIT_FAILURE, "load firmware error");
  }

  printf("Waiting for device");

  while (trycount < 10 && handle == NULL) {
    printf(".");
    fflush(stdout);
    handle = libusb_open_device_with_vid_pid(
      ctx, VID, PID);
    trycount++;
    sleep(1);
  }
  printf("\n");
```

```
    if (handle == NULL) {
      libusb_exit(ctx);
      errx(EXIT_FAILURE, "Unable to find device");
    }

    printf("Device found\n");
  }

  dostuff(handle);
```

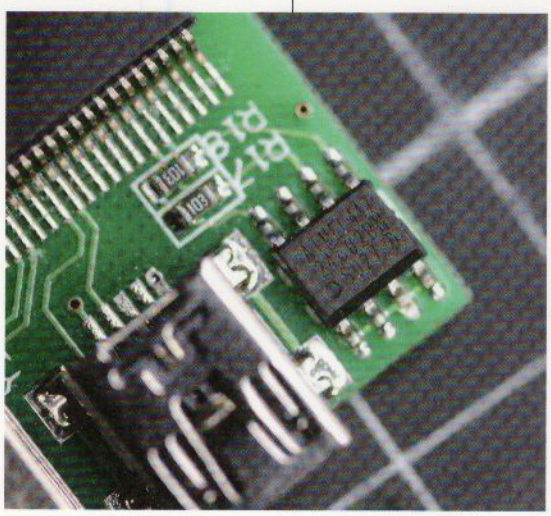
`dostuff()` est la fonction qui dialogue avec le périphérique faisant fonctionner notre *firmware* et qui se limite à envoyer deux requêtes 0x15 (`CMD_SET_LED`), espacées d'une seconde, utilisant la valeur 0x00ff puis 0x0000 pour respectivement allumer puis éteindre la LED. Rien de très différent des appels à `libusb_control_transfer()` de `fx2writeRAM()`, si ce n'est par l'absence de données et donc un pointeur `NULL` pour d'argument `data` et `0` pour `wLength`.

Nous avons maintenant quelque chose presque en tout point similaire à ce que fait Sigrok : initialiser le périphérique et l'utiliser directement. Bien entendu, tant que le *devkit* est sous tension, il n'est pas utile de charger une nouvelle fois le *firmware* et ceci est parfaitement pris en charge par le code client.

Les analyseurs logiques utilisent généralement un firmware chargé dynamiquement, mais comportent tout de même une EEPROM i2c destinée à contenir la configuration sur 128 octets ou moins. C'est la présence de ce composant (et de son contenu) qui fait que le matériel est énuméré avec des identifiants qui ne sont pas ceux par défaut du bootloader.

5. BONUS : RP2040 ET FAUX SALEAE LOGIC ANALYZER

En guise de complément, sachez que ces expérimentations ont également été réalisées avec un analyseur logique du type « clone de Saleae Logic première génération » et ceci fonctionne à merveille. Pour test, la LED rouge intégrée au circuit a été utilisée et sa cathode est reliée à PB0 (port B). Le signal en sortie doit donc également être inversé, mais en dehors de cela, le code du *firmware* ne change pas. Le client, lui aussi, doit être modifié puisque le périphérique, bien que n'utilisant pas un *firmware* stocké en EEPROM i2c, repose sur un chargement *C0*. La puce 24C02 présente sur le circuit contient donc 256 octets de configuration et le matériel est énuméré avec les identifiants 0925:3881 et non 04b4:8613 (*bootloader*) lors de la connexion. Pour que le client fonctionne, il suffit donc d'adapter `FXVID` et `FXPID` avec les nouvelles valeurs.



Le second bonus concernera le RP2040 (non testé sur RP2350) des Raspberry Pi Pico qui, comme les FX2LP, peut également fonctionner sans mémoire de stockage pour le

code. Je n'ai pas cherché à développer un client identique pour cette plateforme, mais cela est parfaitement possible. Il suffirait de s'inspirer des sources de l'outil officiel `picotool` **[13]** qui, justement, sert de « chargeur » pour procéder à un essai. Notez que le chargement d'un tel *firmware* doit être fait en utilisant le fichier ELF (ou UF2) résultant de la construction, le `.bin` ne disposant pas des informations concernant les sections (voir la sortie d'un `arm-none-eabi-readelf -a votre_firmware.elf`). De plus, une ligne précisant le type de binaire à produire lors de la compilation doit être ajoutée au `CMakeLists.txt` de votre projet :

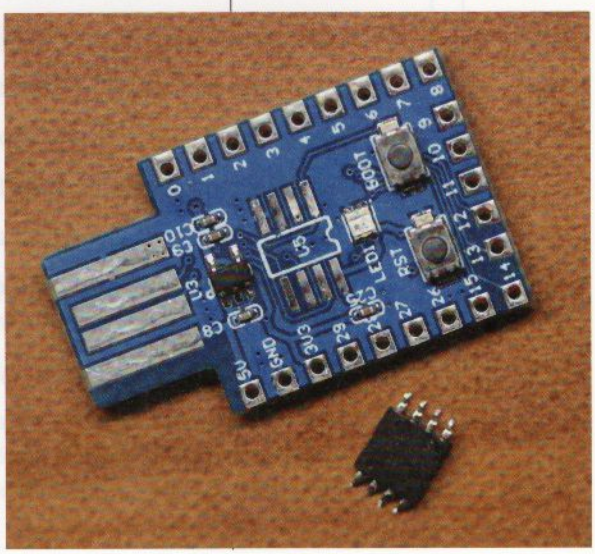
```
pico_set_binary_type(
  ${CMAKE_PROJECT_NAME} no_flash
)
```

La principale différence de fonctionnement entre un FX2LP et un RP2040 concerne précisément la mémoire de stockage. Dans le premier cas, il y a copie de l'EEPROM i2c en RAM, alors qu'avec le RP2040 le code est exécuté directement en flash, sans copie. Mais, oui, il est parfaitement possible d'utiliser une Pico (ou un clone) ainsi, même avec la flash dessoudée, sachant que contrairement au CY7C68013A, vous aurez 264 Kio de SRAM à disposition, et non simplement 16 Kio. Une Pico sans flash n'est donc absolument pas bonne pour la poubelle, loin de là !

CONCLUSION ET REMERCIEMENT

Pour conclure ces explications, je tiens à remercier chaleureusement Jean-Michel Friedt qui, grâce à son article **[14]** sur la carte d'évaluation du MAX2771 de Maxim IC pilotée par un *devkit* FX2LP, a éveillé ma curiosité concernant le CY7C68013A. En effet, bien qu'utilisant fréquemment ce microcontrôleur, presque malgré moi, pour les tâches d'analyse logique, il ne m'était jamais venu à l'idée de l'envisager comme une plateforme de développement au même titre qu'une carte Raspberry Pi Pico ou ESP32, et encore moins soupçonner la possibilité d'utiliser SDCC pour ce faire.

Vais-je poursuivre mes expérimentations avec ce MCU plutôt que de reposer uniquement sur des fonctionnalités équivalentes proposées par le RP2040/RP2350 ou les ESP32 ? Peut-être, mais je pense qu'il s'agira surtout de l'utiliser pour mieux comprendre



Cette carte en provenance d'AliExpress a été sacrifiée sur l'autel de la science et a été sauvagement amputée de sa flash pour vérifier l'utilisabilité d'un MCU RP2040 sans stockage pour le code. Rassurez-vous, le composant a été regreffé par la suite, je ne suis pas totalement un monstre...

le fonctionnement du *firmware* fx2lafw **[3]** du projet Sigrok, et en particulier les fonctions de configuration avancées des *endpoints* avec FX2Lib, ainsi que le transfert rapide de données. Pour la création d'un simple périphérique USB « standard » en revanche, je pense que les environnements de développement comme l'ESP-IDF ou le Pico SDK me paraissent un tantinet plus confortables, ou alors est-ce juste une perception biaisée, du fait que « SDCC » rime encore, pour moi, avec « retro » et « Z80 »... **DB**

RÉFÉRENCES

[1] **https://connect.ed-diamond.com/hackable/hk-056/creez-vos-peripheriques-usb-avec-raspberry-pi-pico**

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

[3] **https://sigrok.org/wiki/Fx2lafw**

[4] **https://www.infineon.com/dgdl/Infineon-CY7C68013A_CY7C68014A_CY7C68015A_CY7C68016A_EZ-USB_FX2LP_USB_Microcontroller_High-Speed_USB_Peripheral_Controller-DataSheet-v31_00-EN.pdf?fileId=8ac78c8c7d0d8da4017d0ec9f7974252**

[5] **https://www.infineon.com/dgdl/Infineon-EZ-USB_TECHNICAL_REFERENCE_MANUAL-AdditionalTechnicalInformation-v08_00-EN.pdf?fileId=8ac78c8c7d0d8da4017d0f9093657d61**

[6] **https://www.infineon.com/dgdl/Infineon-AN50963_EZ-USB_FX1_FX2LP_Boot_Options-ApplicationNotes-v13_00-EN.pdf?fileId=8ac78c8c7cdc391c017d072418a1482a**

[7] **https://fr.aliexpress.com/item/976713163.html**

[8] **https://github.com/djmuhlestein/fx2lib**

[9] **https://github.com/tai/cycfx2prog**

[10] **https://github.com/mbed-ce/fxload**

[11] **https://gitlab.com/0xDRRB/fx2lp_blink**

[12] **https://connect.ed-diamond.com/gnu-linux-magazine/glmf-269/manipulons-les-caracteres-avec-iconv**

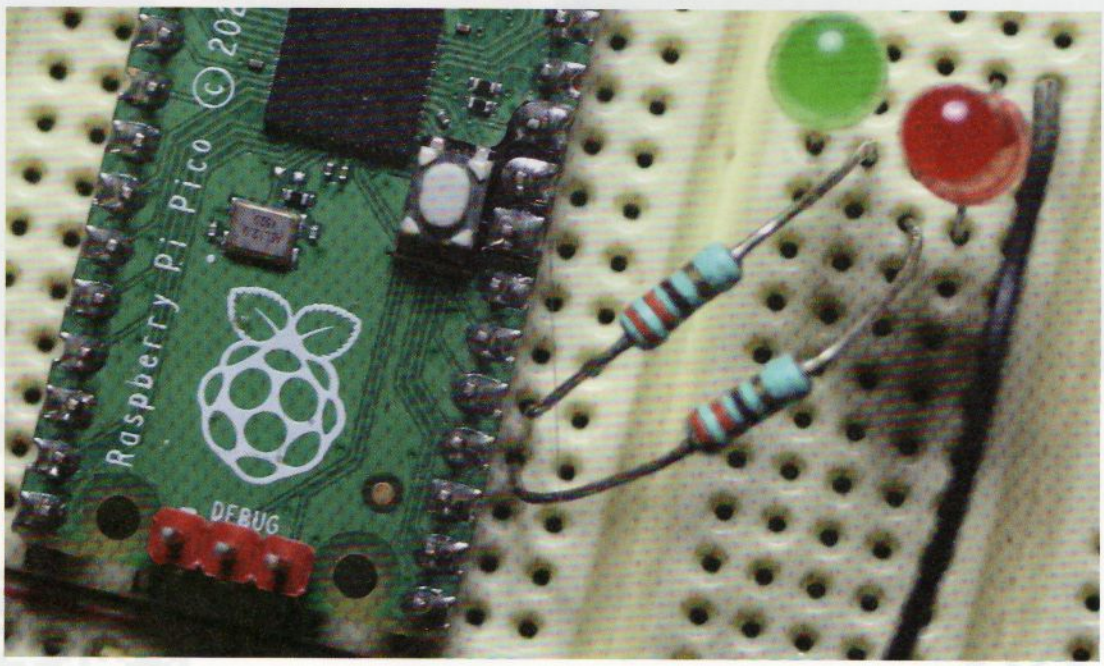
[13] **https://github.com/raspberrypi/picotool**

[14] **https://connect.ed-diamond.com/hackable/hk-057/programmation-usb-sous-gnu-linux-application-du-fx2lp-pour-un-recepteur-de-radio-logicielle-dediee-aux-signaux-de-navigation-par-satellite-1-2**

SÉCURISER TOUT ET N'IMPORTE QUOI AVEC DES MINI-SIGNATURES

Denis Bodor

De nos jours, vérifier l'authenticité d'un message, qu'il s'agisse de celui provenant d'un humain ou d'une machine est devenu presque quelque chose de totalement obligatoire. Tout un tas de mécanismes sont d'ores et déjà en place pour bon nombre de protocoles et de formes d'échanges, qu'il s'agisse du certificat d'un site web, d'un mail signé avec OpenPGP, d'une authentification pour une connexion distante avec SSH, et j'en passe. Mais qu'en est-il pour les petites choses, comme une simple ligne de texte, une URL ou la donnée embarquée dans un QRcode ?



– Sécuriser tout et n'importe quoi avec des mini-signatures –

L'idée derrière cet article provient d'une simple réflexion faite à moi-même en regardant la présentation **[1]** de Melvin Langvik à DEF CON 32 : « Tiens, je me demande si on peut embarquer une signature électronique dans le texte encodé d'un QRcode ». La réponse est évidemment, « oui », puisqu'on peut tout encoder dans un QRcode qui est, je le rappelle, une simple déclinaison du principe du bon vieux code-barre (1D), mais en version 2D, et donc avec une densité d'information bien supérieure à un simple EAN-13 présent sur nos boîtes de petit-pois.

Un QRcode peut encoder bien des choses : un simple texte, une adresse mail, une carte de visite (vCard), une URL... Mais l'espace disponible n'est pas illimité, du moins si l'on veut que le résultat soit lisible et décodable avec un jeu d'équipements le plus large possible, et je pense en particulier aux lecteurs de style « douchettes », souvent bien moins performants que le superordinateur de poche surpuissant et communiquant qu'on appelle vulgairement un smartphone.

La problématique qui se pose alors est celle de la taille de la fameuse signature, car pour affirmer l'authenticité d'une URL, d'une adresse mail ou d'un court message, ne dépassant pas quelques dizaines de caractères, il n'est pas question d'ajouter un bloc de données 5 ou 10 fois plus grand. Un élément important du cahier des charges est donc le choix pertinent de l'algorithme de signature pour minimiser l'encombrement, mais également le fait de limiter les dépendances en termes de bibliothèques utilisées.

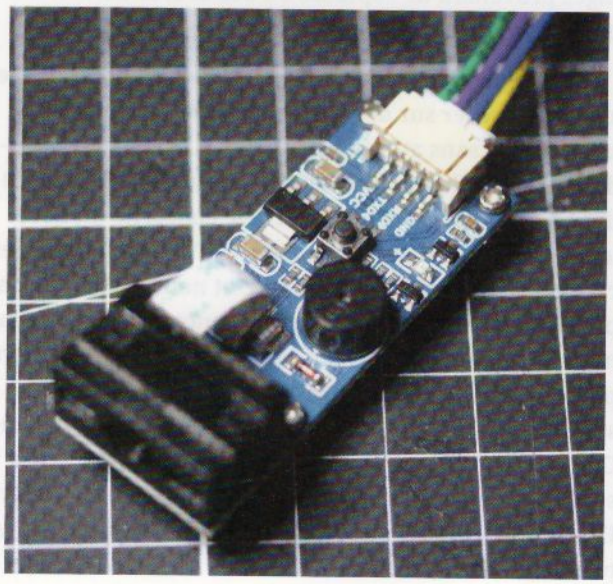
En effet, nous sommes ici coincés entre deux problématiques. La première consiste à absolument éviter (sinon s'interdire purement et simplement) le développement ou la conception d'algorithmes de chiffrement ou de signature et, dans une certaine mesure, leurs implémentations. Réinventer la roue est le meilleur moyen d'ouvrir des brèches et de rendre le résultat peu fiable en se donnant une fausse impression de sécurité. Il en va de même pour « faire le malin » et arrondir les angles avec des approches présomptueuses de type « je vais utiliser la moitié de la signature obtenue, ça suffira ». La cryptographie ne fonctionne pas comme ça. La seconde problématique est de ne pas reposer sur des classiques comme OpenSSL, wolfSSL, Libgcrypt ou encore libsodium, car le code doit être portable et pouvoir être décliné en version MCU (ARM, ESP8266, ESP32, RISC-V, etc.). En effet, il faut penser au-delà de, par exemple, la simple génération de QRcode sur PC ou Raspberry Pi et envisager, à la base, d'utiliser ce code pour d'autres choses, comme authentifier des messages provenant de capteurs ou des ordres donnés à un périphérique IoT « maison ».

Pour régler ce problème, il faut donc faire des concessions et reposer sur des implémentations « simples », portables et sans autres dépendances qu'une libc standard, d'algorithmes connus, validés et réputés sûrs. Ce dont il est question dans cet article concerne un développement sur PC/Mac/RPi, mais, et c'est aussi l'une des raisons de tout implémenter en C et non dans un langage plus « moderne », pour valider le concept et faciliter la mise au point du code, mais devra être facilement transposable sur RPi Pico (RP2040 ou RP2350), sur ESP32, sur STM32 et peut-être même sur Arduino (AVR 8 bits). On pourrait même envisager, à terme, d'intégrer cela, sous forme de contribution, dans quelque chose comme ESPHome, Tasmota ou Home Assistant, mais ça c'est une tout autre aventure...

1. CONDENSÉ, CHIFFREMENT ASYMÉTRIQUE ET SIGNATURE

Signer électroniquement un élément comme un message, un texte, une URL, un e-mail, etc., appelé dans le domaine un **document électronique** a pour objectif d'en garantir d'authenticité. On parle alors de **non-répudiation**, dans le sens où l'identité du signataire est rattachée au document et ne peut être remise en cause. Ceci implique également que la signature garantit le document comme authentique puisque celui-ci ne peut être **falsifié** (modifié ou altéré) sans que la signature devienne invalide et que l'entité l'ayant signé ne peut nier l'opération (notion d'**irrévocabilité**). Enfin, la signature électronique, contrairement à une signature manuscrite humaine, n'est pas réutilisable, on ne peut pas simplement copier-coller une telle signature d'un document à l'autre.

Pour accomplir ce petit tour de force, le mécanisme de signature repose sur le chiffrement asymétrique où une clé est utilisée pour chiffrer et une autre pour déchiffrer. Il n'est pas possible de chiffrer avec la clé de déchiffrement, dite **clé publique** et inversement, il n'est pas possible de déchiffrer avec la clé de chiffrement, dite **clé secrète** ou **clé privée**. La clé publique, comme son nom l'indique, peut donc être transmise à n'importe qui souhaitant déchiffrer un élément chiffré ou **cryptogramme**. En revanche, la clé privée doit impérativement rester secrète et utilisable uniquement par la (ou les) entité(s) devant être en mesure de chiffrer.



Parmi tous les modules qu'on peut trouver sur le Net, c'est ce produit WaveShare que j'ai utilisé pour les tests (parce que je l'avais déjà). Notez que le circuit ne comprend strictement aucun circuit logique ou microcontrôleur. Juste un régulateur de tension, un buzzer, une LED, des MOSFET et une poignée de composants passifs. Toute « l'intelligence » semble être intégrée dans l'optique elle-même.

Il existe de nombreux algorithmes de chiffrement asymétrique, dont le plus connu est sans doute RSA, nommé d'après les initiales de ses trois inventeurs à la fin des années 70 : Ronald Rivest, Adi Shamir et Leonard Adleman. Un autre très connu est DSA (*Digital Signature Algorithm*) standardisé alors que RSA tombait encore sous le coup d'un brevet déposé par le MIT en 1983 (expiré en 2000). Et enfin, nous avons ECDSA (et ses déclinaisons), pour *Elliptic Curve Digital Signature Algorithm*, qui se démarque du duo sacré RSA/DSA par le fait que les clés sont plus courtes et les calculs de chiffrement/déchiffrement plus rapides. Ce qui explique que, par exemple, ECDSA et EdDSA/Ed25519 soient de plus en plus utilisés pour SSH en lieu et place de RSA ou DSA (voir la page de manuel de `ssh-keygen`).

Chiffrer et déchiffrer des « documents électroniques » c'est bien, mais ce que nous

voulons, c'est signer et non rendre nos messages illisibles pour qui n'a pas la clé publique. La solution à ce problème passe par la génération d'un **condensé** ou **valeur de hachage** (ou juste « hachage » ou « *hash* » en anglais). On parle aussi tantôt de condensat, d'empreinte ou de *digest*. Une fonction de hachage permet d'obtenir une donnée de taille fixe en fonction d'une donnée en entrée de taille variable, et ce, avec des propriétés particulières. Nous parlons ici de la sous-catégorie des fonctions de hachage cryptographiques qui se distinguent par le fait qu'elles fonctionnent à sens unique : obtenir un *hash* correspondant à une donnée en entrée est rapide et aisé, mais l'opération inverse, retrouver la donnée à partir du *hash* est impossible en pratique. Autre caractéristique importante, il n'est pas possible de créer deux messages différents résultant en un *hash* identique. On parle dans ce cas de **collision** et ceci signe généralement la fin de l'utilisation de la fonction en question, comme ça a été le cas pour MD4, MD5 et SHA-1 **[2]**. SHA-256 et SHA-512 (pour *Secure Hash Algorithm*) sont typiquement les fonctions de hachage généralement utilisées et reconnues comme fiables et sûres.

De manière générale, pour obtenir une signature électronique, il suffit de combiner une fonction de hachage et un algorithme de chiffrement asymétrique ainsi :

- l'émetteur du message génère une paire de clés (privée et publique) ;
- il diffuse la clé publique ;
- il compose un message de taille arbitraire ;
- il produit un *hash* correspondant ;
- il chiffre le *hash* avec sa clé privée ;
- et enfin, il regroupe le message original et le *hash* chiffré pour le transmettre au destinataire.

À l'autre bout de la communication, le récepteur du message signé peut vérifier la signature ainsi :

- il sépare le message de la signature ;
- il déchiffre la signature à l'aide de la clé publique pour obtenir le *hash* original ;
- il calcule un *hash* identique sur le message réceptionné ;
- et il compare les deux *hashes*. S'ils sont identiques, le message est authentique, inaltéré et la signature est irrévocable. C'est bien l'expéditeur possédant la clé privée correspondant à la clé publique qui est à l'origine du message.

Comme vous pouvez le voir, le chiffrement asymétrique est utilisé sur une information dérivée du message et c'est le fait de pouvoir correctement déchiffrer cette signature qui apporte la preuve de l'intégrité et de la source du message. C'est ainsi que tous les processus de signature fonctionnent, qu'il s'agisse de vérifier un document signé avec un certificat TLS/SSL, en utilisant un outil dédié comme OpenPGP ou lors d'une authentification par clé avec SSH.

Le même principe peut également servir, justement, pour une authentification, sous la forme d'un challenge, mais c'est quelque chose que nous ne couvrirons pas ici. La seule chose qui nous intéresse est la signature et elle doit être la plus petite possible.

La fonction de hachage sera très classiquement SHA-256, mais n'importe quelle autre pourra être utilisée étant donné que la taille du *hash* n'a pas réellement d'importance. En ce qui concerne l'algorithme de chiffrement, nous utiliserons ECDSA qui est, de base, un algorithme de

L'optique utilisée sur le lecteur WaveShare intègre une caméra et deux éléments pour éclairer la cible, une LED blanche pour les codes 2D (à droite) et une LED projetant un faisceau rouge pour les codes 1D (à gauche) type EAN, etc. Le manuel indique toutes les caractéristiques optiques, dont les distances minimum et maximum ainsi que les angles de rotation (X, Y, Z) pour un scan optimal.

signature électronique en lui-même (c'est ECIES qui est utilisé pour le chiffrement). Le *hash* nous sert donc d'intermédiaire entre la signature et le message à signer/vérifier.

En vous penchant sur ECDSA et sur le domaine de la cryptographie sur les courbes elliptiques, vous vous rendrez rapidement compte que la notion de courbe elliptique ne désigne pas un élément précis et unique, mais une famille de courbes algébriques. Les propriétés recherchées pour une application en cryptographie (signature et échange de clé) font qu'il existe des ensembles de paramètres plus adaptés que d'autres et qu'il n'y a donc pas une seule « implémentation » d'ECDSA, mais plusieurs, en fonction du niveau de sécurité recherché et donc des paramètres de la courbe utilisée. Pour faciliter les choses, des organismes spécialisés, comme le NIST américain ou Certicom, recommandent des courbes elliptiques différentes désignées par un nom : P-192, P-256 ou encore P-521 pour NIST, et secp112r1, secp160r1, secp160r2 ou sect409k1 pour Certicom. Il existe même des équivalences, tantôt accompagnées du nom d'un standard ANSI, comme pour NIST P-256 qui est le secp256r1 de Certicom par exemple, mais aussi prime256v1 pour l'ANSI (voir RFC4492 **[3]**). À noter qu'il existe aussi la courbe FRP256v1 dont les paramètres ont été publiés au Journal officiel en 2011 **[4]** et qui est celle recommandée par l'ANSSI à l'époque (ceci a été complété depuis avec d'autres courbes dont Curve25519, alias Ed25519).

Le jeu de paramètres de courbe que nous choisirons d'utiliser ici sera secp160r1 (également appelé wap-wsg-idm-ecid-wtls7 et ansip160r1) qui offre, je pense, un bon équilibre entre l'équivalent RSA/DSA en nombre de bits et la taille de la signature (40 octets), même si nous ne respectons les recommandations de personne (voir **[5]**). À noter qu'avec ECDSA, la taille de la signature est identique à celle de la clé publique, qui est le double de la taille de la clé privée, sauf exception. Et secp160r1 en est une, avec 160 bits (d'où le « 160 » dans le nom), mais 21 octets (160/8+1) pour la clé privée et donc 40 octets pour la clé publique et la signature. Ceci nous donnera donc 80 caractères en hexadécimal, mais il est possible de faire plus court (56 caractères) en encodant le résultat en Base64.

2. POC SUR PC/LINUX (OU PI)

Ce modeste projet est relativement générique et surtout est censé, dès le départ, être portable. Ainsi, la majorité du code utilisé et développé devra fonctionner aussi bien sur un PC ou un SBC ARM ou RISC-V avec des ressources à foison, que sur quelque chose de relativement modeste comme une carte Raspberry Pi Pico (microcontrôleur RP2040) ou un MCU Espressif (ESP8266 ou ESP32). Il doit être également possible d'envisager de porter cela sur Arduino, mais il faut avouer que les

microcontrôleurs AVR 8 bits et leurs très faibles capacités sont maintenant presque totalement obsolètes. Je n'ai donc pas fait d'effort dans ce sens ou testé quoi que ce soit en rapport.

Avant de « porter » sur Pico, nous pouvons donc tout implémenter sur un système GNU/Linux, en profitant pleinement du confort et de la simplicité d'une telle plateforme. Précisons tout de même que l'outil en ligne de commande pour ce type de systèmes est censé faire bien plus que son équivalent sur microcontrôleur : générer des clés, signer et vérifier des signatures. Côté Pico, le but est simplement de vérifier un message signé dans l'idée de coupler la carte avec un module de lecture de QRcode interfacé pour une communication série. Ce genre de matériel se trouve pour environ 20 à 40 euros sur les sites habituels **[6] [7] [8]** et j'ai personnellement utilisé un modèle de chez Waveshare **[9]** traînant dans un tiroir depuis des années.

Notez que le développement sur PC/SBC pourra également valider l'utilisation d'un lecteur de QRcode, étant donné que ce type de périphérique propose généralement deux types d'interfaces : UART et USB-HID. En USB, le lecteur se présente comme un périphérique de saisie (comprendre « un clavier ») et tout ce qui est lu et décodé correspond donc à une communication via l'entrée standard (*STDIN*) du point de vue d'un programme. En d'autres termes, entre un `echo "mon message" | monprogramme` et une lecture de QRcode, il n'y a aucune différence. Les subtilités ne concernent que la configuration du lecteur, mais nous en toucherons un mot plus loin dans l'article.

Ne pas avoir de dépendances d'exécution, comme avec OpenSSL par exemple, ne signifie pas ne pas reposer sur du code tiers. Tout recoder manuellement, en particulier lorsqu'il s'agit de cryptographie, est généralement une très mauvaise idée. Nous avons besoin de produire des *hashes*, de signer et vérifier des signatures ECDSA et d'encoder/décoder du Base64. Pour chaque fonctionnalité, nous allons utiliser une implémentation libre qui pourra être intégrée à notre projet. Comme nous souhaitons rendre cela le plus modulaire possible, les sources seront structurées avec notre code à la racine de notre dépôt (puisque, bien sûr, tout ceci vit dans un dépôt Git **[10]**) et des sous-répertoires par code tiers, prenant la forme de sous-modules Git (`git submodule add`). Ceci aura pour effet de « laisser vivre » le code tiers par ailleurs tout en restant en lien avec notre projet, avec des *commits* précis pour chaque sous-module. Notre dépôt pourra alors être cloné avec l'option `--recurse-submodules`, ou cloné puis complété des sous-modules avec `git submodule init` puis `git submodule update`, pour obtenir un tout compilable.

2.1 SHA-256

Pour procéder à la signature ainsi qu'à la vérification d'une signature, nous devons générer un *hash* cryptographique SHA-256. Nous utiliserons une implémentation de Ilia Levin sous licence MIT **[11]**. Celle-ci est très facile à utiliser puisque nous avons à disposition une simple fonction, `sha256()`, prenant en argument un pointeur vers les données, la taille de ces dernières et un pointeur vers un tableau de `uint8_t` pour stocker le *hash* résultant.

En termes de code, ceci nous donne donc quelque chose comme :

```
char *message = NULL;
uint8_t hash[32] = { 0 };
// allouer la mémoire et peupler message[]
sha256(message, strlen(message), hash);
```

Dans le cas d'une signature, `message` est obtenu soit via un argument en ligne de commande géré par `getopt()`, via un simple `message = strdup(optarg)`, soit via une lecture de l'entrée standard. Nous utilisons ici l'astuce classique consistant à changer de comportement si l'argument de l'option est `-`. Si tel est le cas, nous utilisons `fgets()` pour lire une ligne sur l'entrée standard avec :

```
#define BUFSIZE         LINE_MAX

[...]

if (!(message = malloc(BUFSIZE)))
    err(EXIT_FAILURE, "insign malloc error");

if (!fgets(message, BUFSIZE, stdin))
    errx(EXIT_FAILURE, "fgets error!");

if (message[strlen(message) - 1] == '\n')
    message[strlen(message) - 1] = 0;
else
    errx(EXIT_FAILURE, "STDIN too big or no '\\n'");
```

Notez l'utilisation de `LINE_MAX` provenant de `limits.h`, qui donne la taille maximale d'une ligne de commande pour le système utilisé. C'est une valeur limite arbitrairement choisie ici (pour STDIN), mais cohérente avec l'utilisation d'une option en ligne de commande, sachant que le but n'est pas de signer des masses énormes de données. Il y a d'autres outils pour ce genre de choses, qui n'ont pas besoin d'une signature ridiculement petite (OpenSSL et/ou OpenPGP).

2.2 ECDSA avec micro-ecc

Pour la partie signature, nous utilisons *micro-ecc* de Ken MacKay **[12]**. Cette implémentation est directement écrite pour être portable sur des architectures 8, 32 et 64 bits, et inclut d'ailleurs un exemple pour Arduino (`ecc_test.ino`). La création de Ken est précisément faite pour le type d'utilisation que nous visons et supporte cinq paramètres de courbe elliptique standard : secp160r1, secp192r1, secp224r1, secp256r1 et secp256k1. Le code est sous licence *BSD 2-clause* et n'utilise aucune allocation dynamique, tout en étant développé dans un souci de concision (pour économiser RAM et flash sur microcontrôleur).

Micro-ecc répond à trois de nos besoins pour le projet : la génération d'une paire de clés, la signature et la vérification de signature. Le choix de la courbe est l'affaire d'une simple déclaration de variable qui sera utilisée pour les trois fonctions dédiées à nos trois usages :

```
const struct uECC_Curve_t *curve;

[...]
curve = uECC_secp160r1();
```

Ici, nous n'utilisons (pour l'instant) que secp160r1, mais nous gardons sous le coude le fait d'éventuellement faire évoluer le code pour supporter les quatre autres paramètres de courbe. En cela, nous n'avons pas besoin de nous inquiéter de la taille des tableaux destinés à accueillir la clé privée, la clé publique et la signature :

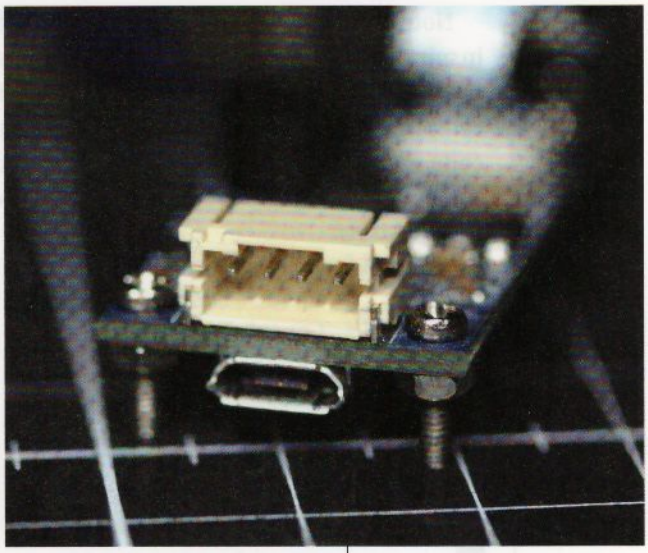
```
uint8_t private[21] = { 0 };
uint8_t public[40] = { 0 };
uint8_t signature[40] = { 0 };
```

Le moment venu, deux options s'offriront à nous, soit utiliser une taille maximum pour ces éléments (32, 64 et 64 `uint8_t`), ou allouer dynamiquement la mémoire avec `malloc()`. Mais pour l'heure, c'est un problème qui n'a pas besoin d'être posé et nous pouvons donc passer à la génération de clés :

```
int genkeys()
{
    if (uECC_make_key(public, private, curve) != 1) {
        errx(EXIT_FAILURE, "Error creating key pair");
    }
    return (0);
}
```

Nous utilisons ici une fonction dédiée pour éviter d'encombrer `main()`, mais aussi pour, plus tard, faire évoluer tout cela. La génération de clés étant dépendante d'une option gérée par `getopt()`, et l'enregistrement des données dans des fichiers est déléguée à une autre fonction dédiée, l'option est alors gérée ainsi :

```
if (optgenkey) {
    genkeys();
    if (savekeys(private, sizeof(private), public,
                 sizeof(public), privkeyfile,
                 pubkeyfile, optoverwrite) != 0) {
        warnx("Error saving key files!");
        ret = EXIT_FAILURE;
        goto getout;
    }
    goto getout;
}
```



Le module propose deux interfaces pour transmettre les données décodées par le lecteur : en haut l'UART configurable avec différents débits et formats de données et, en bas, un connecteur micro-USB pour la liaison USB-HID. Cette dernière permet d'utiliser le lecteur sans le moindre code ou pilote puisqu'il se présente directement comme un « clavier » USB.

Notez le `goto`, également utilisé en cas d'erreur un peu partout dans le code et directement inspiré de ce qui se fait généralement dans le développement de pilote noyau. Contrairement à ce qu'on peut penser, `goto` n'est pas une directive maudite ou tabou, il faut juste l'utiliser avec retenue et pour une bonne raison. Ici, celle-ci renvoie au label `getout:` qui forme la fin du code, avec toutes les libérations de mémoire allouée :

```
getout:
    if (insign)
        free(insign);
    if (optdelim)
        free(optdelim);
    if (message)
        free(message);
    if (privkeyfile)
        free(privkeyfile);
    if (pubkeyfile)
        free(pubkeyfile);
    if (basename)
        free(basename);
    return (ret);
}
```

Non présenté ici, `savekeys()` se charge simplement d'enregistrer par défaut les deux clés dans des fichiers binaires respectivement appelés par défaut `key.sec` et `key.pub`. Les deux constituant une paire intimement liée, une option est disponible pour l'utilisateur afin d'ajuster le nom, mais pas l'extension.

La signature est presque toute aussi simple que la génération de clés :

```
if (uECC_sign(private, hash, sizeof(hash), signature, curve) != 1) {
    warnx("Error signing message");
    ret = EXIT_FAILURE;
    goto getout;
}
```

Et la vérification tout autant :

```
if (uECC_verify(public, hash, sizeof(hash), insign, curve) != 1) {
[...]
```

Je vous fais grâce ici du reste de la portée de la condition, jonglant avec les options et décorant la sortie en conséquence (mode silencieux, couleurs, message verbeux, etc.).

Comme vous pouvez le voir, micro-ecc est très simple d'utilisation, en plus d'être bien testé (le projet à plus de 10 ans et est très bien noté sur GitHub), léger et incluant, de plus, des optimisations pour les plateformes ARM (instructions *Thumb-1* et *Thumb-2* utilisées en fonction des macros définies par le compilateur, ou forcées manuellement, voir `uECC.h`).

2.3 Base64

La dernière pièce du puzzle consiste à essayer de réduire au maximum la taille de la signature ajoutée au message. Dans l'état, et comme l'arrière-pensée reste d'utiliser tout cela avec des QRcodes, une représentation binaire est hors de question. Nous pourrions simplement nous contenter d'utiliser une notation hexadécimale, ce qui, avec une signature de 40 octets, nous ajouterait 80 caractères. Si le message est une URL ou une adresse mail, ne dépassant raisonnablement pas quelques dizaines de symboles, en ajouter 80 n'est pas forcément souhaitable. La question se pose alors de comment représenter 40 octets, avec des caractères lisibles (et encodables dans un QRcode) de façon la plus concise possible, tout en restant standard. Et la réponse est, bien entendu, Base64 **[13]**.

Une fois n'est pas coutume, nous nous tournons une nouvelle fois vers GitHub, pour utiliser le dépôt de Zhicheng Wei **[14]** proposant une implémentation d'un encodeur/décodeur Base64, placée dans le domaine public.

Contrairement au *hash* SHA-256 et à la signature, impliquant tous deux de travailler avec des tailles déterministes, le code de Zhicheng Wei est, bien entendu, censé fonctionner avec n'importe quelle taille de données en entrée et, comme il ne s'agit pas d'un *hash*, la sortie a donc également une taille tout aussi variable. Fort heureusement, le code met à disposition deux macros permettant de « prédire » la taille nécessaire pour stocker un résultat d'encodage ou de décodage :

```
#define BASE64_ENCODE_OUT_SIZE(s) \
((unsigned int)((((s) + 2) / 3) * 4 + 1))

#define BASE64_DECODE_OUT_SIZE(s) \
((unsigned int)(((s) / 4) * 3))
```

Nous pouvons donc nous en servir, soit pour valider le fait que nous avons suffisamment de place dans un tableau (encodage), soit pour allouer la mémoire nécessaire (décodage). Notez bien cependant que la valeur retournée par ces macros n'est **pas** la taille du résultat, mais bien l'espace nécessaire pour l'obtenir (une PR est présente dans le dépôt Git signalant un *buffer overflow* alors qu'il ne s'agit pas d'un vrai problème **[15]**, vous n'êtes pas censé allouer la taille théorique du résultat, mais bien la valeur retournée par ces macros). Ce sont les fonctions d'encodage et de décodage qui retournent la véritable taille des données :

```
unsigned int b64size =
  BASE64_ENCODE_OUT_SIZE(sizeof(signature));

[...]

if (base64_encode(signature, sizeof(signature),
        b64out) != b64size - 1) {
    warn("Base64 encoding error");
    ret = EXIT_FAILURE;
    goto getout;
}
```

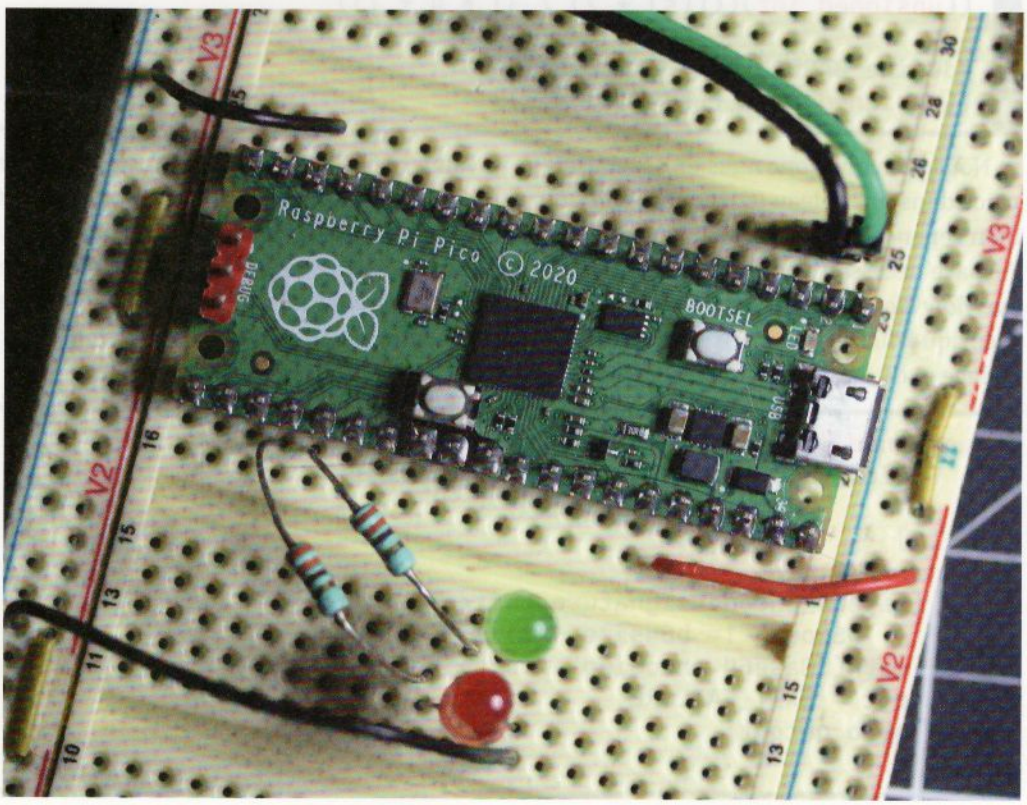
et :

```
if (!(insign = malloc(BASE64_DECODE_OUT_SIZE(strlen(insignenc)))))
    err(EXIT_FAILURE, "insign malloc error");
if ((insignlen = base64_decode(insignenc,
        strlen(insignenc), insign)) != 40) {
    warnx("Bad Base64 signature size!");
    ret = EXIT_FAILURE;
    goto getout;
}
```

Encore une fois, l'une et l'autre opérations sont relativement simples et ne posent aucun problème particulier.

2.4 Vérification de signature

Comme détaillée précédemment, la vérification d'une signature repose sur la génération d'un *hash* pour la partie « textuelle » d'un message reçu, suivie d'une vérification avec ce *hash* et la partie « signature » en argument de `uECC_verify()`, accompagnée de la clé publique et des paramètres de courbe à utiliser (secp160r1 pour le moment).



Le montage expérimental sur platine propose une interface utilisateur relativement simpliste : une LED verte pour dire que la signature est valide et une LED rouge pour dire que non.

La question qui se pose alors est « comment distinguer le message, qui peut être plus ou moins n'importe quoi, de la signature ? », sachant que, si nous voulons garder la perspective d'utiliser d'autres courbes par la suite, il n'est pas question de simplement compter les caractères en partant de la fin de la chaîne en entrée (d'autant que l'idée est de supporter à la fois une signature en Base64 et en hexadécimal, parce que... « pourquoi pas ? »). La solution est simple, il faut utiliser un séparateur, laissé au choix de l'utilisateur via une option dédiée. Il ne s'agit pas d'un simple caractère (qui pourrait être présent dans le message), mais d'une chaîne. Une autre option aurait été de fixer une taille de message et d'utiliser un bourrage (*padding*), mais cela va à l'encontre du fait de rester concis, tout en imposant des contraires pénibles. Le fait de laisser le choix à l'utilisateur (ou à vous) concernant la chaîne à utiliser est ce que j'appellerais un transfert de problème : c'est l'utilisateur qui compose le message et définit le séparateur, donc il est en contrôle, donc c'est son problème, pas le mien...

Ceci nous amène donc à résoudre un problème assez classique : diviser une chaîne en deux sous-chaînes en fonction d'une chaîne séparant les deux. Pas la peine de demander à ChatGPT ou Stack Overflow, c'est un problème classique :

```
char *optdelim = NULL;
char *inmessage;
char *insignenc;

// trouver le séparateur
inmessage = strstr(message, optdelim);
if (!inmessage) {
    warnx("Delimiter not found in message");
    ret = EXIT_FAILURE;
    goto getout;
}

// diviser la chaîne
*inmessage = '\0';
insignenc = inmessage + strlen(optdelim);
```

`optdelim` est notre chaîne de séparation, provenant de la gestion `getopt()` et nous obtenons au final `inmessage` avec la partie « textuelle » et `insignenc` pour la signature encodée (Base64 ou autre). Notez que `message` est modifié au passage, puisque `strstr()` retourne un pointeur sur la première occurrence de la chaîne recherchée (le séparateur), et que nous plaçons un `\0` à cet endroit. Au passage, il est amusant de voir que la page de manuel `strstr(3)` GNU/Linux parle de *needle* et de *haystack* (littéralement « aiguille » et « meule de foin »), alors que celle de FreeBSD est plus « sérieuse » et fait référence respectivement à *little* et *big*, tout comme celle d'OpenBSD.

2.5 Considérations complémentaires

Si l'on veut utiliser ce type de mini-signatures pour des URL, un problème se pose en cas d'encodage Base64. En effet, cet algorithme utilise 64 symboles : les lettres « a » à « z », « A » à « Z », les chiffres « 0 » à « 9 » et les symboles « + » et « / », plus un bourrage avec des « = ». Or,

Voici un exemple de QRcode encodant le message « Hello World » accompagné d'une signature électronique (à gauche), ainsi que (à droite) un simple QRcode encodant la clé publique en hexadécimal permettant la vérification de la signature, si vous voulez jouer ou tester le code...

pour une URL (URI, en réalité), les seuls caractères n'ayant pas une « utilisation réservée » (*unreserved* donc) sont les caractères alphabétiques (minuscule et majuscule), les chiffres et « - », « _ », « . » et « ~ ». Nous avons donc un conflit avec 3 des 64 symboles utilisés par Base64.

On pourrait parfaitement ignorer le problème et se dire que l'utilisateur n'a qu'à supprimer la signature (ou utiliser l'hexadécimal) pour utiliser l'URL, mais imaginez que je vous dise qu'on utilise, comme séparateur, la chaîne `"?/sign="`. Instantanément, le message **et sa signature** deviennent parfaitement utilisables dans des conditions qu'on pourrait considérer comme normales : flasher un QRcode et suivre l'URL obtenue, le serveur HTTPS se débrouillant ou ignorant simplement le paramètre dans la requête GET.

Pour contourner le problème, il nous suffit alors d'inventer une option dédiée pour permuter les symboles, dans un sens ou dans l'autre, et de créer deux petites fonctions à cet effet (oui, j'ai un peu abusé de l'opérateur ternaire, pour le coup) :

```
void urlize(char *b64str, size_t len)
{
    int i;
    for (i = 0; i < len; i++) {
        b64str[i] = b64str[i] == '/' ? '_' :
            b64str[i] == '=' ? '-' :
            b64str[i] == '+' ? '~' : b64str[i];
    }
}

void unurlize(char *b64str, size_t len)
{
    int i;
    for (i = 0; i < len; i++) {
        b64str[i] = b64str[i] == '_' ? '/' :
            b64str[i] == '-' ? '=' :
            b64str[i] == '~' ? '+' : b64str[i];
    }
}
```

Un autre point intéressant, mais cette fois en dehors du code lui-même, concerne la compilation du programme qui est, bien entendu, laissé à la discrétion d'un `Makefile`. Nous avons nos sources dans un dépôt Git et trois sous-dépôts pour le code tiers prenant la forme de sous-répertoires : `micro-ecc/`, `base64/`, et `SHA256/`. Nous pourrions explicitement désigner les sources à utiliser, comme `micro-ecc/uECC.c` par exemple, mais dans ce cas, le `uECC.o` résultant de la compilation trouverait également place dans `micro-ecc/`, ayant pour

conséquence de faire « râler » Git puisque des fichiers non suivis apparaissent de-ci de-là. Il est, bien entendu, possible d'ignorer ces fichiers objet, `**/` est là pour ça, mais personnellement, je n'aime pas du tout modifier le contenu de répertoires appartenant à des sous-modules Git, et préfère mes objets au même endroit que mon binaire.

Pour régler facilement le problème, il nous suffit d'utiliser les `VPATH` de `make`, en utilisant quelque chose comme :

```
TARGET  := stringsign
WARN    := -Wall
DEBUG   := -g
CFLAGS  := $(DEBUG) ${WARN} -O3 -Imicro-ecc -Ibase64 -ISHA256

C_SRCS    = main.c uECC.c base64.c sha256.c
OBJ_FILES = $(C_SRCS:.c=.o)
VPATH=micro-ecc/:base64/:SHA256/

all: ${TARGET}

%.o: %.c
        ${CC} ${WARN} -c ${CFLAGS}  $< -o $@
[...]
```

`VPATH` permet de préciser une liste de répertoires pour trouver les sources, et les objets résultants sont automatiquement placés dans le répertoire courant. En bonus, notez que ceci n'est absolument pas une spécificité de GNU Make et est parfaitement compatible avec les BSD Make de FreeBSD et OpenBSD (testé), contrairement à ces choses comme `C_SRCS = $(wildcard *.c)`.

3. UN MOT SUR L'ASPECT QRCODE ET L'IMPLÉMENTATION PICO/RP2040

Pour la version ligne de commande, l'utilisation d'un lecteur de QRcode, qu'il s'agisse d'un simple module ou d'un « pistolet »/« douchette », n'implique rien de particulier. Ceci se présente sous la forme d'un périphérique de saisie USB-HID émulant une saisie clavier. La seule chose à prendre en considération est la configuration du matériel lui-même qui, chose assez amusante, se fait via l'utilisation de QRcodes listés dans le manuel **[16]**. Ceci permet de choisir différents paramètres comme l'organisation clavier à utiliser, le type d'interface à activer/désactiver et les paramètres de communication (vitesse, etc., et éventuellement les types de code-barres 1D ou 2D à prendre en charge). Est-ce que cette méthode de (re) configuration fonctionne également avec des lecteurs utilisés dans certaines installations publiques mal conçues, dès lors qu'on obtient la documentation adéquate ? À votre avis ;) ?

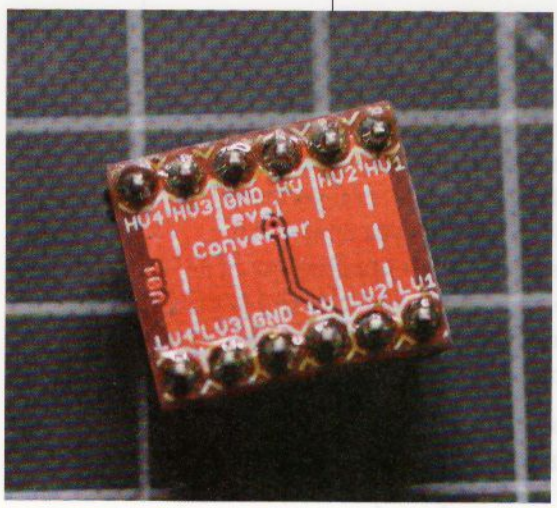
Sur l'aspect microcontrôleur et donc l'utilisation d'un module interfacé en liaison série, l'approche la plus simple est de faire correspondre ce que permet de faire nativement la plateforme avec la configuration du lecteur. Dans le cas d'une carte Raspberry Pi Pico, l'UART0

peut être utilisé très facilement comme l'entrée standard, exactement comme on le fait avec la version en ligne de commande, en utilisant `fgets()`. Il suffit donc de configurer le lecteur en 115200 8N1 et l'affaire est dans le sac.

Produire un QRcode incorporant notre mini-signature électronique est un jeu d'enfant en reposant sur un outil comme `qrencode`, capable de générer des fichiers graphiques comme PNG à partir d'une chaîne de caractères provenant de l'entrée standard. Avec des *pipes*, on peut donc très rapidement signer un message et générer une image, avec `echo "Coucou Monde" | ./stringsign -s -m - -t "###" | qrencode -o qr_code.png` par exemple (`-s` pour signer, `-m -` pour prendre l'entrée standard et `-t` pour préciser `###` comme séparateur). On peut également afficher le QRcode directement dans le terminal en utilisant l'option `-t utf8` et sans préciser de fichier en sortie (`-o`).

C'est là qu'un point excessivement important entre en jeu. La sortie du programme ajoute un saut de ligne après la signature, c'est le comportement standard pour afficher quelque chose dans un terminal, avec `\n` dans un `printf()`. Ce caractère, LF, est présent dans les données passées à `qrencode` et se retrouve encodé dans le QRcode. Et c'est une excellente chose, car, inversement, notre fonction `fgets()` restera bloquée si ce saut de ligne est absent, ou pire, remplacé par un retour chariot (`\r` ou CR). Pour un code fonctionnant sur Pico et sans utiliser de lecteur physique, on pourra très facilement tester le fonctionnement… à condition d'envoyer un LF (`\n`) et non un CR (`\r`). Avec Minicom, par exemple, on s'abstiendra donc d'utiliser la touche « Entrée » pour valider une ligne et on utilisera Ctrl+j. Notez que, pour l'heure, mon implémentation RP2040 attend une entrée conforme pour fonctionner et que l'absence d'un `\n` en fin de message est bloquante. Pour régler ce problème, il faudrait ajouter un *timeout* pour réinitialiser l'entrée et oublier les caractères déjà reçus.

Un des points regrettables concernant le lecteur de code-barres 1D et 2D de WareShare est son fonctionnement en 5 V. Ceci ne posera pas de problème à une carte Arduino, mais pour une Pico (ou un ESP32) il faudra impérativement adapter les tensions des niveaux logiques. Ce type de modules composé de MOSFET fera parfaitement le travail, mais bien d'autres solutions peuvent être utilisées, surtout avec une simple ligne de données (TX 0/+5 V-> RX 0/+3,3 V).



Vous l'aurez compris, une déclinaison microcontrôleur n'a réellement d'intérêt que pour la vérification d'un message signé et on peut tout à fait imaginer, en reprenant l'exemple de l'URL, flasher le QRcode avec un montage indiquant une validation ou une erreur via des LED (vert et rouge), pour ensuite flasher avec son smartphone en ayant l'assurance qu'il n'y a pas de falsification. Vous savez, comme ces QRcodes autocollants placés par des personnages peu scrupuleux sur les panneaux d'information des parkings payants, les terrasses de café et restaurant, etc., pour faire du *phishing* ou pousser l'utilisateur peu méfiant à installer une fausse application qui est en réalité un *malware*.

Ceci dit, en poussant le concept, la vérification de signature pourrait avoir lieu dans l'applicatif Android/iOS directement, ou le système lui-même... Mais je m'égare (et je ne touche plus au dev mobile depuis la mort de Tizen).

Côté organisation des sources, intégrer un code spécifique pour le SDK Pico est relativement facile à faire. Comme on a explicitement spécifié la liste des sources C dans notre `Makefile`, il est parfaitement possible de prévoir un `picomain.c`, ignoré par le *build* standard, mais pris en compte uniquement dans un `CMakeLists.txt`, en compagnie des sources des trois éléments tiers :

```
set(SOURCE_FILES
  picomain.c
  micro-ecc/uECC.c
  base64/base64.c
  SHA256/sha256.c
)
```

On pourra alors très classiquement construire pour le RP2040 en créant un répertoire `build/`, s'y plaçant, et invoquant `cmake ../`, puis `make`. Nous obtiendrons alors les binaires `stringsign.elf`, `stringsign.bin` et `stringsign.uf2` à flasher dans la carte selon votre méthode préférée (`picotool`, donc).

Mais pour que ce code, très majoritairement similaire à la version PC/SBC, fonctionne, il existe une petite subtilité. En effet, si vous appelez `uECC_verify()` après avoir divisé la chaîne et décodé la signature en Base64, celle-ci retournera `0`, indiquant un échec de l'opération. Et il va de même pour `uECC_sign()` et même `uECC_make_key()`. Pourquoi ? Parce que le code de Ken MacKay a besoin d'obtenir un certain nombre de valeurs aléatoires utilisées par l'algorithme ECSDA. Sur PC/SBC, sous GNU/Linux, ceci est automatique en utilisant `/dev/urandom` (voir `platform-specific.inc`) qui fournit ce type de données automatiquement. Mais sur microcontrôleur, ce genre de choses n'existe pas et nous devons alors implémenter une fonction propre à la plateforme. Pour le RP2040 et le SDK Pico, nous pouvons utiliser la fonction `get_rand_32()` et créer :

```
static int eccRNG(uint8_t *dest, unsigned size) {
    while (size) {
        uint8_t val = 0;
        val = (uint8_t)get_rand_32();
        *dest = val;
        dest++;
        size--;
    }
    return (1);
}
```

`get_rand_32()` retourne un `uint32_t` que nous « castons » en `uint32_t` pour remplir un tableau dont le pointeur est passé en argument. Dès le début de `main()`, on spécifiera la fonction en question pour satisfaire micro-ecc avec :

```
uECC_set_rng(&eccRNG);
```

Et ainsi, tout fonctionnera alors parfaitement, comme sur PC/SBC. On n'oubliera pas, toutefois, d'ajouter la bibliothèque `pico_rand` dans son `CMakeLists.txt`, sous peine de se faire insulter par le compilateur :

```
target_link_libraries(${CMAKE_PROJECT_NAME}
    pico_stdlib
    pico_rand
)
```

En utilisant cette approche et en combinant à la fois le développement pour PC/SBC et pour microcontrôleur, on arrivera à maintenir facilement le projet. On pourra également pousser davantage le principe en sortant de `main.c` tout ce qui n'est pas spécifique à une plateforme et adopter la même approche que pour l'enregistrement des clés avec des fonctions dédiées à chaque opération. Les `*main.c` se résumeront alors au strict minimum, par plateforme, et toute évolution dans le code se fera automatiquement pour toutes les cibles.

Les QRcodes ne constituent pas le seul média utilisable pour ce type de choses. La technologie RFID/NFC pourra également stocker vos messages signés, même s'il s'agit d'un modèle dont la sécurité est particulièrement faible, comme les MIFARE Classic 1K/4K maintenant totalement obsolètes (et pourtant toujours massivement utilisés).

CONCLUSION ET ÉVOLUTIONS

Comme d'habitude et puisqu'il ne me semble apparemment plus possible d'avoir une idée d'article sans implémenter un outil, vous trouverez une version très « 0.1 » de ce qui est décrit ici dans mon GitLab **[10]** sous licence *BSD 2-clause* et il est possible que les choses évoluent encore dans l'avenir (utiliser

d'autres courbes ou d'autres fonctions de hachage, en particulier). Le fait d'utiliser de toutes petites signatures électroniques aussi facilement ouvre la voie à plein de réalisations et on imagine facilement intégrer ce genre de choses pour un système de messagerie, une authentification (avec un mécanisme de challenge/réponse) ou encore combiner cela avec d'autres supports et médias comme NFC ou des écrans *e-paper*. On pourrait également imaginer incorporer cela dans un système de communication pour l'IoT, la domotique ou encore des messages radio (Meshtastic, modules 33 MHz **[17]**, etc.), avec des capteurs qui retournent certes des valeurs, mais signées et garanties comme authentiques ! Quid des informations de connexion Wi-Fi encodées en QRcode et présentées sur un écran *e-paper* à flasher ? Ou encore de code promo signé ? Les idées ne manquent pas... **DB**

RÉFÉRENCES

[1] https://www.youtube.com/watch?v=l89qpmb2CAQ

[2] https://shattered.io/

[3] https://www.rfc-editor.org/rfc/rfc4492#appendix-A

[4] https://www.legifrance.gouv.fr/jorf/id/JORFTEXT000024668816

[5] https://www.keylength.com/fr/

[6] https://fr.aliexpress.com/item/4000873144062.html

[7] https://fr.aliexpress.com/item/1005003265115484.html

[8] https://fr.aliexpress.com/item/1005005603412979.html

[9] https://www.waveshare.com/barcode-scanner-module.htm

[10] https://gitlab.com/0xDRRB/stringsign

[11] https://github.com/ilvn/SHA256

[12] https://github.com/kmackay/micro-ecc

[13] https://fr.wikipedia.org/wiki/Base64

[14] https://github.com/zhicheng/base64

[15] https://github.com/zhicheng/base64/pull/13

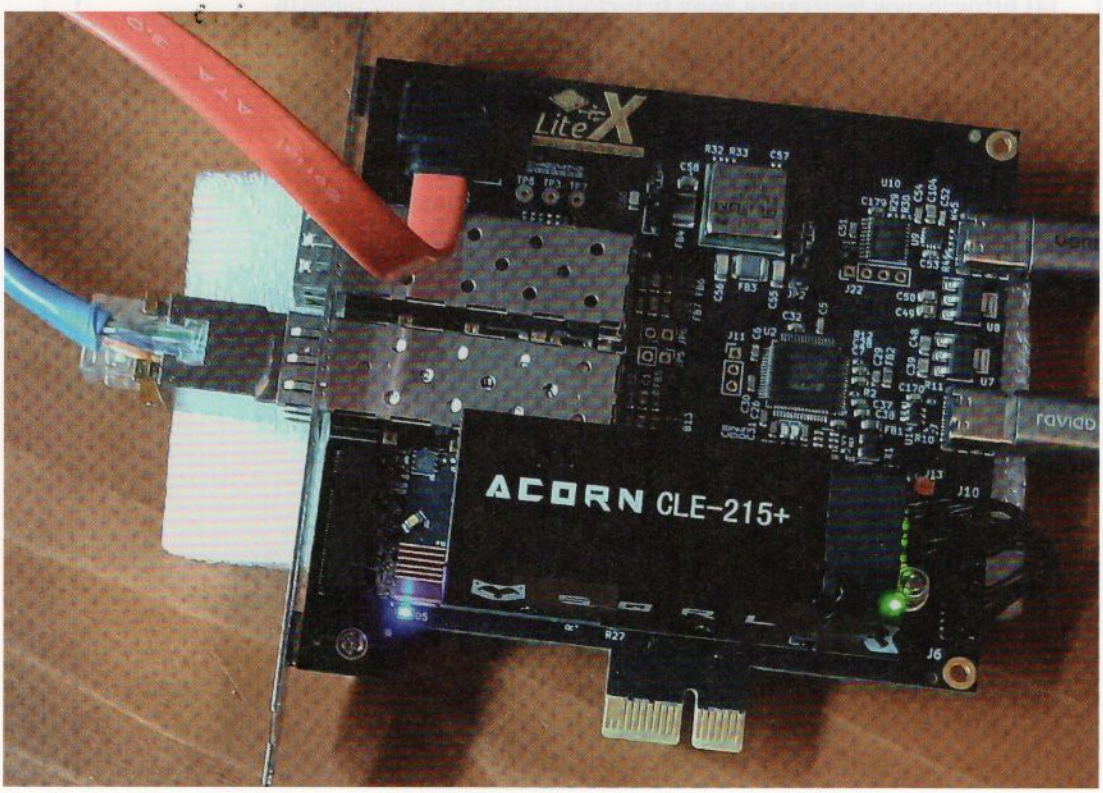
[16] https://files.waveshare.com/upload/d/dd/Barcode_Scanner_Module_Setting_Manual_EN.pdf

[17] https://www.framboise314.fr/transmission-de-donnees-serie-en-433mhz-avec-les-modules-ebyte-e49-400t20d/

LITEX : LINUX SUR UN SOC RISC-V EN FPGA

Denis Bodor

Dans un précédent article [1], nous avons découvert LiteX, une solution permettant de créer des périphériques et SoC en toute simplicité, sans avoir à apprendre à utiliser des langages de description de matériel comme Verilog et VHDL. Aujourd'hui, poussons plus loin l'exploration de ce framework doublé d'une infrastructure de construction et basé sur Python en faisant fonctionner un système d'exploitation GNU/Linux sur une plateforme « full FPGA » RISC-V.



– LiteX : Linux sur un SoC RISC-V en FPGA –

Initialement développé par Enjoy-Digital comme base pour des projets clients et disponible sous licence *open source* très permissive de type BSD, LiteX **[2]** est un *framework* et un ensemble de composants permettant de créer tout type de matériel, avec un accent tout particulier mis sur l'implémentation de systèmes complets à base RISC-V. Principalement écrit en Python, avec la boîte à outils Migen pour la description de circuits logiques, LiteX s'accompagne d'une collection de périphériques (interface DRAM, Ethernet, contrôleur SATA, SPI, i2c, support de carte SD/TF, etc.) qui viendront se greffer à un processeur *softcore* supporté (VexRiscv, Rocket, LM32, Mor1kx, PicoRV32, BlackParrot, etc.) pour créer un SoC sur mesure qu'on pourra ensuite soit simuler (avec Verilator), soit configurer sur l'une des quelque 180 cartes cibles prisent en charge.

Notre objectif ici sera non seulement de créer un SoC RISC-V complet, mais également d'y faire fonctionner un système GNU/Linux construit avec Buildroot, dans un premier temps. Il est d'ailleurs amusant de relever que le slogan de LiteX est « *Build your hardware, easily!* » et celui de Buildroot est « *Making Embedded Linux Easy* ». On ne peut qu'en déduire que ce qui va suivre ne pourra être que simple, voire doublement simple (ou pas) !

1. INSTALLATION ET DÉPENDANCES

Je vais ici éviter au maximum de paraphraser l'article précédent en allant à l'essentiel et sans perdre personne. Pour utiliser LiteX, nous devons disposer d'une installation GNU/Linux complète et à jour, typiquement Debian ou dérivée. Ce à quoi s'ajoutent les paquets suivants :

```
$ sudo apt-get install ninja-build libevent-dev \
libjson-c-dev verilator meson  gcc-riscv64-linux-gnu \
binutils-riscv64-linux-gnu openfpgaloader openocd \
linux-libc-dev git zlib1g-dev python3-venv
```

Notez la présence d'un compilateur croisé à destination de RISC-V ainsi que le support Python pour la gestion d'environnements virtuels, mais aussi Git. L'installation à proprement parler de LiteX se fera dans un répertoire qui fait également office d'environnement virtuel Python et sans aucun besoin de passer en `root`. Pour créer l'environnement, nous activons l'environnement, nous plaçons dans le répertoire cible, téléchargeons le script d'installation et l'exécutons, avec :

```
$ python3 -m venv ~/LITEX
$ source ~/LITEX/bin/activate
$ cd ~/LITEX
$ pip3 install meson
$ wget https://raw.githubusercontent.com/\
enjoy-digital/litex/master/litex_setup.py
$ chmod +x litex_setup.py
$ ./litex_setup.py --init --install
```

S'en suivra une succession de clonages de dépôts Git et d'installations de modules Python. Vous pourrez quitter l'environnement à tout moment avec `deactivate` et y revenir avec un nouveau `source ~/LITEX/bin/activate`. Notez que le répertoire `~/LITEX` est ici choisi de

façon totalement arbitraire, vous pouvez placer ces éléments n'importe où. Toutes les commandes qui vont suivre sont à utiliser dans l'environnement et, par défaut, à la racine du répertoire d'installation de LiteX.

Pour terminer l'installation et prendre les devants, vous pouvez immédiatement cloner le dépôt de `linux-on-litex-vexriscv`, ainsi que celui de la version *upstream* de Buildroot avec :

```
$ git clone \
https://github.com/litex-hub/linux-on-litex-vexriscv.git

$ git clone \
http://github.com/buildroot/buildroot
```

Le *framework* LiteX ne crée pas directement le *bitstream* (configuration binaire pour le FPGA), mais repose sur la présence et l'utilisation d'outils tiers. Ceci signifie que vous devez avoir, dans votre `$PATH`, les exécutables d'une suite de synthèse FPGA adaptée à votre cible. Pour AMD/Xilinx, c'est Vivado, pour Intel/Altera, c'est Quartus Prime, pour Sipeed Tang, c'est l'IDE Gowin, etc. Il s'agit le plus souvent d'outils propriétaires, gratuitement téléchargeables et fonctionnant sous GNU/Linux, qu'il faudra installer en suivant les instructions du fournisseur de la solution. Il existe également des suites entièrement *open source*, en particulier pour les FPGA Lattice **[3]**, mais ce point sera ici laissé à la discrétion du lecteur puisque nous l'avons déjà traité dans le magazine **[4] [5] [6]**. Le travail de LiteX est de produire un code Verilog, ainsi que des fichiers de configuration, qui seront ensuite utilisés avec un simulateur (typiquement Verilator) ou la suite logicielle adaptée au FPGA cible.

Enfin, nous aurons également besoin d'un élément *open source*, mais pour l'instant non disponible nativement pour notre distribution Debian. Le *softcore* VexRiscv est écrit en SpinalHDL, un langage de description matériel basé sur Scala, ou plus exactement, prenant la forme de plusieurs bibliothèques Scala. En cas de nécessité de reconfigurer le *softcore*, SpinalHDL est utilisé pour générer des descriptions de certains éléments et, de ce fait, l'outil de construction dédié est nécessaire : Scala Build Tool ou SBT.

Pour installer SBT sur notre système, un nouveau dépôt de paquets devra être intégré dans la configuration APT. Tout ceci prend la forme des commandes suivantes :

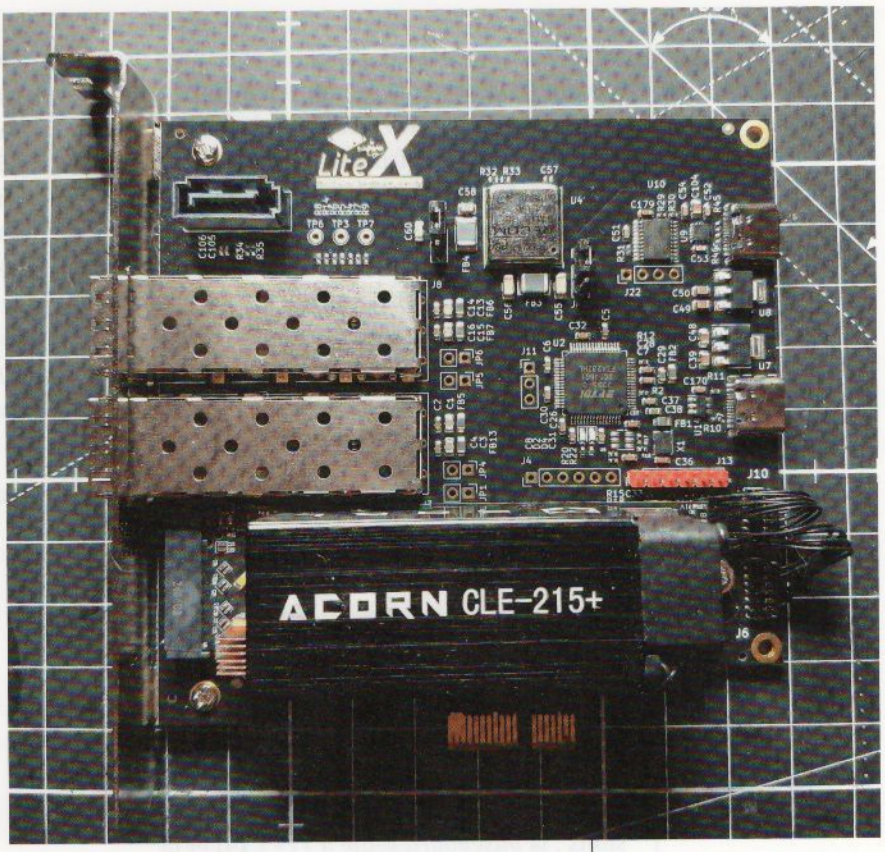
```
$ sudo apt-get install apt-transport-https curl gnupg
$ echo "deb https://repo.scala-sbt.org/scalasbt/debian all main" \
  | sudo tee /etc/apt/sources.list.d/sbt.list
$ curl -sL "https://keyserver.ubuntu.com/pks/lookup?"\
"op=get&search=0x2EE0EA64E40A89B84B2DF73499E82A75642AC823" \
  | sudo -H gpg --no-default-keyring --keyring \
  gnupg-ring:/etc/apt/trusted.gpg.d/scalasbt-release.gpg --import
$ sudo apt-get update
$ sudo apt-get install sbt
```

Dans l'ordre, nous installons les dépendances nécessaires, nous ajoutons le dépôt APT dans la configuration, nous ajoutons la clé GPG permettant d'authentifier et de vérifier l'intégrité des paquets, nous mettons à jour et nous installons SBT.

Tout est maintenant prêt pour la suite, mais avant cela, parlons un peu de matériel...

2. LITEX ACORN BASEBOARD MINI & CLE215+

Bien qu'un certain nombre de cartes et *devkits* soient utilisables pour ce que nous faisons ici, en trouver un parfaitement adapté, qui ne vous obligera pas à casser votre tirelire, n'est pas simple. Faire fonctionner un système complet comme GNU/Linux nécessite des ressources et un certain volume de RAM en particulier. Fort heureusement, les développeurs de LiteX ont trouvé une solution relativement économique en recyclant ce qui est, à l'origine, un périphérique permettant d'accélérer matériellement le minage de cryptomonnaies. Aujourd'hui devenu obsolète pour ce type d'usage, ce matériel est construit autour d'un FPGA AMD/Xilinx Artix7 XC7A200T, relativement conséquent et étoffé de 1 Gio de RAM et 32 Mio de flash. Mieux encore, il prend la forme d'une petite carte PCI Express au format M.2. Le produit n'étant pas un *devkit*, le nombre de broches utilisables pour des entrées/sorties génériques est très limité, mais le connecteur M.2 fournit des fonctionnalités inattendues et très séduisantes. Notez que dans ce qui suit, nous ne ferons pas usage de l'interface PCIe et utiliserons la carte comme un SBC classique. Peut-être reviendrai-je sur le sujet dans un prochain numéro, nous verrons.



Un des matériels permettant de facilement tester LiteX et disposant des ressources suffisantes pour un système Linux est cette carte d'accueil PCIe équipée du module SQRL Acorn CLE-215+, initialement prévue pour le minage de cryptomonnaies (mais étant en réalité un clone du devkit Nitefury-II).

Ce périphérique de minage est un module SQRL Acorn CLE215+ qui, de fait, s'avère être un clone du projet NiteFury/LiteFury [7] presque en tout point similaire, si ce n'est par l'agréable quantité de RAM disponible (le double par rapport au NiteFury). Il semblerait que le fabricant SQRL ait tout simplement basé son design sur le projet de RHSResearchLLC pour créer son produit et que, par la magie du *hacking*, couplé aux efforts d'Enjoy-Digital, la boucle se retrouve parfaitement bouclée.

L'Acorn CLE215+ peut se trouver sur eBay et il existe également quelques annonces proposant LiteFury et NiteFury II sur AliExpress (ainsi que des clones très similaires, cependant sans solution de refroidissement **[8]**), mais il y a mieux. Motivé par le souhait de proposer un *devkit* permettant de découvrir facilement et confortablement LiteX, Enjoy-Digital a décidé de concevoir, produire et vendre une carte d'accueil (*baseboard*) pour le SQRL Acorn CLE215+, fournissant :

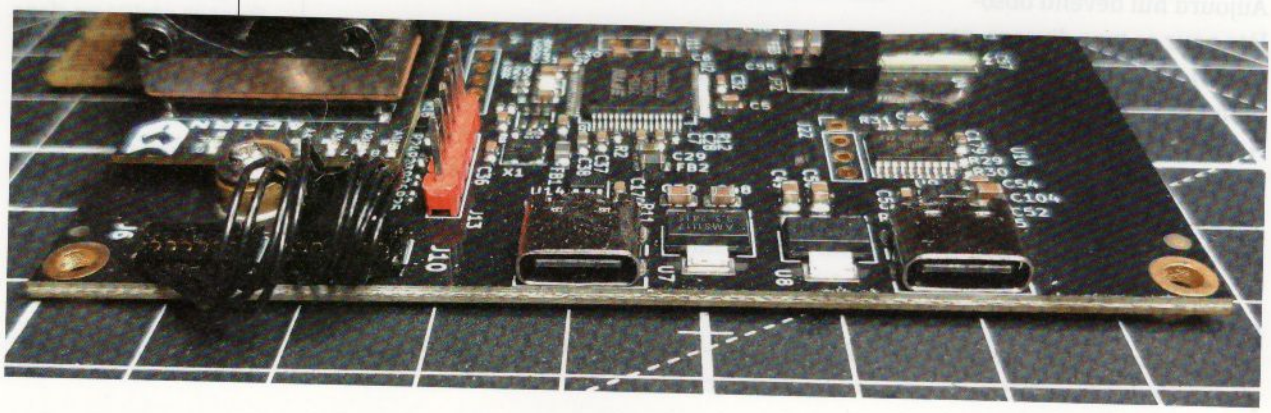
- un connecteur M.2 *key* M ;
- une interface PCIe Gen2 x1 ;
- deux emplacements SFP pour des interfaces réseau ;
- un connecteur SATA pour la connexion d'un disque ;
- une interface JTAG USB-C basée sur un FT4232 ;
- un port USB-C PD (*Power Delivery*) pour l'alimentation (géré par un MCU STM32).

L'ensemble est disponible en deux versions : la *baseboard* seule à 119 € **[9]** et la *baseboard* avec son CLE215+ **[10]**. Notez que la disponibilité de ces cartes est variable, étant donné qu'Enjoy-Digital produit le matériel par lot et que l'objectif n'est pas de faire des bénéfices en commercialisant ce matériel (c'est une société de service, un bureau d'étude), mais de proposer une sorte de plateforme de référence pour le développement autour de LiteX.

Pour utiliser pleinement ce matériel, vous devrez ajouter une alimentation USB PD de qualité (comme mon chargeur UGREEN Nexode 65 W **[11]** ainsi qu'un module SFP cuivre 10/100/1000Base-T RJ45 si vous voulez profiter de la connectivité réseau. Ceci se trouve très facilement et j'ai personnellement opté sur un modèle relativement générique à 20 € sur Amazon **[12]**, sans avoir le moindre problème de compatibilité.

Côté spécifications, le CLE215+ lui-même se compose de :

- un FPGA AMD/Xilinx Artix7 XC7A200T (215360 LC) ;
- une RAM DDR3 16 bits de 1 Gio ;
- 32 Mio de flash QSPI ;
- une interface PCIe Gen2 x4 avec 4 *transceivers* GTP (6,6 Gb/s) ;
- 4 LED ;
- un connecteur Molex Pico-EZmate 6 broches pour le JTAG ;
- un autre Pico-EZmate pour fournir 4 GPIO + masse + Vcc (utilisé pour la liaison/console série avec LiteX) ;
- un connecteur LVDS de 4 paires quasi inaccessible (sous le radiateur).



La carte d'accueil dispose de deux ports USB-C, celui de gauche pour le JTAG (via FT4232) et la communication série, et celui de droite pour l'alimentation USB-PD.

RISC-V : POURQUOI « FIVE » ?

L'ISA RISC-V, à prononcer « *risk five* » (pas « risque v » comme « À mort Louis croix V bâton ! », et pas « risque cinq » non plus), provient tout droit d'UC Berkeley (encore et toujours eux) et plus précisément des travaux de recherche de David Patterson. RISC-V est la cinquième génération de design après RISC-I et RISC-II en 1981, puis RISC-III en 1984, basé sur l'architecture SOAR (*Smalltalk On A RISC*) et enfin RISC-IV en 1988 (architecture SPUR, pour *Symbolic Processing Using RISCs*). Les termes RISC-III et RISC-IV sont apparus après RISC-V (2010), pour remplacer les désignations SOAR et SPUR.

Notez également que ces travaux de recherche, dans leur ensemble, ont également conduit à la création d'autres architectures et processeurs RISC, comme Intel i960, SPARC, DEC Alpha, MIPS, PA-RISC, Hitachi SuperH-4, IBM PowerPC et même ARM (voir **[14]**).

Donc, oui, il y a bien eu un « *risk four* », mais il ne s'appelait tout simplement pas comme ça à sa naissance…

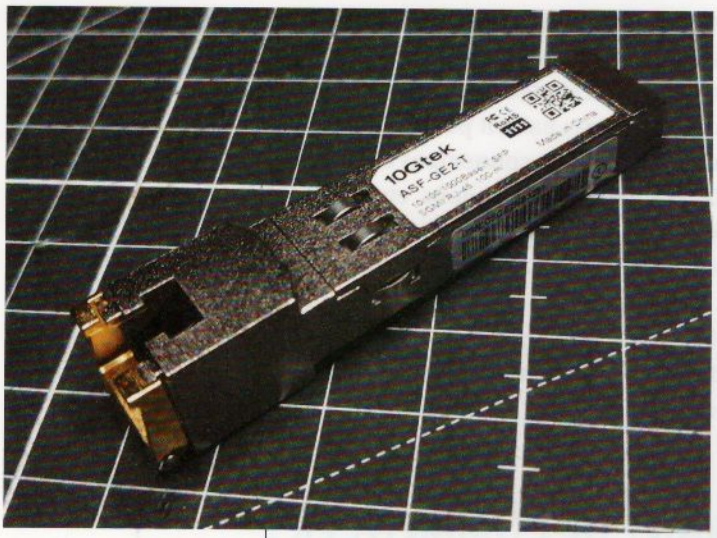
Comme vous pouvez le voir, le duo CLE215+ plus *baseboard* est riche et se prêtera parfaitement à la création d'un SBC capable de faire fonctionner un Linux, mais n'est pas parfait pour tous les usages. Le manque d'I/O limite les connexions de capteurs, supports SD, afficheurs LCD, boutons, etc. Mais d'autre part, l'interface PCIe ouvre des possibilités très intéressantes. On ne peut pas tout avoir, surtout pour ce prix-là. Notez que le schéma de la carte, ainsi que ses déclinaisons sont disponibles dans un dépôt GitHub dédié **[13]**. C'est toujours intéressant de l'avoir sous la main, en cas de doute.

3. AVANT DE COMMENCER : COMPATIBILITÉ !

J'ai passé énormément de temps sur ces expérimentations et il y a eu bien des problèmes, des moments de solitude et aussi quelques noms d'oiseaux lâchés de-ci de-là. Appréhender un tel projet est généralement une petite aventure où on fait, on teste, on rencontre des problèmes et on trouve des solutions, en boucle, étalée sur plusieurs jours, avant de trouver **LE** problème à l'origine de tous les maux. Et ce problème tenait finalement en un mot : « compatibilité ».

Pour vous éviter les mêmes tourments que les miens ces derniers jours, je vous l'annonce clairement : si ça ne marche pas, essayez avec un autre matériel. Dans ce que nous allons voir, et puisque la Baseboard Mini dispose d'un connecteur SATA, nous allons, dans une certaine mesure, faire usage de cette interface après quelques essais préliminaires. Et le choix du disque utilisé semble très important, même si l'origine des problèmes n'est pas entièrement claire pour moi à ce jour (mais ils sont parfaitement reproductibles).

De longues heures ont été passées à tenter de *booter* un système chargé depuis le support SATA et/ou d'accéder à ce dernier depuis le système exécuté. Les comportements étranges se sont succédé, allant de l'impossibilité d'initialiser l'interface avec le disque depuis le « BIOS » à la corruption pure et simple des données en cas d'écriture, en passant par le chargement



Pour connecter la baseboard, et donc le système, au réseau, il vous faudra ajouter un module SFP cuivre comme celui-ci : littéralement le moins cher d'Amazon à ~20 €, qui a fonctionné à merveille.

réussi, mais sans possibilité d'activer le support Linux, ou encore des *segfaults* à répétition. Ces mésaventures ont débuté par l'utilisation d'un HDD mécanique Hitachi HTS545016B9A300 de 160 Go récupéré dans un vieux Samsung NB30, et se sont poursuivies avec un SSD Intel SSDSC2CT120A3 de 120 Go.

C'est par le plus grand des hasards que ma chance a finalement tourné en commandant un SDD, le moins cher d'Amazon, PNY SSD7CS900-250-RB **[15]** de 250 Go, alors que je ne pensais n'avoir aucun SSD sous la main. Imaginez ma surprise, quand subitement, pensant avoir commandé un matériel pour rien (j'ai retrouvé le SSD Intel 2 h après le passage de la commande Amazon), il s'est avéré que l'ensemble des problèmes ont disparu dès le premier essai avec ce disque !

Je vous ferai donc grâce des tentatives de résolution des problèmes, y compris le bidouillage ignoble du *device tree* et du pilote `litesata.c`, pour baser mes explications uniquement et directement sur les manipulations avec ce disque et non les deux autres. Je **sais** que ce produit fonctionne avec la Baseboard Mini et, dans le doute, je ne saurais que trop vous conseiller d'opter pour cette solution également. Au moins, avant de sombrer dans la folie dépressive et jeter l'éponge, testez un ou plusieurs autres SSD SATA...

4. LINUX RISC-V 32 BITS SUR LITEX ACORN BASEBOARD MINI

4.1 La base : bitstream et BIOS

Notre première construction, qui sera une petite mise en jambe avant de passer aux choses sérieuses, consistera à créer et à utiliser un SoC sur une base *softcore* VexRiscv. C'est un processeur RISC-V RV32I, 32 bits donc, écrit en SpinalHDL, directement disponible lorsqu'on installe LiteX (`pythondata-cpu-vexriscv/`). Ce *softcore* est celui que nous avons utilisé dans le précédent article pour découvrir LiteX et exécuter le « BIOS » ainsi que le code de démonstration. Mais, avec suffisamment de ressources, il est possible d'aller bien plus loin qu'un *donut* qui tourne.

LiteX n'est pas qu'un *framework* et un système de construction, il comprend également un certain nombre d'exemples conséquents et travaillés permettant de réellement appréhender toute l'étendue du travail accompli par les développeurs. Ainsi, il nous suffit de reposer sur le contenu du dépôt que nous avons déjà cloné pour l'occasion, et se trouvant dans `linux-on-litex-vexriscv/`. C'est aussi

l'occasion de voir, très superficiellement, à quel point il est facile, avec LiteX, de créer un SoC et tout son environnement logiciel. Ce répertoire contient tout le nécessaire, de la description du matériel aux fichiers de configuration permettant de construire un système compatible avec Buildroot.

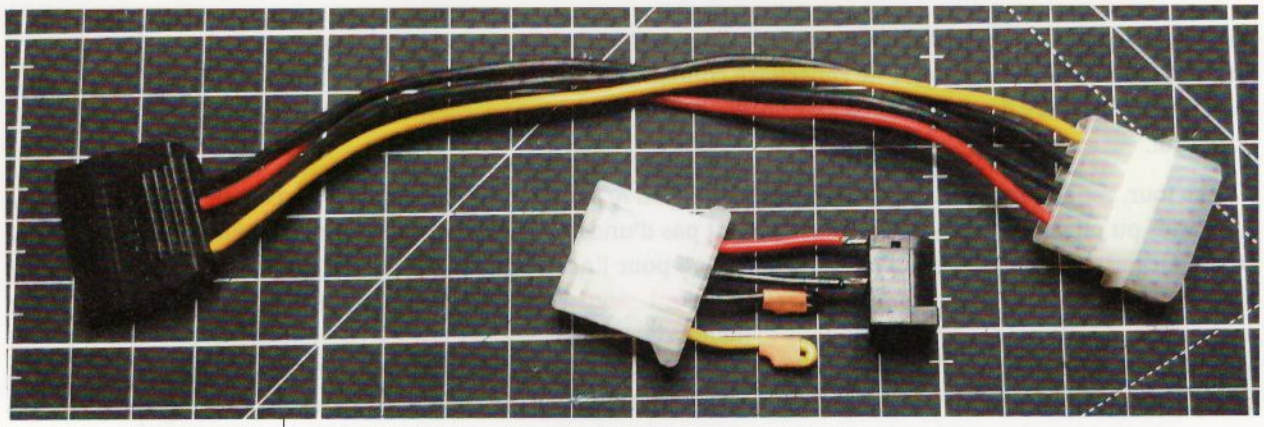
À ce jour, cependant, la Baseboard Mini et son SQRL Acorn CLE215+ ne sont pas directement supportés ou plutôt, devrais-je dire, ne disposent pas d'une description adaptée et clé en main. Ce qui est présent, en revanche, est une description pour l'Acorn telle que référencée dans le fichier `litex-boards/litex_boards/targets/sqrl_acorn.py`, par opposition au `litex_acorn_baseboard_mini.py` présent dans le même répertoire. Ce `sqrl_acorn.py` concerne le module SQRL *hacké* en *devkit*, mais la Baseboard Mini est sensiblement différente. En l'état, vous pouvez lister les *targets* disponibles à l'aide d'un simple :

```
$ cd linux-on-litex-vexriscv
$ python3 make.py --help
[...]
Available boards:
- acorn
- acorn_pcie
- aesku40
- alveo_u250
- alveo_u280
- arty
- arty_a7
- arty_s7
- butter_stick
- cam_link4k
- colorlight_i5
- de0nano
[...]
```

En plus de la liste des cartes, à utiliser avec l'option `--board=`, vous obtiendrez également un certain nombre d'options, supportées par `make.py` permettant d'ajuster quelques réglages.

Inutile ici de s'embêter à bidouiller dans les coins sur la base de `acorn`, nous pouvons ajouter le support qui nous intéresse directement, en éditant `board.py` et en complétant avec :

```
# LiteX Acorn Baseboard Mini ----------------------------------------------------

class LitexAcornBaseboardMini(Board):
    soc_kwargs = {"uart_name": "serial", "sys_clk_freq": int(75e6),
                  "ident_version": True, "sata_gen": "gen1"}
    def __init__(self):
        from litex_boards.targets import litex_acorn_baseboard_mini
        Board.__init__(self, litex_acorn_baseboard_mini.BaseSoC, soc_capabilities={
            "serial",
            "ethernet",
            #"spiflash",
            "sata",
        })
```



La plupart des SSD SATA actuels n'ont pas réellement besoin d'une alimentation proposant toutes les tensions, et le +12 V en particulier. On peut donc se bricoler un adaptateur et alimenter le disque uniquement en +5 V avec, par exemple, un bloc capable de fournir suffisamment de courant (idéalement 1 A, mais mes mesures ont montré que le SSD PNY ne dépasse jamais 120 mA).

Ajouter une carte pour ce projet se résume à créer une classe Python en référençant directement la cible de base telle que décrite dans `boards/litex_boards/targets*.py` (d'où le `from litex_boards.targets import litex_acorn_baseboard_mini`). Tout ce qui est décrit à cet endroit sera ainsi réutilisé et, en lieu et place des options que nous avons vues la dernière fois pour tester LiteX sur différentes cartes FPGA, nous réglons les paramètres via la variable `soc_kwargs` et spécifions ce que le SoC doit intégrer comme périphériques, via `soc_capabilities`. Parmi les arguments ajustant la création du SoC, nous avons :

- `uart_name` pour le type d'interface série, avec ici `serial` correspondant à la liaison accessible via le FT4232 présent sur la carte (troisième port USB/série apparaissant à la connexion de J7 en USB-C) ;
- `sys_clk_freq` précisant la fréquence d'horloge à utiliser pour le SoC, ici 75 MHz (la seule fréquence avec laquelle je n'ai eu strictement aucun problème, comparé à 150 MHz ou 125 MHz, mais j'avoue avoir un peu perdu le fil avec mes disques capricieux) ;
- `ident_version` qui permet simplement d'ajouter des informations de *build* dans le « BIOS », très utile lorsqu'on tâtonne et qu'on ne sait plus quel terminal vient de construire quelle variation ;
- `sata_gen` réglant la version de SATA supportée entre `gen1` (75 MHz, 1,5 Gb/s), `gen2` (150 MHz, 3 Gb/s) et `gen3` (300 MHz, 6 Gb/s). Ceci avec une restriction dans `litex/litex/soc/integration/soc.py` impliquant `self.clk_freq >= sata_clk_freq/2`. J'ai ici opté pour `gen1` qui, avec un SSD fiable, ne présente, apparemment, jamais de problème d'aucune sorte. Ceci peut certainement être affiné, mais après mes déboires, je me montre prudent et vous conseille de faire de même : ce sera dont 75 MHz et Gen1.

Côté périphériques, nous activons bien entendu le port série, mais également l'interface Ethernet (connectée au LAN via le module SFP sur J3) ainsi que le port SATA directement présent sur la carte. Que vous l'utilisiez ou non, dans un premier temps, importe peu. Ceci ne concerne que le design du SoC, pas les périphériques supportés ou non par le futur système.

Une fois la modification apportée au `board.py`, la nouvelle carte apparaît dans la liste :

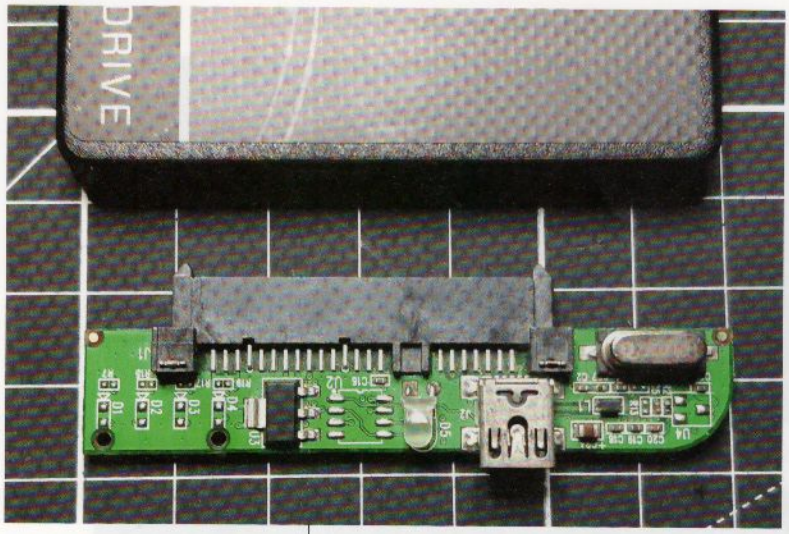
```
$ python3 make.py --help
[...]
- acorn
- acorn_pcie
- aesku40
[...]
- kcu105
- konfekt
- litex_acorn_baseboard_mini
- mini_spartan6
[...]
```

Mais, surtout, nous pouvons lancer la construction de l'ensemble avec :

```
$ python3 make.py --board=litex_acorn_baseboard_mini \
--uart-baudrate=1e6 --build

INFO:SoC:         __   _ __      _  __
INFO:SoC:        / /  (_) /____ | |/_/
INFO:SoC:       / /__/ / __/ -_)>  <
INFO:SoC:      /____/_/\__/\__/_/|_|
INFO:SoC:  Build your hardware, easily!
INFO:SoC:----------------------------------------
INFO:SoC:Creating SoC... (2024-10-18 14:35:23)
INFO:SoC:----------------------------------------
[...]
0 Infos, 0 Warnings, 0 Critical Warnings and 0 Errors encountered.
write_cfgmem completed successfully
# quit
INFO: [Common 17-206] Exiting Vivado at Fri Oct 18 14:39:39 2024...
$
```

Notez que le nom de la carte (`litex_acorn_baseboard_mini`) correspond au nom de la classe (`LitexAcornBaseboardMini`) avec une transformation de notation de *camel case* vers *snake case*, faite par `make.py` avec `camel_to_snake()`.

Le temps de construction dépendra de votre configuration et de la puissance de votre CPU, mais c'est l'affaire de quelques minutes tout au plus. Ce que fait `make.py` est non seulement produire la description Verilog qui est ensuite traitée par la suite Vivado d'AMD/Xilinx, mais également compiler un code de base, le fameux « BIOS », dont le binaire sera embarqué dans le design sous forme de ROM. La structure et les fonctionnalités de ce « BIOS » sont dépendantes des options que nous avons choisies pour la construction, tout comme la définition des registres (`csr.csv`), les entêtes (`csr.h`, `mem.h`, `soc.h`, etc.) et le *Device Tree* (`litex_acorn_baseboard_mini.dts`)) décrivant les périphériques en présence. Tout ceci, en plus du *bitstream* `litex_acorn_baseboard_mini.bit`, la configuration binaire du FPGA, est placé dans un sous-répertoire `litex_acorn_baseboard_mini` du répertoire `build/` créé à la construction.



Pour préparer le SSD à la fois pour les images à charger en mémoire et pour, éventuellement, le système de fichiers racine, un adaptateur USB/SATA est indispensable. Celui-ci, relativement vieux (notez le connecteur mini-USB), a été recyclé d'un ancien boîtier externe pour disque dur.

Notez l'argument `--uart-baudrate=1e6` utilisé à l'exécution de `make.py`, nous permettant de régler le débit du port série sur 1000000 bps en lieu et place des 115200 par défaut. Ceci n'est pas absolument critique, mais vous évitera de mettre votre patience à l'épreuve à l'étape finale de ce premier essai. En effet, dans sa configuration minimale, notre petit projet de SBC Linux utilisera la liaison série pour charger en mémoire les éléments du système (*bootloader* OpenSBI, image du noyau, DTB et *RAMdisk*). Transférer quelque 15 Mio ainsi, à 115200 bps, est une souffrance en soi, d'autant plus lorsqu'il s'agit de le faire plusieurs fois de suite. Le passage de l'UART à 1000000 bps est donc quasi indispensable, mais pourra, par la suite, être abandonné.

Pour l'heure, nous disposons d'un SoC avec une ROM, qu'il est grand temps de configurer dans le FPGA. Pour ce faire, et c'est aussi l'intérêt d'utiliser la cible `litex_acorn_baseboard_mini.py` via la modification du `board.py`, nous pouvons tout simplement brancher le connecteur USB-C en J7 au PC puis alimenter la carte via l'autre port USB-C (USB PD) en J9. Grâce à notre modification, nous disposons automatiquement du support pour la sonde JTAG intégrée (via le FT4232) sans avoir besoin d'utiliser OpenOCD manuellement. Nous pouvons utiliser à nouveau `make.py` en substituant `--build` par `--load` pour configurer le FPGA en SRAM, ou `--flash` pour programmer le *bitstream* en flash et ainsi rendre la configuration résistante à une coupure d'alimentation.

Avant de le faire cependant, ouvrez un autre terminal, rendez-vous dans le répertoire d'installation de LiteX, activez l'environnement virtuel Python et utilisez la commande `litex_term --speed=1e6` suivie de l'entrée `/dev` correspondant au troisième port série créé lors de la connexion du FT4232 en USB-C (`/dev/ttyUSB3` chez moi, mais probablement `/dev/ttyUSB2` pour vous, car j'ai déjà quelque chose sur `/dev/ttyUSB0`).

Le FT4232 propose 4 interfaces, la première (*ADBUS*) est pour le JTAG, la seconde (*ACBUS*) est non connectée, la troisième (*BDBUS*) est pour la liaison série et la dernière (*BCBUS*) est également inutilisée (voir le PDF du schéma de la carte sur le dépôt Git **[13]**).

Chargez ou flashez le *bitstream* avec `make.py` et, sur le terminal `litex_term` nous obtenons :

```
        / /  (_) /____ | |/_/
       / /__/ / __/ -_)>  <
      /____/_/\__/\__/_/|_|
    Build your hardware, easily!

(c) Copyright 2012-2024 Enjoy-Digital
(c) Copyright 2007-2015 M-Labs

BIOS built on Oct 18 2024 14:35:59
BIOS CRC passed (7a982ae7)

LiteX git sha1: 64cf925b3

--------------- SoC ------------------
CPU:            VexRiscv SMP-LINUX @ 75MHz
BUS:            wishbone 32-bit @ 4GiB
CSR:            32-bit data
ROM:            64.0KiB
SRAM:           6.0KiB
SDRAM:          1.0GiB 16-bit @ 600MT/s (CL-7 CWL-5)
MAIN-RAM:       1.0GiB
[...]
--------------- Boot -----------------
Booting from serial...
Press Q or ESC to abort boot completely.
sL5DdSMmkekro
Timeout
Booting from SATA...
Booting from boot.json...
Booting from boot.bin...
SATA boot failed.
Booting from network...
Local IP: 192.168.1.50
Remote IP: 192.168.1.100
Booting from boot.json...
Booting from boot.bin...
Copying boot.bin to 0x40000000...
Network boot failed.
No boot medium found
-------------- Console ----------------

litex>
```

Ceci est le même code, le « BIOS », que nous avons vu la fois précédente pour notre prise en main de LiteX (vous pouvez revenir au shell d'un simple double Ctrl+c), mais l'aide est étoffée de plusieurs commandes supplémentaires :

```
litex> help
LiteX BIOS, available commands:
[...]
```

```
sataboot          - Boot from SATA
netboot           - Boot via Ethernet (TFTP)
serialboot        - Boot from Serial (SFL)
reboot            - Reboot
boot              - Boot from Memory
[...]
sata_rwtest       - SATA read/write test
sata_mem2sec      - Write SATA from memory
sata_write        - Write SATA sector
sata_sec2mem      - Read SATA into memory
sata_read         - Read SATA sector
sata_init         - Initialize SATA

litex>
```

La fois précédente, nous avons utilisé `litex_term` avec l'option `--kernel` suivie d'un chemin vers un binaire pour provoquer le chargement et l'exécution d'un code « utilisateur » de démonstration. Dans ce mode `serialboot`, c'est `litex_term` qui envoie directement les données, sur demande du « BIOS » au démarrage. Ici s'ajoutent `netboot` pour une récupération TFTP et `sataboot` pour une lecture depuis une partition FAT16 sur un SSD. Commençons par la solution la plus simple et le transfert série, ne nécessitant aucun matériel supplémentaire.

Mais encore faut-il avoir quelque chose à transférer...

4.2 Un système pour notre SoC

Pour éprouver notre SoC tout frais sorti de l'œuf, rien de tel qu'un petit code de démonstration compilé pour l'occasion. Mais cette fois, le code est gros, très gros, puisqu'il s'agit d'un système, certes minimaliste, mais complet. Et lorsqu'on parle d'un système Linux pour l'embarqué, on parle, 9 fois sur 10 de quelque chose de construit avec la crème de la crème du système de construction, simple et sobre : Buildroot (oui, on a aussi Yocto/OpenEmbedded, mais j'ai dit « simple et sobre »).

Et heureusement pour nous, le dépôt de ce projet intègre un répertoire `buildroot/` comprenant tous les éléments d'une arborescence externe pour Buildroot (alias `BR2_EXTERNAL`, voir **[16]** et **[17]**). Attention, il ne s'agit pas là d'une archive de Buildroot, mais des éléments (configurations, script, patches, etc.) permettant au système de construction de faire ce travail pour la cible qui nous intéresse ici.

La première chose à faire pour créer le système sera de remonter d'un répertoire et se placer dans le répertoire où nous avons cloné le dépôt de Buildroot en début d'article, puis de charger la configuration par défaut pour notre SoC (« SBC » même à ce stade), avant de tout compiler :

```
$ cd ../buildroot/
$ make BR2_EXTERNAL=../linux-on-litex-vexriscv/buildroot/ \
  litex_vexriscv_defconfig
$ make -j14
```

De ces trois disques testés pour fournir un support de stockage au projet, seul celui de droite, le PNY, fonctionne sans poser de problème. Le SSD Intel (au centre) permet de charger les images en mémoire, mais ne répond pas au pilote Linux et le disque de gauche, mécanique, à un comportement presque totalement aléatoire. Le problème n'est pas l'état du support de stockage puisque ces disques incompatibles fonctionnent à merveille via USB ou dans un PC.

Le `-j14` est l'option permettant de paralléliser les compilations, avec `14` étant le nombre de tâches (ici sur un AMD Ryzen 9 5900HX avec 8 cœurs/16 *threads*). Le processus est long, environ 15 minutes chez moi pour un *build* complet (sans *ccache*). Buildroot prend tout en charge, qu'il s'agisse des outils locaux et du compilateur croisé, les téléchargements et, bien entendu, la compilation. Il faut bien comprendre, en revanche, que la configuration `litex_vexriscv_defconfig` implique l'utilisation de chemins **relatifs** et que l'emplacement du répertoire `buildroot/` (clone Git) **doit** être celui que nous avons défini ici. Ceci concerne en particulier `linux-on-litex-vexriscv/buildroot/board/litex_vexriscv/post-image.sh`, exécuté après la construction, et qui créera le contenu du répertoire `images/` dans `linux-on-litex-vexriscv/` avec :

- `opensbi.bin` : le *bootloader* OpenSBI, typique des plateformes RISC-V ;
- `rv32.dtb` : le *device tree* binaire décrivant les périphériques non détectables, issu du DTS généré en même tant que le *bitstream* ;
- `Image` : l'image du noyau Linux ;
- `rootfs.cpio` : le système de fichiers racine qui constituera le *RAMdisk* utilisé pour `/` ;
- `sdcard.img` : une image pour une carte SD/microSD ;
- `boot.vfat` : le système de fichiers VFAT utilisé pour l'image `sdcard.img` ;
- `boot.json` : la description des éléments à placer en mémoire pour le *boot*.

Ce dernier fichier référence les éléments binaires sous la forme de paire fichier/adresse :

```
{
        "Image"       : "0x40000000",
        "rv32.dtb"    : "0x40ef0000",
        "rootfs.cpio" : "0x41000000",
        "opensbi.bin" : "0x40f00000"
}
```

Et c'est ce fichier que nous allons utiliser, avec `litex_term`, mais en lieu et place de `--kernel`, nous utiliserons l'option dédiée, `--images` ainsi :

```
$ litex_term --speed=1e6 \
  --images=images/boot.json /dev/ttyUSB3
```

Il nous suffira alors d'utiliser la commande `reboot` ou `serialboot` pour provoquer un démarrage, à l'adresse `0x40f00000`, après téléchargement des 4 éléments via la liaison série :

```
litex> serialboot

Booting from serial...
Press Q or ESC to abort boot completely.
sL5DdSMmkekro
[LITEX-TERM] Received firmware download request from the device.
[LITEX-TERM] Uploading images/Image to 0x40000000 (8764048 bytes)...
[LITEX-TERM] Upload calibration... (inter-frame: 10.00us, length: 64)
[LITEX-TERM] Upload complete (85.1KB/s).
[LITEX-TERM] Uploading images/rv32.dtb to 0x40ef0000 (2735 bytes)...
[LITEX-TERM] Upload calibration... (inter-frame: 10.00us, length: 64)
[LITEX-TERM] Upload complete (71.2KB/s).
[LITEX-TERM] Uploading images/rootfs.cpio to 0x41000000 (4219904 bytes)...
[LITEX-TERM] Upload calibration... (inter-frame: 10.00us, length: 64)
|===>             | 19%
```

Une fois le saut effectué, le OpenSBI démarre, passe le relais au noyau et le reste du processus de *boot* se poursuit jusqu'à :

```
[...]
[LITEX-TERM] Booting the device.
[LITEX-TERM] Done.
Executing booted program at 0x40f00000
--============= Liftoff! ===============--
OpenSBI v1.3
[...]
[    0.000000] Linux version 6.9.0 (denis@newbeast)
  (riscv32-buildroot-linux-gnu-gcc.br_real (Buildroot
  2024.08-814-gb98062f730) 13.3.0, GNU ld (GNU Binutils) 2.42)
  #1 SMP Sat Oct 19 08:52:17 CEST 2024
[    0.000000] Machine model: litex_acorn_baseboard_mini
[    0.000000] SBI specification v1.0 detected
[...]
Saving 256 bits of non-creditable seed for next boot
Starting syslogd: OK
Starting klogd: OK
Running sysctl: OK
Starting network: OK
```

```
Starting crond: OK
crond[73]: crond (busybox 1.36.1) started, log level 8
Welcome to Buildroot
buildroot login:
```

Le nom d'utilisateur est `root` et aucun mot de passe n'est défini :

```
buildroot login: root
                    __   _
                   / /  (_)__  __ ____ __
                  / /__/ / _ \/ // /\ \ /
                 /____/_/_//_/\_,_//_\_\
                      / _ \/ _ \
   __   _ __        _  _\___/_//_/          ___  _
  / /  (_) /____ | |/_/__| | / /____ __ / _ \(_)__ _____  __
 / /__/ / __/ -_)>  </___/ |/ / -_) \ // , _/ (_-</ __/ |/ /
/____/_/\__/\__/_/|_|____|___/\__/_\_\/_/|_/_/___/\__/|___/
                    / __/  |/  / _ \
                   _\ \/ /|_/ / ___/
                  /___/_/  /_/_/
  32-bit RISC-V Linux running on LiteX / VexRiscv-SMP.

login[74]: root login on 'console'
root@buildroot:~#
```

Nous avons un système Linux 6.9.0, avec un *userland* principalement basé sur BusyBox, démarré sur un *RAMdisk*, le tout sur un SoC *softcore* RISC-V :

```
root@buildroot:~# uname -a
Linux buildroot 6.9.0 #1 SMP Sat Oct 19 08:52:17
  CEST 2024 riscv32 GNU/Linux
root@buildroot:~# cat /proc/cpuinfo
processor       : 0
hart            : 0
isa             : rv32ima
mmu             : sv32
mvendorid       : 0x0
marchid         : 0x0
mimpid          : 0x0
hart isa        : rv32ima

root@buildroot:~# mount
rootfs on / type rootfs (rw)
devtmpfs on /dev type devtmpfs
   (rw,relatime,size=510508k,nr_inodes=127627,mode=755)
proc on /proc type proc (rw,relatime)
devpts on /dev/pts type devpts
```

```
    (rw,relatime,gid=5,mode=620,ptmxmode=666)
tmpfs on /dev/shm type tmpfs (rw,relatime)
tmpfs on /tmp type tmpfs (rw,relatime)
tmpfs on /run type tmpfs (rw,nosuid,nodev,relatime,mode=755)
sysfs on /sys type sysfs (rw,relatime)
```

Mais ce n'est pas tout. Si vous avez ajouté un module SFP et connecté l'interface de votre carte au LAN, vous pouvez faire ceci :

```
root@buildroot:~# udhcpc
udhcpc: started, v1.36.1
udhcpc: broadcasting discover
udhcpc: broadcasting select for
  192.168.0.108, server 192.168.0.100
udhcpc: lease of 192.168.0.108 obtained
  from 192.168.0.100, lease time 600
deleting routers
adding dns 192.168.0.100
adding dns 81.253.149.13

root@buildroot:~# ping connect.ed-diamond.com
PING connect.ed-diamond.com (91.200.144.250): 56 data bytes
64 bytes from 91.200.144.250: seq=0 ttl=46 time=25.948 ms
64 bytes from 91.200.144.250: seq=1 ttl=46 time=25.685 ms
64 bytes from 91.200.144.250: seq=2 ttl=46 time=25.557 ms
^C
--- connect.ed-diamond.com ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 25.557/25.730/25.948 ms

root@buildroot:~#
```

C'est bien plus qu'un simple test, mais vous disposez d'un système complet et ça, c'est juste extraordinaire !

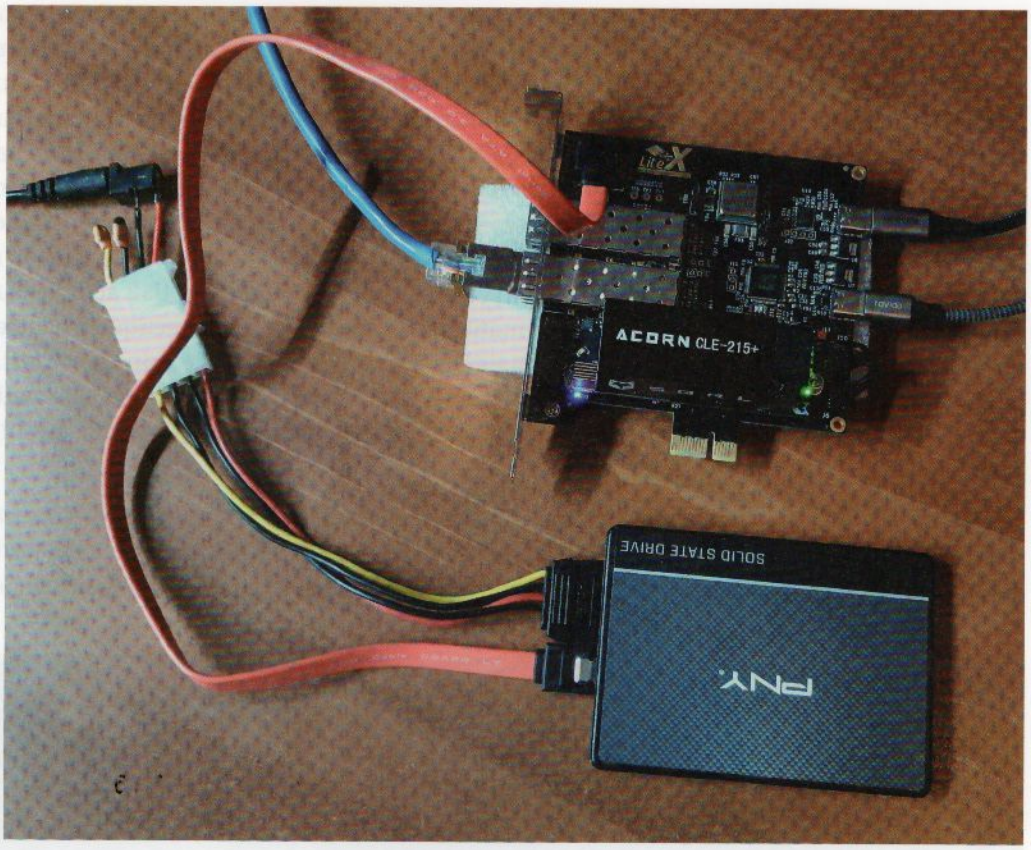
4.3 Ajoutons un SSD en SATA !

Le transfert série des images, même accéléré avec un passage à 1000000 bps, fonctionne mais est tout de même relativement pénible. Une autre solution est de charger ces éléments depuis un disque SATA, de façon beaucoup plus rapide. En termes de praticité, ceci se discute, puisqu'en cas de mise à jour des binaires, ceci implique des déconnexions, connexions, montages, démontages, déconnexions, reconnexions... Mais c'est aussi et surtout une excuse pour utiliser l'interface SATA. Nous allons ici aller plus loin que ce qui est prévu à l'origine par la configuration Buildroot/Linux, en fournissant également au système un accès à ce périphérique de stockage, et ceci implique quelques ajustements.

Mais avant toute chose se pose la problématique de la connexion et surtout de l'alimentation. Connecter un SSD SATA à la carte n'est pas un problème, les connecteurs sont standard. Pour l'alimentation, en revanche, c'est plus délicat. L'approche réflexe consiste à utiliser une alimentation de PC, sensiblement bidouillée pour une activation manuelle, avec un interrupteur entre la masse et le signal PS-ON (vert). Ça fonctionne, mais c'est encombrant et on peut faire plus simple. La très grande majorité des SSD SATA du marché n'ont, en effet, pas besoin de toutes les tensions présentes sur un connecteur d'alimentation SATA à 15 broches. Seules la/les masses et le +5 volts sont alors nécessaires et il est possible de prendre cette tension ailleurs, comme depuis un bloc d'alimentation (type Raspberry Pi) ou un hub USB actif. Il faudra torturer/recycler un connecteur d'alimentation SATA ainsi que la connectique à l'autre bout (jack, barrel, micro-USB femelle, etc.), mais l'effort vaut clairement le gain en confort. La carte ne propose malheureusement pas de connecteur avec masse/+5 V, ce qui est relativement dommage, une solution tout-en-un, basée sur l'USB-PD et un régulateur dédié, aurait été merveilleuse...

Une fois l'aspect matériel traité, nous pouvons d'ores et déjà tester la connectivité directement depuis le « BIOS », en interrompant si besoin le démarrage avec des pressions hystériques sur la touche « Échap » (ou tout simplement supprimer l'option `--images` de la ligne de commande). Là, les commandes `sata*` sont à notre disposition pour :

```
litex> help
[...]
sata_rwtest       - SATA read/write test
sata_mem2sec      - Write SATA from memory
sata_write        - Write SATA sector
sata_sec2mem      - Read SATA into memory
sata_read         - Read SATA sector
sata_init         - Initialize SATA

litex> sata_init
Initialize SATA...
Model:    PNY 250GB SATA SSD
Capacity: 250GB
Successful.

litex> sata_read 0x64
Memory dump:
0x10001530  01 21 02 21 03 21 04 21 05 21 06 21 07 21 08 21  .!.!.!.!.!.!.!.!
0x10001540  09 21 0a 21 0b 21 0c 21 0d 21 0e 21 0f 21 10 21  .!.!.!.!.!.!.!.!
0x10001550  11 21 12 21 13 21 14 21 15 21 16 21 17 21 18 21  .!.!.!.!.!.!.!.!
0x10001560  19 21 1a 21 1b 21 1c 21 1d 21 1e 21 1f 21 20 21  .!.!.!.!.!.!.! !
0x10001570  21 21 22 21 23 21 24 21 25 21 26 21 27 21 28 21  !!"!#!$!%!&!'!(!
0x10001580  29 21 2a 21 2b 21 2c 21 2d 21 2e 21 2f 21 30 21  )!*!+!,!-!.!/!0!
0x10001590  31 21 32 21 33 21 34 21 35 21 36 21 37 21 38 21  1!2!3!4!5!6!7!8!
0x100015a0  39 21 3a 21 3b 21 3c 21 3d 21 3e 21 3f 21 40 21  9!:!;!<!=!>!?!@!
[...]
```



Voici l'installation complète ayant permis de procéder aux différentes expérimentations pour cet article. Notez que l'utilisation d'une alimentation « légère » pour le SSD et non un modèle extrait d'un PC rend les choses beaucoup plus faciles à gérer en termes d'encombrement et de manipulation. Si seulement la baseboard fournissait un connecteur +5 V/1 A, ce serait absolument parfait.

Nous voyons ici que le SSD est parfaitement reconnu lors de l'initialisation et les données accessibles sans problème (ce qui était aussi étrangement le cas avec le SSD Intel). Mais la plus intéressante de ces commandes est, bien entendu, `sataboot` qui fonctionne suivant le même principe que `serialboot`, mais via un média différent : le fichier `boot.json` est lu depuis un système de fichiers FAT16 ou 32, placé sur la **première** partition du support, puis en fonction de son contenu, les binaires sont chargés en mémoire avant de sauter à l'adresse `0x40f00000` où se trouve OpenSBI.

En l'état, nous pourrions simplement préparer le disque et nous contenter de cela, mais nous allons compiler un nouveau noyau pour ajouter le support LiteX SATA qui n'est pas actif par défaut. Pour ce faire, placez-vous dans le répertoire de construction de Buildroot (le clone) et utilisez :

```
$ make linux-menuconfig
```

Ceci vous présentera la classique interface de configuration des sources du noyau. Deux approches sont ici possibles, utiliser et modifier la configuration présente dans le dépôt LiteX (`linux-on-litex-vexriscv/buildroot/board/litex_vexriscv/linux.config`), ou partir de cette dernière pour en créer une nouvelle, stockée dans un autre fichier, ce qui suppose alors de changer la configuration de Buildroot localement (`.config`) ou de l'enregistrer comme

nouvelle configuration par défaut (`*defconfig`). À vous de choisir, mais par simplicité, j'opterai ici pour la première solution. Dans cette interface, chargez la configuration via ***Load*** (en bas) puis allez activer ***LiteX LiteSATA block device support*** (`CONFIG_LITESATA`) dans ***Device Drivers*** et ***Block devices***. Enregistrez ensuite les changements dans le même fichier que celui chargé (***Save*** en bas) et quittez. Il ne vous reste plus alors qu'à reconstruire le noyau ainsi que les images qui en découlent :

```
$ make linux-dirclean
$ make -j14 linux-rebuild all
```

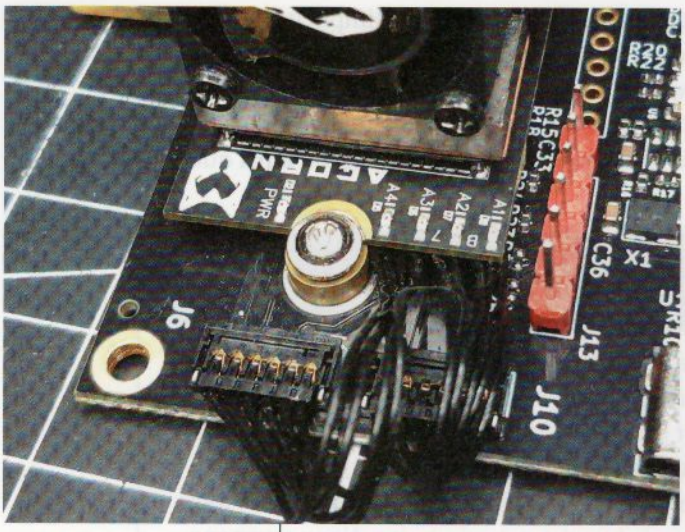
Ceci devrait aller beaucoup plus vite qu'une construction complète et vous pourrez constater que le contenu de `images/` a changé (date de modification). Mais nous n'en avons pas fini, car même si cet ajustement a bien activé le support (pilote) dans le noyau et que le contrôleur est présent dans notre SoC, ce dernier n'est pas détectable automatiquement. Nous devons donc déclarer sa présence dans le *device tree*, qui a été généré précédemment et se trouve dans `build/litex_acorn_baseboard_mini/litex_acorn_baseboard_mini.dts`. Nous revenons donc dans `linux-on-litex-vexriscv/`, copions le fichier en `litex_acorn_baseboard_mini_sata.dts` dans le répertoire courant et l'éditons ainsi :

```
soc {
[...]
  // Example DTS node (adjust with your own addresses from csr.csv):
  litesata0: litesata@f0004000 {
      compatible = "litex,litesata";
      reg = <0xf0004000 0x100>, // sata_identify
            <0xf0005800 0x100>, // sata_phy
            <0xf0006000 0x100>, // sata_sector2mem
            <0xf0005000 0x100>, // sata_mem2sector
            <0xf0004800 0x100>; // sata_irq
      reg-names = "ident", "phy", "reader", "writer", "irq";
      interrupts = <4>;
  };
};
```

Les informations fournies ici ne sortent pas de nulle part, un extrait de DTS est donné dans les sources du pilote (`litesata.c`) et, comme le précise le commentaire, il faut se référer au fichier `csr.csv` (dans `build/litex_acorn_baseboard_mini/`) pour connaître les adresses des cinq registres ainsi que l'IRQ utilisée. Ces valeurs, cependant, peuvent être reprises directement, car elles seront identiques à celles que vous aurez dans votre propre `csr.csv`, avec la même configuration du SoC.

Nous pouvons alors enregistrer les modifications et générer un nouveau fichier `rv32.dtb` avec :

```
$ dtc -O dtb litex_acorn_baseboard_mini_sata.dts \
  -o images/rv32.dtb
```



Le module SQRL Acorn CLE-215+ est assez pauvre en I/O, alors que le FPGA Artix7 XC7A200T en dispose à profusion. Non testé pour cet article, il doit cependant être possible d'utiliser le connecteur LVDS caché sous le radiateur et à peine visible ici (juste au niveau des pointillés, au-dessus de la sérigraphie « ACORN » à l'envers).

Là, deux options s'offrent à vous (encore ?) : tester en chargeant les images via la liaison série comme précédemment, ce qui devrait vous montrer, après démarrage, que le SSD est effectivement détecté, ou passer directement à l'étape d'après en préparant le disque pour *booter* en SATA. Inutile d'être prudent ici, passons à « la totale » sans attendre (d'autant qu'en réalité, cela doit être ma 50e fois, sinon plus, merci les HD/SSD non compatibles). Avant cela, il pourra être intéressant de relancer la construction du SoC et de reflasher la configuration du FPGA en omettant l'option `--uart-baudrate=1e6`, pour laisser l'UART en 115200 bps par défaut. « Flasher » parce qu'à force de débrancher/brancher le disque, utiliser `--load` à répétition est vite pénible. Ceci n'est nullement impératif, mais sachant que ce débit est beaucoup plus standard et que nous n'aurons pas besoin de charger les binaires via la liaison série, autant revenir à quelque chose de « normal ». Si vous choisissez de vous épargner cet effort, pensez à ajouter `--speed=1e6` aux prochaines occurrences de `litex_term` dans la suite de l'article. Au contraire, si vous revenez à 115200 bps, sachez que l'invocation de `python3 make.py` avec l'option `--build` écrasera votre `images/rv32.dtb`. Vous devrez donc relancer `dtc` pour générer la version binaire intégrant la description du contrôleur SATA.

Côté préparation du disque, nous devons disposer d'une première partition, destinée à accueillir un système de fichiers FAT16 (ou 32) pour y placer les images et le JSON. Nous ajoutons également ici une seconde partition pour de l'ext2 et faire quelques essais. Les tailles de ces partitions importent peu, étant donné le volume de données. J'ai opté pour, respectivement, 128 Mio et 512 Mio. Le disque est préparé, sur PC GNU/Linux, via un adaptateur SATA/USB, en utilisant la bonne vieille commande `fdisk`. Notez qu'il semble très important d'activer l'indicateur de compatibilité DOS (commande `c`) avant de créer les partitions (ceci semble être une limitation de la *libfatfs*). Au final, le disque est structuré ainsi :

```
Commande (m pour l'aide) : p
Disque /dev/sdc : 232,89 GiB,
    250059350016 octets, 488397168 secteurs
Modèle de disque : 0GB SAT External
Unités : secteur de 1 × 512 = 512 octets
Taille de secteur (logique / physique) : 512 octets / 512 octets
taille d'E/S (minimale / optimale) : 512 octets / 512 octets
Type d'étiquette de disque : dos
Identifiant de disque : 0xe29d00dc
```

```
Périphérique Amorçage  Début     Fin Secteurs Taille Id Type
/dev/sdc1                 63  262207   262145   128M  6 FAT16
/dev/sdc2             262208 1310784  1048577   512M 83 Linux
```

On créera ensuite les systèmes de fichiers respectivement avec `mkdosfs /dev/sdc1` et `mke2fs /dev/sdc2` et on pourra alors monter le système de fichiers FAT16 pour y copier les fichiers `boot.json`, `Image`, `opensbi.bin`, `rootfs.cpio` et `rv32.dtb`. Ceci fait, il ne restera plus qu'à tout connecter, alimenter le disque **avant** la carte, et observer de ses yeux émerveillés la magie opérer :

```
[...]

Booting from SATA...
Booting from boot.json...
Copying Image to 0x40000000 (8772320 bytes)...
[########################################]
Copying rv32.dtb to 0x40ef0000 (2899 bytes)...
[########################################]
Copying rootfs.cpio to 0x41000000 (4219904 bytes)...
[########################################]
Copying opensbi.bin to 0x40f00000 (263652 bytes)...
[########################################]
Executing booted program at 0x40f00000
--============== Liftoff! ===============--
OpenSBI v1.3
[...]
[   1.673977] LiteX SoC Controller driver initialized
[   4.547227] Initramfs unpacking failed: invalid magic
              at start of compressed archive
[   4.728575] Freeing initrd memory: 8192K
[   6.785872] f0001000.serial: ttyLXU0 at MMIO 0x0
              (irq = 12, base_baud = 0) is a liteuart
[   6.794938] printk: legacy console [liteuart0] enabled
[   6.794938] printk: legacy console [liteuart0] enabled
[   6.805124] printk: legacy bootconsole [liteuart0] disabled
[   6.805124] printk: legacy bootconsole [liteuart0] disabled
[   7.121841] litesata f0004000.litesata:
              250059350016 bytes; PNY 250GB SATA SSD
[   7.163646]  litesata: litesata1 litesata2
[   7.186212] litesata f0004000.litesata:
              probe success; sector size = 512
[   7.241853] liteeth f0002000.mac eth0: irq 14
              slots: tx 2 rx 2 size 2048
```

Les messages de démarrage nous montrent que le disque a bien été détecté, en nous présentant les informations le concernant ainsi que les partitions en présence, prenant la forme des périphériques `litesata1` et `litesata2`, accessibles dans `/dev`. Non, il ne s'agit pas de `sd*` mais d'entrées dédiées, puisque le sous-système *libsata* n'est pas utilisé. Ceci dit, ces disques seront utilisables comme n'importe quel périphérique bloc :

```
root@buildroot:~# mkdir /mnt/plop
root@buildroot:~# mount /dev/litesata2 /mnt/plop/
[   90.483726] EXT4-fs (litesata2): mounted filesystem
   1acc9fdc-989a-49d9-aea7-982122ccc49e r/w without journal.
   Quota mode: disabled.

root@buildroot:~# df -h
Filesystem           Size      Used Available Use% Mounted on
devtmpfs           498.5M         0    498.5M   0% /dev
tmpfs              502.7M         0    502.7M   0% /dev/shm
tmpfs              502.7M         0    502.7M   0% /tmp
tmpfs              502.7M     20.0K    502.6M   0% /run
/dev/litesata2     503.6M      1.2M    476.8M   0% /mnt/plop

root@buildroot:~# mount
rootfs on / type rootfs (rw)
[...]
tmpfs on /tmp type tmpfs (rw,relatime)
tmpfs on /run type tmpfs (rw,nosuid,nodev,relatime,mode=755)
sysfs on /sys type sysfs (rw,relatime)
/dev/litesata2 on /mnt/plop type ext4 (rw,relatime)

root@buildroot:~# umount /mnt/plop/
[  108.480627] EXT4-fs (litesata2): unmounting filesystem
   1acc9fdc-989a-49d9-aea7-982122ccc49e.
root@buildroot:~#
```

Nous obtenons, effectivement un SBC avec un disque SATA parfaitement fonctionnel. Rendez-vous en bien compte, nous avons un *softcore* libre sous licence MIT, des périphériques sous licence BSD, un système sous GPLv2, construit par un système de *build* libre (GPLv2), le tout assemblé par un *framework* sous licence BSD. Si nous avions disposé d'une carte à base de Lattice ECP5, même la suite produisant le *bitstream* aurait été libre et *open source*.

Il est, bien entendu, possible de construire sur cette base, en personnalisant la configuration de Buildroot pour ajouter tout un tas de programmes, d'outils et de fonctionnalités au système. On peut également envisager d'abandonner totalement le *RAMdisk* pour avoir le système de fichiers racine directement sur le disque SATA. Les options ne manquent pas, mais plutôt que d'explorer dans ce sens, que diriez-vous de passer à la vitesse supérieure ? Le FPGA intégré à la carte permet de faire bien plus qu'un modeste SoC RISC-V 32 bits. Et si on passait en 64 bits et, tant qu'à faire, en SMP avec 4 cœurs RISC-V ?

5. ET SI ON PASSAIT EN 64 BITS AVEC 4 CŒURS ?

Le VexRiscv n'est pas le seul *softcore* utilisable avec LiteX qui, en plus, bénéficie de quelques démonstrations avancées avec ce *framework*. Un cran au-dessus, nous avons le *softcore* Rocket Chip, également basé sur l'architecture de jeu d'instructions (ISA) RISC-V, et plus précisément RV64GC, mais cette fois écrit en Chisel (que les lecteurs du magazine doivent connaître, voir article de Fabien Marteau dans le numéro 40 **[18]**, entre autres).

Le Rocket Chip est bien plus complexe que le VexRiscv, avec un adressage 64 bits, qui est donc capable de faire fonctionner un système plus « moderne », dans le sens où certains d'entre eux ont tout simplement décidé de ne pas supporter du tout RV32. Ceci ouvre donc la voie au fait de, à terme, sortir de la « norme » Linux pour s'aventurer dans des contrées plus exotiques (oui, je pense à *BSD). Mais nous n'en sommes pas, là. Avant toute chose, faisons déjà fonctionner ce qui existe et est officiellement supporté.

Pour Linux et Rocket Chip, le nécessaire n'a pas été automatiquement installé par `litex_setup.py` et nous devons, tout d'abord, compléter notre installation en commençant par le support LiteX du *softcore* lui-même :

```
$ git clone https://github.com/litex-hub/pythondata-cpu-rocket.git
$ cd pythondata-cpu-rocket
$ python setup.py install
$ cd ..
```

Souvenez-vous, nous sommes dans un environnement Python virtuel et n'avons donc pas de soucis avec le système de gestion de paquets ou l'installation locale de Python (Pip, etc.). Nous passons ensuite aux téléchargements avec, dans l'ordre, le dépôt contenant scripts et configuration pour le système, les sources du noyau Linux en version LiteX, Busybox pour un *userland* minimal, et les sources du *bootloader* OpenSBI :

```
$ git clone https://github.com/litex-hub/linux-on-litex-rocket.git
$ cd linux-on-litex-rocket
$ git clone https://github.com/litex-hub/linux -b litex-rebase
$ curl https://busybox.net/downloads/busybox-1.36.1.tar.bz2 | tar xfj -
$ git clone https://github.com/riscv-software-src/opensbi.git
```

Contrairement à notre précédente expérience avec VexRiscv, nous n'avons ici pas de constructeur de systèmes comme Buildroot à disposition et devrons tout faire nous-mêmes. Ce qui en soi n'est pas un problème, bien au contraire, cela nous donne deux approches à découvrir ou à redécouvrir.

Nous allons compiler et construire les éléments dans l'ordre manuellement, mais notez que vous trouverez dans le sous-répertoire `scripts/` deux fichiers scripts pouvant être utilisés pour le téléchargement et la construction. Mais avant de nous pencher sur le *software*, occupons-nous du *hardware*, ou plutôt du *gateware*, en créant le *bitstream* et le « BIOS », pour écrire la configuration du FPGA en flash :

```
$ python3 -m litex_boards.targets.litex_acorn_baseboard_mini \
--cpu-type rocket --cpu-variant linux --cpu-num-cores 4 \
--cpu-mem-width 2 --sys-clk-freq 75e6 --with-ethernet \
--with-sata --sata-gen 1 \
--output-dir build/litex_acorn_baseboard_mini_rocket \
--build --flash
```

Notez l'utilisation de l'option `--output-dir` nous évitant d'écraser une éventuelle précédente construction (typiquement celle de l'article précédent) et empêchant de nous mélanger les pinceaux par la suite (la version VexRiscv est dans `linux-on-litex-vexriscv/build/`, pas de problème de ce côté-là). Nous combinons également la construction avec le flashage pour simplifier les choses, car cette étape est bien plus longue que la fois précédente (pause café, donc). À titre de comparaison, les rapports de Vivado (`build/litex_acorn_baseboard_mini/gateware/*.rpt`) indiquent 10260 LUT utilisés pour le SoC VexRiscv contre 114878 pour le SoC Rocket Chip, soit plus de 85 % de ce dont est capable le FPGA Artix7 XC7A200T (contre 8 % pour le VexRiscv).

Une fois le matériel « créé », nous pouvons compiler les éléments du système dans leur ordre de démarrage et en suivant la chaîne de dépendances. La première étape sera de compiler le *device tree*, nécessaire au *bootloader* OpenSBI. En effet, nous suivons ici une procédure différente de celle de Buildroot qui utilise un DTB chargé séparément. Là, nous embarquons le *Device Tree* directement dans le binaire OpenSBI :

```
$ cd linux-on-litex-rocket
$ dtc -O dtb conf/litex_acorn_baseboard_mini.dts \
-o litex_acorn_baseboard_mini.dtb
$ cd opensbi/
$ make -j14 CROSS_COMPILE=riscv64-linux-gnu- \
  PLATFORM=generic FW_FDT_PATH=../litex_acorn_baseboard_mini.dtb \
  FW_JUMP_FDT_ADDR=0x82400000
$ cd ..
```

Le fichier `litex_acorn_baseboard_mini.dts` est fourni par le dépôt Git et ne nécessite pas de modification, il inclut de base la description pour le support du contrôleur SATA, mais il sera judicieux cependant d'en vérifier le contenu, en le comparant à celui de `build/litex_acorn_baseboard_mini_rocket/csr.csv`, généré par LiteX lors de la construction du SoC. Il n'est pas impossible que les choses évoluent entre le moment de la rédaction de l'article et celui où vous reproduirez ces manipulations. Une fois OpenSBI compilé, nous obtenons un binaire `fw_jump.bin` dans le sous-répertoire `build/platform/generic/firmware/`. Notre première brique est prête.

Nous pouvons alors passer au noyau, qui est une version modifiée pour supporter le SoC Rocket Chip et les périphériques provenant de LiteX, et possédant donc une `defconfig` adaptée et incluant le support SATA :

```
$ cd linux
$ make ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu- \
  litex_rocket_defconfig
$ make -j14 ARCH=riscv CROSS_COMPILE=riscv64-linux-gnu-
$ cd ..
```

Nous obtenons ainsi le binaire `arch/riscv/boot/Image` sans aucun problème et pouvons passer à BusyBox qui, lui aussi, dispose d'une configuration toute prête :

```
$ cd busybox-1.36.1/
$ cp ../conf/busybox-1.36.1-rv64gc.config .config
$ make -j14 CROSS_COMPILE=riscv64-linux-gnu-
$ cd ..
```

Pour rappel, BusyBox est un binaire (`busybox`) intégrant les fonctionnalités d'un ensemble d'outils et commandes typiques d'un système Unix, réagissant de manière différente en fonction du nom de fichier (ou lien symbolique) qui lui est donné (`argv[0]`). Le seul élément qui nous intéresse ici est donc ce binaire, directement présent dans le répertoire courant après compilation. Celui-ci nous permet directement de construire le *RAMdisk* compressé avec :

```
$ mkdir initramfs
$ cd initramfs/
$ mkdir -p bin sbin lib etc dev home proc sys tmp \
  mnt nfs root usr/bin usr/sbin usr/lib
$ cp ../busybox-1.36.1/busybox bin/
$ ln -s bin/busybox ./init
$ cat > etc/inittab
::sysinit:/bin/busybox mount -t proc proc /proc
::sysinit:/bin/busybox mount -t devtmpfs devtmpfs /dev
::sysinit:/bin/busybox mount -t tmpfs tmpfs /tmp
::sysinit:/bin/busybox mount -t sysfs sysfs /sys
::sysinit:/bin/busybox --install -s
/dev/console::sysinit:-/bin/ash
^D
$ fakeroot
# find . | cpio -H newc -o > ../initramfs.cpio
2422 blocs
^D
$ cd ..
$ gzip -c initramfs.cpio > initrd
```

Tout ceci est largement et honteusement inspiré du `scripts/build_software.sh` présent dans le dépôt Git et se résume à la création d'une arborescence classique d'un système Linux et de quelques éléments de configuration, le tout rassemblé dans une archive CPIO compressée avec Gzip et prenant place dans le fichier `initrd`. Notez l'appel à `busybox --install -s` créant automatiquement les quelque 150 liens symboliques vers `busybox` nous donnant accès aux commandes courantes, telles qu'activées dans la configuration BusyBox.

Il ne nous reste maintenant qu'à utiliser ces trois éléments binaires, *bootloader*+DTB, noyau et *RAMdisk*, pour créer notre JSON listant les fichiers à charger en mémoire :

```
$ mkdir images
$ cp opensbi/build/platform/generic/firmware/fw_jump.bin images/
$ cp linux/arch/riscv/boot/Image images/
```

```
$ cp initrd images/
cat images/boot.json
{
    "initrd":      "0x82000000",
    "Image":       "0x80200000",
    "fw_jump.bin": "0x80000000"
}
$ cd ..
```

Et enfin, comme avec le *softcore* VexRiscv, nous pouvons copier le contenu de `images/` sur le SSD pour obtenir un démarrage automatique. Après reconnexion du SSD et alimentation de la carte, nous voyons effectivement le système démarrer et détecter le disque. Nous arrivons directement sur un shell root et pouvons utiliser quelques commandes pour explorer ce système très basique :

```
[...]
--=============== SoC ==================--
CPU:            RocketRV64[imac] @ 75MHz
BUS:            wishbone 32-bit @ 4GiB
CSR:            32-bit data
ROM:            128.0KiB
SRAM:           8.0KiB
SDRAM:          1.0GiB 16-bit @ 600MT/s (CL-7 CWL-5)
MAIN-RAM:       1.0GiB

[...]

[    3.629705] litesata 12003000.litesata:
       250059350016 bytes; PNY 250GB SATA SSD
[    3.652889]  litesata: litesata1 litesata2
[...]
[    4.469545] Run /init as init process
[    4.472130]   with arguments:
[    4.475073]     /init
[    4.477331]   with environment:
[    4.480763]     HOME=/
[    4.482797]     TERM=linux
#

# uname -a
Linux litex 6.12.0-rc2-ga11f53dc13c1-dirty #17 SMP
Thu Oct 17 11:10:20 CEST 2024 riscv64 GNU/Linux

# cat /proc/cpuinfo
processor       : 0
hart            : 0
isa             : rv64imafdc_zicntr_zicsr_zifencei_zihpm_zca_zcd
```

```
mmu             : sv39
uarch           : sifive,rocket0
mvendorid       : 0x0
marchid         : 0x1
mimpid          : 0x20181004
hart isa        : rv64imafdc_zicntr_zicsr_zifencei_zihpm_zca_zcd
[...]
processor       : 3
hart            : 3
isa             : rv64imafdc_zicntr_zicsr_zifencei_zihpm_zca_zcd
mmu             : sv39
uarch           : sifive,rocket0
mvendorid       : 0x0
marchid         : 0x1
mimpid          : 0x20181004
hart isa        : rv64imafdc_zicntr_zicsr_zifencei_zihpm_zca_zcd

# ifconfig -a
eth0      Link encap:Ethernet  HWaddr D6:E9:EA:B7:04:40
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
          Interrupt:14

[...]

# mkdir /mnt/plop
# mount /dev/litesata1 /mnt/plop
# df -h
Filesystem      Size      Used Available Use% Mounted on
devtmpfs        494.5M       0    494.5M   0% /dev
tmpfs           496.0M       0    496.0M   0% /tmp
/dev/litesata1  127.7M   73.1M     54.6M  57% /mnt/plop
# umount /mnt/plop
```

Nous avons donc un ensemble fonctionnel ayant accès au SSD. Tout n'est cependant pas parfait, puisque nous avons là un système réduit à son strict minimum. Pour utiliser l'interface Ethernet par exemple, via `udhcpc`, il convient de tout d'abord rendre actif `eth0` avec `ifconfig up` et, même là, l'interface ne « prend » pas l'IP et `route` affiche clairement que rien n'a été configuré. Tout ceci est parfaitement normal, et absolument pas lié à LiteX ou aux pilotes noyau, nous avons là un système qu'on pourrait qualifier de largement incomplet, ce n'est qu'un noyau avec un `init` lançant un shell, rien de plus. Mais c'est une excellente base de départ pour construire tout le reste à la main (ou éventuellement basculer sur Buildroot), ce qui ne peut que constituer une formidable piqûre de rappel (vous avez dit « LFS » ?).

6. POUR FINIR

Je n'irai pas plus loin dans cet article déjà énorme, si ce n'est en vous disant que, oui, il est parfaitement possible de démarrer réellement sur le SSD. Pour cela, tout ce que vous avez à faire est de modifier le DTS pour changer `root=/dev/ram0` en `root=/dev/litesata2`. Il faudra ensuite peupler le système de fichiers de la seconde partition du SSD (on ne l'a pas créé par hasard en ext2) avec peu ou prou la même chose que ce qui se trouve dans le *RAMdisk*. L'approche que j'ai adoptée personnellement a été de démarrer le système sans `init`, mais avec un lien symbolique entre `bin/busybox` et `bin/sh` me donnant un shell « brut » du *boot*. On peut alors monter le nécessaire, remonter `/` en lecture/écriture et ajouter les commandes de BusyBox :

```
# /bin/busybox mount -t proc proc /proc
# /bin/busybox mount -t tmpfs tmpfs /tmp
# /bin/busybox mount -t sysfs sysfs /sys
# /bin/busybox mount -o remount,rw /dev/litesata2 /
# /bin/busybox --install -s
```

On en profitera pour créer un fichier `/etc/inittab` sensiblement différent :

```
::sysinit:/bin/busybox mount -t proc proc /proc
::sysinit:/bin/busybox mount -t tmpfs tmpfs /tmp
::sysinit:/bin/busybox mount -t sysfs sysfs /sys
::sysinit:/bin/busybox mount -o remount,rw /dev/litesata2 /
/dev/console::sysinit:-/bin/ash
```

Ceci nous donnera un système avec un *rootfs* sur SSD et donc une solide base pour étoffer à volonté ensuite. Notez qu'il vaut mieux être prudent, car avec un stockage en lecture/écriture, des redémarrages intempestifs seront probablement synonymes de corruption de données. `mount -o remount,ro /dev/litesata2 /` est votre ami...

Je terminerai en disant que je suis absolument bluffé et émerveillé par ce que permet de faire LiteX. Le travail réalisé par Florent Kermarrec et les autres contributeurs/développeurs de LiteX (dont Gabriel L. Somlo pour la partie Linux sur Rocket Chip) est absolument phénoménal, sinon titanesque. Cette carte va rester un bon bout de temps sur mon bureau et sera abusivement exploitée, comme système GNU/Linux RISC-V bien sûr, mais également plein d'autres choses. Après tout, c'est un *devkit* FPGA et cela a donc une signification très claire : il peut devenir absolument tout et n'importe quoi ! **DB**

RÉFÉRENCES

[1] **https://connect.ed-diamond.com/hackable/hk-057/fpga-facile-petite-presentation-et-prise-en-main-de-litex**

[2] **https://github.com/enjoy-digital/litex**

[3] **https://yosyshq.readthedocs.io/projects/yosys/en/stable/cmd/synth_lattice.html**

[4] **https://connect.ed-diamond.com/Hackable/hk-032/des-kits-de-developpement-fpga-a-moins-de-30-eus**

[5] **https://connect.ed-diamond.com/Hackable/hk-035/une-carte-pilote-de-led-rgb-hackee-en-kit-de-developpement-fpga-a-bas-cout**

[6] **https://connect.ed-diamond.com/contenu-premium/installez-un-environnement-open-source-pour-le-developpement-fpga-sur-rpi**

[7] **https://github.com/RHSResearchLLC/NiteFury-and-LiteFury**

[8] **https://fr.aliexpress.com/item/1005006844453359.html**

[9] **https://enjoy-digital-shop.myshopify.com/products/litex-acorn-baseboard-mini**

[10] **https://enjoy-digital-shop.myshopify.com/products/litex-acorn-baseboard-mini-sqrl-acorn-cle215**

[11] **https://www.amazon.fr/gp/product/B091BRJSRN**

[12] **https://www.amazon.fr/dp/B01L6PQ6I2**

[13] **https://github.com/enjoy-digital/litex-acorn-baseboard**

[14] **https://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-6.pdf**

[15] **https://www.amazon.fr/dp/B07XZLW68F**

[16] **https://connect.ed-diamond.com/open-silicium/os-010/creer-une-configuration-buildroot-sur-mesure**

[17] **https://connect.ed-diamond.com/open-silicium/os-011/personnaliser-buildroot-precisions-et-complements**

[18] **https://connect.ed-diamond.com/hackable/hk-040/chisel-construire-du-materiel-en-langage-scala**

LE SEUL ÉVÈNEMENT
100% SÉCURISÉ

WWW.IT-AND-CYBERSECURITY-MEETINGS.FR

18, 19 & 20 MARS 2025

PALAIS DES FESTIVALS ET DES CONGRÈS DE CANNES

IT AND CYBERSECURITY MEETINGS FRANCE

one to one Meetings Exhibition
by Weyou Group

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

ILS SONT DÉJÀ INSCRITS

6cure, ACKnowledge, Adaptive Shield, adista, AIS, Akamai, ALL4TEC, allentis, Anozr Way, ARMIS, APIXIT, Avantdecliquer, betaeasterns, betcobe, BeyondTrust, Bitdefender, BITSIGHT, BlueFiles, BlueTrusty, board of cyber, bouygues entreprises, CATO, cegedim, centhys, CLOUDFLARE, conscio, Contrast, CROWDSTRIKE, CRYPT.ON IT, CUSTOCY, Cyber Guru, CYBERARK, cybereason, cyberwatch, CYBLEX, DATACORE, dbSQWare, Delinea, DELL Technologies, devensys, DIFENSO, DIGITAL REALTY, digital.ar, dust MOBILE, EASYVISTA, EATON, efficient IP, elastic, elba, EPOS, ERCOM, eurofiber, EV Observe, EXCUBE, F24, Filigran, FORTINET, GATEWATCHER, GitGuardian, HarfangLab, HCLSoftware, HEXANET, HOLISEUM, IC CONSULT, Ilex IAM Platform Inetum, infinigate, infoblox, interdata, ITS Ibelem, Jabra GN, Kamaé, KDDI France, KEEPER, koesio, Linkt, LOGPOINT, LogRhythm, Lutessa, Maltem, ManageEngine, Mantra, merox, METSYS, MIEL paloalto, mimecast, mindflow, monacodigitalgroup, my memority, nameshield, NEOTRUST, NeoVAD, NetApp, netskope, netwrix, ninjaOne, nomios, NOZOMI NETWORKS, Oeil pour Oeil, OGO, okta, Olfeo, ONE IDENTITY, codrive, opentext Cybersecurity, Cyberdefense, OVERSOC, OWN, Panasonic CONNECT, PatrOwl, Ping Identity, pradeo, PRIMX, proofpoint, PURESTORAGE, Qualys, radware, RAPID7, REEL iT, Riot, rubrik, rubycat, SailPoint, sea tpi, sewan, SFR BUSINESS, Sharekey, SILVERFORT, SKYBOX, SNS SECURITY, SOPHOS, sosafe, SPIE, splunk>, STORMSHIELD, SUSE, synetis, sysdig, TANIUM, EDF, TENTRIS, TELEHOUSE, tenable, Tenacy, THEGREENBOW, TREND, TrustBuilder, tufin, TYREX, UBIKA le nouveau DenyAll, vade, VARONIS, VEEAM, VERACODE, VirtualBrowser, WITH secure, WIZ, xmco, ZeroTrust, Zerto a Hewlett Packard Enterprise company, zpe

Liste des exposants inscrits arrêtée au 12/11/2024

BY weyou
Professional Exhibitions
one to one Meetings Exhibitions