ÉLECTRONIQUE | EMBARQUÉ | RADIO | IOT

HACKABLE

L'EMBARQUÉ À SA SOURCE

N° 57
NOVEMBRE / DÉCEMBRE 2024

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

L 19338 - 57 - F: 14,90 € - RD

CPPAP : K92470

RASPBERRY PI / DÉCENTRALISATION / PEER-TO-PEER

Comment échapper à la surveillance et protéger votre vie privée ?

ANONYMISEZ ET SÉCURISEZ VOS CONNEXIONS ...AVEC I2P p.38

- Comprenez les limitations des VPN
- Rejoignez l'Internet invisible
- Transformez votre RPi en routeur I2P
- Créez des tunnels entre vos machines

ÉDITO

Ils l'ont fait !

Au moment où je rédige ceci, le Super Heavy Booster de Starship vient tout juste de se poser délicatement sur Mechazilla, la tour de lancement équipée de « bras » et située à Starbase, la plateforme de lancement de SpaceX à Boca Chica, Texas. Voir ce moment réellement historique et presque irréel, en direct, est un événement en soi. C'est être témoin de la progression technologique humaine, qui, on ne peut le nier, ne cesse de s'accélérer.

L'impact de ces images est évident, même pour une personne n'ayant aucune affinité ou connaissance dans le domaine spatial, aéronautique ou technologique de manière générale. Mais elles sont encore plus impressionnantes lorsqu'on se doute du nombre incroyable de problématiques à régler pour arriver à un tel résultat. Je parle, bien évidemment, d'un point de vue électronique et informatique. La masse de calculs nécessaires et la précision à atteindre dépassent presque l'entendement. Imaginez un instant la tâche à accomplir : poser un cylindre de 70 mètres de haut et de 9 mètres de diamètre en le positionnant à l'aide de trois moteurs Raptor (sur les 33 utilisés au lancement), montés sur un cardan en assurant l'orientation. Le tout après une descente de quelque 70 kilomètres avec un pic de vitesse à 4000 km/h.

C'est un exploit d'ingénierie, bien sûr, tout autant que scientifique, mathématique, physique... Mais pour moi, c'est aussi, et surtout, une démonstration de ce que permettent de faire les technologies informatiques, électroniques et embarquées aujourd'hui.

Je ne sais plus quel scientifique (peut-être Carl Sagan) s'est vu poser la question de savoir si, du fait d'avoir la connaissance nécessaire à comprendre le mécanisme de formation des arcs-en-ciel, cela rendait le phénomène moins appréciable, moins magique. Chose à quoi il a répondu que non, bien au contraire, le fait de savoir, de comprendre, ne fait que rendre le moment encore plus unique et merveilleux.

Il en va de même ici. Tenter d'imaginer, en plus du spectacle époustouflant, le nombre de systèmes, de sous-systèmes, de capteurs, de MEMS et la masse de données à traiter, avec des contraintes physiques énormes, donne le vertige. Et pourtant, pourtant... ils l'ont fait. Et nous, nous vivons tous une époque incroyable...

Denis Bodor

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.

CONFÉRENCE EUROPEAN GNU RADIO DAYS 2024 : ANNONCE DE GNU RADIO 4.0 ET TUTORIEL SUR LES BLOCS DE TRAITEMENT PYTHON

Jean-Michel Friedt

Quelques retours sur la conférence européenne dédiée au logiciel libre de traitement de signaux radiofréquences échantillonnés en temps discret GNU Radio, et le développement de blocs Python dédiés au traitement du signal en flux tendu.

La conférence European GNU Radio Days (Fig. 1), qui vise à regrouper les utilisateurs et développeurs de l'infrastructure libre de traitement numérique de signaux radiofréquences GNU Radio, a vu sa septième édition organisée en Allemagne dans les locaux de l'accélérateur d'ions GSI (*GSI Helmholtzzentrum für Schwerionenforschung*) près de Darmstadt. Ce grand instrument est en effet utilisateur de GNU Radio dans sa gestion des signaux radiofréquences nécessaires à accélérer les ions, et plusieurs centaines de points de mesure à base de systèmes commerciaux accessibles au grand public sont disséminés sur les accélérateurs linéaire et circulaire pour surveiller les instruments qui ne sont pas munis par défaut de systèmes de supervision. Alors que GNU Radio est l'héritage d'un long développement initié en 2001, un développement totalement innovant est engagé comme une potentielle refonte de GNU Radio. Josh Morman en particulier, maintenant président du projet GNU Radio, a participé à la réflexion sur l'architecture des interfaces et du code. La conférence, qui s'est tenue pendant la dernière semaine d'août au GSI/FAIR, fut l'opportunité d'annoncer officiellement le développement de GNU Radio 4.0 et introduire les développeurs aux nouveaux concepts **[1]** proposés dans cette refonte de notre infrastructure favorite de traitement de signaux radiofréquences échantillonnés en temps discret.

Figure 1 : Logo de l'édition 2024 de European GNU Radio Days qui s'est tenue cette année en Allemagne.

1. ASPECTS HISTORIQUES

Après plusieurs mises en contexte de la recherche sur les ions et l'état exotique de la matière dans des situations aussi variées que les premières étapes de l'expansion de l'univers aux étoiles les plus denses, sujets d'étude de GSI **[2, 3]**, la parole fut donnée à Eric Blossom (Fig. 2), créateur de la première version de GNU Radio **[4]**. Son travail, hérité d'un premier travail mené au MIT nommé *pspectra* dont les sources ont maintenant disparu, fut poursuivi sous les auspices bienveillants de John Gilmore, fondateur de l'EFF. Eric insiste sur l'aspect collaboratif du travail qui passe par des relations personnelles autant que sur des aspects techniques, et décrit l'évolution de ses collaborations au cours de développement de GNU Radio et en particulier la nécessité de trouver

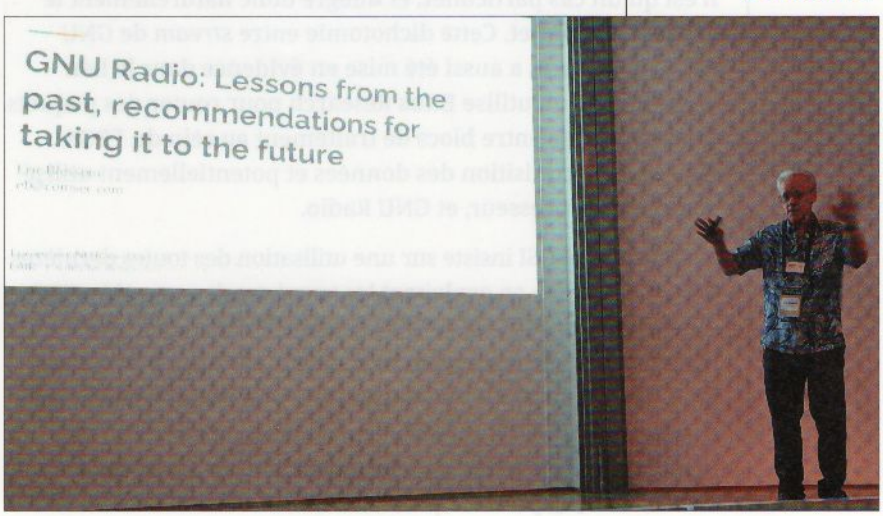


Figure 2 : Eric Blossom, fondateur de GNU Radio, présente l'histoire du projet depuis 2001 et ses recommandations pour les évolutions futures, en insistant sur l'aspect collaboratif et les relations personnelles entre développeurs et utilisateurs.

un financement pérenne – potentiellement étatique, mais aussi de mécénat – pour développer un projet aussi imposant qu'une infrastructure de radio logicielle qu'un bénévole, aussi motivé fût-il, ne pourrait probablement pas mener à terme. Eric est désormais employé par Planet Labs **[5]**, société qui lance des essaims de satellites en orbite basse pour la prise de vues depuis l'espace : la radio logicielle n'a pas encore sa place sur les véhicules spatiaux, mais deux présentations portent sur le prototypage rapide de stations au sol pour la réception de signaux venus de l'espace **[6, 7]**.

Du point de vue de GSI/FAIR, l'intérêt pour la radio logicielle vient de l'ambition de surveiller l'accélérateur en temps réel, notamment par la signature électromagnétique des ions en mouvement, et réagir à des événements. Une dichotomie est mise en évidence entre le principe sous-jacent de la radio logicielle et de GNU Radio en particulier qu'est le flux continu de données acquises par le convertisseur analogique numérique – mode *streaming* – et l'encapsulation d'informations dans des paquets disjoints. La gestion des paquets – PDU pour *Protocol Data Unit* – a été ajoutée de façon quelque peu ad hoc dans GNU Radio et la refonte de GNU Radio 4 vise à corriger une telle déficience. Dans le cas de GSI, un PDU est la détection d'un événement qu'il faut propager le long de la chaîne de traitement. Dans le cas d'une communication numérique, il s'agit de la détection du préambule d'un paquet, par exemple par son mot de synchronisation, qui déclenche le décodage de l'entête qui contient les informations pour ensuite décoder le contenu du message. GNU Radio 4 étend le PDU à tout type de structure, dont la valeur scalaire (entier, réel ou complexe) n'est qu'un cas particulier, et intègre donc naturellement le concept de paquet. Cette dichotomie entre *stream* de GNU Radio et paquets a aussi été mise en évidence dans le lien entre RFNoC qu'utilise Ettus Research pour router des paquets d'informations entre blocs de traitement au sein du FPGA chargé de l'acquisition des données et potentiellement utilisé comme co-processeur, et GNU Radio.

L'équipe de GSI insiste sur une utilisation des toutes dernières moutures du C++ en exploitant les *templates* dans un objectif de vitesse en optimisant le code au travers des blocs individuels de traitement et en visant à tirer le meilleur parti des instructions SIMD par le processeur, et non par une bibliothèque dédiée telle que le propose VOLK actuellement. Cette approche, bien que démontrant d'excellentes performances en termes de temps d'exécution **[1]**, présente pour le moment un inconvénient rédhibitoire : toute la chaîne de traitement doit être recompilée à chaque modification, et compte tenu du nombre d'optimisations possibles en observant ce qui se passe dans l'ensemble de la chaîne et pas juste dans chaque bloc individuel, ainsi que la multiplication des types de variables supportés par types polymorphes (PMT), la compilation s'avère très longue, de plusieurs minutes à plusieurs dizaines de minutes. Cet aspect pourrait passer anodin, si ce n'est que GNU Radio est promu comme un environnement de prototypage rapide et interactif : modifier un paramètre de bloc et relancer le traitement dans GNU Radio Companion est instantané lors de la génération du code Python qui instancie les bibliothèques contenant les méthodes de chaque bloc de traitement. Ce problème devra être réglé pour que GNU Radio 4 soit exploitable : des pistes pourraient venir de LLVM avec Clang qui met moins de la moitié du temps de compilation que GCC pour un exécutable au moins aussi rapide, mais trop long pour considérer le développement de la chaîne de traitement comme interactif. Le lien entre C++ et Python, tel qu'il existe aujourd'hui dans GNU Radio 3.x, fournira une capacité d'intégrer les blocs de traitement sous forme de greffons

précompilés et retrouver l'interactivité actuelle. Les auteurs de GNU Radio 4.0 nous informent que désactiver les diverses optimisations et analyses statiques de code permet d'abaisser le temps de compilation sous la barre des 2 minutes.

Parmi les bénéfices de la refonte du code au cœur de GNU Radio avec la proposition de la version 4 :

- accélération de la gestion des tampons et de la vitesse de calcul en général, notamment par une utilisation efficace de SIMD ;
- transition d'un traitement en flux tendu vers un traitement par paquets ;
- organisation du code pour permettre une reconfiguration par le développeur de la stratégie d'ordonnancement selon divers critères (débit, latences, prise en compte des ressources matérielles). Les auteurs de GNU Radio 4.0 prévoient d'ouvrir un concours pour sélectionner le meilleur ordonnanceur ;
- intégration aisée dans tout matériel, indépendant d'une dépendance à un fournisseur spécifique, et en particulier en visant les applications embarquées, même en l'absence de système d'exploitation ;
- exploitation des dernières évolutions du C++ pour produire un code plus « propre » visant une meilleure accessibilité aux utilisateurs et développeurs de modules hors cœur (*Out Of Tree modules*) ;

et ce, avec une diffusion des codes en suivant la philosophie promue par la FSFE (*Free Software Foundation Europe*) de *Public Money, Public Code* **[8]** par des auteurs financés par un institut de recherche public (Fig. 3).

Les développeurs de GNU Radio 4.0 envisagent de passer en licence LGPL dans l'espoir d'attirer des utilisateurs et donc potentiellement des contributeurs de sociétés privées. Cette ouverture des logiciels libres vers les acteurs privés ne me semble pas souhaitable. En effet, il me semble que cette idée que des sociétés privées – et en particulier du complexe militaro-industriel français – vont contribuer à du logiciel libre me paraît non seulement naïve et contraire aux observations, mais dangereuses pour le logiciel libre puisque la société privée s'approprie les quelques projets qui ont péniblement émergé, pour suffisamment en modifier l'API ou le protocole pour garantir un monopole et une incompatibilité avec la solution originale qui va être noyée dans le marketing faute de ressources d'une communauté de libristes obsédés par la technique, tel que nous le constatons par exemple pour White Rabbit. Une GPLv3 garantit au moins la protection de la FSF (**https://en.wikipedia.org/wiki/Free_Software_Foundation,_Inc._v._Cisco_Systems,_Inc.**), quitte à repousser les contributeurs privés, qui sont de toute façon peu enclins à participer activement à des développements libres. On notera que GSI/FAIR – comme nombre d'accélérateurs de particules et détecteurs distribués de rayons cosmiques – est contributeur et développeur de plateformes White Rabbit dédiées à leur contrôle

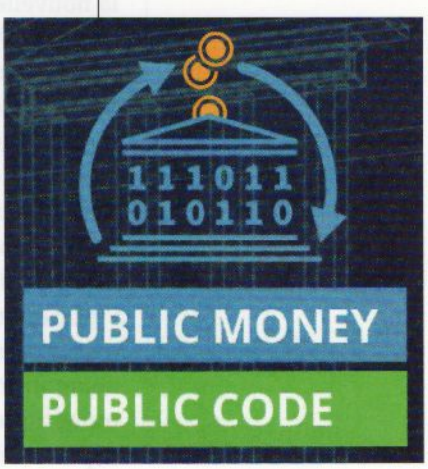


Figure 3 : L'initiative « Public Money, Public Code » de la Free Software Foundation Europe (FSFE) qui encourage les projets financés par de l'argent public à libérer leur code, proposition suivie par bien peu de projets français, mais qui fut au moins discutée en Allemagne... sans beaucoup plus de résultats d'après https://fsfe.org/news/2022/news-20220315-01.en.html, mais avec quelques succès tout de même, si l'on en croit la présentation « Some updates on public code in Germany » au FOSDEM 2024.

Figure 4 : Josh Morman, président de GNU Radio, résume les principales évolutions du projet au cours de l'année et la vision pour les futures évolutions.

de l'accélérateur sous forme de cartes PCIe. Les développeurs autour de Dietrich Beck ont eu la gentillesse de me prêter gracieusement certaines de ces cartes pour que je puisse illustrer le bénéfice mutuel de GNU Radio et White Rabbit pour former un système distribué cohérent et synchrone de génération et d'acquisition de signaux radiofréquences, démontré avec la mesure de direction d'arrivée de signaux issus du RADAR GRAVES et réfléchis par des avions, depuis un site situé à une quarantaine de kilomètres de l'émetteur **[9]**.

Josh Morman, président actuel de GNU Radio, fait un récapitulatif de l'état d'avancement du logiciel **[10]**, avec la promesse de maintenir 3.10 aussi longtemps que nécessaire tant que 4 ne sera pas utilisable (Fig. 4) et que les blocs de traitement actuels n'auront pas été implémentés pour la nouvelle version, puisque rien n'est compatible, de l'API existante aux chaînes décrites pour le moment en YAML dans GNU Radio Companion. GSI dédie actuellement du personnel à ce projet, mais en vue de répondre aux besoins de l'accélérateur : la communauté de développeurs devra mettre la main à la pâte pour obtenir les liens vers Python ou une interface graphique comparable à GNU Radio Companion. En vue d'évaluer l'utilisation de GNU Radio 4.0, l'auteur expérimenté qu'est Daniel Estévez est financé pour démontrer le gain en débit d'un radio-modem numérique sous GNU Radio 4.0 et fait part de son expérience **[11]**.

2. TUTORIELS

Pour entrer dans des considérations plus techniques, l'auteur de cette prose a été sollicité pour introduire GNU Radio 3.10 aux non-développeurs de la conférence, et en particulier les chercheurs et ingénieurs de GSI qui auraient entendu parler du logiciel sans s'en être approprié les concepts et le maniement. Il est intéressant de noter que le changement de paradigme qui semblait avoir repoussé certains de ces utilisateurs est le passage au traitement d'un flux continu de données par des blocs agencés séquentiellement, plutôt que par un programme statique bouclant sur des paquets d'éléments acquis par une interface de conversion analogique numérique. Ainsi, l'objectif était d'attirer des utilisateurs qui connaissent déjà la programmation Python, qui post-traitent déjà des fichiers de données,

mais cette fois les amener à du traitement temps réel bénéficiant de la mise en forme et du prétraitement par une chaîne GNU Radio.

Après une rapide introduction des concepts de base – couleurs associées aux types de données transmises entre blocs de traitement, échantillonnage en temps discret, tension imaginaire et fréquence négative, repliement spectral – nous avons abordé trois types d'interaction de GNU Radio avec Python :

1. L'insertion de quelques lignes de code Python dans la chaîne de traitement produite par GNU Radio Companion.
2. L'insertion d'un code conséquent Python dans le code source produit par GNU Radio Companion.
3. Le traitement du flux IQ par un bloc de traitement écrit en Python.

Ces trois points définissent la séquence des sections qui suivent pour en développer les principes.

2.1 Python Snippet

Le premier cas est le plus simple : nous voulons afficher la longueur d'un vecteur lors de l'exécution d'une chaîne de traitement. Considérons le cas trivial d'un filtre passe-bas (Fig. 5) : ce filtre est caractérisé par une fréquence de coupure et une bande de transition. La fréquence de coupure se comprend bien – la fréquence à laquelle l'amplitude du signal transmis par le filtre commence à décroître – mais qu'est-ce que la bande de transition ? Il s'agit d'un paramètre, que nous noterons δf, qui détermine à quel écart de la fréquence de coupure l'amplitude du signal transmis par le filtre doit avoir baissé. Un utilisateur naïf pourrait se dire que le signal doit passer « instantanément » d'une fonction de transfert passante (amplitude unitaire) à coupante (amplitude nulle). Un tel filtre ne peut exister, ou nécessite un nombre infini de coefficients. De toute façon, GNU Radio Companion nous insulte si nous choisissons une bande de transition nulle en disant `IndexError: firdes check failed: transition_width > 0`. Comment donc choisir δf ?

L'argumentaire est le suivant : si nous définissons une bande de transition δf, il faut que la transformée de Fourier puisse déterminer

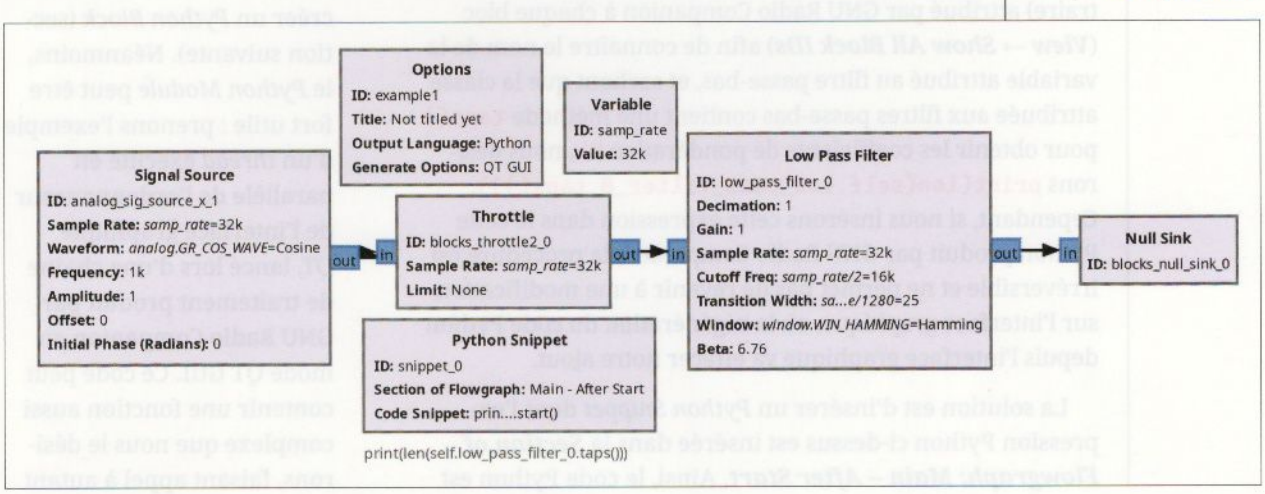


Figure 5 : Exploitation du Python Snippet pour exécuter quelques lignes de code dans un emplacement configurable du code généré par GNU Radio Companion. Ici, nous avons activé la visualisation de l'identifiant de chaque bloc pour connaître le nom du filtre passe-bas et en afficher le nombre de coefficients.

l'amplitude à une fréquence f et une fréquence adjacente f+δ f, donc avoir une résolution spectrale d'au moins δ f. Une transformée de Fourier d'un signal temporel sur N points donne un signal dans le domaine spectral sur N points aussi, dans une gamme de fréquences s'étendant de $-f_s/2$ à $+f_s/2-f_s/N$ avec f_s la fréquence d'échantillonnage en temps discret du signal. Nous approximerons pour cette discussion la borne supérieure à $+f_s/2$ en supposant N grand. Ainsi, la résolution spectrale du spectre est f_s/N, et cette résolution doit être au moins δ f, ou en d'autres termes $N>f_s/\delta f$. Nous voyons que si f_s est petit, N va devenir démesurément grand. Or, l'expression temporelle d'un filtre, défini par son gabarit dans le domaine spectral, est un filtre à réponse impulsionnelle finie (FIR) reliant sortie y avec valeurs passées des entrées x par

$$y_n = \sum_{k=0}^{N-1} b_k \cdot x_{n-k}$$

et il s'agit bien du même N que précédemment par bijection de la transformée de Fourier. Ainsi donc, lorsque nous exprimons un filtre, chaque échantillon de sortie nécessite N multiplications, et on voit bien que choisir N trop grand, ou δ f trop petit, va nécessiter trop de ressources de calcul, même sur un processeur moderne.

Ainsi donc, nous affirmons que GNU Radio choisit N le nombre de coefficients de l'ordre de $f_s/\delta f$, et nous voulons le vérifier expérimentalement.

Pour ce faire (Fig. 5), nous affichons l'identifiant (arbitraire) attribué par GNU Radio Companion à chaque bloc (***View → Show All Block IDs***) afin de connaître le nom de la variable attribué au filtre passe-bas, et sachant que la classe attribuée aux filtres passe-bas contient une méthode `taps()` pour obtenir les coefficients de pondération `bk`, nous désirons `print(len(self.low_pass_filter_0.taps()))`. Cependant, si nous insérons cette expression dans le code Python produit par GNU Radio Companion, la procédure est irréversible et ne permet pas de revenir à une modification sur l'interface graphique, ou la régénération du code Python depuis l'interface graphique va effacer notre ajout.

La solution est d'insérer un *Python Snippet* dont l'expression Python ci-dessus est insérée dans la ***Section of Flowgraph: Main -- After Start***. Ainsi, le code Python est appelé après initialisation des variables, et en choisissant une bande de transition égale à `samp_rate/128` dans le filtre passe-bas, nous sommes informés que le nombre de coefficients est 309. Si nous divisons par 10 la bande de transition pour choisir `samp_rate/1280`, nous constatons que le nombre de coefficients croît bien à 3083 donc d'un facteur 10, la théorie est vérifiée.

2.2 Python Module

Il est fondamental de comprendre que les *Python Snippets* et *Python Modules* ne peuvent pas accéder aux données IQ à traiter, mais ne peuvent qu'agir sur le comportement des blocs de traitement et les variables qui en définissent le comportement. Afin de pouvoir interagir avec les données radiofréquences, il faut créer un *Python Block* (section suivante). Néanmoins, le *Python Module* peut être fort utile : prenons l'exemple d'un *thread* exécuté en parallèle de l'ordonnanceur de l'interface graphique QT, lancé lors d'une chaîne de traitement produit par GNU Radio Companion en mode QT GUI. Ce code peut contenir une fonction aussi complexe que nous le désirons, faisant appel à autant

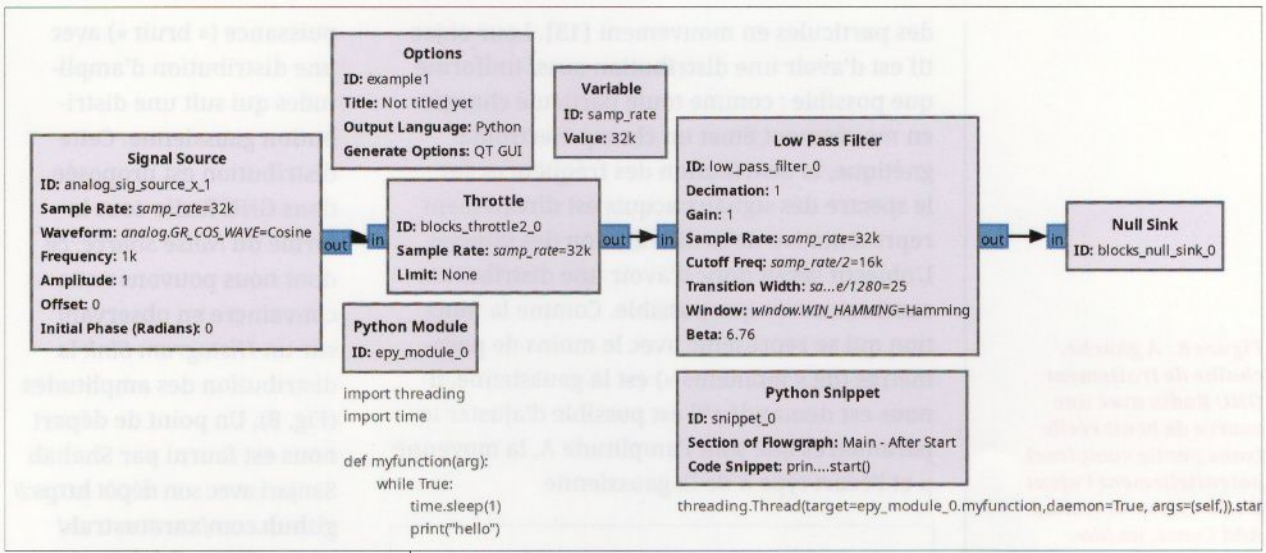


Figure 6 : Un thread séparé de l'ordonnanceur Qt qui se charge de gérer les événements de l'interface graphique, qui se contente ici de simplement afficher un message chaque seconde. Ce thread sera exécuté par un Python Snippet sans lequel le programme contenu dans le Python Module est du code mort. Noter l'option daemon=True au lancement qui garantit la mort du fils lorsque le père achève son exécution.

de classes que nécessaire, et être lancé depuis le *Python Snippet* que nous venons de voir sous forme de *thread* indépendant de l'exécution du père (Fig. 6).

2.3 Python Block

Les utilisateurs au GSI/FAIR (Fig. 7) de GNU Radio analysent la distribution des vitesses des ions dans la boucle de stockage en observant les signaux issus d'antennes placées à proximité

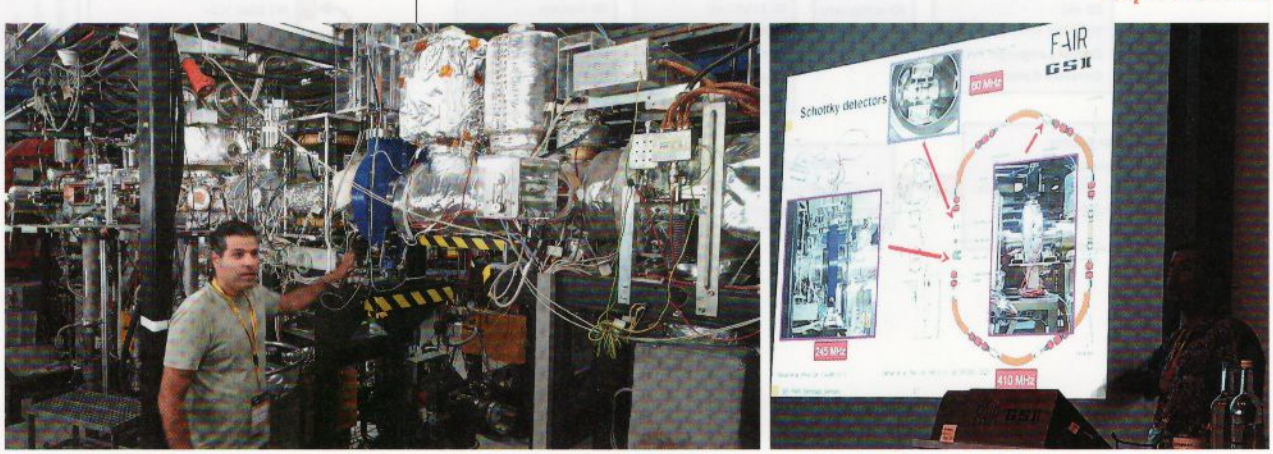


Figure 7 : Shahab Sanjari présente l'anneau de stockage des ions (gauche) dans lequel il observe la durée de vie des divers isotopes et leur fission en divers éléments, et à droite une description du dispositif expérimental dans lequel il exploite GNU Radio pour traiter les signaux acquis par LimeSDR. Notez les fréquences de 60, 245, et 410 MHz [12] parfaitement commensurables avec les plateformes de radio logicielle couramment disponibles. Il est à l'origine de la demande pour un tutoriel sur l'ajustement des paramètres d'une gaussienne sur une courbe expérimentale.

des particules en mouvement **[13]**. Leur objectif est d'avoir une distribution aussi uniforme que possible : comme toute particule chargée en mouvement émet un champ électromagnétique, la distribution des fréquences sur le spectre des signaux acquis est directement représentative de la distribution des vitesses. L'objectif serait donc d'avoir une distribution aussi resserrée que possible. Comme la fonction qui se représente avec le moins de paramètres (de « moments ») est la gaussienne, il nous est demandé s'il est possible d'ajuster les paramètres que sont l'amplitude A, la moyenne μ et l'écart type σ de la gaussienne

$$y = A \cdot \exp\left(\frac{-(x-\mu)^2}{2\sigma^2}\right)$$

connaissant les observations aux abscisses x. Afin de démontrer ce principe, nous produisons une distribution gaussienne pour effectuer nos développements, non pas sur un spectre en fréquences, mais sur les amplitudes d'un bruit caractérisé par AWGN (*Additive White Gaussian Noise*) dans lequel toutes les fréquences sont représentées avec la même puissance (« bruit ») avec une distribution d'amplitudes qui suit une distribution gaussienne. Cette distribution est proposée dans GNU Radio sous la forme du *Noise Source*, ce dont nous pouvons nous convaincre en observant sur un *Histogram Sink* la distribution des amplitudes (Fig. 8). Un point de départ nous est fourni par Shahab Sanjari avec son dépôt **https://github.com/xaratustrah/curve-fitting/blob/main/curve_fit.py** qu'il utilise actuellement en post-traitement et désire insérer dans la chaîne d'analyse en temps réel GNU Radio.

Le tutoriel **https://wiki.gnuradio.org/index.php?title=Creating_Your_First_Block** et le motif (*template*) de bloc

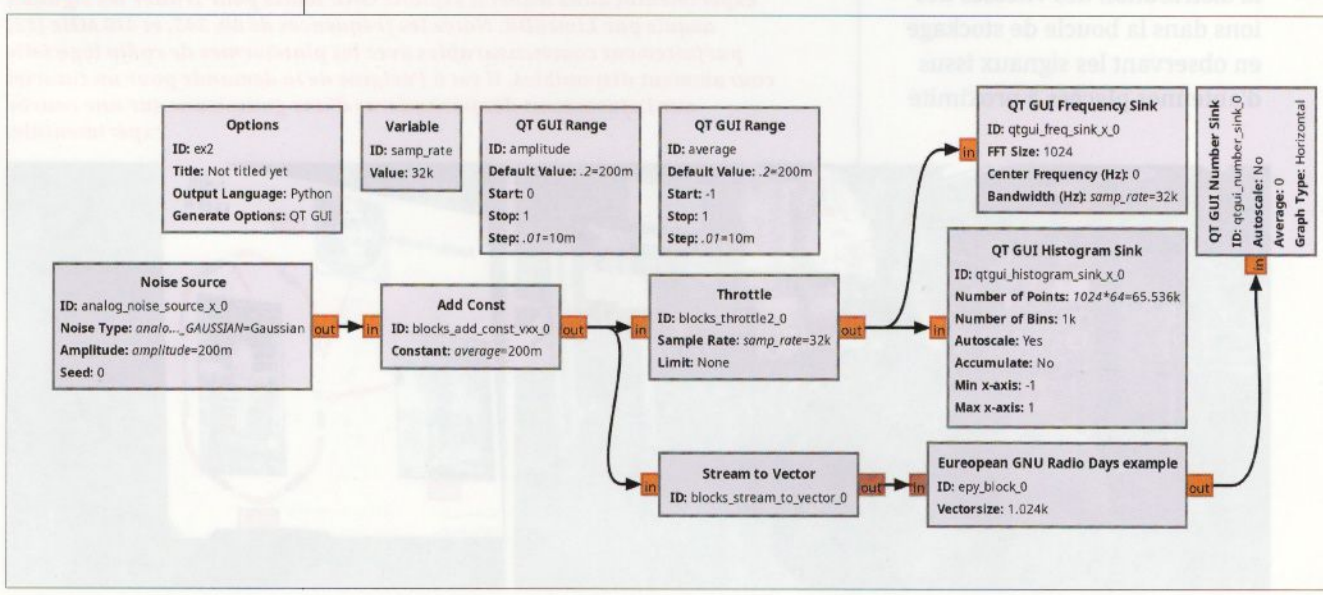


Figure 8 : À gauche, chaîne de traitement GNU Radio avec une source de bruit réelle (sans partie complexe), potentiellement l'ajout d'une valeur moyenne Add Const, un bloc de cadencement en l'absence d'horloge physique Throttle, en haut l'affichage du spectre, au milieu de l'histogramme des amplitudes, et en bas le bloc de traitement spécifiquement conçu pour extraire les paramètres de la gaussienne. Droite : distribution en amplitude (haut) et spectrale (bas) de la source de bruit.

de traitement sont limpides et permettent de rapidement commencer le développement, si ce n'est que pour pouvoir tracer un histogramme, il faut avoir accumulé un nombre minimum de points avant de pouvoir remplir les cases correspondant aux différentes valeurs possibles d'amplitudes. Afin de garantir un nombre minimum de points, nous convertissons le flux continu de données (*stream*) en un vecteur de longueur correspondant au nombre de points que nous désirons distribuer dans les cases de l'histogramme. Un second tutoriel **https://wiki.gnuradio.org/index.php?title=Python_Block_with_Vectors** indique comment fournir au bloc des entrées qui ne sont pas scalaires, mais des vecteurs sous la forme de `in_sig=[(np.float32,vectorSize)]`. De cette façon, le flux de données acquis par la fonction `work` exécutée en boucle infinie est une structure à trois dimensions telle que le démontre :

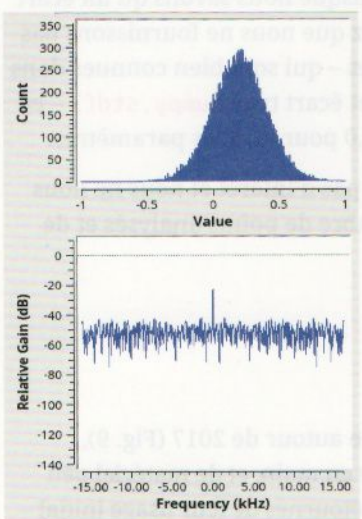
```
print(f"{len(input_items[0])} {input_items[0][0].size} {input_items[0].size}")
```

qui indique 8 1024 8192 qui s'interprète comme :

- `input_items[0]`, car nous avons une entrée. Si le bloc avait plusieurs entrées, cet indice indiquerait de quelle entrée il s'agit ;
- `input_items[0][k]` indique le `k`-ième vecteur, puisque l'ordonnanceur GNU Radio n'a aucune raison de nous fournir un seul vecteur, mais peut avoir accumulé suffisamment de points en amont pour fournir plusieurs vecteurs ;
- `input_items[0][k][m]` est le `m`-ième point du `k`-ième vecteur. Cependant, chaque vecteur contient bien les 1024 points requis par la conversion de `stream` en `vector` en amont de ce bloc dédié.

Une fois cette structure comprise, la suite est limpide. Nous calculons l'histogramme des points en entrée supposés dans l'intervalle [−1, +1] au moyen de Numpy par :

```
y,x=np.histogram(input_items[0][k],bins=32,range=(-1,1))
```



puis effectuons l'ajustement gaussien avec la fonction définie dans `fit_function()` des mesures issues de l'histogramme. Seule subtilité, comme les intervalles et les piquets, le vecteur `x` issu de `np.histogram()` contient les extrémités de chaque classe, donc n+1 valeurs s'il y a n classes. Nous déduisons la valeur moyenne de chaque classe comme `x=0.5*(x[0:-1]+x[1:])` qui a bien n valeurs comme y. Finalement, nous affichons l'amplitude, la valeur moyenne et l'écart type, et renvoyons pour traitement ultérieur ce dernier paramètre qui alimente dans notre chaîne de traitement un QT GUI Number Sink.

```
import numpy as np
from gnuradio import gr
from scipy.optimize import curve_fit

class blk(gr.decim_block):
    def __init__(self, vectorSize=1024):  # default args
        gr.sync_block.__init__(
            self,
            name='European GNU Radio Days example',
            in_sig=[(np.float32,vectorSize)],
            out_sig=[np.float32]
        )

    def fit_function(self, x, A, mu, sigma): # fitting function
        return A*np.exp(-(x-mu)**2/(2.*sigma**2))

    def work(self, input_items, output_items):
        for k in range(0,len(output_items)) :
            y,x=np.histogram(input_items[0][k],bins=32,range=(-1,1))
            x=0.5*(x[0:-1]+x[1:])  # x=middle of each bin
            popt, pcov = curve_fit(self.fit_function, x, y)
            print(f"{popt[0]:.2f}\t{popt[1]:.2f}\t{np.abs(popt[2]):.3f}")
# sigma ** 2 so either positive or negative solutions are acceptable
            output_items[0][k]=np.abs(popt[2])
        return len(output_items[0])
```

Le lecteur qui exécute ce code pourra se convaincre que la valeur moyenne introduite par le bloc ***Add Const*** est bien identifiée dans le second paramètre μ de l'ajustement, tandis que l'« amplitude » telle que l'appelle ce paramètre du bloc source de bruit est bien identifiée par l'écart type. Seule petite subtilité : l'écart type σ n'apparaît que par son carré dans l'équation d'ajustement, et les deux solutions de σ positive ou négative sont parfois atteintes selon la nature du bruit et le cheminement de la descente de gradient. Puisque nous savons qu'un écart type est forcément positif, nous affichons la valeur absolue. Notez que nous ne fournissons pas ici le dernier paramètre optionnel que sont les conditions initiales – qui sont bien connues dans le cas d'une gaussienne comme valeur moyenne `numpy.mean()` et écart type `numpy.std()` – et par défaut le point de départ de la recherche de solution prend 1,0 pour tous les paramètres.

Le dernier paramètre qu'est l'amplitude de l'histogramme n'a pas d'intérêt et nous ne nous attarderons pas dessus – il est principalement dépendant du nombre de points analysés et de leur étalement dans les diverses classes de l'histogramme.

CONCLUSION

La radio logicielle, et GNU Radio en particulier, a vu son apogée autour de 2017 (Fig. 9), combinaison d'un environnement logiciel stable et simple de prise en main, et de matériel peu coûteux – récepteurs de télévision numérique terrestre RTL-SDR détournés de leur usage initial – permettant d'aborder nombre de protocoles de communication radiofréquences.

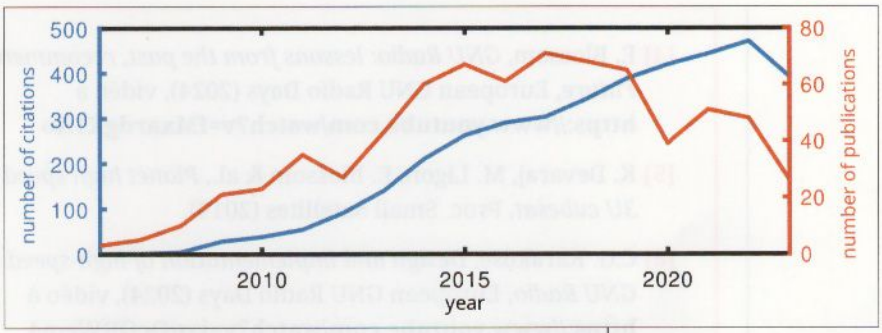


Figure 9 : Nombre de publications (rouge) et de citations (bleu) contenant le terme « GNU Radio » en fonction des années. Source : Web of Science.

Sept ans plus tard, les contributions scientifiques et techniques s'épuisent, probablement parce que l'outil est devenu tellement courant et adopté qu'il ne mérite plus à être mentionné (qui cite l'utilisation d'un oscilloscope dans une mesure électronique ?). Pourtant, le développement de GNU Radio se poursuit, notamment pour tirer parti des dernières évolutions des plateformes de calcul hétérogènes, tentant d'aborder de façon transparente CPU généralistes, GPU et FPGA pour distribuer sur les ressources les plus appropriées les diverses étapes d'une chaîne de traitement. La cinquantaine de participants à la conférence a conclu les échanges sur les utilisations innovantes de la radio logicielle, même pour éventuellement détecter en temps réel un ion rare qu'il faut absolument capturer avant qu'il ne disparaisse. Les vidéos des présentations de la conférence sont disponibles sur le canal European GNU Radio Days sur YouTube à **https://youtube.com/@europeangnuradiodays1445** dans la série « *European GNU Radio Days 2024* » avec leurs transparents comme liens dans le programme à **https://events.gnuradio.org/event/23/**.

REMERCIEMENTS

Cyrille Morin a relu les codes Python pour tenter d'en éliminer les plus grossières maladresses, mais toute erreur résiduelle ne peut être imputée qu'à l'auteur de ces lignes. JMF

RÉFÉRENCES

[1] GNU Radio 4.0 Developer Tutorials à **https://www.youtube.com/playlist?list=PLCfH8xIFcsLlkmVHKLjCr1UaYiMwewiFS**

[2] J. Messchendorp, *Pushing boundaries: femtoscale research with large-scale tech (in accelerator-driven nuclear physics)*, European GNU Radio Days (2024), vidéo à **https://www.youtube.com/watch?v=66kqn-2W6I8**

[3] J. Stadlmann, *... and one to accelerate them all*, European GNU Radio Days (2024), vidéo à **https://www.youtube.com/watch?v=CiaVABVZ71U**

[4] E. Blossom, *GNU Radio: lessons from the past, recommendations for taking it to the Future*, European GNU Radio Days (2024), vidéo à **https://www.youtube.com/watch?v=fMxardgX1Ao**

[5] K. Devaraj, M. Ligon, E. Blossom & al., *Planet high speed radio: Crossing Gbps from a 3U cubesat*, Proc. Small Satellites (2019).

[6] C.G. Karaköse, *Design and implementation of high-speed data link receiver with GNU Radio*, European GNU Radio Days (2024), vidéo à **https://www.youtube.com/watch?v=jxqQcORWem4**

[7] G. Ari Özcan, *Design and Implementation of an Adaptive Data Rate LoRa Modem for LEO Satellites Using SD*, European GNU Radio Days (2024), vidéo à **https://www.youtube.com/watch?v=Jc0aT4i2VaM**

[8] **https://wiki.fsfe.org/Activities/PMPC** et **https://publiccode.eum/**

[9] J.-M Friedt, *Distributed coherent SDR systems: GNU Radio rides the White Rabbit*, European GNU Radio Days (2024), vidéo à **https://www.youtube.com/watch?v=iyUabco0z4A**

[10] J. Morman, *GNU Radio Project Overview and Update*, European GNU Radio Days (2024), vidéo à **https://www.youtube.com/watch?v=89l9_7PitTE**

[11] D. Estévez, *gr4-packet-modem: a QPSK packet modem for GNU Radio 4.0*, European GNU Radio Days (2024), vidéo à **https://www.youtube.com/watch?v=1EPuhaIjxCk**

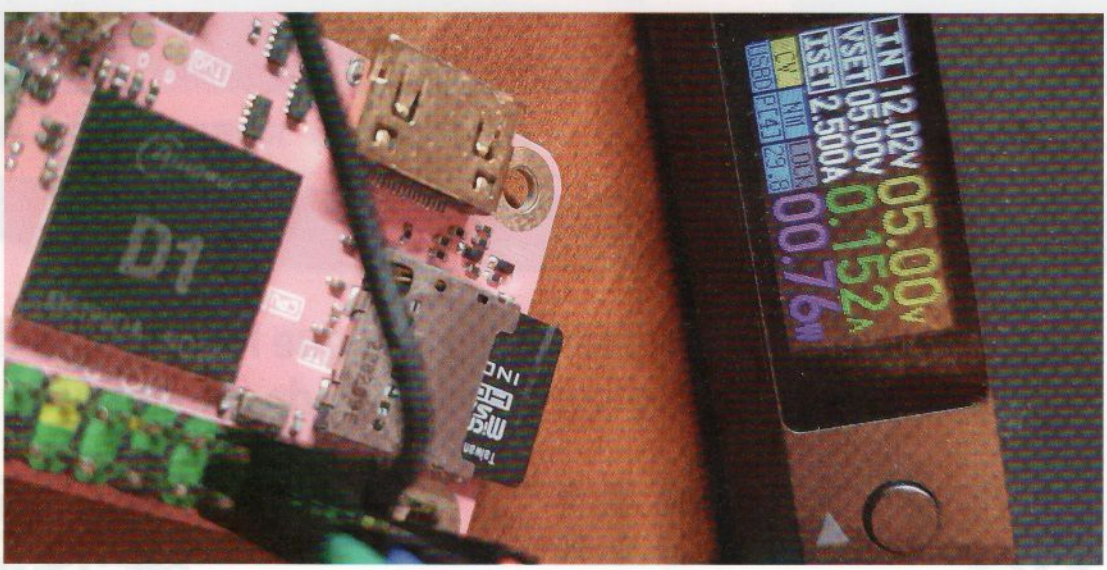
[12] M.S. Sanjari & al., *A 410 MHz resonant cavity pickup for heavy ion storage rings*, Rev. Sci. Instrum. 91(8) 083303 (2020).

[13] P. Kienle, F. Bosch, P. Bühler, T. Faestermann, Yu A. Litvinov, N. Winckler, M. S. Sanjari & al., *High-resolution measurement of the time-modulated orbital electron capture and of the β^- decay of hydrogen-like $^{142}Pm^{60+}$ ions*, Physics Letters B 726 (4-5) 638–645 (2013).

EFFORT MAXIMUM : OPENBSD SUR UNE CARTE RISC-V 1 GHZ/1 GIO À 30 €

Denis Bodor

Dans un précédent édito, je râlais (comment ça, « comme toujours » ?) à propos de l'absence de « leader » dans le monde des SBC RISC-V, qui permettrait, à l'instar de Raspberry Pi, d'avoir au moins une plateforme de référence pour l'ensemble des projets open source. Pour asseoir ma complainte et démontrer cette carence, j'ai décidé d'opter pour une petite séance de masochisme en mode « poisson globe roulé sous les aisselles » en me penchant sur le SBC MangoPi MQ-Pro D1 pour y installer un OS qu'on trouve rarement dans l'embarqué…



– Effort maximum : OpenBSD sur une carte RISC-V 1 GHz/1 Gio à 30 € –

Ce qui est condensé ici est le résultat de trois jours d'essais, de fausses joies, de déception et, au final, d'une bonne dose de dopamine. Pourquoi ? Parce que même si l'on trouve facilement cette carte pour environ 30 € sur AliExpress (~27 € avec 512 Mio de RAM et ~32 avec 1024 Mio), elle est déjà plus ou moins obsolète. Pas techniquement et matériellement bien sûr, surtout vu son prix, mais de fait, elle est clairement « abandonnée ». Le SoC qui est à la base de cette carte est un Allwinner D1 à cœur (au singulier) RISC-V XuanTie C906. Ce nom vous rappelle-t-il quelque chose ? C'est normal, nous l'avons évoqué dans un précédent article sur la minuscule Milk-V Duo **[1]**, intégrant un SoC CVITEK CV1800B comprenant deux de ces processeurs… pour 8 €. Ce n'est pas tout. Très classiquement, le BSP du fondeur du SoC, appelé Tina Linux, est disponible via des dépôts GitHub **[2]**, mais ceux-ci sont « archivés » (en lecture seule) depuis janvier 2024. La page officielle de MangoPi, elle, référence une distribution Armbian de 2022, téléchargeable via un *cloud* chinois, ainsi qu'un dépôt GitHub qui n'a pas bougé depuis des années.

Bref, à ce stade il est clair qu'on est dans le très classique syndrome type *fire'n'forget* : « voici une carte, voilà un système qui fonctionne dessus et nous, on passe à autre chose ».

Pour 30 € cependant, nous avons là un objet très intéressant : une carte au format Raspberry Pi Zero, avec un SoC RISC-V à 1,0 GHz, 1 Gio de RAM, une sortie mini HDMI, un *chip* Wi-Fi/BT (avec antenne), 40 broches avec GPIO, deux ports USB-C (un hôte et un OTG) et l'indispensable emplacement pour microSD. Le rapport ressources/prix est déjà plaisant, mais lorsqu'on ajoute à cela le caractère « exotique » de pouvoir utiliser un « vrai » système (pas un RTOS) sur une architecture RISC-V, c'est difficile de ne pas céder à la tentation. Et, cerise sur le gâteau, cette carte est listée sur la page « riscv64 » du projet OpenBSD **[3]**.

Utiliser une distribution GNU/Linux n'est pas vraiment un challenge ni une démonstration, c'est le système généralement choisi par les constructeurs pour accompagner leurs produits et il n'y a donc pas grand-chose à apprendre au passage, si l'on se contente d'utiliser ce qui existe. Si déjà on sort des sentiers battus en s'éloignant du coutumier ARM, autant se tourner vers un système qui est à la fois sobre, carré et avec une philosophie qui, il n'y a pas si longtemps, était aussi celle strictement appliquée par GNU/Linux (oui, je parle de KISS **[4]**). Et à ces superlatifs s'ajoute également, dans le cas d'OpenBSD, celui de « spartiate », dans le bon sens du terme (avec les abdos, les barbes et le « réalisme historique » de Zack Snyder (sarcasmes)). Cette carte est l'une des 5 listées sur la page, on peut donc supposer que c'est une plateforme choisie pour être une référence, mais comme nous allons le voir, dans le petit monde RISC-V les choses ne sont pas aussi simples.
À noter au passage, FreeBSD propose également un ancien début de support fonctionnel pour le D1, mais là encore, le dépôt **[5]** est archivé depuis début 2024 et le système ne démarre que sur un RAMdisk, pas avec un *rootfs* sur SD.

1. C'EST PARTI !

En jetant un œil aux notes d'installation d'OpenBSD/riscv64 7.5 **[6]**, on se rend rapidement compte que le support est très préliminaire. Plus exactement, l'installation se fait théoriquement en démarrant le système sur une microSD initialisée avec le système du constructeur. L'idée ici est d'utiliser le *bootloader*, ou plus exactement les *bootloaders* déjà présents

pour ensuite démarrer l'installation (chargeur EFI, noyau OpenBSD et installeur) depuis une clé USB et non la microSD, un peu comme démarrer une distribution GNU/Linux *live* depuis GRUB-uEFI sur PC.

Comme avec de nombreuses plateformes embarquées, le démarrage initial est géré par U-Boot, qui fait bien plus que de simplement charger un noyau et lui passer le relais. Dans sa configuration la plus complète, U-Boot sait parfaitement prendre en charge plusieurs types de média (µSM, USB, réseau, etc.) et systèmes de fichiers, tout en permettant l'utilisation de scripts de configuration relativement avancés. Mais sur RISC-V, U-Boot n'arrive que tardivement dans la séquence de *boot* qui prend la forme suivante : *bootcode* en ROM, chargeur (ou SPL), OpenSBI, *bootloader* (U-Boot), noyau OS, etc.

OpenSBI est spécifique aux plateformes RISC-V et est une implémentation de référence *open source* pour la SBI ou *Supervisor Binary Interface* RISC-V. Il s'agit d'une API et d'une couche d'abstraction permettant l'implémentation portable du mode dit « superviseur » (*S-mode*) dans lequel fonctionne le noyau du système d'exploitation ou l'hyperviseur dans un environnement virtualisé (*VS-mode*).

Le mode de plus bas niveau dit « machine » (*M-mode*), autrement dit le mode offrant le maximum de privilèges, est celui dans lequel fonctionne l'implémentation SBI et est désigné par l'acronyme SEE pour *Supervisor Execution Environment*. Cet élément est spécifique à la plateforme, mais offre une interface normalisée pour le système d'exploitation (*S-mode*) qui lui-même contrôle l'exécution des applications en espace utilisateur (*U-mode*). Grossièrement, l'architecture est similaire à la notion de *rings* du monde x86 ou d'*exception levels* (EL3 à EL0) d'ARM. Notez au passage que les EL d'ARM ont un ordre inverse des *rings* x86 avec EL0 pour les applications, EL1 pour le noyau, EL2 pour l'hyperviseur et EL3 pour le mode moniteur (généralement *Trustzone*). RISC-V se débarrasse de cette problématique de classement parfois peu intuitive en utilisant des désignations non numériques.

Le mécanisme d'installation d'OpenBSD part du principe de réutiliser cette chaîne de démarrage jusqu'à U-Boot pour ensuite, via quelques scripts déjà présents dans sa configuration, basculer sur le système d'installation (chargeur EFI + noyau + RAMdisk) mis à disposition via une clé de stockage USB. Sur le papier, c'est très bien, mais on utilise un morceau de système dont nous n'avons pas la maîtrise et, de fait, qui est susceptible de changer en provoquant des problèmes de compatibilité. Et c'est précisément ce qui arrive, car en tentant l'opération avec l'image Armbian disponible pour la MangoPi MQ-Pro, le processus ne fonctionne pas. Quelque chose est cassé entre ce que faisait la chaîne de *boot* avant, et ce qu'elle fait maintenant. Et dieu sait que j'ai testé bien des combinaisons de binaires et bien des déclinaisons de DTB (*Device Tree Blob*, la structure de données décrivant les périphériques non détectables automatiquement pour une plateforme précise).

Dans cette situation, il ne reste qu'une seule solution : ne pas reposer sur du binaire existant et recompiler les éléments de la chaîne de *boot* à la main, pour créer un contenu minimal pour la carte microSD et pouvoir démarrer le processus d'installation...

2. PRÉPARATION DE LA MICROSD

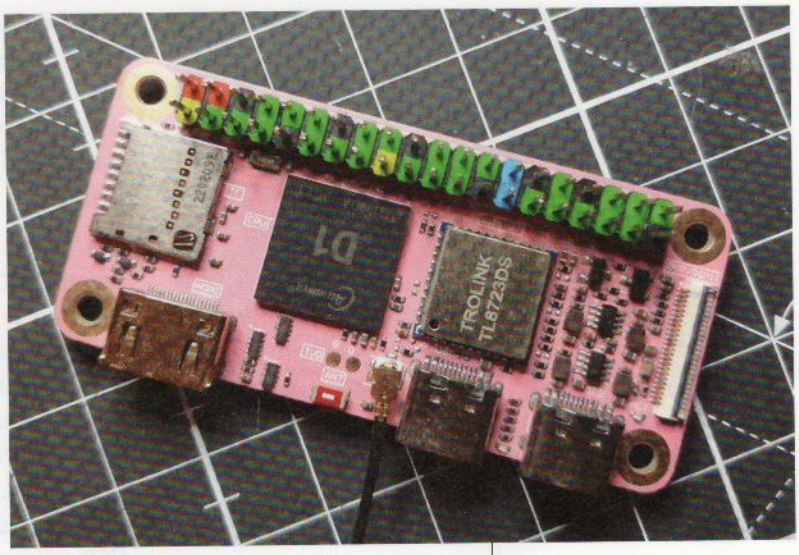
Les spécifications UEFI ou *Unified Extensible Firmware Interface* sont devenues la norme dans le monde x86/amd64/x86_64 et c'est également de plus en plus le cas dans l'embarqué, avec les SBC ARM en particulier.

– Effort maximum : OpenBSD sur une carte RISC-V 1 GHz/1 Gio à 30 € –

Les plateformes RISC-V n'échappent pas au phénomène, car cette approche permet d'avoir une API stable permettant de développer des chargeurs en minimisant la dépendance au matériel. Le système (le noyau) est donc de moins en moins souvent chargé directement par U-Boot, qui se contente de charger un binaire EFI qui lui s'occupe du reste des opérations, exactement comme c'est le cas sur PC.

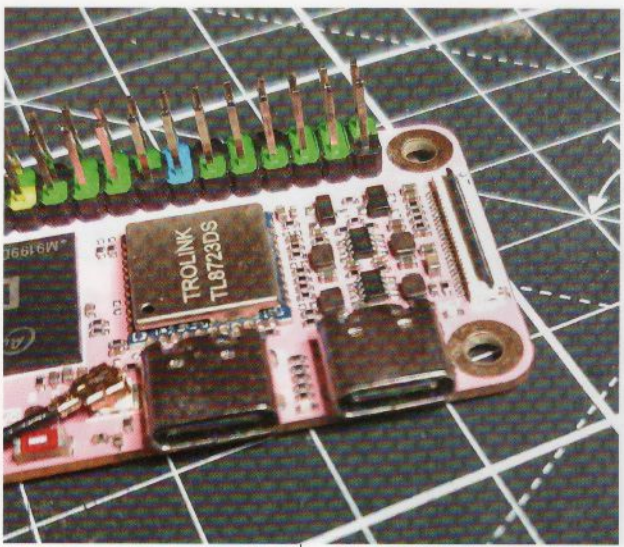
Ce binaire (`bootriscv64.efi` pour RISC-V, `bootaa64.efi` pour AARCH64, `bootx64.efi` sur x86_64, etc.) réside généralement dans une arborescence (`EFI/boot/`) dans un système de fichiers FAT16/FAT32 présent dans une partition de type `0B` (Win95 FAT-32) en MBR ou `EE`/`EF` (EFI GPT / UEFI) en GPT. Ce système de fichiers est présent, quelle que soit l'architecture utilisée, dès lors qu'UEFI est utilisé (y compris sur votre PC). Ceci va beaucoup plus loin que les simples binaires susnommés, avec des choses comme Clover [7] qui me permet de démarrer Devuan GNU/Linux, FreeBSD, NetBSD et OpenBSD en *quadruple boot* sur mon vieux MacBook A1278, ou tout simplement GRUB/EFI, l'implémentation UEFI du célèbre *bootloader*. UEFI permet, techniquement, de développer un OS dont la tâche principale est de *booter* un autre OS.

Pour créer une microSD qui sera, à l'avenir, capable de démarrer automatiquement, nous devons d'ores et déjà préparer ce support en incluant cette partition et ce système de fichiers. Notre microSD jouera deux rôles : permettre le démarrage du système pour l'installation d'OpenBSD en nous donnant accès à l'interface d'U-Boot, puis une fois le système installé, charger ce système sans intervention de notre part.



Le SBC MangoPi MQ-Pro D1 se présente sous le même format que le Raspberry Pi Zero, mais se caractérise par deux points notables, le SoC est basé sur un processeur RISC-V et la carte est violemment rose bonbon ! Quel rapport avec une mangue ? Mystère…

Comprenez bien que le partitionnement de la microSD avant d'installer le/les *bootloader(s)* est une manœuvre pour nous faciliter la vie et simplifier le processus d'installation. Cette préparation des partitions de la microSD, ainsi que les compilations qui vont suivre, sont ici décrites pour OpenBSD, mais sont tout à fait applicables avec un autre système d'exploitation, à condition qu'il s'agisse d'un OS digne de ce nom (un Unix libre, donc). Il suffira de désigner les périphériques de stockage avec leurs bons noms et d'utiliser un compilateur croisé adapté au système (testé avec FreeBSD et le préfixe `riscv64-none-elf-` (paquets `riscv64-none-elf-gcc`/`riscv64-none-elf-binutils`)). La configuration des labels nécessite un système sachant les manipuler, mais ceci peut également être fait durant le processus



Deux ports USB-C sont accessibles sur la carte. Celui de gauche est USB hôte uniquement, mais celui de droite est OTG. Il peut donc jouer le rôle d'hôte ou de périphérique selon la configuration appliquée, comme votre smartphone.

d'installation qui inclut une étape de partitionnement. Créer au minimum la partition pour UEFI au tout début, comme ici, est toutefois la meilleure approche, et cela peut être fait avec GParted sous Raspberry Pi OS, par exemple, sans le moindre problème.

L'image d'installation, `install75.img` fournie via le CDN OpenBSD **[8]**, qui est un système *bootable* (`miniroot75.img`) utilisant un RAMdisk en guise de racine accompagné des *sets* d'installation (également présents dans le même répertoire du site), utilise un partitionnement MBR et non GPT. Pour minimiser le risque d'éventuels problèmes, nous allons utiliser la même chose pour préparer la microSD, glissée dans un lecteur USB connecté à une machine OpenBSD.

Nous commençons par créer une nouvelle table de partition MBR avec :

```
# fdisk -i sd2
Do you wish to write new MBR? [n] y
Writing MBR at offset 0.
```

Nous pouvons ensuite commencer le partitionnement en utilisant le mode interactif de la même commande :

```
# fdisk -e sd2
Enter 'help' for information
```

L'option `-i` nous aura créé automatiquement une partition 3 de type `A6` pour OpenBSD occupant la totalité de l'espace disponible. Ce n'est pas ce que nous voulons, et nous la supprimons avec la commande `e 3`, pour « éditer la partition 3 ». Attribuer un type `00` supprime la partition, et c'est ce que nous faisons.

L'étape suivante consistera à créer la partition destinée à accueillir le système de fichiers FAT pour le binaire EFI. Nous ne pouvons cependant pas créer cette partition n'importe où, car la chaîne de *boot* (SPL + OpenSBI + U-Boot) doit être placée en début de « disque », juste après la table de partition. Notre partition EFI doit donc démarrer assez loin pour ne pas gêner ces éléments. Nous choisissons donc de placer la première partition à quelque 10 Mio du début du support en éditant 0 avec `e 0` :

```
Partition id ('0' to disable) [01 - FF]: [00] (? for help) 0B
Do you wish to edit in CHS mode? [n]
Partition offset [0 - 7744511]: [0] 20480
Partition size [1 - 7724032]: [1] 102400
```

Le type à choisir est `0B` (« Win95 FAT-32 ») avec un décalage de 20480 secteurs de 512 octets (20480*512/1024/1024 = 10 Mio) et une taille de 50 Mio (102400 secteurs). Puisque nous sommes dans l'édition de partitions, autant immédiatement prévoir celle pour le système. Nous en ajoutons donc une nouvelle, type `A6`, occupant le reste du disque :

```
Partition id ('0' to disable) [01 - FF]: [00] (? for help) A6
Do you wish to edit in CHS mode? [n]
Partition offset [0 - 7744511]: [0] 122880
Partition size [1 - 7621632]: [1] 7621632
```

Voici ce que nous obtenons :

```
sd2*: 1> p
Disk: sd2       geometry: 482/255/63 [7744512 Sectors]
Offset: 0       Signature: 0xAA55
        Starting       Ending       LBA Info:
#: id  C   H   S -   C   H   S [  start:    size ]
------------------------------------------------------------------------
0: 0B  1  70   6 -   7 165  30 [  20480:  102400 ] Win95 FAT-32
1: A6  7 165  31 - 482  18  48 [ 122880: 7621632 ] OpenBSD
2: 00  0   0   0 -   0   0   0 [      0:       0 ] Unused
3: 00  0   0   0 -   0   0   0 [      0:       0 ] Unused
```

Et finalement, nous inscrivons tout cela sur le support en quittant avec `q` (ou `w` puis `ex`).

OpenBSD, contrairement à d'autres systèmes de type UNIX (je pense à GNU/Linux), utilise deux notions différentes pour le terme « partition ». Nous avons d'une part ce que nous venons de faire et, de l'autre, les partitions de système de fichiers ou *disklabel partition*, parfois simplement nommée « labels ». Une unique partition MBR (ou GPT) dédiée à OpenBSD peut contenir plusieurs labels permettant d'accueillir les systèmes de fichiers montés par le système. Traditionnellement et par convention, les labels « a », « b » et « c » ont des rôles prédéterminés, respectivement, le *rootfs*, la *swap* et « c » désignant le disque dans son ensemble.

Comme pour les partitions MBR (*slices* dans le jargon FreeBSD), les labels ne sont que des noms référençant des zones du support de stockage. Si nous éditons interactivement les labels avec `disklabel -E`, nous voyons immédiatement :

```
# disklabel -E sd2
Label editor (enter '?' for help at any prompt)
sd2> p
OpenBSD area: 122880-7744512; size: 7621632; free: 7621632
#        size      offset  fstype [fsize bsize   cpg]
  c:   7744512           0  unused
  i:    102400       20480   MSDOS
sd2>
```

c est la totalité du disque et i est notre partition EFI référencée ici sous forme de label. Nous pouvons alors ajouter un label a pour notre système de fichiers racine (nous n'avons pas besoin de *swap*) et enregistrons le tout avec :

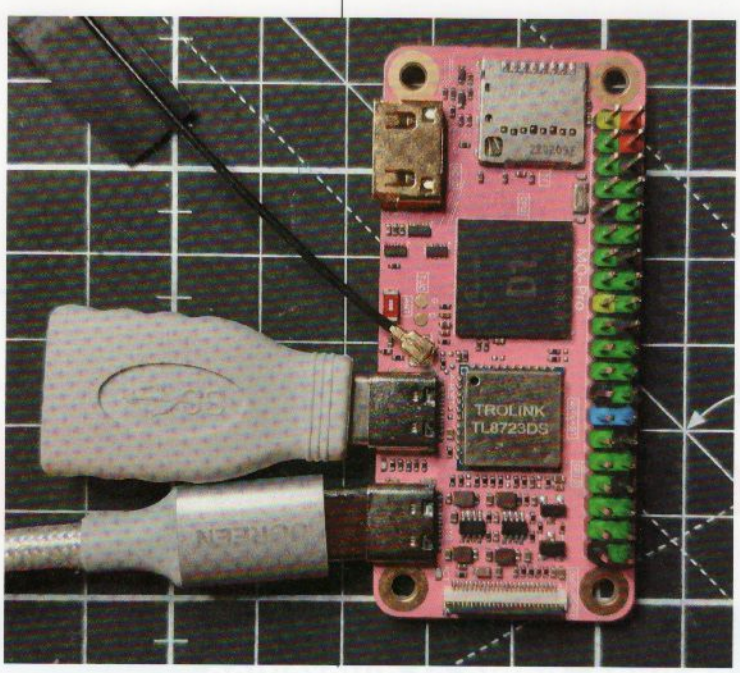
```
sd2> a a
offset: [122880]
size: [7621632]
FS type: [4.2BSD]
sd2*> p
OpenBSD area: 122880-7744512; size: 7621632; free: 0
#         size    offset  fstype [fsize bsize   cpg]
  a:   7621632    122880  4.2BSD   2048 16384     1
  c:   7744512         0  unused
  i:    102400     20480   MSDOS

sd2*> q
Write new label?: [y]
```

Et tant qu'on y est, on en profite pour créer le système de fichiers FAT qui accueillera le ou les binaire(s) EFI plus tard :

```
$ doas newfs_msdos sd2i
/dev/rsd2i: 102164 sectors in
25541 FAT16 clusters (2048 bytes/cluster)
bps=512 spc=4 res=1 nft=2 rde=512 mid=0xf8
spf=100 spt=63 hds=255 hid=20480 bsec=102400
```

Notez la désignation utilisée, sd2i référençant le label « i » du disque SCSI (sd) numéro 2.



La proximité des deux connecteurs USB-C peut être un problème avec certains adaptateurs (ou changeur de genre) USB OTG. On voit clairement ici que forcer la connexion appliquerait une force non négligeable susceptible d'endommager la carte. Heureusement, il est parfaitement possible d'alimenter le SBC directement en +5 V via le connecteur 40 broches.

3. CONSTRUCTION DE LA CHAÎNE DE BOOT

Comme dit précédemment, alors que la documentation OpenBSD recommande d'utiliser une microSD inscrite avec l'image fournie par le constructeur (un système GNU/Linux, donc), nous allons tout reconstruire à la main. On peut supposer que, à un moment, l'approche de la documentation a fonctionné, mais les briques évoluant individuellement sans grande cohérence, quelque chose s'est cassé dans les dépendances. Le processus

de *boot* est bien plus complexe que ce que j'ai décrit en début d'article, avec chaque élément initialisant certaines fonctionnalités et passant des informations à la brique suivante. On retrouve, de-ci de-là, des rapports de *bugs* (ou *issues* sur GitHub) signalant ces incompatibilités et les mêmes symptômes qu'on constate lorsqu'on suit la procédure standard (non-initialisation des périphériques, plantage immédiat du noyau, erreur mémoire, blocage en plein milieu du *boot*, message signalant qu'on n'a pas de RAM (?!), etc.).

Nous compilerons le nécessaire ici exactement comme nous avons partitionné la microSD : sous OpenBSD. Tout ce qu'il nous faut, c'est un compilateur croisé RISC-V et quelques outils habituels (GNU Make, Git, etc.). Ceci s'obtiendra très facilement sous OpenBSD en installant des paquets binaires avec :

```
$ doas pkg_add riscv-elf-binutils riscv-elf-gcc \
riscv-elf-newlib swig gmake git
```

Historiquement, et comme décrit sur le wiki *sunxi* **[9]**, il était nécessaire de construire le SPL (*Secondary Program Loader* exécuté juste après le code en ROM), OpenSBI et U-Boot pour ensuite les combiner et les inscrire directement sur la microSD. Depuis fin 2022 cependant (donc après la diffusion de la distribution du constructeur), tout ceci a évolué et un développeur, Samuel Holland, a *forké* et adapté U-Boot pour ce SoC (entre autres). Le SPL fait maintenant directement partie du mécanisme de construction de son *fork* d'U-Boot qui permet également d'intégrer un binaire OpenSBI pour créer une image unique qu'il suffit de transférer, en une fois, sur le support.

La solution ne sera pas, cependant, de simplement compiler les deux éléments, car dans leur configuration de base, le système ne démarrera pas. Nous commençons donc par récupérer les sources d'OpenSBI depuis le dépôt officiel via Git (branche *master*, commit *bb7267a* du 05/07/2024 ici) :

```
$ mkdir kkpart
$ cd kkpart
$ git clone https://github.com/riscv-software-src/opensbi
```

Les versions récentes d'OpenSBI utilisent une interface de configuration Kconfig (comme Linux, ESP-IDF, U-Boot, etc.) et nous allons ajuster le profil par défaut avec :

```
$ cd opensbi
$ gmake CROSS_COMPILE=riscv64-unknown-elf- \
  PLATFORM=generic menuconfig
```

Dans l'interface qui se présente, allez dans ***Platform Options*** et désactivez le support de tous les SoC, sauf ***Allwinner D1 support***. Dans ***Serial Device Support***, nous pouvons également désactiver la prise en charge des contrôleurs série inutiles, c'est-à-dire tout sauf ***8250 UART FDT driver***. Et enfin, dans ***Interrupt Controller Support***, nous faisons de même pour tous les pilotes, sauf ***Platform Level Interrupt Controller (PLIC) FDT driver***. Ceci fait, quittez l'interface en enregistrant les changements et lancez la construction :

```
$ gmake -j CROSS_COMPILE=riscv64-unknown-elf- \
  PLATFORM=generic FW_PIC=y
```

À l'issue de l'opération, vous trouverez, entre autres, le binaire `fw_dynamic.bin` dans le sous-répertoire `build/platform/generic/firmware/`. C'est la version « dynamique » d'OpenSBI (voir `docs/firmware/*.md`), que nous pourrons utiliser avec U-Boot, que nous construisons ensuite :

```
$ cd ..
$ git clone https://github.com/smaeul/u-boot\
  -b d1-wip u-boot_smaeul
$ gmake CROSS_COMPILE=riscv64-unknown-elf- \
  mangopi_mq_pro_defconfig
```

Nous utilisons ici la branche *d1-wip* de Samuel et la configuration par défaut (`defconfig`) pour la MangoPi MQ-Pro D1. Cependant, comme j'ai rencontré des problèmes avec le *watchdog* (peut-être lié à la lenteur de la microSD utilisée) chargé de redémarrer le système automatiquement s'il se bloque, nous allons le désactiver. Comme avec OpenSBI, une interface Kconfig rend cela très aisé :

```
$ gmake CROSS_COMPILE=riscv64-unknown-elf- \
  menuconfig
```

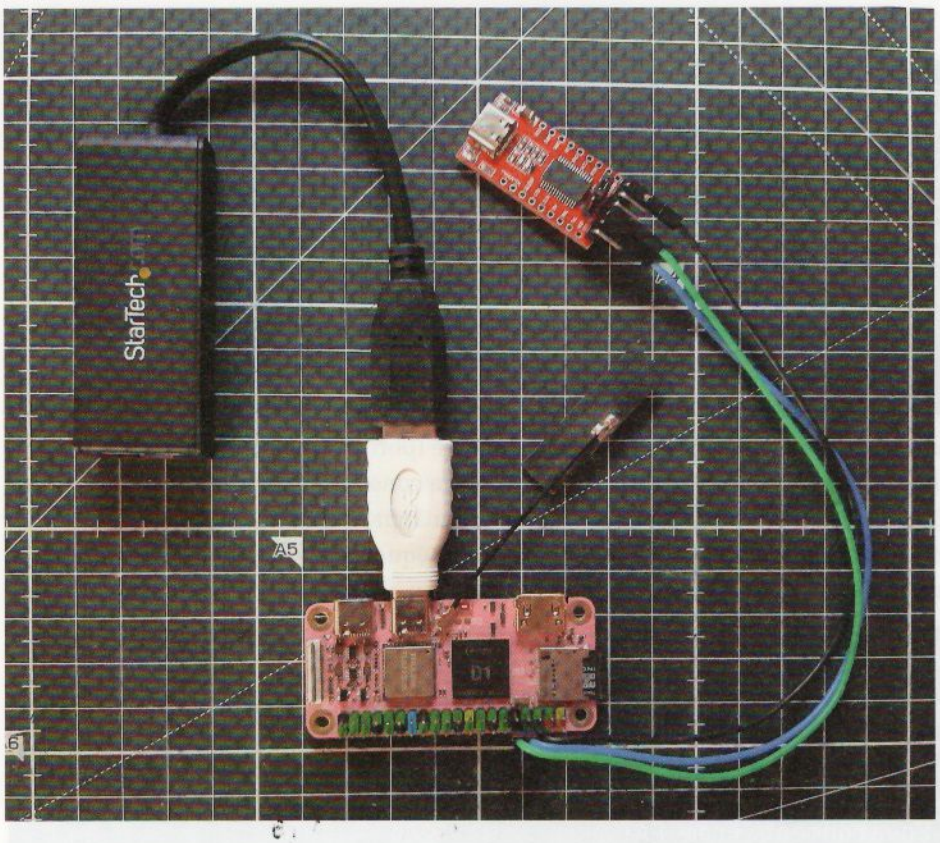
Rendez-vous dans ***Device Drivers*** puis ***Watchdog Timer Support*** et désactivez (décochez) l'option ***Automatically start watchdog timer***. Ainsi, le *watchdog* ne sera pas armé directement au démarrage. Quittez en enregistrant les changements et lancez la construction :

```
$ gmake CROSS_COMPILE=riscv64-unknown-elf- \
OPENSBI=../opensbi/build/platform/generic/\
firmware/fw_dynamic.bin
```

Nous référençons ici directement le binaire OpenSBI sur la ligne de commande et notez que nous nous abstenons également d'utiliser l'option `-j` de GNU Make, permettant la compilation parallèle. Pour une raison qui reste mystérieuse, il semblerait que les avertissements soient considérés comme des erreurs en parallélisant, du moins sous OpenBSD (le problème ne se pose pas sous FreeBSD). Quoi qu'il en soit, nous obtenons le binaire `boot-sunxi-with-spl.bin` qui est un regroupement des trois éléments, SPL + OpenSBI + U-Boot, que nous pouvons inscrire immédiatement sur le support de stockage avec :

```
$ doas dd if=u-boot-sunxi-with-spl.bin \
  of=/dev/sd2c bs=1024 seek=8
939+1 records in
939+1 records out
962077 bytes transferred in 0.730 secs
  (1316892 bytes/sec)
```

– Effort maximum : OpenBSD sur une carte RISC-V 1 GHz/1 Gio à 30 € –



Parmi les caractéristiques notables du MangoPi MQ-Pro D1, on remarquera un connecteur mini HDMI, un module Wi-Fi/BT RTL8723ds, un connecteur U.FL/UMCC pour une antenne Wi-Fi/BT (livrée avec la carte), un connecteur DVP/RGMII pour l'ajout d'une interface Ethernet et vidéo et, à l'arrière, un connecteur DSI/CTP/LVDS pour une caméra ou un écran. La plupart de ces fonctionnalités ne disposent pas encore d'un support pour OpenBSD.

Les données sont enregistrées à 8 Kio du début du support (pour ne pas écraser le MBR) et font moins de 1 Mio. Nous avons prévu une « marge » de 10 Mio, la partition EFI n'est donc pas impactée. Nous pouvons passer au démarrage de notre adorable carte toute rose…

4. DÉMARRAGE DE L'INSTALLEUR

Après toutes ces étapes, nous revenons au processus décrit dans la documentation officielle : nous allons utiliser la chaîne de *boot* pour manuellement démarrer un système se trouvant, non pas sur la microSD, mais sur une clé USB. Un adaptateur USB OTG (C vers A femelle) sera nécessaire, ainsi qu'un adaptateur USB/série 3,3 V pour la console (broches 8-TX0 et 10-RX0, plus masse (9)). Étant donnée la proximité des deux connecteurs USB-C, il est parfois difficile de connecter l'alimentation et l'adaptateur OTG sans risquer d'endommager la carte. J'ai donc préféré, finalement, alimenter la Mango en 5 volts via les broches 2 ou 4 (VCCIN) et une masse (6), plutôt que via USB.

La clé USB devra être initialisée avec l'image d'installation `install75.img` et non `miniroot75.img`, car étant donné que nous n'avons pas de réseau dans l'immédiat, nous devons disposer des *sets* (composants d'installation) sur un support de stockage. On inscrira l'image avec :

```
$ cd kkpart
$ ftp https://cdn.openbsd.org/pub/OpenBSD/\
7.5/riscv64/install75.img
$ ftp https://cdn.openbsd.org/pub/OpenBSD/\
7.5/riscv64/SHA256.sig
$ signify -C -p /etc/signify/openbsd-75-base.pub
  -x SHA256.sig install75.img
Signature Verified
install75.img: OK

$ doas dd if=install75.img of=/dev/rsd2c bs=1m
```

On connectera ensuite la clé via l'adaptateur sur la carte (port USB-C « HOST »), placera la microSD dans le support et connectera l'adaptateur pour la console (115200 8N1) avec Minicom, par exemple. On mettra alors sous tension la carte et verra apparaître les premiers messages de *boot*. Là, vous avez deux secondes pour interrompre la séquence au moment où U-Boot est lancé, en appuyant sur une touche :

```
U-Boot 2024.01-rc1-45338-g2e89b706f5
(Jul 26 2024 - 15:13:00 +0200) Allwinner Technology
DRAM:  1 GiB
Core:  48 devices, 20 uclasses, devicetree: separate
WDT:   Not starting watchdog@6011000
MMC:   mmc@4020000: 0, mmc@4021000: 1
Loading Environment from FAT...
  Unable to read "uboot.env" from mmc0:1...
In:    serial@2500000
Out:   serial@2500000
Err:   serial@2500000
Net:   No ethernet found.
starting USB...
Bus usb@4200000: USB EHCI 1.00
Bus usb@4200400: USB OHCI 1.0
scanning bus usb@4200000 for devices...
  2 USB Device(s) found
scanning bus usb@4200400 for devices...
  1 USB Device(s) found
       scanning usb for storage devices...
         1 Storage Device(s) found
Hit any key to stop autoboot:  0
=>
```

Notez que U-Boot détecte bien la présence du stockage USB. À ce stade et parce que nous n'utilisons pas la chaîne de *boot* d'origine, nous pouvons en profiter pour modifier ce délai très court de 2 secondes et enregistrer la modification :

```
=> setenv bootdelay 5
=> saveenv
Saving Environment to FAT... OK
=>
```

Cette fonctionnalité qui n'est souvent pas présente par défaut nous permet d'enregistrer l'environnement U-Boot dans un fichier `uboot.env` sur la partie FAT de la microSD et ainsi conserver une configuration modifiée d'un démarrage à l'autre. Pour procéder à l'installation, nous n'avons ensuite plus qu'à dire à U-Boot de démarrer sur le support USB. Les scripts intégrés (voir la sortie de la commande `printenv`) se chargent de scanner le périphérique, trouver les éléments nécessaires, charger le binaire EFI et lui passer la main. Tout ceci avec un simple :

```
=> run bootcmd_usb0
[...]
Scanning usb 0:1...
Card did not respond to voltage select! : -110
No EFI system partition
No EFI system partition
Failed to persist EFI variables
BootOrder not defined
EFI boot manager: Cannot load any image
Found EFI removable media binary efi/boot/bootriscv64.efi
148476 bytes read in 13 ms (10.9 MiB/s)
Booting /efi\boot\bootriscv64.efi
disks: sd0* sd1
>> OpenBSD/riscv64 BOOTRISCV64 1.5
boot>
[...]
OpenBSD 7.5 (RAMDISK) #0: Fri Mar 22 19:50:20 MDT 2024
    deraadt@riscv64.openbsd.org:
    /usr/src/sys/arch/riscv64/compile/RAMDISK
real mem  = 1073741824 (1024MB)
avail mem = 1005133824 (958MB)
SBI: OpenSBI v1.5, SBI Specification Version 2.0
[...]
Welcome to the OpenBSD/riscv64 7.5 installation program.
(I)nstall, (U)pgrade, (A)utoinstall or (S)hell?
```

La procédure d'installation est tout à fait standard, tout en considérant que nous n'avons pas de réseau et devons donc préciser où se trouvent les *sets*. `sd0` désigne la microSD et `sd1` le support USB, et c'est là le seul point où il ne faut pas se mélanger les pinceaux. Concernant le partitionnement, nous n'avons presque rien à faire, si ce n'est, à l'étape de configuration des labels, de spécifier le point de montage avec :

```
sd0> n a
mount point: [none] /
```

Point important, vers la fin de l'installation, OpenBSD va « réorganiser » le noyau pour produire un binaire unique. C'est une fonctionnalité de sécurité appelée KARL et cette génération prend du temps, beaucoup de temps. Ne vous inquiétez donc pas en voyant le message « `Relinking to create unique kernel...` » et rien d'autre pendant un bon bout de temps. C'est normal et ceci débouchera, à un moment, sur :

```
CONGRATULATIONS! Your OpenBSD install has been
successfully completed!

When you login to your new system the first time,
please read your mail using the 'mail' command.

Exit to (S)hell, (H)alt or (R)eboot? [reboot]
```

Nous choisissons « h » ici, car le système n'est pas encore *(re)bootable*. Nous avons préparé la microSD avec les *bootloaders* et une zone pour UEFI, mais avons démarré sur USB. Notre microSD ne possède pas le binaire EFI nécessaire. Corrigeons cela avant le premier démarrage.

5. INSTALLER LE CHARGEUR EFI ET PREMIER DÉMARRAGE

Placer le chargeur EFI (`bootriscv64.efi`) sur la microSD est un jeu d'enfant puisque nous venons de l'utiliser à l'instant : il suffit de le copier, depuis le système de fichier FAT de la clé USB vers la microSD :

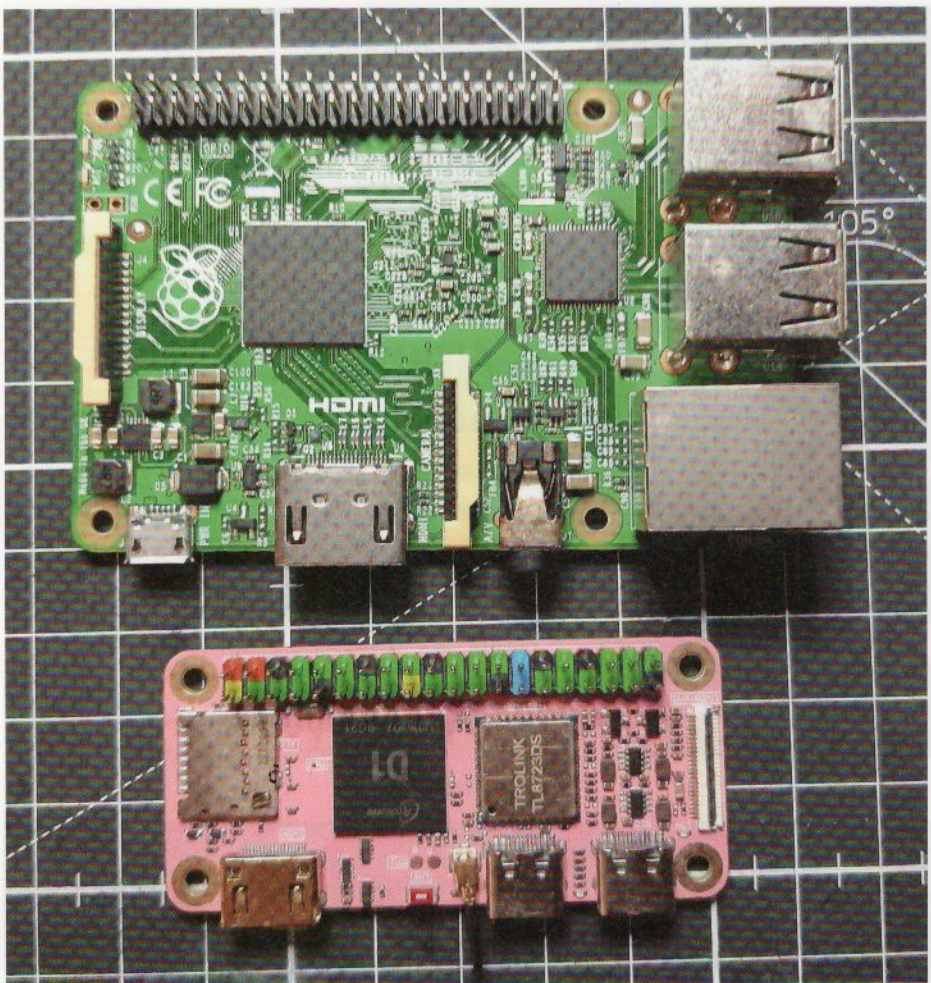
```
# mkdir tmp
# mount /dev/sd2i /mnt
# cp -r /mnt/efi tmp/
# umount /mnt/
```

puis :

```
# mount /dev/sd2i /mnt
# cp -r tmp/efi /mnt/
# umount /mnt/
# rm -rf tmp
```

Comme les scripts U-Boot cherchent automatiquement ce binaire sur les supports scannés et que le premier d'entre eux est la microSD, il nous suffit de remettre la carte dans son emplacement et d'alimenter le tout. Nous verrons alors, comme précédemment, démarrer le système, mais cette fois sur la microSD. Durant le *boot* (et après), vous aurez droit à divers ralentissements :

```
reordering: ld.so  libc libcrypto sshd.
openssl: generating isakmpd RSA keys... done.
openssl: generating iked ECDSA keys... done.
ssh-keygen: generating new host keys: RSA ECDSA ED25519
sshd: (ED25519) SHA256:SeYVtcV1n0KIn7J6zutN736PXzALEJ0+iAy7uVXIM3c
starting early daemons: syslogd pflogd ntpd.
```



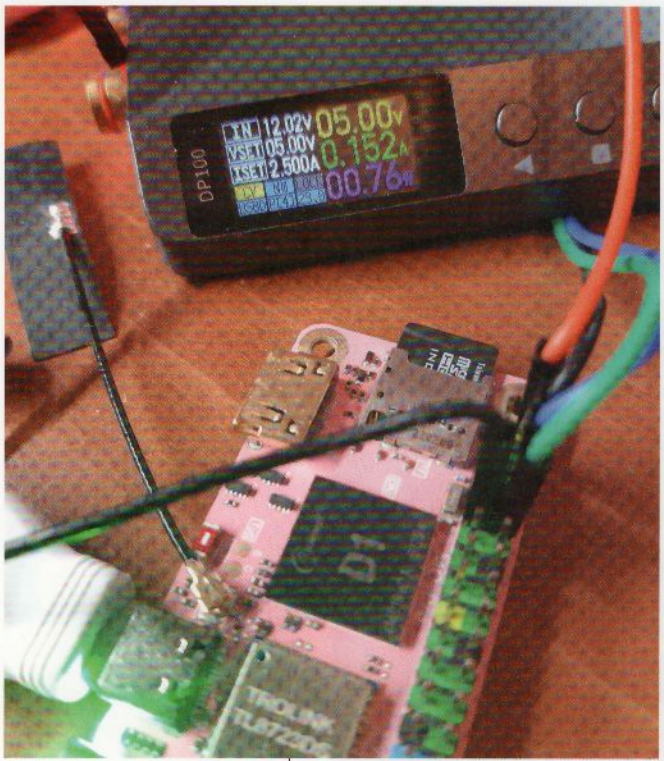
Voici une petite comparaison de taille entre la MangoPi et une très classique Raspberry Pi 2. Au-delà des couleurs et dimensions cependant, rien ne saurait différencier davantage ces deux SBC que l'architecture utilisée : ARM en haut et RISC-V en bas.

Ce `reordering` est un dispositif de sécurité, réorganisant les symboles dans les bibliothèques afin de rendre les attaques plus difficiles. Et ceci sera fait **à chaque démarrage**, ce qui, sur une plateforme embarquée, peut être très pénalisant. Vous pouvez désactiver cela avec :

```
# rcctl disable library_aslr
```

Ce n'est pas tout, le même style de réorganisation est appliqué au noyau (KARL), mais ceci se fait en tâche de fond et augmente la charge système quelques dizaines de secondes après le *boot*. Ceci signifie que le système sera moins réactif dans l'immédiat, ce qui peut également être évité (pour les tests) en utilisant :

```
# sha256 -h /var/db/kernel.SHA256 \
  /dev/null
```



L'alimentation de la carte via le connecteur 40 broches permet accessoirement d'avoir une idée de sa consommation en temps réel. 150 mA en 5 V pour le système « au repos » est plus que respectable. Chargée, la carte n'a à aucun moment dépassé 2 W lors des tests.

Notez toutefois que vous aurez droit à un message d'erreur « `reorder_kernel: failed` » à chaque démarrage. Le fichier de log spécifié dans le message contient la commande permettant de réactiver le mécanisme de sécurité le moment venu.

6. AJOUTONS DU RÉSEAU

La MangoPi MQ-Pro D1 ne dispose pas d'interface Ethernet et, pour l'heure, le contrôleur Wi-Fi n'est pas pris en charge. Cependant, rien ne vous empêche de connecter un adaptateur USB/Ethernet puisque le port OTG est disponible. Si le modèle est supporté, vous devrez voir apparaître des messages en conséquence dans la console série :

```
axen0 at uhub0 port 1 configuration 1 interface 0
"ASIX Elec. Corp. AX88179" rev 2.10/1.00 addr 2
axen0: AX88179, address 00:24:9b:77:20:eb
rgephy0 at axen0 phy 3: RTL8169S/8110S/8211 PHY, rev. 5
<code>
```

Et l'interface sera alors également visible avec les outils habituels :

```
<code>
# ifconfig
[...]
axen0: flags=8802<BROADCAST,SIMPLEX,MULTICAST> mtu 1500
        lladdr 00:24:9b:77:20:eb
        index 4 priority 0 llprio 3
        media: Ethernet autoselect (10baseT half-duplex)
        status: no carrier
```

Automatiser la prise en charge est très simple, puisqu'il suffit de créer un fichier `hostname` dans `/etc` avec, comme extension, le nom de l'interface (ici, `axen0`). Pour une configuration automatique via DHCP, le fichier contiendra :

```
# cat > /etc/hostname.axen0
inet autoconf
^D
```

Et vous pourrez prendre les modifications en compte immédiatement sans avoir à redémarrer avec :

```
# sh /etc/netstart
WARNING: /etc/hostname.axen0
is insecure, fixing permissions.


# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8): 56 data bytes
64 bytes from 8.8.8.8: icmp_seq=0 ttl=115 time=12.861 ms
64 bytes from 8.8.8.8: icmp_seq=1 ttl=115 time=12.522 ms
64 bytes from 8.8.8.8: icmp_seq=2 ttl=115 time=12.557 ms
^C
```

À présent connecté au réseau, le système est non seulement accessible via SSH, qui est bien plus agréable à utiliser que la console série, mais aussi, et surtout, vous pouvez installer des paquets binaires et commencer à réellement vous amuser :

```
# pkg_add cowsay
quirks-7.14:updatedb-0p0: ok
quirks-7.14 signed on 2024-03-25T22:12:52Z
quirks-7.14: ok
cowsay-0.2.1v0:p5-Text-Template-1.61: ok
cowsay-0.2.1v0: ok


# cowsay "Hackable <3 OpenBSD/RISC-V"
 ____________________________
< Hackable <3 OpenBSD/RISC-V >
 ----------------------------
        \   ^__^
         \  (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||
```

CONCLUSION

Maintenir un système pour une carte RISC-V spécifique est presque en tout point similaire à faire de même pour une plateforme de *retrocomputing*. Certes, le matériel est là, accessible à peu de frais (dans certains cas) et un système « officiel » existe, mais celui-ci est d'ores et déjà abandonné (ou le sera dans quelques mois). On se retrouve donc dans une situation où,

le temps que le support soit stabilisé, la carte n'est plus la starlette du moment, remplacée par une autre, plus complète, plus rapide, plus ce que vous voulez. Ceci revient à tenter de faire un magnifique château de sable, juste pour le voir emporté par la marée, à peine prend-il réellement forme. On comprend alors parfaitement pourquoi si peu de systèmes, que ce soit des GNU/Linux *mainstream* ou des alternatives comme [Free|Open|Net]BSD, supportent officiellement et directement certaines cartes et plateformes, préférant souvent se concentrer sur des émulations avec QEMU. De fait, QEMU RISC-V **est** la plateforme de référence, et même là, comme le précise le site officiel, « *Unfortunately many of the RISC-V boards QEMU supports are currently undocumented* »...

C'est totalement paradoxal, car on ne peut même pas réellement parler de carence en matériel, puisque la gamme de cartes et de SoC est non négligeable. Au contraire, cette impression de « ça part dans tous les sens » fait plus de tort qu'autre chose. Je pense réellement que, tant que cette situation perdurera et qu'aucune plateforme ne se dégagera comme étant **la** référence, l'architecture RISC-V appliquée aux SBC (ou même aux *laptops*) n'a que peu de chance de faire de l'ombre à ARM et donc de devenir une alternative réellement viable. Mais j'ai bon espoir, car le phénomène était exactement le même dans l'univers des MCU avant l'arrivé massive des ESP32-C*... **DB**

RÉFÉRENCES

[1] **https://connect.ed-diamond.com/hackable/hk-054/milk-v-duo-un-minuscule-sbc-risc-v-a-8-eu**

[2] **https://github.com/Tina-Linux**

[3] **https://www.openbsd.org/riscv64.html**

[4] **https://fr.wikipedia.org/wiki/Principe_KISS**

[5] **https://github.com/freebsd-d1/freebsd-d1**

[6] **https://cdn.openbsd.org/pub/OpenBSD/7.5/riscv64/INSTALL.riscv64**

[7] **https://sourceforge.net/projects/cloverefiboot/**

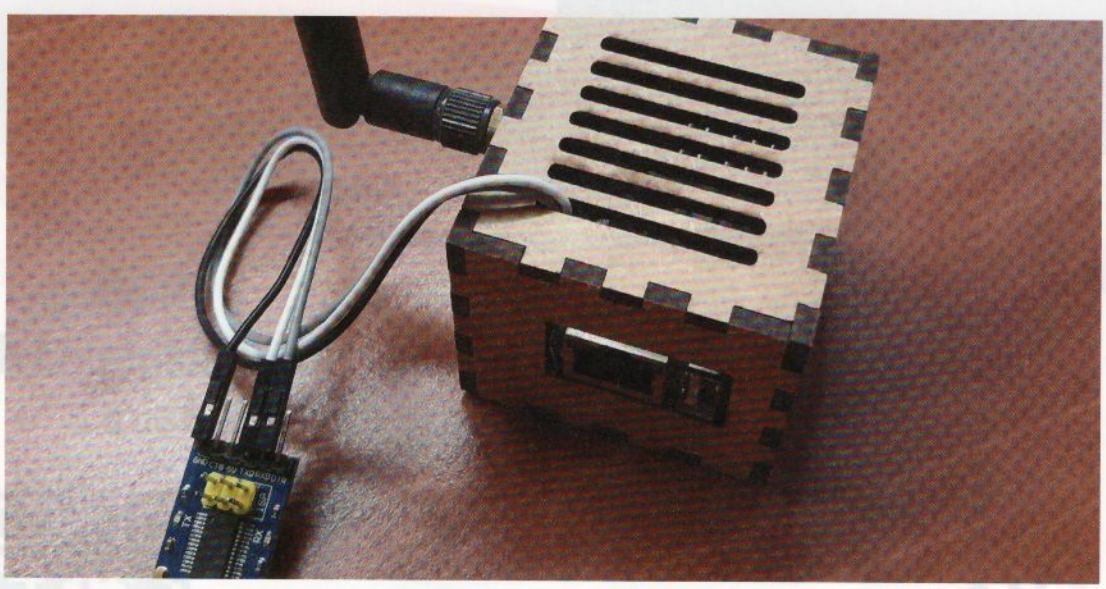
[8] **https://cdn.openbsd.org/pub/OpenBSD/7.5/riscv64/**

[9] **https://linux-sunxi.org/Allwinner_Nezha**

RPI & I2P : ANONYMISER SON TRAFIC AVEC L'INTERNET INVISIBLE

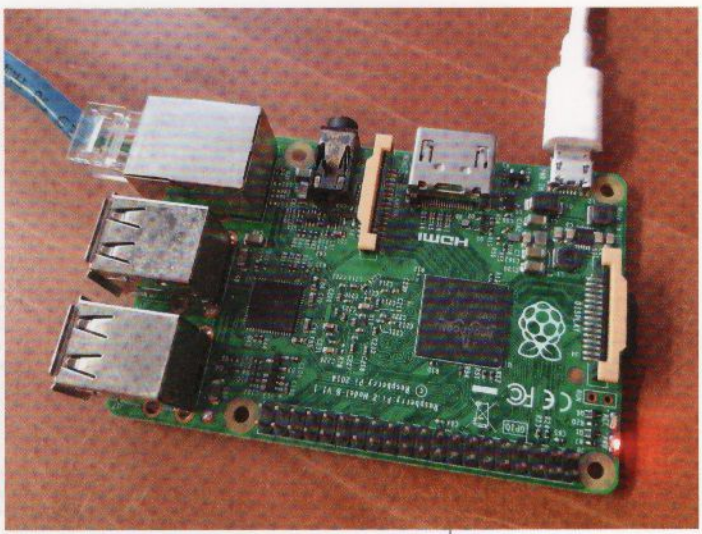
Denis Bodor

La surveillance de masse, la censure et les nombreuses restrictions qui pèsent sur Internet, et les communications en général, représentent un énorme problème pour la vie privée et la liberté d'expression. Bien entendu, en Europe, nous ne sommes certainement pas les moins bien lotis, en particulier en comparaison avec des pays aux régimes totalitaires. Mais le fait de dissimuler ses communications et ses échanges de données n'est pas l'apanage des journalistes, des lanceurs d'alerte, des freedom fighters ou même, à l'autre extrême, des groupes mafieux ou terroristes et des réseaux pédophiles. L'anonymisation, le chiffrement et la dissimilation sont devenus l'affaire de tout un chacun désormais.



Si nous faisons le choix de rester très terre à terre et concentré sur notre vie de tous les jours, on peut se dire qu'un réseau décentralisé utilisant un chiffrement de bout en bout n'a que peu d'intérêt dans notre quotidien. Après tout, nous n'essayons pas de renverser un régime tyrannique avec un dictateur psychopathe à sa tête, ne sommes pas en train de collecter des informations sur une mégacorporation satanique pour faire éclater la vérité au grand jour et n'échangeons pas non plus de messages avec des réseaux de trafic et de distribution de drogues, d'armes ou que sais-je encore. Nous sommes des citoyens « normaux » (autant qu'on puisse l'être en lisant ce magazine, du moins aux yeux du commun des mortels), utilisant Internet de façon on ne peut plus légale et cherchant simplement à apprendre et expérimenter autour des technologies qui nous intéressent. Nous utilisons des machines parfois dispersées de-ci de-là, et éventuellement participons à différents projets et communautés en lien avec des centres d'intérêt qui ne regardent que nous.

Ce comportement qu'on pourrait qualifier de « standard » et absolument pas répréhensible n'en est pas moins susceptible d'être surveillé, espionné et analysé. Je pars du principe ici que, étant un minimum versé dans les subtilités de la technologie et du fonctionnement des systèmes d'information, vous n'êtes pas non plus du genre à disséminer vos informations personnelles naïvement sur tous les réseaux sociaux disponibles, n'utilisez pas de connexions non chiffrées pour accéder à vos machines et ne choisissez pas « 123456 » comme mot de passe *root* dans vos systèmes embarqués (parce que c'est le même code que sur mes valises ?!). Peut-être même avez-vous mis en place votre propre VPN avec OpenVPN et/ou WireGuard. Mais cela est-il réellement suffisant pour vous protéger ?



Une « bonne vieille » Raspberry Pi 2b fera parfaitement l'affaire en guise de routeur/proxy I2P pour un LAN. Les performances globales n'étaient impactées que très peu par les capacités du processeur et/ou la vitesse de l'interface Ethernet (100 Mb derrière une box fibre à 500 Mb, par exemple, importe peu).

0.1 VPN ET VPN : DE LA NON-ANONYMISATION DES CONNEXIONS

Avant toute chose, commençons par faire une distinction très claire entre deux choses totalement différentes d'un point de vue pratique, lorsqu'on parle de VPN. Cet acronyme est devenu un terme fourre-tout visant à vendre des abonnements à des services qui promettent monts et merveilles, mais qui, en réalité, ne sont généralement utilisés que pour une chose : regarder des contenus géographiquement restreints sur les sites de VOD et de *streaming*.

Ainsi, lorsque le « grand public » parle de VPN, il s'agit généralement de services en ligne, incluant éventuellement des greffons pour les navigateurs ou des outils exécutés localement (souvent propriétaires). Dans ce genre d'architecture, votre

La page de statistiques des tunnels I2P de l'applicatif officiel en Java vous donnera tous les détails concernant les liaisons établies. Pour autant, rien de tout cela ne mettra en péril votre sécurité ou celle des autres pairs.

machine se connecte à un serveur puis fait transiter l'ensemble des communications vers ce serveur avant d'atteindre le site ou service que vous souhaitez utiliser. Votre fournisseur d'accès, par exemple, n'est pas capable de voir et d'analyser ce trafic chiffré, ni même de déterminer la destination finale des données. Le point de sortie du réseau privé est souvent laissé à discrétion de l'utilisateur, opérant un choix dans une interface en ligne et/ou directement dans l'application localement installée. C'est ce qui permet, par exemple, d'accéder à des contenus qui sont limités dans le pays où vous vous trouvez physiquement, puisque les connexions semblent venir d'une autre région du monde.

L'autre « type » de VPN est celui que vous allez choisir de configurer par vos propres moyens. Ceci se résume à utiliser les services d'un hébergeur pour, par exemple, déployer une machine virtuelle accueillant le serveur VPN, comme OpenVPN. Les clients contactent ce serveur pour former le réseau virtuel où les différentes machines peuvent communiquer comme si elles se trouvaient sur un réseau local, fonctionnant par-dessus Internet. C'est un réseau dans le réseau, d'où les termes de « réseau », « privé », « virtuel ». Une alternative possible, en utilisant par exemple WireGuard, consiste à établir des connexions individuellement, sous la forme de tunnels. Il n'y a plus de serveur, mais en interconnectant les clients, on obtient également un réseau.

Et bien entendu, comme rien ne vous empêche de mettre en place de la translation d'adresses et des règles de routage, vous pouvez parfaitement configurer le tout pour reproduire ce que propose un fournisseur de services. Déployer une VM dans un autre pays que le vôtre, installer un client VPN et rediriger votre trafic local, vers ce point, de manière à ce que votre accès à Internet l'utilise comme point de sortie, et le tour est joué. Bravo, vous êtes votre propre TrucmucheVPN !

Une certaine forme d'anonymisation est donc effectivement en place, mais elle n'est pas universelle. Dans les deux cas, personne ne peut inspecter votre trafic entre vous et le point de sortie. Mais là, à cet endroit précis, tout est visible. Pas les données bien sûr, puisque vous utilisez HTTPS ou un autre protocole *over* SSL/TLS, mais les métadonnées : origine, destination, durée, moment, taille, etc. Si vous reposez sur un service, vous devez également faire confiance au fournisseur pour qu'il n'enregistre pas tout cela. Mais ce n'est pas tout, dans les deux cas, l'adresse IP d'origine des connexions est liée à votre identité. Il est donc possible de faire le rapprochement entre l'IP et le compte (service VPN ou hébergement de VM), le compte et vous, et même le moment et la durée des connexions au VPN avec les mêmes informations en sortie du VPN. Même d'un point de vue public, votre connexion à un service, un site, un forum, un serveur... est identifiable et identifiée par l'IP que vous utilisez. Et cette information peut, par définition, être corrélée avec celles de précédentes utilisations et toutes autres sortes d'information.

Bref, chiffré oui, totalement anonyme, non.

1. I2P : INVISIBLE INTERNET PROJECT

I2P est une solution très différente d'un simple VPN, mais offre le même genre de services avec, en prime, une anonymisation complète du trafic. Le principe de fonctionnement est relativement simple puisque tous les hôtes faisant fonctionner un bout de code I2P (client ou routeur) forment un réseau dynamique, faisant transiter les données chiffrées, sans qu'aucun d'entre eux (ou presque) ne sache d'où elles viennent, où elles vont ou ce qu'elles sont. Si cela vous rappelle à la fois Tor (*The Onion Routing*) et/ou BitTorrent, ce n'est pas un hasard, le principe est le même. Les objectifs, cependant, ou plus exactement la manière de les atteindre est sensiblement différente. Nous avions abordé l'utilisation de Tor en point d'accès Wi-Fi dans le numéro 6 **[1]** il y a bien longtemps.

On retrouve ainsi le fonctionnement en « routage en oignon », appelé pour l'occasion « routage en ail » (*garlic routing* en anglais) et le chiffrement, quatre couches au total. Chaque participant au réseau I2P, appelé nœud ou routeur (*node* ou *router*), est connecté en utilisant des chemins, ou tunnels, qui peuvent être entrants ou sortants, et qui servent à communiquer avec d'autres routeurs. Ils sont identifiés par une valeur cryptographique qui les caractérise. D'autres entités sur le réseau, les clients, utilisent les tunnels mis en place temporairement entre les routeurs pour communiquer entre eux, ceux-ci sont désignés par le terme de « destination » dans la nomenclature I2P. Lorsque deux clients échangent des données ou messages, les routeurs ne voient pas le trafic, car il est chiffré, et les chemins empruntés par ces messages changent dynamiquement. Personne ne peut savoir d'où viennent les messages et où ils vont, car le trafic entre deux clients est noyé dans la masse. Bien entendu, plus il y a de routeurs entre deux destinations, moins il est possible de savoir ce qui se passe sans avoir la maîtrise complète de l'ensemble du réseau. Chose quasi impossible à réaliser puisque tout le système est un réseau pair à pair décentralisé.

Comme vous pouvez le voir, et étant donné que les destinations ne sont pas identifiées par une adresse IP, nous sommes dans une situation bien plus avantageuse qu'un simple VPN. Tout ce que votre fournisseur d'accès peut voir, c'est un trafic chiffré entre des machines, en utilisant un certain nombre de ports TCP reconnaissables, mais ça s'arrête là. De plus, comme par défaut, un client est également un routeur pour le réseau, il n'y a strictement aucun moyen de faire la distinction entre vos données et celles des autres, transitant simplement par vous.

Si vous avez déjà utilisé Tor, tout ceci ne vous est pas étranger puisque le fonctionnement est très similaire. Il y a cependant quelques différences qui sont davantage architecturales que purement techniques :

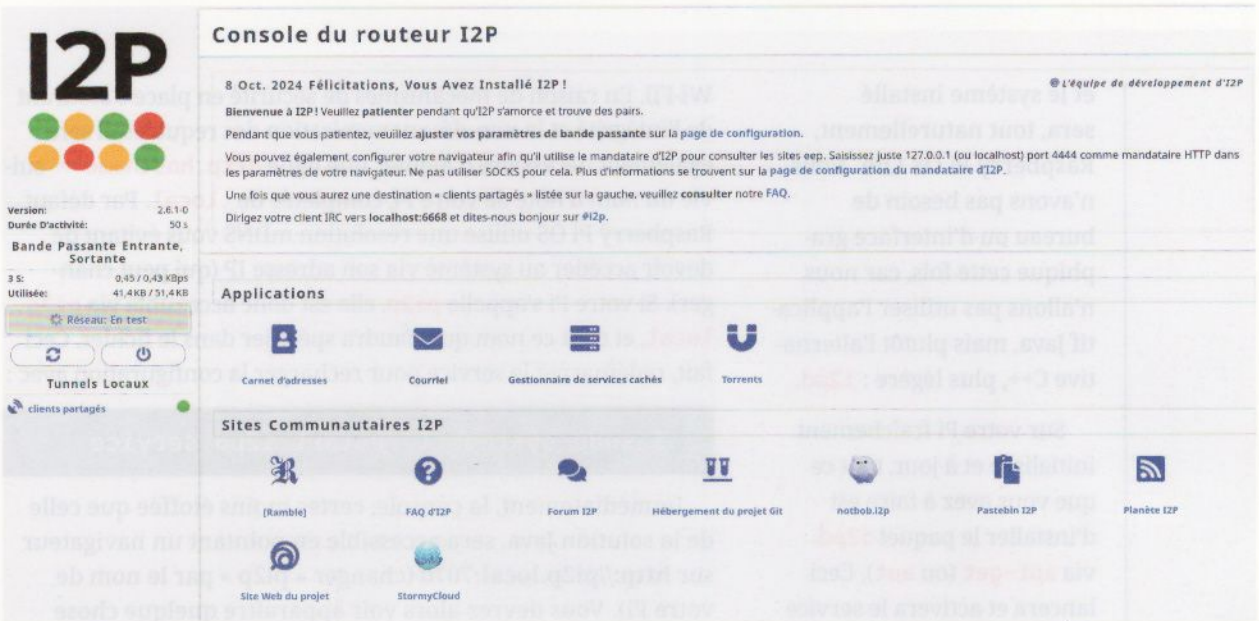
- I2P est plus petit que Tor et en conséquence sensiblement moins populaire bien qu'étant apparu une année seulement après Tor (2002). Ceci n'est pas nécessairement un problème, car I2P subit moins d'attaques et est moins bloqué que Tor. La popularité est à la fois un avantage et une malédiction.
- Par défaut, tous les pairs dans I2P participent au fonctionnement du réseau alors qu'avec Tor, c'est un choix de configuration.
- I2P n'est **pas** initialement prévu pour accéder à l'Internet « standard » et il n'y a pas de « points de sortie » comme c'est le cas avec Tor ou un service de VPN. L'objectif d'I2P n'est pas de permettre l'accès à des sites bloqués par des instances étatiques totalitaires, par exemple, même s'il est parfaitement possible de configurer une destination dans ce sens. Pour I2P, les bénéfices découlant de l'utilisation du réseau proviennent précisément du fait de ne pas accéder à l'Internet public. C'est une différence **fondamentale**.
- Tor, comme I2P, permet la mise en place de « services cachés » (*hidden services*) comme des serveurs web, IRC ou de messagerie n'étant accessibles **que** à l'intérieur du réseau. Mais I2P est conçu et optimisé spécifiquement pour cet usage précis, et de ce fait, plus rapide et performant que Tor dans ce domaine.
- Comme I2P n'est pas conçu pour fournir un accès à l'Internet standard, il n'y a pas de problème de configuration lié au fait de ne pas laisser « sortir » de trafic tout en participant au réseau. Dans le cas de Tor, les nœuds de sortie, en plus d'être des points d'attaque privilégiés, induisent une responsabilité de la part de la personne en charge de la machine, car c'est l'adresse IP de ce dernier qui sera vue par les services accédés et potentiellement surveillés. IP associée à votre identité et donc susceptible de vous impliquer en cas d'utilisation frauduleuse ou illégale par des tiers.

Comme vous pouvez le voir, Tor et I2P répondent à deux cas d'usage très différents. Avec, pour I2P, une double utilité : accéder aux services cachés et établir des connexions entre machines, d'un groupe, d'une communauté ou même, comme pour moi, d'une seule personne. Remplaçant alors, d'une certaine manière, un VPN « maison ».

2. ESSAYER I2P LOCALEMENT

Il existe principalement deux options pour installer I2P sur PC, Mac ou SBC. La première **[2]** est celle proposée par le projet lui-même **[3]**, écrite en Java et fonctionnant pour Windows, macOS et GNU/Linux (ainsi qu'Android, mais c'est un cas à part), et la seconde **[4]**, plus technique, en C++, est orientée principalement vers les systèmes Unix libres (GNU/Linux, FreeBSD, OpenBSD, etc.), mais pas seulement.

Pour essayer rapidement I2P sur une machine *desktop*, commençons par l'outil Java. Ce langage permettant précisément un fonctionnement sous plusieurs systèmes et architectures, et l'installation sous Windows largement documentée, nous ferons le choix d'utiliser ici une machine GNU/Linux.
Notez au passage que de base, si l'anonymat et la vie privée sont des considérations importantes pour vous, peut-être vaut-il mieux éviter Windows ainsi que les systèmes propriétaires en général (cf. les décriées « fonctionnalités » *Recall & Copilot+*). Notez qu'il en va de même pour les navigateurs et qu'utiliser Google Chrome et son lot de *trackers* fait perdre tout son intérêt à I2P en termes de vie privée.



Console web de l'applicatif Java officiel d'I2P.

Pour installer I2P sur une machine, il vous faudra un environnement d'exécution Java fonctionnel, chose qui sur Debian, par exemple, pourra être réglée d'un simple `apt-get install openjdk-17-jre-headless`. Vous pourrez ensuite récupérer l'installeur depuis la page de téléchargement **[2]** sous la forme d'un fichier `i2pinstall_2.6.1.jar`. Vous pourrez ensuite exécuter ce dernier via `java -jar` suivi du nom du fichier. Choisissez ensuite le répertoire d'installation et terminez la procédure. Démarrer I2P sur votre machine reviendra ensuite à vous placer dans le répertoire choisi et utiliser la commande `./i2prouter start`. Ceci démarrera le service et ouvrira automatiquement la page **http://127.0.0.1:7657/home** dans votre navigateur par défaut, c'est la console I2P à laquelle vous pouvez retourner n'importe quand en utilisant la même URL. `./i2prouter stop` arrêtera le routeur I2P lorsque vous n'en avez plus besoin.

Les différentes pages du « site » vous permettront de connaître l'état de votre routeur ainsi que de sa configuration. La page d'accueil propose également les liens vers les services cachés dont le nom se termine par « .i2p ». Dans l'état, cependant, un clic sur l'un de ces éléments provoquera une erreur, votre navigateur ne sait pas accéder à ces éléments aux TLD étranges. Pour régler ce problème, vous devez configurer votre navigateur pour utiliser le service de proxy intégré à I2P. Dans Firefox, allez dans ***Paramètres***, cherchez « proxy », et dans ***Paramètres de connexion*** spécifiez une ***Configuration manuelle du proxy*** avec ***Proxy HTTP*** à 127.0.0.1 et ***Port*** à 4444 (idem pour HTTPS). C'est tout, votre navigateur est maintenant capable de visiter les « .i2p ».

3. RASPBERRY PI EN PROXY

Faire fonctionner I2P sur votre poste de travail est une chose, mais pour fournir ce même type de connectivité à l'ensemble de votre LAN, mieux vaudra héberger le routeur sur une machine consommant peu de courant, tout en pouvant rester fonctionnelle en permanence : une carte Raspberry Pi, donc. N'importe quelle Pi devrait faire l'affaire, ou même un autre SBC équivalent,

et le système installé sera, tout naturellement, Raspberry Pi OS Lite. Nous n'avons pas besoin de bureau ou d'interface graphique cette fois, car nous n'allons pas utiliser l'applicatif Java, mais plutôt l'alternative C++, plus légère : `i2pd`.

Sur votre Pi fraîchement initialisée et à jour, tout ce que vous avez à faire est d'installer le paquet `i2pd` via `apt-get` (ou `apt`). Ceci lancera et activera le service automatiquement, il sera donc disponible à chaque démarrage. Cependant, la configuration par défaut fonctionne exactement comme l'I2P en Java et écoutera les connexions uniquement sur l'hôte local (127.0.0.1). Ce n'est pas ce que nous voulons.

Pour modifier la configuration, vous devez éditer le fichier `/etc/i2pd/i2pd.conf` et trouver les lignes :

```
[http]
address = 127.0.0.1
port = 7070
```

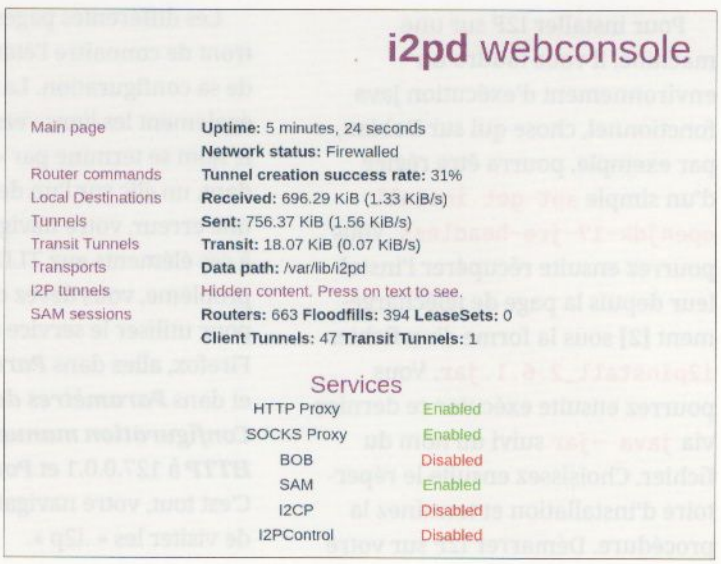
et :

```
[httpproxy]
address = 127.0.0.1
port = 4444
```

Là, changer simplement `127.0.0.1` en `0.0.0.0` pour écouter sur toutes les interfaces disponibles et donc Ethernet (et éventuellement Wi-Fi). En raison de mécanismes de sécurité en place s'assurant de l'intégrité et la non-désanonymisation des requêtes, il sera également nécessaire d'ajouter une ligne `http.hostname =` suivie du nom d'hôte de votre Pi, complétée de `.local`. Par défaut, Raspberry Pi OS utilise une résolution mDNS vous évitant de devoir accéder au système via son adresse IP (qui peut changer). Si votre Pi s'appelle `pi2p`, elle est donc accessible via `pi2p.local`, et c'est ce nom qu'il faudra spécifier dans le fichier. Ceci fait, redémarrez le service pour recharger la configuration avec :

```
$ sudo systemctl restart i2pd.service
```

Immédiatement, la console, certes moins étoffée que celle de la solution Java, sera accessible en pointant un navigateur sur **http://pi2p.local:7070** (changer « pi2p » par le nom de votre Pi). Vous devrez alors voir apparaître quelque chose comme ceci :



Cette page vous donne plusieurs indications sur la configuration et l'état de la connexion au réseau I2P, et nous reviendrons dessus dans un instant. Vous pouvez également choisir de passer l'interface en français (***Routers Commands*** et ***Change language***). Le proxy est également actif, ce qui signifie que vous pouvez alors l'utiliser avec toutes les machines de votre réseau en spécifiant le nom d'hôte (ici, « pi2p.local ») et le port 4444 dans les configurations des navigateurs. Remarquez le port 7070, différent du 7657 de la console en Java.

Votre Pi en routeur I2P est en place et prêt à vous permettre de surfer dans l'Internet invisible…

4. UTILISATION POUR DES CONNEXIONS POINT À POINT

Si vous êtes comme moi, vous devez être en train de vous dire que tout ceci est bien joli, mais que se promener sur des sites cachés, anonymement, à la vitesse d'un escargot sous benzodiazépines, c'est amusant quelques minutes, mais ça s'arrête là. Et je ne peux que vous donner raison, car effectivement, c'est bien plus lent qu'une connexion à Internet qu'on a désormais l'habitude de considérer comme normale, mais c'est la nature même du système qui veut ça, mais lorsqu'on a un usage standard du Net, cela reste une découverte anecdotique. Ce qu'il nous faut, c'est une vraie et concrète utilisation pratique pouvant être utile dans notre quotidien.

Que diriez-vous donc de pouvoir accéder anonymement, discrètement et de façon sécurisée à cette Pi fraîchement installée, et ce, depuis n'importe où, y compris via le Wi-Fi WEP d'un cybercafé malfamé en plein centre de Pyongyang ? En guise d'exemple, et étant donné que nous avons activé la connexion SSH sur la Pi, nous allons utiliser I2P pour établir un tunnel entre la machine utilisant l'applicatif Java et la Pi. L'idée est, au final et en guise d'exemple, de ne plus ouvrir le port 22 au réseau, mais de permettre l'établissement d'une connexion SSH via I2P.

Pour cela, nous allons tout d'abord modifier la configuration côté Pi en créant le fichier `/etc/i2pd/tunnels.d/monssh.conf` contenant :

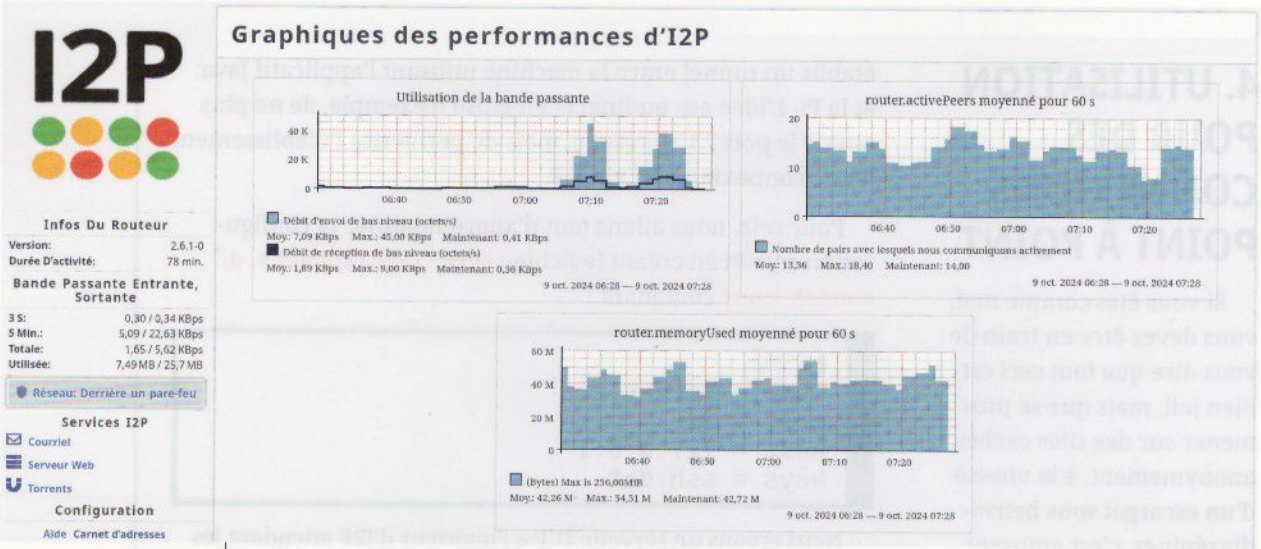
```
[SSH]
type = server
port = 22
host = 127.0.0.1
keys = ssh.dat
```

Nous créons un serveur TCP à l'intérieur d'I2P attendant les connexions et les redirigeant vers l'hôte local sur le port 22. Le fichier `ssh.dat` sera automatiquement créé dans `/var/lib/i2pd` et contiendra la clé de chiffrement permettant également d'identifier le service dans le réseau. En rechargeant la configuration via `systemctl` ou avec la console web, nous pouvons constater dans cette dernière qu'à la section ***I2P tunnels***, une ligne supplémentaire a fait son apparition : « SSH ⇒ ozxzy7zm6ra223imtq2i6ctekypl6jvpj74y33nybj34ctill2aa.b32.i2p:22 ». Cette longue chaîne (sans le « :22 ») est le « nom » ou identifiant de votre serveur dans le réseau I2P.

Passons maintenant du côté de l'applicatif Java et, dans la console cliquez sur l'icône ***Gestionnaire de services cachés***. Tout en bas de la page, cliquez sur ***Créer*** pour ajouter une configuration cliente. Donnez un nom au profil (ici « SSH denis »), éventuellement une description, et complétez :

- port : 2222 ;
- accessible par : 127.0.0.2 ;
- destination du tunnel : collez ici le nom I2P copié de la console d'i2pd.

Le tunnel ne sera pas créé (démarré) automatiquement, mais il apparaîtra sur la page, tout en bas. Cliquez sur ***Démarrer*** et le tunnel sera mis en place. Ceci prendra un peu de temps, mais au final, l'état passera du rouge au jaune, puis au vert (ainsi que, sur la gauche, sous ***Tunnels Locaux***). Lorsque c'est le cas, vous disposez effectivement d'un tunnel entre 127.0.0.1:2222 sur cette machine et le service caché sur la Pi. En vous connectant, avec OpenSSH sur ce port localement (`ssh-p 2222 localhost`), vous arriverez sur le serveur SSH de la Pi. Soyez patient, en particulier si vous venez tout juste



Les performances sont à la fois la faiblesse et la pierre angulaire d'un réseau comme I2P. Il n'y a pas de miracle et les différentes couches cryptographiques, ainsi que l'architecture même du réseau, rendent impossible le fait d'envisager des débits comme ceux d'un Internet public non protégé. C'est ainsi…

de démarrer ou redémarrer vos connexions I2P, il faut du temps au réseau pour optimiser le trafic, mais à terme, ceci devrait être parfaitement utilisable, même si toujours un peu lent.

Comprenez bien la teneur de ce qui se passe. Il ne s'agit pas d'une connexion directe, mais d'un tunnel anonymisé, invisible et sécurisé, passant par I2P. Il est donc parfaitement possible de changer la configuration du serveur SSH côté Pi pour n'accepter que les connexions depuis 127.0.0.1 et non plus via l'interface Ethernet. Ceci est également valable pour un serveur possédant une IP publique, comme une VM chez un hébergeur et le service sera alors parfaitement invisible. Bien entendu, cela fonctionnera de la même manière avec un serveur HTTP pour héberger un site web accessible uniquement dans I2P (un forum, un Gitea ou encore quelque chose comme un Nextcloud, si vous êtes de nature très patiente).

Notez que, dans la console de l'applicatif Java, vous pouvez cliquer sur le nom de votre tunnel (à gauche), pour voir les informations concernant les tunnels construits ou acheminés. Vous y retrouverez votre tunnel (« SSH denis »), la liste des participants avec leur géolocalisation, ainsi que les données statistiques.

Nous n'avons ici utilisé i2pd que d'un seul côté, mais il est parfaitement possible de reproduire cette configuration en abandonnant totalement Java. Le serveur ne change bien entendu pas, mais côté client, nous installons également le paquet `i2pd` et utilisons la configuration par défaut (écoute sur 127.0.0.1 uniquement). Nous ajoutons ensuite un fichier `/etc/i2pd/tunnels.d/sshpi.conf`, mais qui cette fois décrira une connexion sortante et non un serveur :

```
[ssh-pi2p]
type = client
address = 127.0.0.1
port = 2222
destination = [...]34ctill2aa.b32.i2p
```

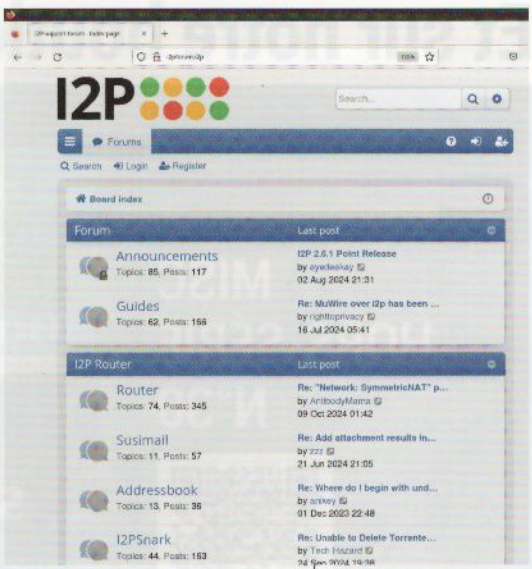
La structure est la même, mais nous précisons le type `client`, ainsi que la destination qui devient le nom du serveur fonctionnant sur la Pi. Dès la configuration rechargée ou le service redémarré, nous obtiendrons le même comportement qu'avec l'applicatif

Java, mais avec, je trouve, des performances sensiblement supérieures. Le tunnel fonctionnera localement de la même façon, mais il est possible de le rendre accessible à l'ensemble du LAN en changeant « 127.0.0.1 » par « 0.0.0.0 » par exemple, comme le proxy web.

CONCLUSION

Cet article n'est qu'une brève introduction à I2P, mais il y a beaucoup plus de choses à dire, à faire et à tester. Ceci dépendra totalement de l'utilisation que vous comptez avoir de ce système, qu'il s'agisse d'un usage individuel ou pour une structure comme un groupe, ou même une entité commerciale. Parmi les limitations d'I2P, on retrouve celles qui sont typiquement associées à ce genre de réseau, à commencer par les performances. Il est possible d'optimiser dans une certaine mesure, mais la lenteur des connexions est inhérente à la nature même de la protection fournie. Une utilisation interactive, comme SSH, ou un service web avec des pages légères et/ou majoritairement statiques sera parfaitement adapté (la cache du navigateur fait son travail), mais n'espérez pas transférer joyeusement des fichiers de plusieurs mégas avec `scp`, par exemple. I2P n'est pas fait pour cela et, si nous parlons d'échange de données volumineuses, c'est vers BitTorrent et sa structure distribuée qu'il faudra vous tourner. I2P a d'ailleurs été pensé dans ce sens, pour BitTorrent.

Un autre point très important concernant I2P, mais plus encore pour Tor, est la nécessité d'adapter votre comportement d'utilisation au but poursuivi. Disséminer des informations à gauche et à droite, établir des points de connexions entre plusieurs éléments qui vous caractérisent, tolérer des systèmes de *tracking*... sont autant de choses qui réduiront à néant la protection offerte par un réseau comme Tor, FreeNet ou I2P.



Ce site, i2pforum.i2p, est l'un des très nombreux services cachés qu'il n'est possible d'utiliser, et même de voir, qu'en étant connecté à I2P. Tout ceci est parfaitement invisible du reste d'Internet.

Protéger sa vie privée, assurer son anonymat et dissimuler des activités ou des échanges qui ne regardent que vous ne se limite pas à installer un code Java ou un paquet Raspberry Pi OS. Le monde dans lequel on vit est à l'affût de toute trace permettant de créer ou de compléter un profil pour chaque internaute, qui ensuite viendra s'ajouter à l'énorme masse de méta-informations échangées, commercialisées et utilisées par des acteurs qui, trop souvent, n'ont que peu de bonnes intentions à votre égard.

Donc entre laxisme et paranoïa, chacun doit trouver son juste milieu. Et I2P n'est qu'une variable de l'équation, que chaque personne doit résoudre pour elle-même... **DB**

RÉFÉRENCES

[1] **https://connect.ed-diamond.com/Hackable/hk-006**

[2] **https://geti2p.net/en/download**

[3] **https://geti2p.net/en/**

[4] **https://i2pd.website/**

CROSS-COMPILATION D'OPENBSD : C'EST MAL (TM), MAIS C'EST PAS GRAVE...

Denis Bodor

OpenBSD est un système spécial, adorablement spécial même, et parmi ses spécificités, on trouve par exemple le fait de privilégier la compilation native, par opposition à la compilation croisée. Je n'invente rien, c'est dans la documentation [1]. Ceci dit, même si les raisons évoquées sont parfaitement compréhensibles et légitimes, ce n'est pas parce que quelque chose n'est pas recommandé qu'il ne faut pas le faire. Cross-compilons donc un noyau OpenBSD pour arm64 depuis amd64...

– Cross-compilation d'OpenBSD : c'est mal (tm), mais c'est pas grave... –

« *Cross-compiling tools are in the system, for use by developers bringing up a new platform. However, they are not maintained for general use.* » (« Les outils de compilation croisée sont dans le système, à l'usage des développeurs abordant de nouvelles plateformes. Cependant, ils ne sont pas maintenus pour un usage général »). Comment voulez-vous qu'on résiste à un tel appel du pied ? Notez que ceci provient de la FAQ traitant de la construction du système depuis les sources **[1]** et que dans ce contexte, la mention de « *for general use* », qui ne concerne **que** la *cross-compilation*, ne peut que faire sourire. C'est aussi pour ça que j'aime OpenBSD, parce que les notions d'utilisateur, de développeur, d'usage général et de ce qui est attendu ou non d'une personne qui interagit avec le système sont assez différentes des autres systèmes. J'irai même jusqu'à dire que c'est plus agréablement *old school*.

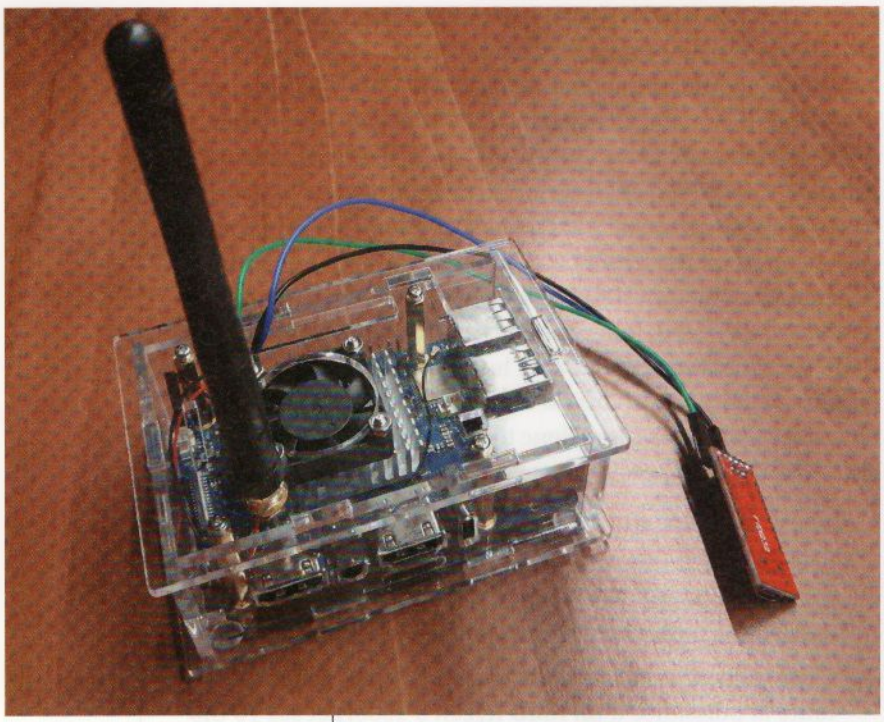
Plus concrètement cependant, et même si compiler tout un système est effectivement un excellent moyen de tester une plateforme, il est des situations où les options s'avèrent limitées. Prenons par exemple ma récente expérience d'une carte Orange Pi 5 max (SoC Rockchip RK3588), non supportée de base par le système (la version « plus » l'est, mais est suffisamment différente pour ne pas fonctionner de base) et cependant (bêtement ?) acquise en raison de ses huit cœurs ARM (4x Cortex-A76 + 4x Cortex-A55), plus 16 Gio de RAM, et destinée spécifiquement à servir de plateforme de construction pour d'autres cartes SoC ARM 64 bits. Le problème initial, en dehors du cadre du présent article, était d'arriver à obtenir un système fonctionnant sur un disque NVMe avec, au minimum, une interface réseau fonctionnelle (native ou via un adaptateur USB). Ceci suppose de jouer avec le *device tree* et un certain nombre de pilotes, conduisant naturellement, à un moment, à bidouiller dans les coins et donc tester différents noyaux (au moins pour savoir précisément ce qui se passe).

Dans ce genre de situations, il n'y a que deux solutions :

- utiliser une autre plateforme ARM 64 bits sous OpenBSD pour compiler (la Raspberry Pi 4b 2 Gio étant la plus capable de mon stock, d'où l'achat de l'Orange Pi 5 max) ;
- ou se tourner vers la compilation croisée et bénéficier de toute la puissance d'un AMD Ryzen 5 5600H et ses 16 Gio de RAM.

La première option permet d'utiliser la chaîne de compilation native, ce qui évite une étape dans le processus, mais est dépendante de performances souvent sous-optimales, ne serait-ce qu'à cause de la SD/MMC, tout en monopolisant une plateforme pour cet usage (même temporairement) et la seconde implique donc la construction d'une chaîne de compilation (certes, une seule fois) avant de réellement profiter de la puissance d'une machine de développement. Notez au passage que ceci ne se limite absolument pas au monde de l'embarqué et à ARM en particulier, et est tout aussi vrai avec des plateformes comme PowerPC, MIPS64, SuperH, Octeon, etc. Je n'ai pas testé toutes ces cibles, mais `macppc` (vieux iMac G4), `riscv64` (VisionFive 2 avec un SoC StarFive JH7110) et `armv7` (Orange Pi Zero avec SoC Allwinner H2+) n'ont posé strictement aucun problème.

Ce qui va suivre peut être fait pour un système en version *release* ou *stable*, mais vous comprendrez aisément que la motivation étant de chercher à faire fonctionner un noyau et ses pilotes correctement, ceci ne présente d'intérêt que pour *current*, la version actuelle de développement, avec les pilotes les plus à jour. En l'occurrence donc, la cible comme l'hôte sont en *current*.



Il est recommandé, avec l'Orange Pi 5 max (et « plus », et « pro »), d'utiliser un boîtier permettant l'ajout d'un système de refroidissement digne de ce nom. Des solutions peu chères, comme celle-ci (~6 € et 100 % acrylique transparent), font généralement l'affaire, à condition de remplacer le ventilateur inclus par quelque chose de plus puissant. Théoriquement, un simple radiateur sur le SoC pourrait suffire, mais je ne prends pas de risque avec une carte qui m'a coûté environ 160 €.

1. PRÉPARATION

Pour *cross-compiler* un noyau OpenBSD, tout ce qu'il vous faut, c'est un système OpenBSD fonctionnel sur une machine puissante (l'hôte) et les sources les plus à jour du système. Celles-ci peuvent être récupérées via un serveur AnonCVS **[2]** ou via Git, puisqu'un miroir est maintenu sur GitHub **[3]**. Personnellement, je maintiens un dépôt Git local régulièrement synchronisé avec celui de GitHub et une simple copie de l'arborescence source complète fait l'affaire. On peut faire de même avec `/usr/src` et CVS.

Pour nous faciliter la vie et éviter de taper des lignes de commandes à rallonge, nous allons définir quelques variables d'environnement :

```
$ export CROSSBASE=${HOME}/mycrossbuild
$ export CROSSTARGET=arm64
$ export CROSSCONF=GENERIC.MP
```

`CROSSBASE` est notre répertoire « racine » pour les manipulations et celui-ci est destiné à accueillir les sources ainsi que les objets et binaires produits pour chaque cible. En effet, il n'y a aucun intérêt à s'éparpiller inutilement et nous pourrons réitérer à loisir ces opérations pour plusieurs plateformes cibles après nous être occupés d'*arm64*. Chaque cible verra ses éléments répartis dans un répertoire différent suffixé du nom de la plateforme. Celle qui nous occupe présentement est donc spécifiée dans `CROSSTARGET`. Et enfin, nous avons `CROSSCONF` qui est la configuration du noyau que nous utiliserons pour procéder à une première compilation. J'avais évoqué la manière de configurer un profil pour une compilation de noyau OpenBSD dans l'article sur l'écriture de pilote dans le numéro 269 **[4]**, mais vous pouvez aussi, tout simplement, vous référer à la FAQ citée en début d'article **[1]**. Notez que nous utilisons ici `GENERIC.MP`, avec `MP` faisant référence à un système multiprocesseur puisque nous avons affaire à un monstre à huit cœurs, mais que certaines cibles peuvent se contenter de `GENERIC` (l'iMac G4, par exemple).

Ces variables définies, nous pouvons créer le répertoire et copier les sources :

```
$ mkdir ${CROSSBASE}
$ cd ${CROSSBASE}
$ cp -r ~/OpenBSDsrc.git ./src
$ cd src
```

Vient ensuite la phase la plus lourde de l'opération, puisque nous devons construire une chaîne de compilation capable de fonctionner sur notre système (amd64), mais produisant des binaires pour la cible (arm64). Pour ce faire, nous n'avons qu'à utiliser le `Makefile.cross` dédié, en passant les bons arguments, en commençant par créer l'arborescence où trouveront place les fichiers produits :

```
$ doas make -f Makefile.cross \
TARGET=${CROSSTARGET} \
CROSSDIR=${CROSSBASE}/dest.${CROSSTARGET} \
cross-dirs
[...]

$ ls ../dest.${CROSSTARGET}/
TARGET_ARCH   TARGET_CANON  TARGET_CPU    altroot/
bin/          dev/          etc/          home/
mnt/          root/         sbin/         sys@
tmp/          usr/          var/
```

Nous obtenons, dans le répertoire parent, un `dest.arm64` avec une arborescence similaire à celle d'un système OpenBSD et pouvons alors lancer la construction des outils :

```
$ doas make -j `sysctl -n hw.ncpuonline` \
-f Makefile.cross TARGET=${CROSSTARGET} \
CROSSDIR=${CROSSBASE}/dest.${CROSSTARGET} \
cross-tools
```

Attendez-vous à devoir mettre votre patience à l'épreuve, car même sur un Ryzen 5 5600H par exemple, vous pouvez compter sur une bonne heure de décrassage de ventilateurs. Fort heureusement, ceci n'est à faire qu'une seule fois. À propos de performances justement, remarquez l'utilisation de `sysctl` nous permettant de récupérer le nombre de CPU *online* qui est utilisé en argument pour paralléliser les tâches via l'option `-j` de `make`. Notez également que nous utilisons `hw.ncpuonline` et non `hw.ncpu` puisque, depuis l'apparition des exploits de type Spectre, rendant le système vulnérable aux *Side-Channel Attacks*, OpenBSD a décidé de tout simplement désactiver l'*Hyper-Threading* par défaut. Vous pouvez éventuellement réactiver cette fonctionnalité avec `sysctl hw.smt=1` si nécessaire (cf. en fin d'article).

Une fois la construction terminée avec succès, nous obtenons les exécutables tant attendus :

```
$ ls ../dest.${CROSSTARGET}/usr/bin
[...]
aarch64-unknown-openbsd7.6-addr2line*
aarch64-unknown-openbsd7.6-ar*
aarch64-unknown-openbsd7.6-c++filt*
[...]
c++*
cc*
clang-cpp*
ld.lld*
[...]

$ ../dest.${CROSSTARGET}/usr/bin/cc --version
OpenBSD clang version 16.0.6
Target: aarch64-unknown-openbsd7.6
Thread model: posix
InstalledDir:
/home/denis/mycrossbuild/src/../dest.arm64/usr/bin
```

La dernière étape consistera à, encore une fois, nous simplifier la vie pour plus tard en créant un lien symbolique pointant directement vers l'arborescence contenant les binaires et en changeant les permissions pour ne plus avoir à utiliser `doas` :

```
$ ln -sf ${CROSSBASE}/dest.${CROSSTARGET}/usr/obj \
${CROSSBASE}/obj.${CROSSTARGET}

$ doas chown -R `id -un`.`id -gn` \
${CROSSBASE}/obj.${CROSSTARGET}/*
```

Petite astuce au passage (surtout pour un article et faciliter les copier/coller), on utilise encore une fois le *backtick* avec `chown` pour obtenir le nom d'utilisateur courant et son groupe avec `id`. Notez que ceci fonctionne malgré le `doas` puisque le *backtick* est évalué par le shell **avant** l'exécution de la ligne en `root`.

En répétant ces longues opérations pour toutes les plateformes qui nous intéressent à présent et dans le futur, notre `$CROSSBASE` ressemblera à :

```
$ ls ..
dest.arm64/   dest.armv7/   dest.macppc/  dest.riscv64/
obj.arm64@    obj.armv7@    obj.macppc@   obj.riscv64@
src/
```

Nous pouvons alors nous pencher sereinement sur la tâche la plus importante et potentiellement répétitive.

2. COMPILATION DE KERNEL

L'un des principaux avantages de la OPi 5 max (et de ses consœurs), en dehors des 8 cœurs du RK3588, est la possibilité d'utiliser un support de stockage NVMe, beaucoup (vraiment beaucoup) plus rapide que la SD/MMC. Notez au passage qu'ici, une carte MMC est toujours utilisée, pour accueillir le bootloader U-Boot capable de charger le firmware UEFI depuis le NVMe. Lui-même étant chargé par le U-Boot (ne supportant pas UEFI) placé en flash SPI et qui, pour l'heure, est une version spécifique à Orange Pi.

Ceci ira beaucoup plus vite que tout ce qui a précédé, car les sources du noyau sont beaucoup moins étoffées que celles d'une chaîne de compilation complète. La première chose à faire avant de nous lancer est de définir un certain nombre de variables d'environnement permettant au `Makefile` de retrouver ses petits. Fort heureusement pour nous, l'une des cibles de `Makefile.cross` est spécialement faite pour cela : `cross-env`. Nous pouvons alors l'utiliser pour directement exporter les variables en question :

```
$ eval export $( make -f Makefile.cross \
TARGET=${CROSSTARGET} \
CROSSDIR=${CROSSBASE}/dest.${CROSSTARGET} \
cross-env )
```

Ceci devra être fait à chaque fois que l'environnement n'est pas initialisé et donc à l'ouverture d'un terminal (ou d'une session SSH), dès lors que vous souhaitez *cross-compiler*. Il en va, bien entendu, de même pour nos propres variables `CROSSBASE`, `CROSSTARGET` et `CROSSCONF`.

Comme vous le savez peut-être, une construction de noyau OpenBSD se passe en deux temps avec une configuration suivie d'une compilation. Contrairement à un système « local » où un simple `config` suivi du fichier de configuration (ici, dans `$CROSSCONF`) dans le bon répertoire suffit, nous devons ici prendre en considération les chemins particuliers que nous avons définis. Ceci étoffe sensiblement la ligne de commande, mais, dans les grandes lignes, le principe est le même :

```
$ config -b ${CROSSBASE}/obj.${CROSSTARGET}/sys/arch/${CROSSTARGET}/compile/${CROSSCONF} \
-s ${CROSSBASE}/src/sys \
sys/arch/${CROSSTARGET}/conf/${CROSSCONF}
```

`-b` et `-s` permettent respectivement de préciser les répertoires de construction et des sources qui, par défaut, seraient `../compile/GENERIC.MP` et `compile/GENERIC.MP/../../../..` (quatre répertoires au-dessus du répertoire de construction par défaut). Nous pouvons alors nous rendre dans le répertoire de construction, créé pour nous, et lancer la compilation avec :

```
$ cd ${CROSSBASE}/obj.${CROSSTARGET}/sys/arch/${CROSSTARGET}/
compile/${CROSSCONF}
$ make -j `sysctl -n hw.ncpuonline`
[...]
text    data    bss     dec     hex
7240316 223536  706384  8170236 7caafc
mv bsd bsd.gdb
```

Ceci ne prendra, avec ma configuration matérielle, qu'une petite minute pour obtenir les binaires attendus :

```
$ file bsd
bsd:  ELF 64-bit LSB executable, AArch64, version 1
```

Et, en termes de performances, si vous vous posez la question du `hw.smt`, voici quelques données. D'abord, sans *Hyper-Threading* :

```
real    1m15.026s
user    5m37.400s
sys     1m8.650s
```

Et ensuite avec (avec un `make clean` entre, bien entendu) :

```
real    1m6.899s
user    9m44.260s
sys     1m53.920s
```

Quelques secondes de différences, qui certes peuvent devenir des minutes pour la chaîne de compilation, ne sont pas quelque chose qu'on pourrait qualifier de réellement significatif. Du moins, on ne divise clairement pas par deux le temps de construction, mais obtenons juste ~10 % de différence.

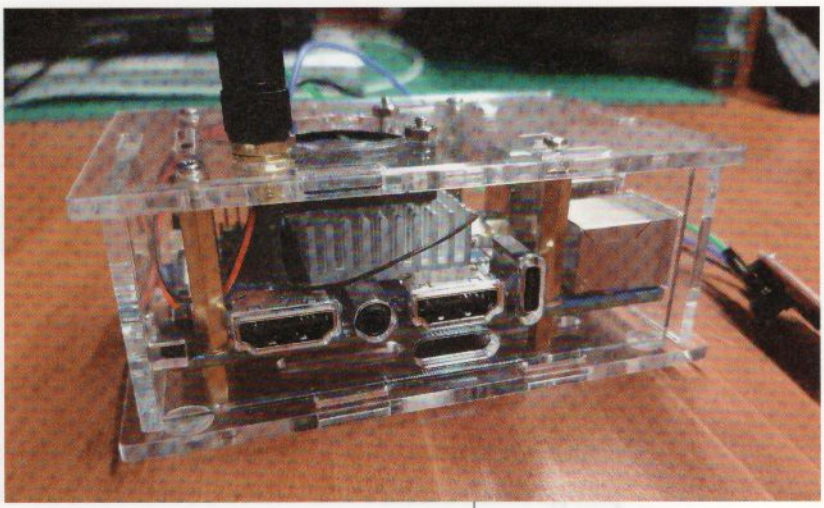
3. POUR FINIR

Nous nous sommes surtout intéressés ici au noyau, car c'est la problématique ayant déclenché la rédaction autour du sujet, mais il est également possible de construire le *userland* avec :

```
$ doas make -j `sysctl -n hw.ncpuonline` -f Makefile.cross \
TARGET=${CROSSTARGET} CROSSDIR=${CROSSBASE}/dest.${CROSSTARGET} \
cross-distrib
```

– Cross-compilation d'OpenBSD : c'est mal (tm), mais c'est pas grave... –

Ceci a d'ailleurs permis de mettre le doigt sur un problème n'apparaissant, *a priori*, que lors d'une *cross-compilation* avec `share/tabset/Makefile` référençant le mauvais répertoire (`${.CURDIR}/obj` au lieu de `${.OBJDIR}`). Donc, non seulement cela permet de jouer et de tester des modifications sur les outils intégrés au système, mais montre bien aussi que, finalement, faire des choses qui ne sont pas « *for general use* » n'est pas une si mauvaise idée que cela.
Au moins, ça permet de remonter des *bugs* et celui-ci en particulier est maintenant corrigé.



La carte dispose de deux sorties HDMI pour l'instant non supportées par OpenBSD et/ou le Device Tree, on doit se contenter de la console série et de SSH. Il en va de même pour l'USB (EHCI et xHCI), même si le pilote est bien présent dans le noyau. Adapter le Device Tree (DTS) du noyau Linux (et/ou d'U-Boot) version Orange Pi n'est pas une mince affaire, en particulier en l'absence de schéma du circuit qui pourrait expliciter clairement quelle GPIO du SoC pilote quel régulateur pour alimenter chaque composant (PHY, en particulier)...

Pour la petite histoire, l'Orange Pi 5 max fonctionne... partiellement. Toujours pas d'USB pour le moment, mais le système est installé sur le NVMe et l'interface réseau RTL8125 est correctement prise en charge. Il reste bien du travail à faire, étant donné que le *Device Tree* est réduit à sa plus basique expression, mais ça avance. Dans l'état, la carte peut donc effectivement déjà *builder* système et ports pour ses cousines ARM64 moins puissantes (Pi3 et Pi4, par exemple). Victoire partielle, donc.

Et enfin, peut-être vous demandez-vous « *mais pourquoi donc s'amuser avec un système comme OpenBSD alors qu'il y a pléthore de distributions GNU/Linux parmi lesquelles choisir ?* »
Une partie de la réponse est donnée en tout début d'article : c'est un système spécial et différent. À cela s'ajoute, bien sûr, la sécurité, la concision, la philosophie très UNIX et enfin, la communauté, tournée vers l'essentiel, technique, experte, et disons-le franchement, ne s'encombrant pas de considérations qui pourraient sortir du cadre de ce que chacun peut apporter à l'ensemble (une « avenante méritocratie », si vous voulez). Qui sait, tout ceci pourrait aussi vous plaire à vous aussi, alors pourquoi ne pas faire l'essai ? ;) **DB**

RÉFÉRENCES

[1] **https://www.openbsd.org/faq/faq5.html**

[2] **https://www.openbsd.org/anoncvs.html**

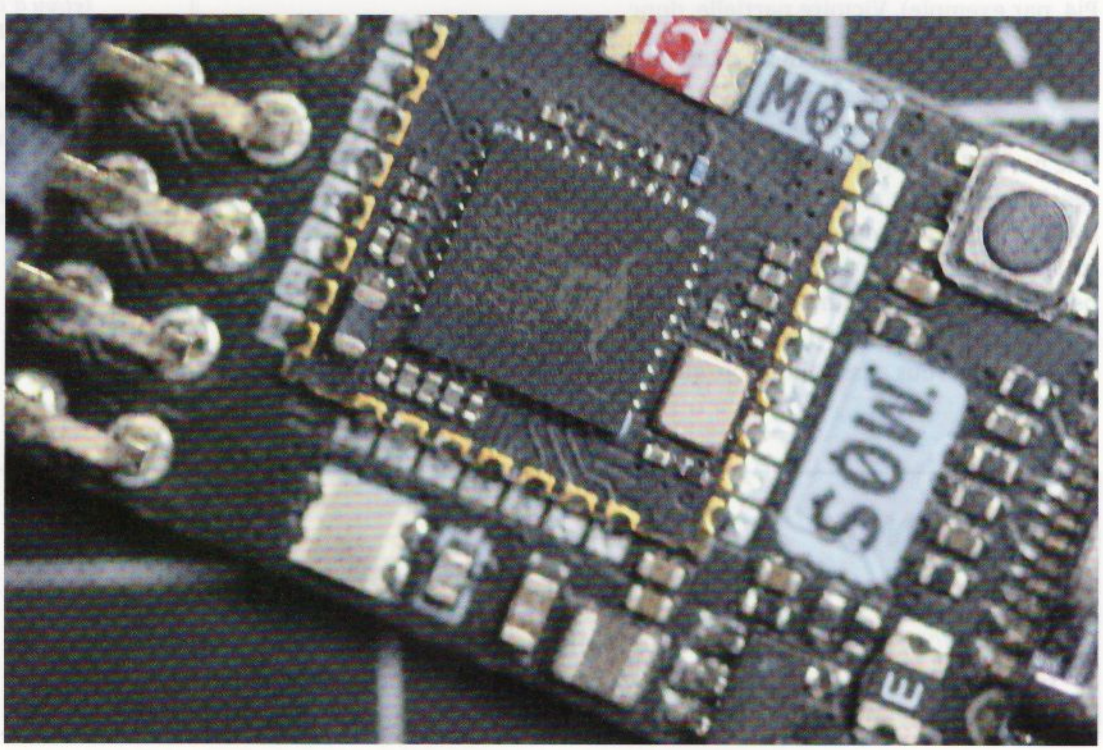
[3] **https://github.com/openbsd/src**

[4] **https://connect.ed-diamond.com/gnu-linux-magazine/glmf-269/ecrire-son-premier-pilote-pour-openbsd**

SIPEED SLOGIC COMBO 8 : UN MULTIOUTIL TRÈS UTILE... UN JOUR

Denis Bodor

Dans la trousse à outils de l'amateur de systèmes embarqués et de programmation sur microcontrôleur se trouvent généralement plusieurs accessoires indispensables : l'adaptateur USB/série, le programmeur/débogueur JTAG/SWD ou encore l'analyseur logique USB. Le fabricant chinois Sipeed a eu l'excellente idée de réunir ces trois éléments dans un unique périphérique, absolument minuscule. C'est le SLogic Combo 8.



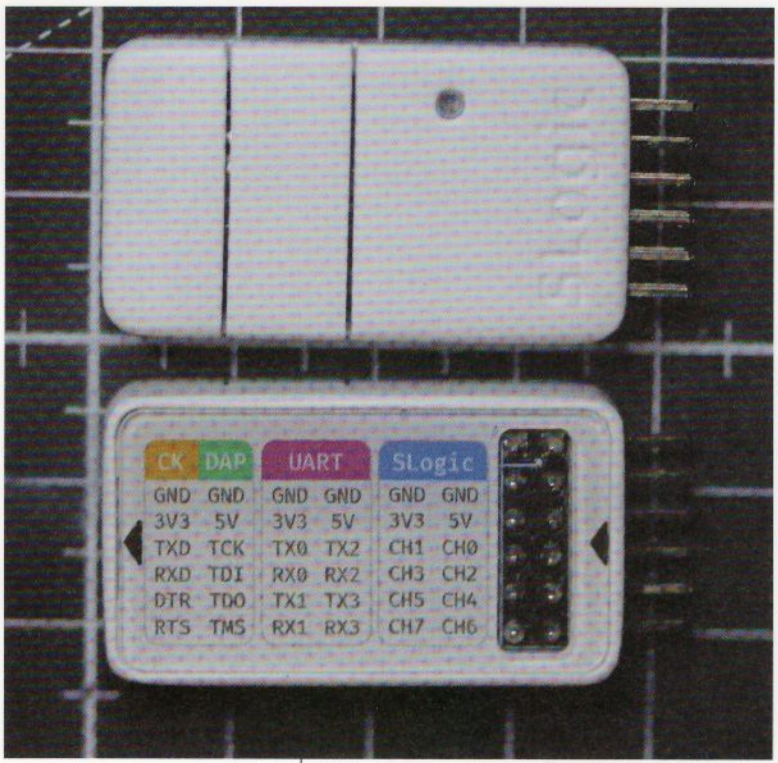
Un seul périphérique pouvant servir de couteau suisse pour la manipulation de microcontrôleurs ou de cartes SoC, traînant en permanence sur le *bench*, le bureau ou dans la poche semble être à la fois un gain de temps et d'ergonomie. Imaginez simplement ne plus avoir à jongler entre les appareils et passer d'une fonction à l'autre par une simple pression sur un bouton. Voilà précisément ce que propose le SLogic Combo 8 de Sipeed **[1]**.

L'objet se présente sous la forme d'un boîtier plastique de très petite taille (36 mm x 20 mm x 10 mm), disponible en blanc et en noir, présentant d'un côté un port USB-C (câble A vers C fourni) et de l'autre deux rangés de 6 connecteurs (au pas de 2,54 mm), avec, à l'arrière, un bouton et une LED. L'utilisation physique du périphérique est relativement simple, vous le connectez à votre machine de développement (PC, Mac ou SBC) et vous appuyez sur le bouton jusqu'à ce que la LED présente la couleur associée à la fonction que vous souhaitez utiliser :

- rouge : mode interface série. Vous disposez alors de pas moins de 4 ports série (ttyACM0 à ttyACM3), disposant chacun de lignes RX et TX sur les broches, ainsi que de deux masses et deux lignes d'alimentation (3,3V et 5V). Les deux premiers sont capables d'assurer une vitesse de communication allant jusqu'à 20 M/s et les deux autres jusqu'à 1 M/s. Ceci est peu ou prou l'équivalent d'une connexion de 4 adaptateurs USB/série type CDC/ACM. ID USB = 359f:3101.

Le fait d'avoir en façade les différents modes, le brochage et le code couleur de la LED est une brillante idée.

- bleu : mode analyseur logique. Les broches disponibles deviennent 8 entrées logiques (0-3,6V) capables d'un taux d'échantillonnage de 40 MSamp/s (80 MSamp/s en utilisant seulement 4 et 8 entrées, et théoriquement 160 MSamp/s avec 2). Les broches de masse et d'alimentation restent identiques au mode série. ID USB = 359f:0300.
- vert : mode programmeur/débogueur DAP. Le périphérique devient une sonde JTAG/SWD utilisant le protocole CMSIS-DAP v2.x permettant une utilisation, par exemple avec OpenOCD, pour programmer des microcontrôleurs à base de CPU ARM Cortex M, comme une carte Raspberry Pi Pico et contrôler l'exécution des programmes (*debug* avec GDB). Le protocole DAP existe en deux versions, la première présente une interface USB-HID et la seconde nécessite un contrôle direct en USB. Le SLogic Combo 8 ne propose que la version 2 (pas d'USB-HID, donc). Côté brochage, on retrouve les signaux classiques d'une sonde JTAG avec TCK, TMS, TDI et TDO. Notez que dans ce



« Pourquoi se priver quand on peut avoir le double pour seulement deux fois le prix ? » - S. R. Hadden (John Hurt) - Contact (1997).

mode, les quatre broches restantes deviennent celles d'une interface série (CDC/ACM) disposant, en plus, des signaux DTR/RTS pour le contrôle de flux matériel. ID USB = d6e7:3507.

- jaune : mode programmeur/débogueur CK-Link. Similaire au mode précédent, mais le protocole de communication entre le périphérique et l'hôte est CK-Link, généralement utilisé sur MCU RISC-V avec un outil comme T-Head DebugServer. Ceci semble être relativement courant avec les MCU chinois, mais extrêmement rare (pour l'instant) sur les plateformes généralement utilisées en Occident. Les signaux disponibles sur les broches sont identiques au mode DAP puisqu'il s'agit également d'une interface JTAG/SWD, seul le protocole entre le périphérique et l'hôte change. Le port série CDC/ACM complémentaire et les signaux de contrôle de flux ne changent pas non plus. ID USB = 42bf:b210.

À noter que la documentation présente sur le wiki de Sipeed précise une plage de tensions en entrée (*Signal Input Range*) uniquement pour le mode analyseur logique, avec 0 à 3,6 volts et des spécifications pour l'état haut et bas respectivement à VIH>2V et VIL<0,8V. Il n'est donc précisé nulle part les tensions pour les autres modes (très certainement identiques avec un fonctionnement en sortie à 0-3,3V), ni si le matériel est tolérant au 5 volts (probablement pas).

Ce matériel est disponible sur les sites habituels (comprendre AliExpress, Amazon et consorts), et j'ai acquis mes deux exemplaires sur le store AliExpress de Sipeed **[2]** pour 18 € pièce en août (aujourd'hui, 21 €), mais d'autres offres le proposent autour de 15 € à ce jour.

1. ESSAIS DU COMBO 8

1.1 Mode série

Il n'y a pas grand-chose à dire sur ce mode puisqu'il ne s'agit, après tout, que d'un quadruple adaptateur USB/série. Mais nous pouvons, cependant, profiter de la vitesse de communication exceptionnelle pour procéder à quelques expérimentations qui pourraient, peut-être, vous rendre service un jour.

Pour mettre en œuvre le Combo 8 dans ce mode, nous allons tout simplement établir une liaison série entre

deux machines (PC, Mac, SBC, Pi, peu importe) utilisant un Unix libre (GNU/Linux, Raspberry Pi OS, OpenBSD, peu importe également). Nous connectons deux Combo 8 dans ce mode (LED rouge) aux hôtes et les relions via de simples connecteurs Dupont : masse sur masse, RX0 sur TX0, et TX0 sur RX0. Nous n'obtenons rien de moins qu'un câble null modem USB.

Sur les deux systèmes, nous installons Picocom et Lrzsz (paquets `picocom` et `lrzsz` avec Debian ou RPi OS). Pourquoi Picocom et pas GNU Screen ? Parce que Screen ne sait pas facilement échanger des fichiers (mais ça reste possible, voir Hackable 5 **[3]**), et que Minicom semble avoir un problème dès qu'on tente d'imposer une vitesse de communication supérieure à 115200 bps ou non standard. De plus, il s'avère que Picocom est vraiment un excellent outil et remplace désormais mon choix par défaut de Minicom, lui-même ayant remplacé celui de Screen il y a quelques années. Lrzsz, pour sa part, nous permet de disposer des protocoles d'échange de fichiers XMODEM, YMODEM et ZMODEM que les lecteurs les plus... heu... *vintage* (comme moi) auront connus il y a bien longtemps en utilisant des BBS.

Notre test sera relativement simple puisqu'il s'agira de transférer un fichier quelconque de quelque 18 Mio entre les deux systèmes en ZMODEM. Pour cela, nous lançons Picocom sur les deux hôtes avec :

```
$ piccocom -b 9830400 -f n /dev/ttyACM0
```

Les débits les plus couramment utilisés pour les liaisons série sont 2400, 4800, 9600, 19200, 38400, 57600, 115200, 230400... Au-delà, nous arrivons dans un territoire particulier où il n'existe pas réellement de norme. J'ai donc opté pour un multiple de 9600 bps, soit 9600 x 1024 = 9830400. Et si vous vous posez la question, oui, j'ai testé avec 19660800 (soit la limite maximum de 20 Mio/s des deux premiers ports série) et la liaison est relativement instable, très certainement en raison de l'utilisation de la connectique elle-même (câbles de 20 cm non blindés, non torsadés), plus que du matériel Sipeed.

Une fois Picocom lancé des deux côtés, nous pouvons utiliser :

- Ctrl+A, Ctrl+R et valider sans spécifier de nom (ZMODEM n'en a pas besoin) pour attendre les données, côté receveur ;
- Ctrl+A, Ctrl+S et saisir le chemin vers le fichier à envoyer côté émetteur.

Le transfert se met en route et, une fois, terminé, nous voyons côté receveur :

```
$ rz -vv
Receiving: plop.mkv
Bytes received: 18756069/18756069   BPS:936825
Transfer complete
```

Et émetteur :

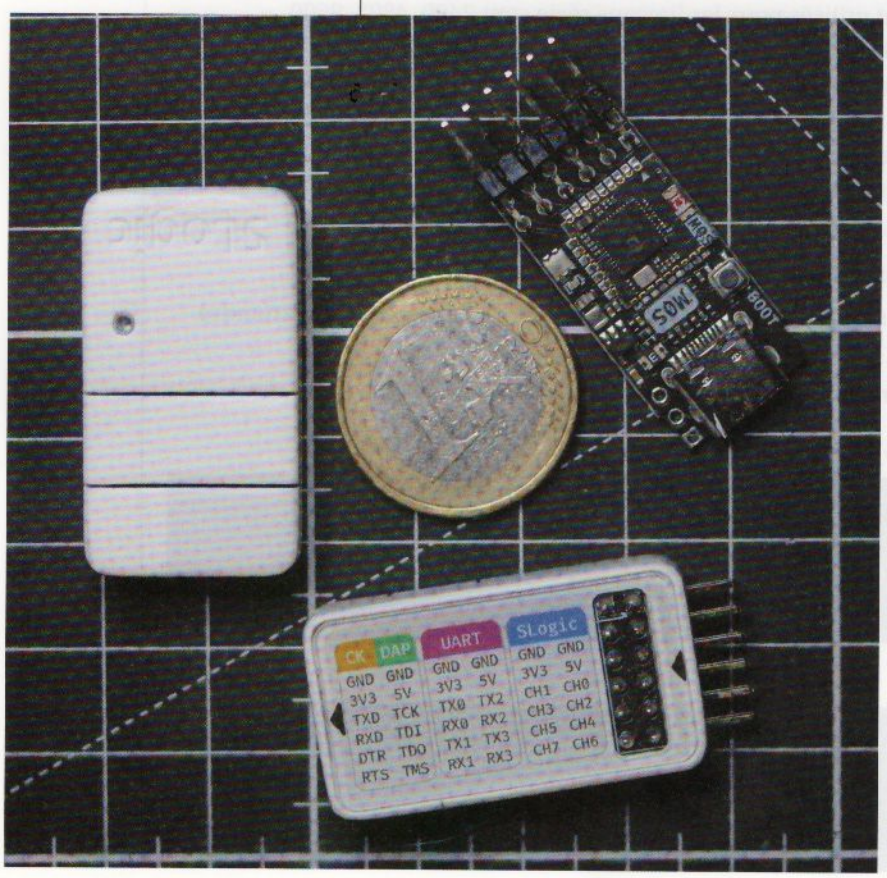
```
$ sz -vv /tmp/plop.mkv
Sending: plop.mkv
Bytes Sent:18756069   BPS:936698
Transfer complete
```

Quelque 914 Kio/s est on ne peut plus raisonnable et effectivement, correspond au débit spécifié en bits par secondes avec 8 bits de données, 1 bit de *start* et un bit de stop (sans parité). Nous pouvons donc valider ce mode et estimer qu'un quadruple convertisseur USB/série dans un tel format est déjà, en soi, un avantage intéressant.

1.2 Mode DAP : JTAG/SWD

Pour tester ce mode consistant à utiliser le Combo 8 comme une sonde de *debug* JTAG, nous nous limiterons au protocole DAP. Je n'ai, en effet, aucun outil utilisant CK-Link et, je pense, vous non plus. Le principal intérêt de ce mode est de pouvoir remplacer aisément une sonde JTAG plus volumineuse pour faciliter la mise au point de programmes sur des architectures ARM, et des cartes microcontrôleurs en particulier. CMSIS-DAP ne concerne pas directement la plateforme cible, accédée via JTAG et/ou SWD, mais la communication entre la sonde et la machine de développement et constitue une solution pour « standardiser » ce type de liaisons. Avant CMSIS-DAP (pour *Common Microcontroller Software Interface Standard - Debug Access Port*), chaque modèle de sonde nécessitait un support dédié dans les logiciels comme OpenOCD, PyOCD, Keil MDK, SEGGER Embedded Studio, etc. DAP est en train de devenir la norme et, par exemple, l'outil de mise au point Raspberry Pi DebugProbe (anciennement PicoProbe) utilise également CMSIS-DAP. De plus, une implémentation de démonstration, *open source*, nommée DAPLink sert de référence et facilite donc la popularisation du système **[4]**.

Bien sûr que j'en ai démonté un ! Et la surprise est de taille, ce qui se trouve dans les entrailles du SLogic Combo 8 n'est autre que le devkit de Sipeed pour le Maix M0S.



Comme précisé précédemment, il existe deux versions du protocole, une reposant sur USB-HID et l'autre sur un accès direct au matériel en USB. Le Combo 8 ne supporte pas la version USB-HID, mais comme nous allons le voir dans un instant, CMSIS-DAPv2 est parfaitement utilisable avec OpenOCD.

Notre victime pour cet essai sera une Raspberry Pi Pico, disposant d'un emplacement pour un connecteur à souder à trois broches proposant une interface SWD (*Serial Wire Debug*). La Pico possède donc un port SWD, mais la sonde ne parle que le JTAG. Pas de problème cependant, puisqu'il existe une correspondance de signaux entre les deux interfaces (il existe même une séquence permettant à la sonde de spécifier au MCU le standard à utiliser) :

- SWDIO = TMS, échange de données ;
- SWCLK = TCK, horloge ;
- SWO = TDO (optionnel, inexistant sur Pico), communication unidirectionnelle complémentaire (MCU vers sonde/hôte).

Contrairement à ce qu'il était nécessaire de faire du temps de PicoProbe (Pico flashée avec le *firmware* la transformant en sonde de *debug*), nous pouvons ici utiliser un OpenOCD parfaitement standard tel que proposé par le système de gestion de paquets de notre Debian. En effet, la version *upstream* supporte à la fois CMSIS-DAP et le RP2040 (mais pas le RP2350 de la Pico2, voir **[5]**). Nous pouvons donc connecter le Combo 8 à la Pico et le tout à la machine de développement et :

```
$ openocd -f interface/cmsis-dap.cfg -f target/rp2040.cfg
Open On-Chip Debugger 0.12.0
[...]
Info : Using CMSIS-DAPv2 interface with
       VID:PID=0xd6e7:0x3507, serial=012345ABCDEF
Info : CMSIS-DAP: SWD supported
Info : CMSIS-DAP: JTAG supported
Info : CMSIS-DAP: Atomic commands supported
Info : CMSIS-DAP: Test domain timer supported
Info : CMSIS-DAP: FW Version = 2.1.0
Info : CMSIS-DAP: Interface Initialised (SWD)
Info : SWCLK/TCK = 0 SWDIO/TMS = 0
       TDI = 1 TDO = 1 nTRST = 0 nRESET = 1
Info : CMSIS-DAP: Interface ready
Info : clock speed 100 kHz
Info : SWD DPIDR 0x0bc12477, DLPIDR 0x00000001
Info : SWD DPIDR 0x0bc12477, DLPIDR 0x10000001
Info : [rp2040.core0] Cortex-M0+ r0p1 processor detected
Info : [rp2040.core0] target has 4 breakpoints, 2 watchpoints
Info : [rp2040.core1] Cortex-M0+ r0p1 processor detected
Info : [rp2040.core1] target has 4 breakpoints, 2 watchpoints
Info : starting gdb server for rp2040.core0 on 3333
Info : Listening on port 3333 for gdb connections
Info : starting gdb server for rp2040.core1 on 3334
Info : Listening on port 3334 for gdb connections
Info : accepting 'telnet' connection on tcp/4444
[...]
```

OpenOCD a parfaitement reconnu le Combo 8 et la cible. Nous pouvons alors utiliser un autre terminal pour lancer une connexion Telnet locale et obtenir des informations sur la cible :

```
$ telnet 127.0.0.1 4444
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
Open On-Chip Debugger
> targets
    TargetName    Type       Endian TapName     State
--  ------------ ---------- ------ ------------------
0* rp2040.core0  cortex_m   little rp2040.cpu  running
1  rp2040.core1  cortex_m   little rp2040.cpu  running

> flash list
{name rp2040.flash driver rp2040_flash base 268435456
size 0 bus_width 0 chip_width 0 target rp2040.core0}
{name rp2040.alias driver virtual base 268435456 size
0 bus_width 0 chip_width 0 target rp2040.core1}

> halt
[rp2040.core0] halted due to debug-request,
current mode: Thread
xPSR: 0x21000000 pc: 0x10000a84 msp: 0x20041fc0

> flash probe rp2040_flash
Found flash device 'win w25q16jv' (ID 0x001540ef)
RP2040 B0 Flash Probe:
2097152 bytes @0x10000000, in 32 sectors

flash 'rp2040_flash' found at 0x10000000

> exit
```

Il est, bien entendu, également possible d'utiliser GDB (port 3333) pour tracer, inspecter et analyser le code actuellement développé, directement sur le RP2040. Mais nous pouvons aussi nous servir d'OpenOCD pour programmer un *firmware* dans la flash :

```
$ openocd -f interface/cmsis-dap.cfg \
  -f target/rp2040.cfg \
  -c "targets rp2040.core0; program picoledusb.elf \
  verify reset exit"
[...]
** Programming Started **
Info : Found flash device
       'win w25q16jv' (ID 0x001540ef)
Info : RP2040 B0 Flash Probe: 2097152 bytes
       @0x10000000, in 32 sectors
```

```
Info : Padding image section 1 at 0x10004c0c with
       244 bytes (bank write end alignment)
Warn : Adding extra erase range,
       0x10004d00 .. 0x1000ffff
** Programming Finished **
** Verify Started **
** Verified OK **
** Resetting Target **
shutdown command invoked
```

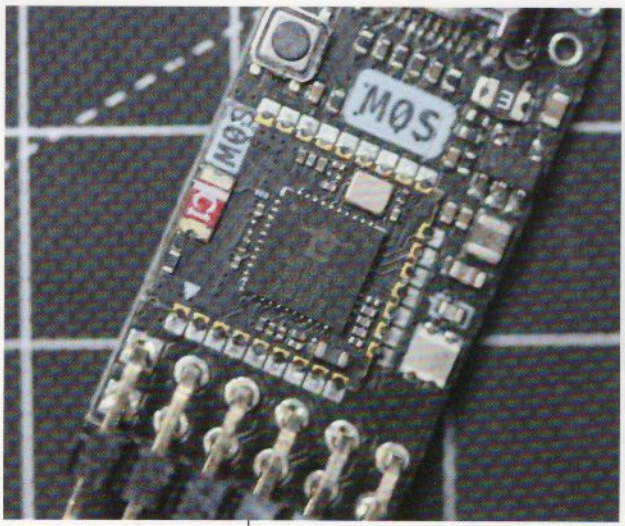
Le fonctionnement du Combo 8 en tant qu'outil de mise au point est donc validé, et nous avons donc bien entre les mains deux outils très utiles, combinés en un seul. Serait-ce un sans-faute ?

1.3 Mode analyseur logique

Je vais répondre de suite à la question : non, pas pour l'instant. Le mode analyseur logique propose des caractéristiques très intéressantes et bien supérieures à ce que, par exemple, un périphérique basé sur le MCU Cypress EZ-USB FX2 peut offrir. Nous avons à disposition 8 canaux et une fréquence d'échantillonnage entre 40 et 160 MSamp/s (pour respectivement 8, 4 et 2 canaux simultanés), avec un débit allant jusqu'à 320 Mb/s. Tout ceci est très séduisant, mais encore faut-il disposer du logiciel adéquat et, ô surprise, il s'agit de PulseView, reposant sur la libsigrok, du projet éponyme **[6]**. Une solution *open source*, développée de longue date, supportant déjà une longue liste de matériel allant des clones Saleae Logic aux fameux Kingst LA2016/LA5016/LA5032 en passant par l'Openbench Logic Sniffer, les vrais Saleae 8 et 16, ou encore le Buspirate de Dangerous Prototypes.

Sipeed propose PulseView à la fois pour Windows et pour GNU/Linux sous la forme d'une AppImage, un binaire statique à la mode macOS, incluant tout le nécessaire et ne nécessitant aucune installation. Et c'est là que le bât blesse. En effet, même si cela fonctionne correctement et avec des performances tout à fait acceptables pour un matériel de ce prix, jusqu'à très récemment (12/09/2024), Sipeed était tout bonnement en train de violer les clauses de la licence GPL v3 en reprenant le travail d'un projet libre et en intégrant ses modifications sans rediffuser les sources. Pire encore, le PulseView/libsigrok de Sipeed en version AppImage ne supporte **que** le SLogic Combo 8 et rien d'autre, obligeant l'utilisateur à jongler entre la version proposée par son système et celle, modifiée, de Sipeed.

Cette situation est presque passée inaperçue depuis mars 2023, moment auquel un certain *taorye* propose un *patch* pour libsigrok pour un matériel appelé alors « *SLogic Lite8* » (en plus du SLogic Combo 8). Suite aux échanges et demandes de modifications, et avec une massive contribution de Martin Herren qui maintient sa propre version, subitement le dépôt source disparaît début 2024, ce qui clôt la proposition (PR) sur GitHub. Le problème, bien entendu, est le fait que *taorye* semble intimement connaître le *firmware* du matériel et que personne d'autre n'est en mesure de compléter les fonctionnalités encore absentes ou de corriger les dysfonctionnements liés à la gestion du matériel lui-même et de son protocole.



Ce microcontrôleur est le BL616 de Bouffalo Lab. Un cœur RISC-V RV32GCP à 320 MHz avec 480 Kio de SRAM, 4 Mio de flash, I2C, UART, SPI, Bluetooth 5.2, Wi-Fi 6 et USB HighSpeed OTG.

Et c'est là que nombre de contributeurs et utilisateurs se rendent compte que l'approche de Sipeed s'inscrit en violation de la GPL. S'en suivent quelques e-mails et demandes directs pour finalement aboutir à la mise à disposition en septembre 2024 d'un dépôt GitHub public par Sipeed : **https://github.com/sipeed/sigrok_slogic**, avec pour principal contributeur... *taorye*, et un code inchangé depuis 2 à 5 ans en moyenne. Que ceci soit bien clair, c'est la réaction de la communauté, en particulier vis-à-vis d'un *tweet* de Sipeed à propos du matériel **[7]** qui a provoqué cette « mise en conformité ». Ceci est inacceptable.

À ce stade, rien n'a bougé et aucune nouvelle contribution n'a été proposée au projet sigrok. Le dépôt maintenu par Sipeed contient une version de libsigrok relativement ancienne par rapport à l'actuelle version de développement, et un travail d'adaptation est nécessaire. Mais ce n'est pas tout, le comportement de ce *fork* n'est pas totalement stable et, dans l'état, ne peut être considéré comme fiable. Le premier pas consistant à respecter les termes de la licence libre est une chose, mais le protocole de communication n'est pas documenté et l'évolution du code semble, à ce stade, peu probable. De ce fait, l'intégration dans sigrok est très incertaine à ce jour. Sans compter le fait que ces modifications devront ensuite passer dans la version stable, qui elle-même devra être utilisée par les responsables de paquets des distributions.

On constate donc que ceci semble être, comme trop souvent pour ce style de matériel prometteur et économique, un *one shot*, sans suivi, mise à jour ou évolution. Quelqu'un a développé un *firmware* et un support logiciel pour mettre le matériel en production et en vente, et on passe à la suite.

En ce qui me concerne, la partie analyseur logique, celle la plus intéressante du Combo 8, n'est pas réellement utilisable. Ce PulseView AppImage nécessite non seulement plusieurs redémarrages en cas de captures répétées et, soyons honnête, est d'ores et déjà obsolète.

2. CONCLUSION ET AVIS

Ce produit s'avère matériellement très intéressant, en particulier pour son prix. Mais force est de constater que son avenir en tant qu'outil universellement adopté est compromis. Il mériterait un *firmware* alternatif *open source*, comme c'est le cas pour le *fx2lafw* des analyseurs à base de Cypress FX2, et c'est une éventualité techniquement envisageable. Le MCU autour duquel il est construit est un Bouffalo Lab BL616, un processeur RISC-V (RV32GCP) agrémenté de bon nombre de périphériques (BT 5.2, Wi-Fi 6, USB OTG, etc.), et disposant

d'un SDK *open source* sous licence Apache 2.0 **[8]**. Au cœur du Combo 8 se cache un module Maix M0S de Sipeed **[9]**, dont le M0S Dock (la carte d'accueil) n'est autre que le Combo 8 lui-même. Il existe même un dépôt GitHub **[10]**, peuplé d'exemples, permettant de prendre en main la plateforme. Il suffirait donc que Sipeed décide d'ouvrir le code du *firmware* du Combo 8 pour débloquer la situation et faire de ce matériel l'un des fleurons de la « flotte » d'outils supportés par sigrok/PulseView. C'est dommage, tellement dommage... **DB**

RÉFÉRENCES

[1] https://wiki.sipeed.com/hardware/en/logic_analyzer/combo8/index.html

[2] https://sipeed.fr.aliexpress.com/store/911876460

[3] https://connect.ed-diamond.com/Hackable/hk-005/transferer-des-fichiers-entre-pc-et-raspberry-pi-sans-reseau

[4] https://github.com/ARMmbed/DAPLink

[5] https://github.com/raspberrypi/openocd

[6] https://sigrok.org/wiki/Main_Page

[7] https://x.com/SipeedIO/status/1833809669120315779

[8] https://github.com/bouffalolab/bouffalo_sdk

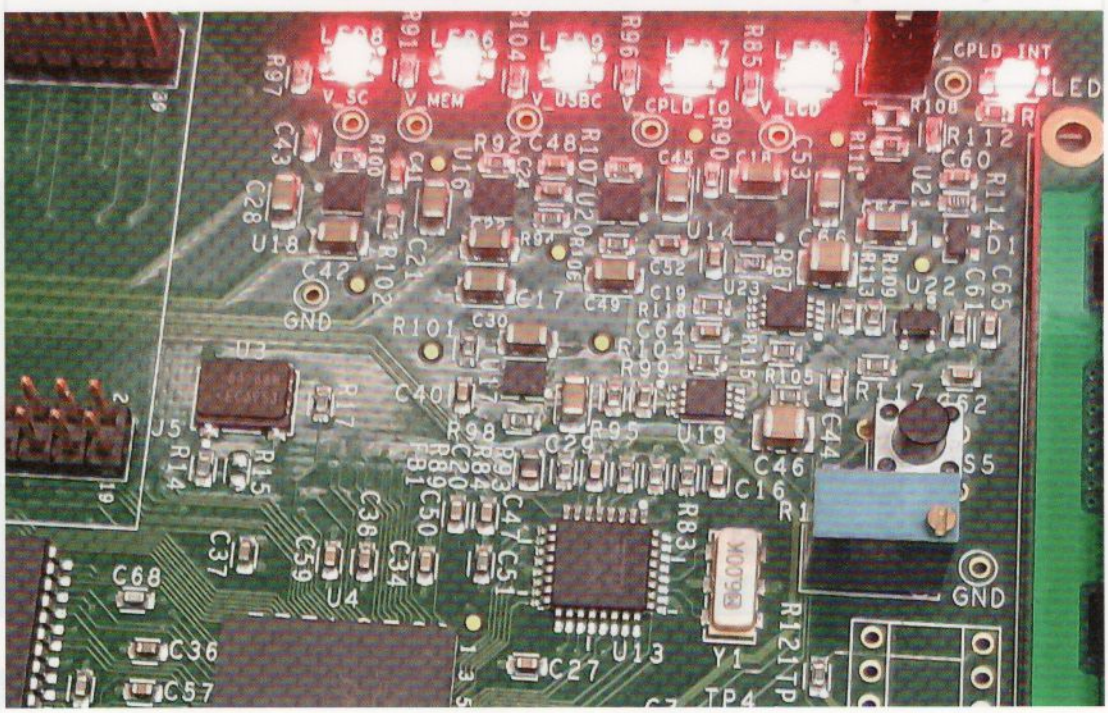
[9] https://wiki.sipeed.com/hardware/en/maixzero/m0s/m0s.html

[10] https://github.com/sipeed/M0S_BL616_example

FPGA FACILE : PETITE PRÉSENTATION ET PRISE EN MAIN DE LITEX

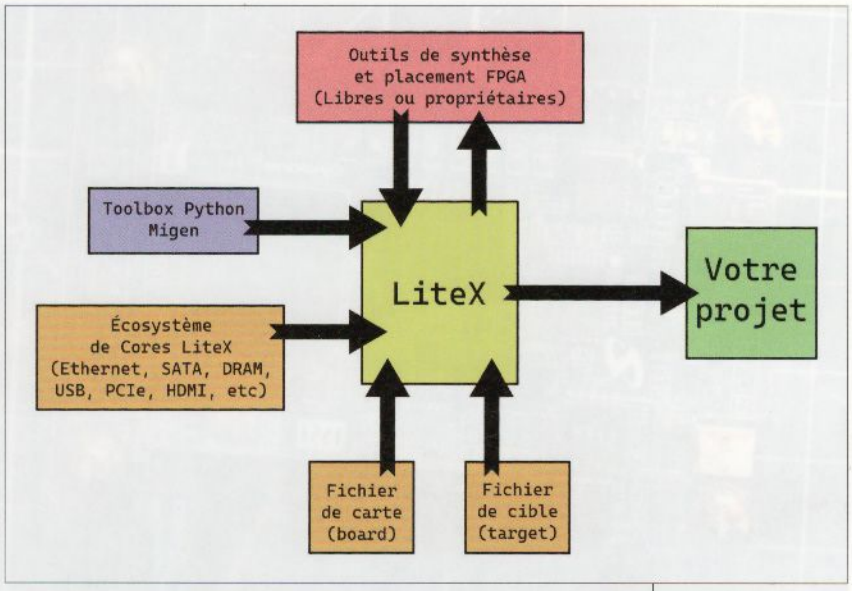
Denis Bodor

LiteX est un projet qui aurait dû faire l'objet d'un article depuis bien longtemps, puisqu'il s'agit sans doute de la façon la plus simple d'aborder le monde des circuits logiques programmables (PLD), et des FPGA en particulier. Plus exactement, ce framework permet de composer son matériel avec des briques élémentaires, sans utiliser de Verilog ou de VHDL et d'obtenir, au final, un système embarqué ou un périphérique sur mesure qu'on peut ensuite programmer. Découvrons cela...



Nous l'avons vu dans le numéro 55 **[1]**, l'univers des FPGA est absolument fantastique, mais il nécessite souvent un investissement personnel, en temps, en énergie et en finances, qui est loin d'être négligeable. Même si l'aspect budgétaire pèse de moins en moins lourd dans la balance, avec des kits et cartes relativement étoffés et à un prix tout à fait abordable aujourd'hui, même à titre individuel, il s'agit là d'un environnement très différent de celui consistant simplement à utiliser et programmer des microcontrôleurs et des SoC (avec ou sans OS). L'autre chose que nous avions constatée est, qu'en pratique, composer un système ou un périphérique avec des langages de description comme Verilog ou VHDL implique une grosse part de réutilisation de composants, ou *IP cores*, existants. Dans notre petit projet, nous avions assemblé un *core* Z80, l'UART et de la mémoire (RAM et ROM) de cette façon, sans rien réellement concevoir à ce niveau. Le code VHDL avait principalement servi pour créer le « liant », connectant ces éléments entre eux en quelque chose de fonctionnel.

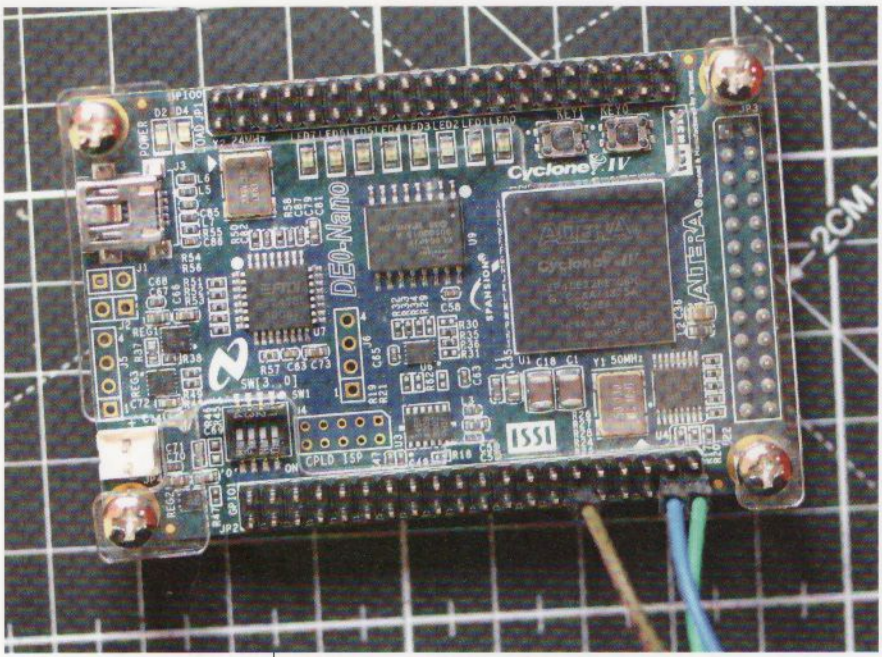
Cette logique, qui existe sans doute depuis aussi longtemps que les circuits logiques programmables eux-mêmes, est également celle de LiteX,



Architecture générale de LiteX.

mais poussée au point d'en faire l'objet même du projet. LiteX est un *framework*, créé par la société française (Bretonne, même) EnjoyDigital, basé sur tout un écosystème de composants sous licence libre (*BSD-2-Clause*), comme le projet lui-même. Vous pouvez voir cela comme le pendant matériel d'un système de construction comme Buildroot, mais appliqué au matériel. Les composants incluent de nombreuses pièces essentielles comme une interface Ethernet, un contrôleur SATA, un bus SPI ou encore le support de la RAM dynamique (DRAM), qui se greffent sur un *softcore* (processeur programmé dans un design FPGA) pouvant être de l'OpenRISC 1000, du LatticeMico32, et bien entendu du RISC-V (picorv32 de Clifford Wolf, VexRiscv, RocketChip, BlackParrot, NEORV32, ou Minerva décrit en nMigen).

La plupart des composants et périphériques portés par le projet LiteX sont écrits en Migen (condensé de *Milkymist generator*), une boite à outils proposée sous la forme de modules Python permettant de décrire des composants. La philosophie ici est de tirer parti de la richesse d'un langage de programmation orienté objet et de l'appliquer à la conception de circuits. On décrit le composant avec du code Python pour ensuite générer une description HDL qui servira pour la synthèse, le placement et le routage (avec un outil libre



La carte DE0-Nanon de Terasic est un classique du genre pour s'initier aux FPGA. Bien que le Cyclone IV intégré et ses 22320 éléments logiques commencent à se faire vieillissants, celle-ci est toujours vendue et constitue toujours un choix relativement intéressant.

ou non). La description binaire, ou *bitstream*, résultante pourra ensuite être chargée/programmée dans un FPGA, là encore avec un utilitaire d'un constructeur ou une solution *open source* (comme le fantastique openFPGALoader).

Notons cependant qu'il est également possible d'intégrer des descriptions écrites dans des langages plus traditionnels dans un projet LiteX, comme VHDL, Verilog ou SystemVerilog, ainsi que nMigen et Spinal-HDL. Et inversement, on peut aussi utiliser LiteX pour créer une base de travail pour ensuite intégrer le résultat, en Verilog, dans une réalisation et un processus plus « classique ».

Enfin, pour conclure cette introduction, et puisque LiteX est vraiment une solution visant à simplifier les choses de bout en bout, il est non seulement possible de faire fonctionner sa réalisation dans un simulateur (Verilator), mais aussi, et surtout, sur une carte de développement physique. La liste des cartes et FPGA directement supportés est relativement conséquente et on y retrouve tous les classiques du genre, des Xilinx (maintenant AMD) aux Altera (maintenant Intel), en passant par les Lattice ECP5 et iCE40, les FPGA Gowin (Sipeed Tang nano), ou encore les Trions d'Efinix. Bien entendu, comme LiteX ne s'occupe pas directement de la synthèse, les outils et chaînes de compilation externes sont nécessaires, mais directement utilisés lors du processus de construction. Je le répète, LiteX n'est ni une solution pour remplacer les logiciels propriétaires des constructeurs par de l'*open source*, ça, c'est le travail d'autres projets comme Yosys, NextPNR ou Apicula, ni même une chaîne de compilation, ça, c'est le travail de GCC et/ou Clang/LLVM.

En parlant de compilation justement, même si LiteX n'est pas lié uniquement à la construction de systèmes reposant sur un SoC *softcore* et qu'il est parfaitement possible de créer des périphériques sans CPU, la plupart des exemples reposent sur l'utilisation d'une telle architecture, le plus souvent à base de *softcore* VexRiscv. En ce sens, LiteX se charge non seulement de créer le SoC et les périphériques qui y sont rattachés, mais également de la partie logicielle, en produisant, au minimum, un « BIOS », exécuté de base par la plateforme construite.

Mais tout ceci deviendra plus clair avec un exemple concret, que nous allons voir de ce pas...

1. PRÉREQUIS ET INSTALLATION

Vous l'avez compris, LiteX repose sur de nombreux éléments externes ainsi que sur Python, ce qui induit également un certain nombre de dépendances. Fort heureusement, avec une distribution digne de ce nom comme Debian, Devuan ou encore Ubuntu, tout le nécessaire est installable sous la forme de paquets. Notez que ceci n'a pas été testé sur Raspberry Pi OS, pour deux raisons. D'une part, selon le FPGA utilisé, il n'existe pas nécessairement de solutions *open source* pour la synthèse (typiquement Altera/Intel Cyclone) et le constructeur ne fournit quasi systématiquement que des binaires pour Windows et GNU/Linux sur x86/AMD64, et non ARM. D'autre part, la synthèse, le placement et le routage permettant d'obtenir le *bitstream* à programmer dans le FPGA sont un processus lourd, nécessitant des ressources (RAM et CPU) très conséquentes. Un SBC, même aussi puissant qu'une Pi5, n'est, par définition, pas un bon choix pour une plateforme de développement de ce genre.

Pour utiliser LiteX, vous devrez donc installer 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 \
git zlib1g-dev python3.11-venv
```

En plus de tout ceci, vous devrez également disposer de la suite logicielle associée au matériel que vous possédez et/ou une alternative *open source* pour synthétiser. Dans le cadre de cet article, j'ai opté pour deux plateformes : une carte Tang Nano 20K de Sipeed équipée d'un FPGA Gowin GW2AR-18, ainsi qu'un DE0-Nano de Terasic construit autour d'un Altera/Intel Cyclone IV EP4CE22F17C6N. La première utilise l'IDE Gowin ou les outils du projet Apicula **[2]** (non testés), et la seconde nécessitera la suite Intel Quartus Prime Lite Edition. Dans les deux cas, ceci est téléchargeable gratuitement depuis le site du constructeur et les utilitaires permettant une utilisation en ligne de commande (`gw_sh` ou `quartus_sh`) doivent se trouver dans le `PATH`, même temporairement. Notez que je recommande de placer le chemin en question **à la fin** du `PATH`, car par exemple, Quartus Prime intègre une version spécifique d'OpenOCD qui n'est pas nécessairement compatible (ou assez complète) avec ce qu'attend LiteX.

Une fois tout ceci installé, on pourra se pencher sur LiteX lui-même, mais là encore, cela se fera sans avoir à toucher quoi que ce soit pouvant nuire au système de gestion de paquets, grâce à l'utilisation d'un environnement virtuel Python. Nous commençons donc par créer cet environnement, l'activer et nous placer dans le répertoire correspondant :

```
$ python3 -m venv ~/LITEX
$ source /home/denis/LITEX/bin/activate
$ cd ~/LITEX
$ pip3 install meson
Collecting meson
  Using cached meson-1.5.2-py3-none-any.whl (968 kB)
Installing collected packages: meson
Successfully installed meson-1.5.2
```

Nous pourrons quitter cet environnement à tout moment en utilisant la commande `deactivate` et y revenir plus tard en sourçant à nouveau `bin/activate`. Nous récupérons ensuite le script d'installation et de configuration, directement depuis le dépôt GitHub, le rendons exécutable et procédons à l'installation :

```
$ wget https://raw.githubusercontent.com/\
enjoy-digital/litex/master/litex_setup.py
$ chmod +x litex_setup.py
$ ./litex_setup.py --init --install
          __   _ __      _  __
         / /  (_) /____ | |/_/
        / /__/ / __/ -_)>  <
       /____/_/\__/\__/_/|_|
     Build your hardware, easily!
          LiteX Setup utility.

[   0.001] LiteX Setup auto-update...
[   0.001] Initializing Git repositories...
[   0.001] --------------------------------
[   0.001] Cloning migen Git repository...
Clonage dans 'migen'...
remote: Enumerating objects: 12709, done.

[...]

[  63.795] Installing pythondata-cpu-vexriscv-smp Git repository...
Obtaining file:///home/denis/LITEX/pythondata-cpu-vexriscv-smp
  Preparing metadata (setup.py) ... done
Installing collected packages: pythondata-cpu-vexriscv-smp
  Running setup.py develop for pythondata-cpu-vexriscv-smp
Successfully installed pythondata-cpu-vexriscv-smp-1.0.1.post325
```

Git sera utilisé pour directement récupérer les composants et les *IP cores* de LiteX et les placer dans le répertoire courant. Faisant également partie du lot, figurent les fichiers de définition pour les cartes et les plateformes matérielles actuellement supportées, ainsi que les *softcores* (CPU) utilisables. Tout ceci se retrouve ici, sous la forme de répertoires (et dépôt Git) avec :

- les composants LiteX : `litedram/`, `liteeth/`, `liteiclink/`, `litejesd204b/`, `litepcie/`, `litesata/`, `litescope/`, `litesdcard/`, `litespi/` ;
- les *softcores* : `pythondata-cpu-lm32/`, `pythondata-cpu-mor1kx/`, `pythondata-cpu-naxriscv/`, `pythondata-cpu-serv/`, `pythondata-cpu-exiiriscv/`, `pythondata-cpu-vexriscv/`, `pythondata-cpu-vexriscv-smp/` ;
- les cartes et plateformes cibles, ainsi que les sondes JTAG (pour OpenOCD) : `litex-boards/litex_boards/platforms/`, `litex-boards/litex_boards/targets/`, `litex-boards/litex_boards/prog`.

On retrouve également les répertoires `bin/`, `include/` et `lib/` faisant partie de l'environnement, nous permettant de disposer d'outils propres à LiteX, leurs bibliothèques et dépendances. Et tout ce petit monde est confiné dans le répertoire d'installation, sans interférer avec le reste du système. C'est beau quand tout est fait proprement et est bien rangé...

À ce stade, et grâce au simulateur Verilator, vous pouvez déjà procéder à un premier test en simulant une architecture de démonstration basée sur le *softcore* VexRiscv avec :

```
$ litex_sim --cpu-type=vexriscv
INFO:SoC:         __   _ __      _  __
INFO:SoC:        / /  (_) /____ | |/_/
INFO:SoC:       / /__/ / __/ -_)>  <
INFO:SoC:      /____/_/\__/\__/_/|_|
INFO:SoC:  Build your hardware, easily!
INFO:SoC:----------------------------------------
INFO:SoC:Creating SoC... (2024-09-30 15:40:36)
INFO:SoC:----------------------------------------
INFO:SoC:FPGA device : SIM.
INFO:SoC:System clock: 1.000MHz.
[...]
Build machine cpu family: x86_64
Build machine cpu: x86_64
Host machine cpu family: riscv
Host machine cpu: vexriscv
Target machine cpu family: riscv
Target machine cpu: vexriscv
Compiler for C supports arguments -nostdlib: YES
Checking if "long double check" compiles: YES
[...]
ROM usage: 23.55KiB     (18.40%)
RAM usage: 1.62KiB      (20.21%)
rm crt0.o
make : on quitte le répertoire
   " /home/denis/LITEX/build/sim/software/bios "
INFO:SoC:Initializing ROM rom with contents (Size: 0x5e3c).
INFO:SoC:Auto-Resizing ROM rom from 0x20000 to 0x5e3c.
[...]
[ethernet] loaded (0x555b25009ef0)
[gmii_ethernet] loaded (0x555b25009ef0)
[jtagremote] loaded (0x555b25009ef0)
[serial2console] loaded (0x555b25009ef0)
[serial2tcp] loaded (0x555b25009ef0)
[clocker] loaded
[spdeeprom] loaded (addr = 0x0)
[xgmii_ethernet] loaded (0x555b25009ef0)
[clocker] sys_clk: freq_hz=1000000, phase_deg=0


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

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

BIOS built on Sep 30 2024 15:43:12
BIOS CRC passed (3d448994)

LiteX git sha1: 87ee6ec3a
--=============== SoC ==================--
CPU:            VexRiscv @ 1MHz
BUS:            wishbone 32-bit @ 4GiB
CSR:            32-bit data
ROM:            128.0KiB
SRAM:           8.0KiB
--============== Boot ==================--
Booting from serial...
Press Q or ESC to abort boot completely.
sL5DdSMmkekro
Timeout
No boot medium found
--============= Console ================--

litex>
```

LiteX va ici faire beaucoup de choses, puisqu'il ne se contente pas de simplement générer du Vérilog qui sera simulé, mais créera également un programme de base servant de « BIOS » ou de moniteur permettant de charger des programmes. Ce que vous avez là, sous les yeux, est un code en C compilé et exécuté par un processeur RISC-V simulé en compagnie d'un lot de périphériques, tous faisant partie du projet LiteX. L'interface qui se présente à vous vous permet d'utiliser des commandes (`help`) pour tester la mémoire, l'inspecter ou la modifier, mais aussi de contrôler le SoC simulé. Vous pouvez quitter cette « console série » avec un simple Ctrl+C.

Plus récente dans le petit monde des FPGA, la série des Tang Nano de Sipeed est une alternative peu chère et relativement riche. Ici, la Tang Nano 20k ayant fait office de victime pour nos expérimentations. Remarquez le logo LiteX sur la boite. La carte arrive, en effet, flashée par défaut avec précisément ce que nous construisons dans cet article.

Si vous n'obtenez pas ce résultat, assurez-vous en premier lieu que l'installation s'est correctement déroulée et qu'il n'existe pas d'éléments manquants n'ayant pu être téléchargés. Il y en a beaucoup, une erreur passe facilement inaperçue. Dans le doute, utilisez la commande `./litex_setup.py --update` qui, même s'il n'y a pas de mise à jour, vous assurera de récupérer ce qui manque (via Git). Une autre source de

problèmes peut être une dépendance non satisfaite. J'ai donné en début de procédure une liste de paquets à installer qui couvre tous les besoins au moment de la rédaction de cet article, mais les choses auront peut-être évolué entre temps. Lisez attentivement les messages d'erreurs précisant le fichier ou la commande absente, elle y est forcément listée (c'était le cas pour `zlib.h` par exemple, m'ayant fait ajouter `zlib1g-dev` à la liste des paquets à la dernière minute).

2. FAISONS ÇA AVEC DU VRAI MATÉRIEL, SVP

Simuler, c'est bien (et nécessaire), mais l'objectif final est tout de même de voir un beau *softcore* RISC-V fonctionner sur une carte bien physique. C'est là qu'on remarquera le phénoménal travail achevé par les développeurs de LiteX, car rendre quelque chose simple est sans le moindre doute ce qui demande le plus d'efforts et de compétences. En effet, à ce stade, vous avez déjà appliqué le plus gros des manipulations nécessaires et passer du virtuel au réel est l'affaire d'une poignée de commandes.

Notre victime sera une carte Tang Nano 20K de Sipeed (environs 35 € sur AliExpress **[3]**), ne nécessitant aucun programmeur ou sonde JTAG complémentaires. Il vous faudra télécharger l'IDE Gowin, également utilisable en ligne de commande, et l'installer quelque part dans votre système, tout en le rendant accessible via le `PATH` :

```
$ export PATH="$PATH:/chemin/GowinTang/IDE/bin"
```

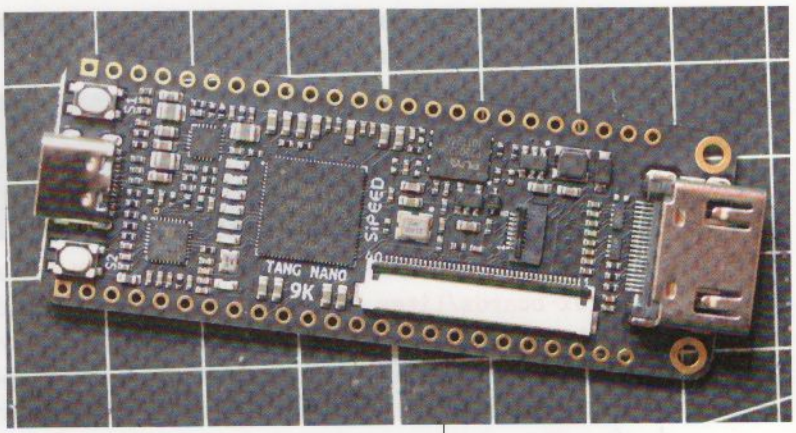
Ou mettre quelque chose d'équivalent dans votre `~/.bashrc`. Et ensuite, LiteX fera le travail pour vous, puisqu'il suffit de se glisser dans le sous-répertoire `litex-boards/litex_boards/targets/`, de repérer le script Python correspondant à la carte (parmi les quelque 180 présents !) et l'exécuter avec les bons arguments :

```
$ ./sipeed_tang_nano_20k.py \
--toolchain gowin \
--output-dir build_tang20k \
--build --load
[...]
[5%] Running netlist conversion ...
Running device independent optimization ...
[10%] Optimizing Phase 0 completed
[15%] Optimizing Phase 1 completed
[25%] Optimizing Phase 2 completed
Running inference ...
[30%] Inferring Phase 0 completed
[40%] Inferring Phase 1 completed
[50%] Inferring Phase 2 completed
[55%] Inferring Phase 3 completed
Running technical mapping ...
[60%] Tech-Mapping Phase 0 completed
[65%] Tech-Mapping Phase 1 completed
```

```
[75%] Tech-Mapping Phase 2 completed
[80%] Tech-Mapping Phase 3 completed
[90%] Tech-Mapping Phase 4 completed
GowinSynthesis finish
Running placement......
[10%] Placement Phase 0 completed
[20%] Placement Phase 1 completed
[30%] Placement Phase 2 completed
[50%] Placement Phase 3 completed
Running routing......
[60%] Routing Phase 0 completed
[70%] Routing Phase 1 completed
[80%] Routing Phase 2 completed
[90%] Routing Phase 3 completed
Running timing analysis......
[95%] Timing analysis completed
Placement and routing completed
Bitstream generation in progress......
Bitstream generation completed
Running power analysis......
[100%] Power analysis completed
[...]
Done
DONE
Jtag frequency : requested 2.50MHz   -> real 2.00MHz
erase SRAM Done
Flash SRAM: [==============================] 100.00%
Done
```

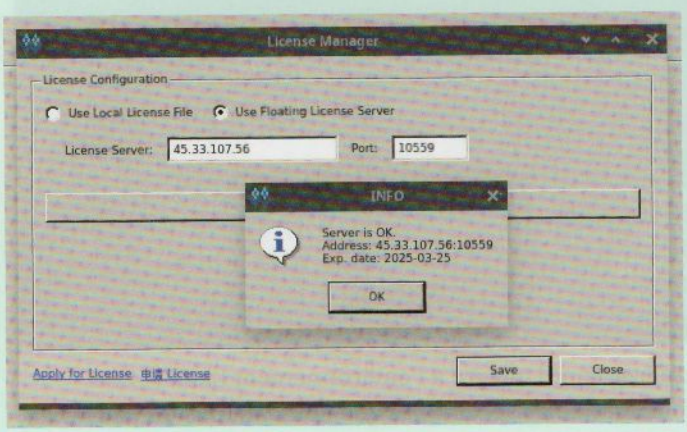
Il est possible, si vous venez d'installer l'IDE Gowin, que vous obteniez un message d'erreur juste avant le début de la synthèse : « *License verification failed Server not responding.[...] OSError: Error occured during Gowin's script execution* » (oui, les messages d'erreur Python sont illisibles, mais il y a pire, Java par exemple). Selon la version que vous avez choisi d'utiliser, une licence, même temporaire, doit être vérifiée et, pour une raison mystérieuse, le serveur ne semble pas toujours disponible. Pour régler le problème (comme détaillé dans la documentation Sipeed **[4]**), lancez l'IDE en mode graphique (`gw_ide`) puis, dans l'interface, passez par ***Help*** et ***Manage License***. Dans la fenêtre qui apparaît figure une adresse IP et un port. Si celle déjà présente ne fonctionne pas, pointez un navigateur sur **http://gowinlic.sipeed.com/** et utilisez l'une de celles qui s'affichent. Vous devez obtenir un message comme celui-ci pour « activer » la licence :

Les options utilisées sont sensiblement différentes de celles précisées dans la documentation en raison de préférences personnelles, tout en sachant que chaque script dans ce répertoire peut en ajouter qui lui sont propres, comme ici avec `--toolchain` précisant la « suite » à utiliser pour synthétiser le *bitstream*. Vous pouvez obtenir la liste des options disponibles en invoquant le script avec `--help`. `--build` et `--load` parlent d'elles-mêmes en permettant respectivement de construire le *bitstream* (avec le code compilé du BIOS en « ROM ») et de le charger dans le composant (ou la flash associée). `--output-dir` en revanche est une préférence et permet de stocker tous les fichiers résultants du processus de construction (*software* et *hardware*) dans un répertoire explicitement désigné, et non simplement `build/`. Ceci afin de ne pas tout mélanger en travaillant éventuellement avec plusieurs plateformes.



La Tang Nano 9k est la petite sœur de la 20k. Plus modeste en termes de ressources proposées, elle est aussi plus économique (~16 €) tout en étant parfaitement adaptée pour faire connaissance avec LiteX.

Les quelques lignes en fin du présent extrait sont celles résultant de l'utilisation d'openFPGALoader, qui est utilisé par défaut pour charger le *bitstream* en SRAM du composant (`--flash` ajoutera l'option `-fr` à l'appel pour une écriture en flash). Le fichier contenant ce *bitstream* est `build_tang20k/gateware/sipeed_tang_nano_20k.fs`, que vous pourrez d'ailleurs utiliser indépendamment de la construction, avec openFPGALoader, par la suite.



Validez le message, cliquez ***Save***, fermez la fenêtre, puis l'IDE. Le tout sera normalement prêt à être utilisé, y compris en ligne de commande. De la même manière, si une IP ne répond pas toujours, optez simplement pour l'autre (actuellement, les deux serveurs sont 106.55.34.119:10559 et 45.33.107.56:10559).

Notez qu'il est possible également d'invoquer un script d'une autre façon, tout en restant à la racine de l'arborescence LiteX, avec :

```
$ python3 -m litex_boards.targets.sipeed_tang_nano_20k \
--toolchain gowin --output-dir build_tang20k \
--build --load
```

Le résultat sera le même, mais `build_tang20k/` sera créé à cet endroit plutôt que dans `litex-boards/litex_boards/targets/`, ce qui a tendance à déranger Git, puisque seul `build/` est dans le `.gitignore` (ce qui est logique). Si vous souhaitez conserver et inspecter le répertoire de construction, mieux vaudra peut-être procéder ainsi. Notez qu'ici, en l'absence de l'option `--output-dir`, les fichiers issus de la construction trouvent place automatiquement dans un sous-répertoire de `build/`, ce qui assure également une distinction par défaut entre plateformes/cartes.

Une fois le *bitstream* chargé dans le périphérique, on pourra se connecter à ce dernier avec un outil comme `screen` ou `minicom`. Dans le cas du Tang Nano 20k, une première interface série (`/dev/ttyUSB0` en général) représente l'accès JTAG, mais la seconde (`/dev/ttyUSB1`) vous permettra d'accéder au « BIOS » LiteX (115200 8N1) où vous obtiendrez la même chose qu'avec la simulation précédente. Quelques commandes supplémentaires permettent de manipuler le matériel spécifique au design pour cette carte/plateforme :

```
litex> help
LiteX BIOS, available commands:
leds              - Set Leds value
buttons           - Get Buttons value
[...]
sdram_mr_write    - Write SDRAM Mode Register
sdram_test        - Test SDRAM
sdram_init        - Initialize SDRAM (Init + Calibration)

litex> ident
Ident: LiteX SoC on Tang Nano 20K 2024-09-30 16:39:55

litex> mem_list
Available memory regions:
ROM       0x00000000 0x20000
SRAM      0x10000000 0x2000
MAIN_RAM  0x40000000 0x800000
CSR       0xf0000000 0x10000
```

Il sera cependant plus judicieux d'utiliser l'outil intégré à l'environnement LiteX (`litex_term /dev/ttyUSB1`) apportant des fonctionnalités spécifiques que nous verrons un peu plus loin. Si vous ne voyez pas les messages de démarrage, en particulier lors d'une seconde connexion, utilisez simplement la commande `reboot`.

3. UN AUTRE EXEMPLE : DE0-NANO

Les plateformes à base de FPGA Intel/Altera nécessitent souvent un programmeur comme cet USB Blaster. Ce n'est pas le cas pour DE0-Nano qui intègre, de base, une telle interface accessible via le connecteur mini-USB.

Répéter la procédure avec un FPGA Alter/Intel Cyclone sera une opération très similaire à celle avec la carte Tang Nano, mais diffèrera, bien entendu, par la suite logicielle à installer et à utiliser pour la synthèse. Là encore, il faudra déployer l'applicatif propriétaire du constructeur qui, s'il est bien dans le `PATH`, sera automatiquement utilisé par LiteX, y compris l'outil permettant de transférer le *bitstream* dans la mémoire du composant (`quartus_pgm`).

La construction du projet sera, cette fois, directement lancée avec :

```
$ python3 -m litex_boards.targets.terasic_de0nano \
--uart-name=jtag_uart --build --load
```

Celle-ci devrait se solder par la création du fichier `build/terasic_de0nano/gateware/terasic_de0nano.sof` (pour *SRAM Object File*) qui sera ensuite transféré dans le FPGA pour le configurer.

Contrairement à la Tang Nano, le DE0-Nano ne dispose pas d'une interface série reliée au chip USB, mais fait usage d'un « câble de téléchargement » USB-Blaster (une sonde JTAG) directement intégré à la plateforme et accessible via le connecteur mini-USB. Pour tout de même accéder au « BIOS » LiteX, nous pouvons nous reposer sur OpenOCD en spécifiant une option spécifique de `litex_term` :

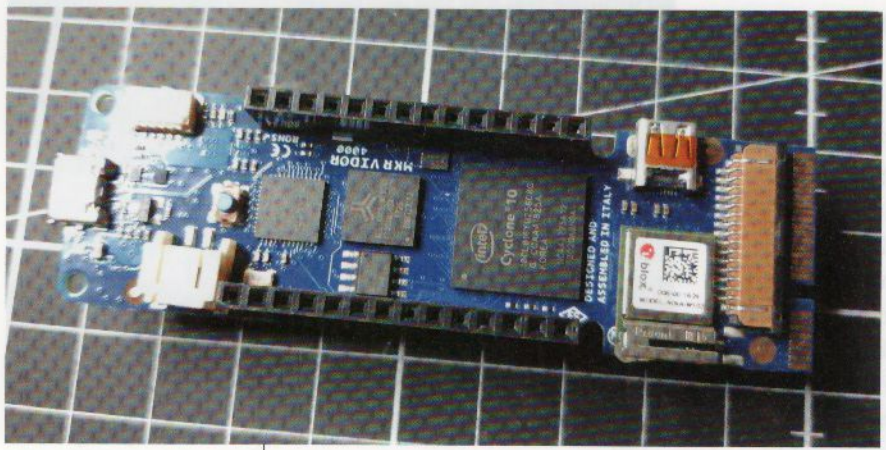
```
$ litex_term --jtag-config \
litex-boards/litex_boards/prog/openocd_usb_blaster.cfg \
jtag
```

Là encore, nous obtenons le même résultat que précédemment avec, plus ou moins, les mêmes commandes à notre disposition :

```
litex> ident
Ident: LiteX SoC on DE0-Nano 2024-10-01 06:07:13

litex> mem_list
Available memory regions:
ROM      0x00000000 0x20000
SRAM     0x10000000 0x2000
MAIN_RAM 0x40000000 0x2000000
CSR      0xf0000000 0x10000
```

Notez le volume de `MAIN_RAM` bien supérieur à ce qu'on trouve avec la Tang Nano 20k.



Parmi les quelque 180 cartes supportées de base par LiteX, on retrouve des surprises, comme cette Arduino MKR Vidor 4000 intégrant un FPGA Cyclone 10CL016. Cette plateforme, introduite en 2018 par le projet Arduino, n'a pas rencontré un franc succès et n'est aujourd'hui plus proposée à la vente.

4. LE BIOS C'EST BIEN, DU « VRAI » CODE, C'EST MIEUX

Nous avons à présent un SoC RISC-V complet fonctionnant dans un FPGA et l'avons construit et « exécuté » d'une manière qui n'a jamais été aussi simple, démontrant précisément la raison d'être de LiteX. De façon presque involontaire, nous avons également compilé et exécuté du code C pour RISC-V. La preuve :

```
$ file  ./build/terasic_de0nano/software/bios/bios.elf
./build/terasic_de0nano/software/bios/bios.elf:
  ELF 32-bit LSB executable, UCB RISC-V, soft-float ABI,
  version 1 (SYSV), statically linked,
  with debug_info, not stripped
```

La version purement binaire, `bios.bin`, est celle intégrée dans la « ROM » du SoC et que vous voyez s'exécuter automatiquement en vous connectant via la liaison série. Mais nous pouvons aller plus loin en faisant en sorte de lancer une « application utilisateur » juste après le BIOS. Pour cela, l'une des solutions, et sans doute la plus simple, est de compiler le binaire de démonstration et le transférer, via la liaison série, dans la mémoire du SoC.

Commençons par construire ce programme :

```
$ litex_bare_metal_demo \
--build-path=build/sipeed_tang_nano_20k
CC       donut.o
CC       helloc.o
CC       crt0.o
CC       main.o
CC       demo.elf
/usr/lib/gcc-cross/riscv64-linux-gnu/12/
  ../../../../riscv64-linux-gnu/bin/ld:
  attention: demo.elf a un segment LOAD avec
  les permissions RWX
chmod -x demo.elf
OBJCOPY  demo.bin
chmod -x demo.bin
```

```
$ file demo/demo.elf
demo/demo.elf: ELF 32-bit LSB executable,
UCB RISC-V, soft-float ABI, version 1 (SYSV),
statically linked, with debug_info, not stripped
```

Vous avez déjà eu un aperçu du principe de fonctionnement du transfert sans le savoir avec le message qui s'affiche au démarrage (réinitialisation du SoC) :

```
--============== Boot ==================--
Booting from serial...
Press Q or ESC to abort boot completely.
sL5DdSMmkekro
Timeout
No boot medium found
```

En réalité, le « BIOS » tente un *boot* du code attendu via la liaison série (« *Booting from serial* »), et ne voyant rien venir (« *Timeout* »), il finit par simplement basculer sur la console interactive. Une option de `litex_term` nous permet cependant de changer ce comportement ou plutôt, de le satisfaire. Tout ce que nous avons à faire est de spécifier une option, suivie du chemin vers le binaire à transférer :

```
$ litex_term /dev/ttyUSB1 --kernel=demo/demo.bin
```

Ou avec le DE0-Nano :

```
$ litex_term --jtag-config \
litex-boards/litex_boards/prog/openocd_usb_blaster.cfg \
--kernel=demo/demo.bin jtag
```

Vous verrez alors le code en question être chargé, puis exécuté :

```
litex>
litex> reboot

[...]
[LITEX-TERM] Received firmware download
   request from the device.
[LITEX-TERM] Uploading demo/demo.bin
   to 0x40000000 (7776 bytes)...
[LITEX-TERM] Upload calibration...
   (inter-frame: 10.00us, length: 64)
[LITEX-TERM] Upload complete (9.9KB/s).
[LITEX-TERM] Booting the device.
[LITEX-TERM] Done.
Executing booted program at 0x40000000
```

```
--============= Liftoff! ===============--

LiteX minimal demo app built Oct  1 2024 09:15:47

Available commands:
help                - Show this command
reboot              - Reboot CPU
led                 - Led demo
donut               - Spinning Donut demo
helloc              - Hello C
```

Vous pourrez alors utiliser l'une des commandes proposées pour voir la démonstration jouée pour vous (`donut` est très rigolo). Une fois l'amusement passé, vous pourrez faire un tour dans `litex/litex/soc/software/demo/` et étudier le code ainsi que le système de construction via `bin/litex_bare_metal_demo`. Ceci pourra être relativement facile à adapter et à personnaliser pour tester du code « maison ».

5. POUR FINIR

Arrêtons-la cette exploration qui n'est, somme toute, qu'une petite introduction à LiteX, mais je pense que la démonstration est faite, et bien faite. Ce projet facilite énormément la conception, ou plutôt la composition, d'un SoC, du *softcore* aux codes exécutés en réutilisant simplement des briques existantes. Je ne suis pas particulièrement enjoué à l'idée d'utiliser Python pour produire mes binaires RISC-V, mais on comprend tout à fait la logique consistant à tout orienter vers ce langage du début à la fin. De plus, rien ne vous empêche de développer votre code C en dehors de LiteX et de tester le binaire via `litex_term`.

Pour la petite histoire, la genèse de cet article provient d'un de mes tweets, montrant cette vieille carte PCI (pas PCIe) à base de CPLD Max-II, utilisée pour la rédaction d'un futur tutoriel de développement de pilotes pour FreeBSD et/ou OpenBSD (pour GNU/Linux Magazine). Parmi les réponses et échanges autour de la difficulté à trouver des IP cores libres se trouvait un commentaire me conseillant de jeter un œil à LiteX et à LitePCIe en particulier. Un week-end d'essais plus tard, l'article était sur les rails...

BONUS : TROIS QUESTIONS À FLORENT KERMARREC DE ENJOY-DIGITAL

▶ **Hackable** : Le monde du design FPGA est relativement « fermé » en général, pourquoi avoir choisi une licence *open source* pour le projet ? Et pourquoi *BSD-2-Clause*, par opposition avec une licence avec *copyleft* ?

Florent : Le monde des FPGA est en effet dominé par des solutions propriétaires. LiteX s'appuie sur Migen et MiSoC, deux outils *open source* développés par M-Labs Ltd, auxquels j'ai initialement contribué. Ces projets étaient déjà sous licence *BSD-2-Clause*, et j'ai donc naturellement poursuivi avec cette licence permissive et offrant une grande liberté d'utilisation, que ce soit pour des projets *open source* ou commerciaux. Avec le recul, je me rends compte que cette ouverture a permis de nouer de nombreuses collaborations et de saisir des opportunités incroyables !

▶ **Hackable** : On trouve de plus en plus de *devkits* FPGA assez économiques, certains sont-ils mieux adaptés pour démarrer sereinement un projet basé sur LiteX ?

Florent : Aujourd'hui, il existe des *devkits* FPGA très accessibles, parfois dès 15 euros, souvent grâce à des cartes du commerce reversées par la communauté. Ces plateformes démocratisent l'accès au FPGA. Pour ceux qui souhaitent démarrer avec LiteX, les cartes comme la Digilent Arty A7 ou les FPGA Lattice ECP5 et iCE40 sont particulièrement adaptées. Ces FPGA Lattice permettent d'ailleurs d'utiliser une *toolchain* entièrement *open source* (Yosys/nextpnr), ce qui offre un gros avantage : des cycles d'itération plus rapides et la possibilité de comprendre et de corriger les *bugs* soi-même.

Nous développons également notre propre *devkit* : la LiteX Acorn *baseboard*. Elle permet de travailler avec des *transceivers* et des interfaces série rapides (PCIe, SFP, SATA, DDR3), ce qui reste assez rare sur les cartes d'entrée de gamme. Elle donne aussi un accès complet aux IP LiteX comme LitePCIe, LiteSATA, LiteEth, et LiteICLink. Un des projets en cours de développement actuellement avec cette carte est une carte réseau PCIe/Ethernet entièrement *open source* qui devrait bientôt être disponible.

▶ **Hackable** : Comment peut-on aider le projet LiteX, y a-t-il des contributions qui sont plus attendues que d'autres ?

Florent : LiteX s'est développé grâce aux contributions de la communauté, et toute aide est la bienvenue. Que ce soit des corrections de *bugs*, de nouvelles fonctionnalités ou des améliorations de la documentation, chaque contribution compte. En ce moment, des exemples d'utilisation sur différentes plateformes seraient particulièrement utiles pour aider les nouveaux utilisateurs à adopter LiteX.

L'*open source* est puissant, car il apporte des idées et des contributions inattendues, rendant le projet riche et stimulant. Pour une petite société comme EnjoyDigital, cela nous permet d'aller bien au-delà de ce que nous aurions pu accomplir seuls !

Je ne me suis, pour l'instant, pas encore penché sur la personnalisation des configurations des SoC, comme par exemple utiliser certaines broches du FPGA du DE0-Nano pour y attacher l'UART et un adaptateur USB/série, et donc éviter un passage par OpenOCD. De la même façon, rencontrant des problèmes avec la connexion série sur Tang Nano 9k (non traités ici), chose que j'avais déjà subie avec le projet z80, je souhaiterai faire cela sur toutes les plateformes utilisées. Un simple coup d'œil à `litex-boards/litex_boards/platforms/sipeed_tang_nano_20k.py` montre clairement toute la simplicité de la chose et les bénéfices apportés par Migen :

```
_io = [
    # Clk / Rst.
    ("clk27",  0, Pins("4"), IOStandard("LVCMOS33")),
    # Serial.
    ("serial", 0,
        Subsignal("rx", Pins("70")),
        Subsignal("tx", Pins("69")),
        IOStandard("LVCMOS33")
    ),
[...]
```

Il s'avère que ceci est même déjà prévu pour le DE0-Nano :

```
# Serial
("serial", 0,
    # Compatible with cheap FT232 based cables
    # (ex: Gaoominy 6Pin Ftdi Ft232Rl Ft232)
    # GND on JP1 Pin 12.
    Subsignal("tx", Pins("JP1:10"), IOStandard("3.3-V LVTTL")),
    Subsignal("rx", Pins("JP1:8"),  IOStandard("3.3-V LVTTL"))
)
```

Malheureusement, la documentation détaillant la façon d'ajouter une carte **[5]** est pour l'heure en *TODO*, mais je ne doute pas un instant qu'en explorant et en cherchant dans les coins, cloner un script de `litex-boards/litex_boards/targets` et `litex-boards/litex_boards/platforms` ne doit pas nécessairement demander beaucoup d'efforts, s'il ne s'agit pas d'une plateforme drastiquement différente ou complexe.

Il reste donc bien des choses à explorer et nous reparlerons donc peut-être de LiteX dans de futurs numéros... **DB**

RÉFÉRENCES

[1] **https://connect.ed-diamond.com/hackable/hk-055/mon-premier-projet-fpga-un-ordinateur-8-bits-complet-en-vhdl**

[2] **https://github.com/YosysHQ/apicula**

[3] **https://fr.aliexpress.com/item/1005006536679799.html**

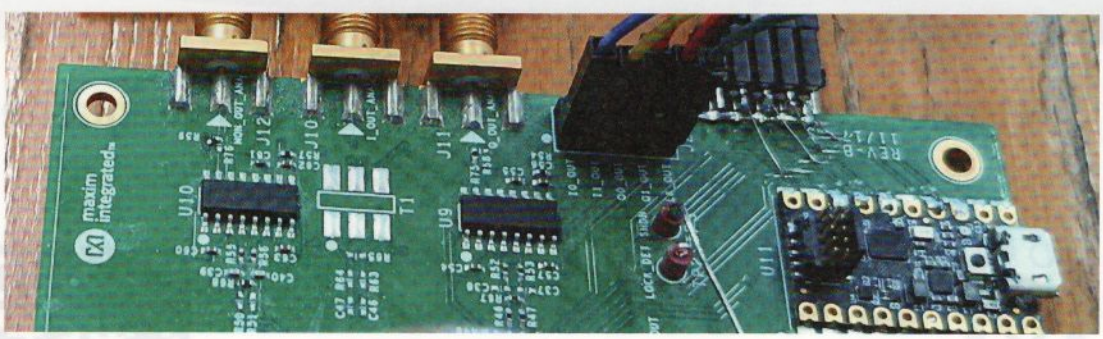
[4] **https://wiki.sipeed.com/hardware/en/tang/Tang-Nano-Doc/install-the-ide.html**

[5] **https://github.com/enjoy-digital/litex/wiki/Add-A-New-Board**

PROGRAMMATION USB SOUS GNU/LINUX : APPLICATION DU FX2LP POUR UN RÉCEPTEUR DE RADIO LOGICIELLE DÉDIÉ AUX SIGNAUX DE NAVIGATION PAR SATELLITE (1/2)

Jean-Michel Friedt, enseignant-chercheur à l'université de Franche-Comté à Besançon

Alors que l'USB est souvent abordé comme un bus émulant un port série, tirer pleinement profit de sa bande passante nécessite d'exploiter les interfaces disponibles les plus appropriées, en particulier Human Interface Device (HID) et transferts en volume (Bulk). Nous proposons d'appréhender le bus USB exposé par le noyau Linux en vue d'en tirer le maximum du débit disponible, et appliquer cette connaissance en réalisant un récepteur de radio logicielle dédié à la réception des signaux de navigation par satellite (GNSS) en bande L (1–2 GHz) grâce au MAX2771. Nous démontrons le bon fonctionnement du circuit avec l'acquisition et le traitement de signaux issus de diverses constellations, de GNSS en orbite intermédiaire MEO et Iridium en orbite basse LEO, observés avec une bande passante pouvant aller jusqu'à 44 MHz.



Notre objectif est de réaliser un récepteur, par radio logicielle, des signaux de navigation par satellite (GNSS pour *Global Navigation Satellite Systems*) et autres signaux satellitaires transmis dans la bande 1–2 GHz qualifiée de bande L. Les modes de modulation les plus récents (Galileo européen, L5 américain) occupant jusqu'à 41 MHz de bande passante (pour Galileo E6, voir **https://gssc.esa.int/navipedia/index.php/Galileo_Signal_Plan**), nous devons aborder le transfert « rapide » de données entre l'interface d'acquisition et l'ordinateur chargé de stocker les mesures – sans prétention de les traiter en temps réel, mais tout de même de capturer tous les échantillons sans en perdre un seul. Pour ce faire, nous devons apprendre à aborder les divers modes de communication du bus USB, et leur mise en œuvre dans un composant de Cypress, le CY7C68013A aussi nommé FX2LP, aussi quelque peu abusivement nommé EZ-USB. Ce composant embarque un vénérable cœur de processeur 8051 **[1]** pour configurer les diverses interfaces de communication : le 8051 est trop petit, avec son accumulateur et demi (A et B) et ses huit (R0–R7) registres 8 bits, pour être abordé par GCC et les codes seront donc compilés au moyen du *Small Device C Compiler* SDCC. Nous nous inspirerons au cours de ces développements du système fonctionnel proposé par Tomoji Takasu de l'université de Tokyo des sciences et technologies marines (*Tôkyô University of Marine Science and Technology*), aussi auteur de RTKLib, célèbre parmi les utilisateurs qui visent la résolution centimétrique de leur récepteur GPS **[2]**. Ce système est nommé PocketSDR et disponible à **https://github.com/tomojitakasu/PocketSDR**. Bien que le circuit soit librement distribué au format KiCad sur ce dépôt, le choix de l'auteur a été d'imposer de travailler dans deux bandes de fréquences différentes – dites haute (L1) et basse (L2, L5) dans la nomenclature du composant MAX2771 chargé de recevoir les signaux GNSS – et d'incorporer l'interface de communication parallèle vers USB FX2LP. Comme l'EZ-USB FX2LP peut être obtenu assemblé et prêt à l'emploi pour un peu moins de 5 euros sur AliExpress quand le composant seul coûte près de 20 euros chez Farnell (code commande 1269134), nous avons fait le choix de travailler sur un circuit prêt à l'emploi (**https://fr.aliexpress.com/item/1005006134347046.html**) et de lui adjoindre l'interface d'acquisition à base de MAX2771, dans un premier temps sous forme de la carte d'évaluation commercialisée par Maxim IC (maintenant Analog Devices), fabricant du composant. Finalement, Tomoji Takasu utilise un compilateur propriétaire pour convertir son code source en binaire à destination du 8051, solution évidemment inacceptable à laquelle il faudra remédier en convertissant ses sources à SDCC, ce qui ne se fera pas sans douleur comme nous le verrons ci-dessous. À l'issue de ces développements, nous commencerons par aborder la constellation en orbite basse Iridium qui, bien qu'émettant dans des fréquences juste au-dessus de la bande GNSS occupée par le système russe GLONASS, reste dans la bande de fréquences accessible au MAX2771 avec un signal beaucoup plus facile à détecter en vue de déverminer le bon fonctionnement des interfaces d'acquisition et de communication, avant de finalement décoder GPS.

Le plan des deux articles relatant ces explorations est le suivant :

1. Partant d'une carte d'évaluation du MAX2771 fournie avec un logiciel propriétaire communiquant en HID sous MS-Windows, nous identifions le protocole de communication et l'implémentons en Python sous GNU/Linux en nous appuyant sur libusb.

2. Ayant découvert que de toute façon ladite carte d'évaluation n'est pas conçue pour acquérir et transmettre des données, mais uniquement configurer le périphérique (? !), nous reprenons la partie numérique en éliminant le microcontrôleur fourni d'origine et en le remplaçant par un FX2LP.
3. Après avoir développé le premier programme de base (faire clignoter une LED) sur le microcontrôleur équipant le FX2LP pour valider la compréhension du compilateur, de la bibliothèque associée et des outils de communication entre le PC et le microcontrôleur pour exécuter le binaire compilé sur la cible, nous portons l'implémentation logicielle (*bitbang*) du bus SPI (*Serial Peripheral Interface*) proposée par PocketSDR vers le compilateur SDCC, et apprenons à communiquer par des *Vendor Requests* pour exécuter des ordres au travers du bus USB.
4. Finalement, nous complétons le portage du *firmware* fourni par PocketSDR en ajoutant les communications *Bulk* sur USB, et validons les divers débits de communication en fonction des registres de configuration de l'horloge cadençant les convertisseurs analogiques numériques.
5. Le bon fonctionnement du montage est validé sur les signaux « puissants » des satellites en orbite basse Iridium, puis sur les signaux sous le bruit thermique des satellites de navigation.
6. Dans le prochain article, nous aborderons la conception et la réalisation d'une carte fille dédiée embarquant deux MAX2771 pour des mesures différentielles de signaux radiofréquences acquises par FX2LP et transmises par USB *Bulk* vers le PC pour post-traitement, en particulier pour pallier les déficiences du couplage entre signaux portés par des fils de communication trop longs dans le premier circuit de prototypage.
7. Finalement, nous ajouterons la capacité à corriger la source de fréquence qui cadence le MAX2771 afin de corriger son écart à la fréquence nominale GPS, et ce en ajoutant de nouvelles *Vendor Requests* afin de programmer ce nouveau périphérique ajouté au bus s'apparentant à SPI.

La seule modification lors de l'utilisation de la carte d'évaluation du MAX2771 est de remplacer l'oscillateur avec sa fréquence par défaut de 16,368 MHz par un oscillateur à 24 MHz afin de pouvoir utiliser tels quels les fichiers de configuration proposés par PocketSDR. Nous verrons en section 2 comment recompiler le *firmware* et ainsi pouvoir changer la configuration de PocketSDR pour supporter un oscillateur à 16,368 MHz.

Ainsi, dans le prochain épisode, quitte à remplacer le microcontrôleur de la carte d'évaluation à plus de 400 euros, nous poursuivrons en proposant notre propre implémentation d'une carte fille munie de deux MAX2771 s'adaptant à une carte faible coût (5 euros) munie d'un FX2LP, finalisant la démarche sur un circuit coûtant moins d'une centaine d'euros tout compris (MAX2771, FX2LP et antennes GNSS multibandes ou Iridium). En particulier, nous éliminerons de ce fait les rayonnements électromagnétiques indésirables lorsque des signaux à plusieurs MHz circulent sur des fils non blindés d'une dizaine de centimètres de long, induisant corruption des signaux numériques et bruit en bande de base lors de l'analyse des signaux transposés par l'oscillateur local avant conversion analogique numérique, tel que nous le verrons en conclusion.

1. LE BUS USB

Historiquement, un ordinateur communique en point à point par un protocole sérialisant dans le temps les données au lieu de les communiquer en parallèle – la seconde approche est limitée en débit de communication par le couplage inductif entre fils adjacents et la taille de la nappe de fils nécessaire pour porter tous les bits. RS232 est un exemple de bus série, qualifié d'asynchrone puisque les interlocuteurs ne partagent pas d'horloge. Dans la même veine, SPI est un bus série synchrone puisqu'un maître distribue un signal d'horloge vers ses esclaves. Le vénérable protocole qu'est RS232, encore largement utilisé dans l'embarqué, n'est cependant que rarement disponible sur les ordinateurs personnels actuels, et les interfaces USB-RS232 sont pléthores. Ainsi, le protocole de communication CDC d'USB – *Communications Device Class* – fait croire à l'hôte (l'ordinateur) que le périphérique parle le RS232, quand en réalité les informations sont portées sur bus USB. Ce mode de communication est très pratique puisqu'il permet d'utiliser les outils associés à RS232 – `minicom` et `screen` pour ne citer qu'eux – mais ne permet pas de tirer parti de tout le débit accessible sur USB. Dans LUFA par exemple (**http://www.fourwalledcubicle.com/LUFA.php**), un Atmega32U4 se configure facilement en CDC par :

```
#define F_CPU 16000000UL  //T=62.5ns
#define F_USB 16000000UL
#include "VirtualSerial.h"
#define N 1024

extern USB_ClassInfo_CDC_Device_t VirtualSerial_CDC_Interface;
extern FILE USBSerialStream;

int main(void){
  char tab[N]; memset(tab,'U',N);
  SetupHardware();
  CDC_Device_CreateStream(&VirtualSerial_CDC_Interface, &USBSerialStream);
  GlobalInterruptEnable();
  while(1) {fwrite(tab,1,N,&USBSerialStream); USB_USBTask(); }
}
```

et connecter le microcontrôleur programmé par cette séquence d'instructions au bus USB d'un PC fait apparaître une interface `/dev/ttyACM0` accessible par `minicom -D /dev/ttyACM0` sans que le débit (*baudrate*) n'ait d'importance ici. Quelques mesures de débit, pour N allant de 64 à 1024 en puissance de deux, indique par :

```
timeout 10 cat < /dev/ttyACM0 > fichier
```

une taille de fichier de l'ordre 965± 5 kB donc des débits de l'ordre de 100 kB/s, en accord avec les tests de **https://www.pjrc.com/teensy/benchmark_usb_serial_receive.html** pour l'Atmega32U4 qui équipe une carte Arduino Leonardo. D'après ce même site, un microcontrôleur un peu plus puissant doit atteindre le MB/s, encore loin de la dizaine de MB/s que nous visons pour une application de radio logicielle. Il faut donc se tourner vers une utilisation optimisée du bus USB et ne pas se contenter de juste exposer un port série virtuel.

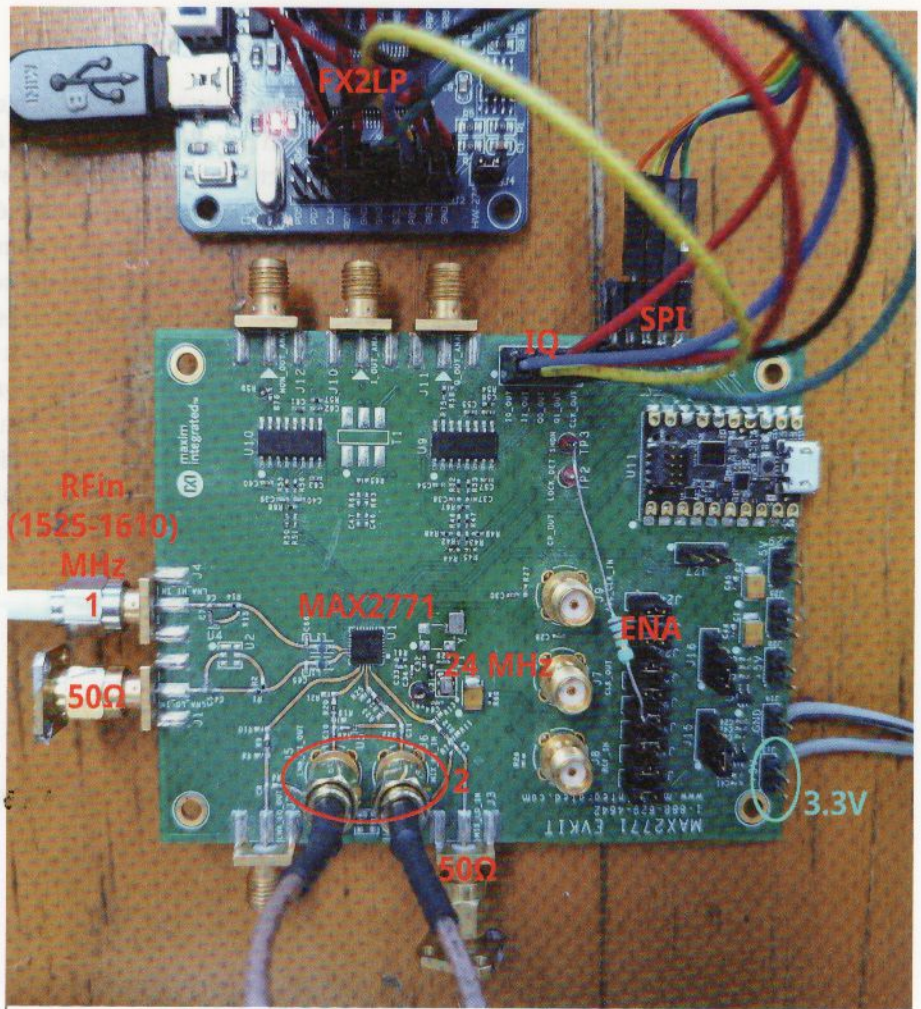


Figure 1 : La carte d'évaluation du MAX2771 de Maxim IC, source de tous nos déboires, avec son microcontrôleur (en haut à droite) qui ne peut communiquer qu'avec le logiciel propriétaire exécuté sous MS-Windows, et qui ne s'avère finalement capable que de convertir des commandes USB en SPI. Cette carte est déjà équipée des liaisons SPI et IQ (bornier à côté du microcontrôleur) que nous redirigerons vers l'EZ-USB afin de corriger ces déficiences. Seule l'alimentation 3 V (en bas à droite, en cyan) est nécessaire pour faire fonctionner la carte, les alimentations symétriques ne servent que pour les amplificateurs opérationnels qui ne sont pas utilisés. Pour tester la bande supérieure GNSS, on branchera un signal ou une antenne active sur le connecteur de gauche (marque 1) et relierons la sortie de l'amplificateur faible bruit (LNA) à l'entrée du mélangeur (ellipse rouge 2). Une résistance de tirage vers l'alimentation sort le MAX2771 de son mode veille (ENA) lorsque le microcontrôleur de la carte n'est pas alimenté.

Nombre de dispositifs « récents » n'exploitent pas CDC, mais exposent une interface plus riche, et c'est par exemple le cas du microcontrôleur qui équipe la carte d'évaluation du MAX2771 (Fig. 1). En effet, ayant acquis cette carte il y a plusieurs années, elle devint un très beau presse-papiers quand j'ai réalisé que ni GNU/Linux ni Wine ne pouvaient communiquer avec elle au travers du logiciel propriétaire fourni par Maxim IC. Comme un presse-papiers à 450 euros acquis avec l'argent du contribuable est discutable, nous nous interrogeons à comprendre son mode de communication, et les échanges entre le microcontrôleur dont la carte d'évaluation est munie et le PC. Bien entendu, sans accès au code source du *firmware* exécuté sur ledit microcontrôleur, nous ne pourrons que tenter une rétro-ingénierie en sondant le bus USB lorsque le logiciel original communique avec la carte d'évaluation : ce logiciel est exécuté depuis une machine virtuelle VirtualBox munie de MS-Windows et exécutant le logiciel propriétaire disponible sur le site du fabricant **[3]**.

À la connexion de l'interface USB du microcontrôleur, Linux nous informe (`dmesg`) que :

```
[X] usb 1-2: Product: HID DEVICE
[X] usb 1-2: Manufacturer: mbed.org
[X] usb 1-2: SerialNumber: 0123456789
[X] hid-generic 0003:1234:0006.0005: hiddev1,hidraw2: USB HID v1.11 Device
[mbed.org HID DEVICE] on usb-0000:00:14.0-2/input0
```

donc il s'agit d'un *Human Interface Device* dont nous devons comprendre le protocole de communication.

1.1 usbmon pour la rétro-ingénierie d'une interface HID

HID est l'interface de nombre de périphériques USB relativement bas débit, incluant les souris et les claviers, et son analyse est donc l'objet de nombreuses études, notamment **https://www.baeldung.com/linux/usb-sniffing** qui nous enseigne que le noyau Linux propose un mode de déverminage du bus USB pour afficher les octets échangés. En effet :

```
$ sudo mount -t debugfs device_debug /sys/kernel/debug
$ sudo modprobe usbmon
```

crée les pseudofichiers dans `/sys/kernel/debug/usb/usbmon/Xu`, avec `X` le numéro du bus, donnant accès aux ressources du noyau. Après avoir identifié sur quel bus USB le périphérique est connecté par `lsusb` du genre :

```
Bus 001 Device 039: ID 1234:0006 Brain Actuated Technologies HID DEVICE
```

nous utiliserons le premier argument (ici `1`) en place de `X` pour sonder les messages transmis sur ce bus. Noter qu'il peut être malin de ne pas brancher le microcontrôleur de la carte d'évaluation sur le même bus USB qu'une souris communiquant par cette interface pour ne pas être pollué par les messages de celle-ci chaque fois que nous déplaçons le curseur de l'interface graphique. On pourra au pire `|grep YYY` avec `YYY` le numéro de *Device* pour filtrer.

Tel quel, le microcontrôleur n'est pas causant. Cependant, si nous exécutons le logiciel propriétaire proposé par ADi pour communiquer depuis MS-Windows **[3]**, dans VirtualBox, nous obtenons une série de messages sous réserve que l'utilisateur appartienne au groupe `vboxusers` et que l'*Extension Pack* soit installé (support USB de VirtualBox), de la forme :

```
# cat /sys/kernel/debug/usb/usbmon/3 u
ffff89247452b840 3730820207 S Io:3:106:3 -115:1 64 = c9000000 00000000 00000000 0000...
ffff89247452b840 3730821534 C Io:3:106:3 0:1 64 >
ffff89245ffbec00 3730821653 C Ii:3:106:4 0:1 64 = cb010000 00000000 00000000 0000000...
ffff89245ffbe3c0 3730822364 S Ii:3:106:4 -115:1 64 <
ffff8923b4e30480 3731821079 S Io:3:106:3 -115:1 64 = c9000000 00000000 00000000 0000...
ffff8923b4e30480 3731822566 C Io:3:106:3 0:1 64 >
```

```
ffff89245ffbe3c0 3731822679 C Ii:3:106:4 0:1 64 = cb010000 00000000 00000000 0000000...
ffff8923b4e309c0 3731823216 S Ii:3:106:4 -115:1 64 <
ffff89245ffbec00 3732762122 S Io:3:106:3 -115:1 64 = c20000a2 24160300 00000000 0000...
...
```

donc le logiciel sous MS-Windows demande périodiquement au microcontrôleur s'il est présent. Le microcontrôleur acquitte de chaque requête (message `c9` suivi de plusieurs `0`). En plus de ces requêtes périodiques, le logiciel propriétaire permet de configurer les registres du MAX2771, et dans ce cas une série de messages commençant par `c2` indique le numéro du registre et la valeur à y stocker. Une fois le protocole identifié, il ne reste plus qu'à l'implémenter, par exemple en s'inspirant de **https://github.com/david0/durgod-keymapper/blob/master/remap.py** qui nous enseigne comment communiquer un message à un périphérique HID dont on connaît le *Vendor ID* et le *Product ID* (`VID:PID` – ici `1234:0006` tel qu'indiqué auparavant par `lsusb`), de la forme :

```
def tohex(data):
    return ' '.join(map(lambda x: "%02x" % x, data))

import hid
VENDOR_ID=0x1234
PRODUCT_ID=6
RESET = b"\xc9".ljust(31, b"\x00")
device_info = next(device for device in hid.enumerate()
... if device['vendor_id'] == VENDOR_ID and device['product_id'] == PRODUCT_ID)
device=hid.device()
device.open_path(device_info['path'])
device.write(RESET)                    # envoi du message C9 00 00 ...
resp=device.read(64, timeout_ms=500) # réception de la réponse
resp=bytearray(resp).rstrip(b'0x00');
print(tohex(resp))
```

pour envoyer `c9` suivi de plusieurs `0`, et en effet **[0]** le microcontrôleur répond :

```
$ python3 ./max2771.py
cb 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ...
```

donc `cb 01` indiquant que le microcontrôleur acquitte notre requête formée de `0xc9` suivi de 31 zéros (`b"\xc9".ljust(31, b"\x00")`). La voie est donc toute tracée, il ne reste plus qu'à comprendre les diverses commandes du logiciel propriétaire et les implémenter en Python. Ce faisant, un point attire cependant une interrogation : aucun bouton sur le logiciel propriétaire de communication ne permet d'acquérir les données, seulement de communiquer au travers du bus SPI des configurations des registres du MAX2771. Ayant sollicité l'aide des ingénieurs de Maxim, nous avons eu confirmation que la communication des données acquises par le MAX2771 était prévue... mais n'a jamais été implémentée. Nous avons donc acquis un convertisseur USB vers SPI à 450 euros ! Inutile de câbler les résistances R31/R32 et R44/R45 (I0/I1 to host et Q0/Q1 to host) sur la carte d'évaluation, le microcontrôleur ne saura que faire de ces signaux IQ représentatifs du signal électrique converti en signal numérique par le MAX2771. Nous

devons donc nous débrouiller seuls pour connecter une interface parallèle/USB sur le bornier J26 propageant les signaux IQ issus du récepteur de signaux GNSS pour les transmettre au PC, et c'est là qu'intervient le Cypress CY7C68013A.

En effet, Tomoji Takasu étant moins incompétent que Maxim IC – ou surtout plus motivé que les mercenaires payés ponctuellement à développer le logiciel de la carte d'évaluation – il fournit la solution avec l'EZ-USB du FX2LP : ce composant se connectera aux broches IQ du bornier et son horloge associée (communication synchrone) pour traduire vers un flux USB les mesures proposées au format parallèle par le MAX2771. Mais pour en arriver là, il faut programmer le microcontrôleur 8051 embarqué dans le CY7C68013A pour en configurer les interfaces...

1.2 EZ-USB FX2LP

L'EZ-USB ne date pas d'hier **[5]**, mais malheureusement la majorité des projets sur GitHub qui le concernent ont plus de dix ans et bien du mal à encore fonctionner aujourd'hui. Heureusement, la bibliothèque qui est associée à SDCC **[6]** pour le FX2LP, fx2lib à **https://github.com/djmuhlestein/fx2lib** fonctionne encore, et une version déclinée de fx2lib est maintenue par *sigrok* **[7]**.

Deux outils pour transférer le *firmware cross-compilé* sur PC vers le microcontrôleur sont `fxload` à **https://github.com/mbed-ce/fxload**, que nous préférerons à **https://github.com/esden/fxload** ou au paquet binaire Debian GNU/Linux issu de SourceForge compte tenu des améliorations récemment amenées, pour transférer le programme en mémoire volatile RAM ou non volatile

I0/I1 ET Q0/Q1 : CONVERSION ANALOGIQUE NUMÉRIQUE SUR 2 OU 3 BITS !

On pourrait être surpris de la nomenclature I0/I1 et Q0/Q1 qui laisse penser que les signaux complexes I+jQ sont codés sur deux bits seulement, ou un bit de signe et un bit de valeur. Ce codage est courant dans l'analyse des signaux GNSS qui se situent 20 dB sous le bruit thermique (voir section 3) et pour lesquels un nombre important de bits ne ferait que coder le bruit, pas le signal. La puissance du code pseudoaléatoire (CDMA) qui encode les messages GNSS est que la corrélation fait ressortir le signal du bruit d'un facteur égal au facteur de compression (*pulse compression ratio* donné par le produit de la bande passante multiplié par la durée du code, ou en numérique le nombre de bits du code). Avec un code sur 1023 bits pour GPS L1, le gain de compression est $\log_2(1023) \approx 10$ bits, donc les trois bits deviennent 13 bits après corrélation par le code connu, voire 16 bits pour les codes dix fois plus longs de GPS L5. À notre grande surprise, aussi peu de bits de codage resteront suffisants pour décoder Iridium qui ne bénéficie pas d'un gain de compression du codage CDMA, mais profite d'un signal bien plus puissant transmis par ses satellites en orbite basse. Ainsi, le MAX2771 peut soit fournir les mesures complexes (parties réelle et imaginaire) en bande de base sous forme I+jQ avec I et Q codés sur 2 bits chacun, ou bien en présence d'une fréquence intermédiaire acquérir la partie réelle uniquement (donc spectre pair) codée sur 3 bits selon I1, I0, Q1 (donc le bit de poids le plus faible sur Q1 même si l'information porte sur une valeur réelle), et il sera à la charge de l'utilisateur d'effectuer numériquement la transposition de fréquence (multiplication par un oscillateur local de fréquences égales à la fréquence intermédiaire) pour produire les valeurs complexes en bande de base.

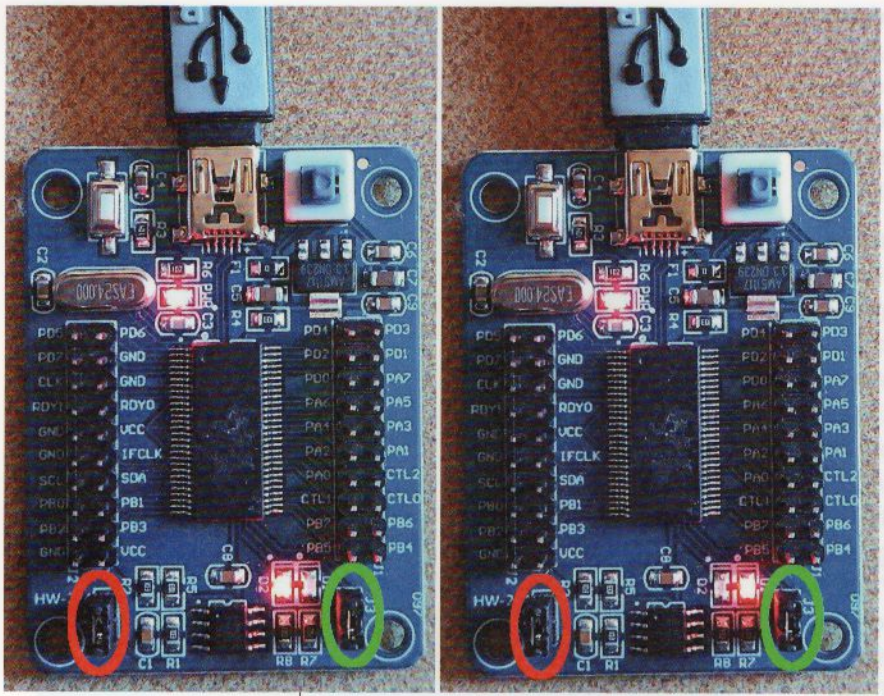


Figure 2 : Deux photographies de la carte de développement FX2LP alors que la LED de gauche est allumée et celle de droite éteinte (gauche) ou le contraire (droite), à côté du jumper entouré en vert qui les relie à l'alimentation. Le jumper entouré en rouge sert à invalider l'adresse mémoire de l'EEPROM au démarrage pour permettre de la reprogrammer (voir section 1.4).

EEPROM, et plus rapide dans un premier temps sera `cycfx2prog`, aussi disponible comme paquet binaire Debian. Muni de ces outils, le premier objectif est de faire clignoter une LED pour valider la compréhension du 8051 et des outils de programmation. Malheureusement, sélectionner une carte de développement au rabais a bien sûr quelques conséquences : la version 56 broches du CY7C68013A ne route pas les interfaces de communication asynchrones (UART, compatible RS232, uniquement disponible sur les boîtiers avec plus de broches) **[8, page 18]**, donc nous devrons nous contenter de cette LED comme mode de communication jusqu'à maîtriser les interfaces USB.

1.3 Clignotement d'une LED

Ce premier exemple s'appuie sur **https://github.com/sidd-kishan/fx2lp-blinky** faute de documentation du fonctionnement des registres du FX2LP, le cœur de 8051 étant entouré de bien plus de périphériques que le microcontrôleur original. La description des registres dans la documentation technique (incluant leur emplacement en mémoire) est des plus succinctes.

La carte de développement du FX2LP est munie de deux LED qui sont ou non activables selon qu'un *jumper* les relie ou non à l'alimentation (GPIO en puits de courant, schéma de la carte à **[9]**). Ces deux LED sont commandées depuis le port A et sont connectées aux broches 0 et 1. Ainsi, le programme trivial :

```
#include <fx2regs.h>
#include <delay.h>
#define led12 3  // 1<<0 | 1<<1
void main(void)
{unsigned char val=1;
 OEA=(led12);    // PA0, PA1 output
```

```
 while (1)
 {val=led12-val; // 2 <-> 1
  IOA = val;
  delay(1000);    // wait 1 s
 }
}
```

initialise deux bits du port A en sortie (registre OEA, bit à 1 en sortie) puis manipule le registre définissant l'état des broches en sortie IOA, alternant la LED allumée et éteinte chaque seconde – LED de gauche allumée sur la photographie de gauche et LED de droite sur la photographie de droite dans ces illustrations. Ce programme, compilé puis lié avec la bibliothèque fx2lib de **https://github.com/djmuhlestein/fx2lib** que nous avons au préalable compilée par `make` pour générer `lib/fx2.lib`, est converti en fichier hexadécimal compatible Intel (extension `ihx`) par `sdcc` avec :

```
sdcc -mmcs51  mainPA.c -I../fx2lib/include/ -L../fx2lib/lib/ fx2.lib
```

pour produire implicitement `mainPA.ihx`, qui est exécuté depuis la RAM du 8051 par `sudo cycfx2prog prg:mainPA.ihx run`. Nous constatons sur les deux figures accompagnant le code que les deux *jumpers* sont en place, le rouge pour faire *booter* le FX2LP en mode *bootloader* et charger l'exécutable au format Intel hexadécimal (`ihx`) en RAM, et le vert pour connecter les LED à l'alimentation. Divers `Makefiles` pour FX2LP ajoutent des options d'édition de liens du type `--code-size 0x1c00 --xram-size 0x0200 --xram-loc 0x1c00`, mais les valeurs par défaut semblent suffisantes pour ne pas devoir les expliciter.

Ce faisant, nous avons programmé le 8051 comme n'importe quel microcontrôleur généraliste, sans tirer parti d'une de ses originalités. En effet pour faire clignoter une autre LED externe que nous connectons entre PD7 et la masse par exemple, le code :

```
#include <fx2regs.h>
#include <delay.h>
void main()
{OED = (1 << 7);                  // direction PD7 out
 while (1)
 {PD7 = 0; // IOD = 0;            // IOD: set register
  delay(1000);
  PD7 = 1; // IOD = (1<<7); // setbit sets one bit only
  delay(1000);
 }
}
```

(Fig. 3) manipule l'unique bit PD7, en laissant en commentaire la manipulation du registre de données IOD associé au port D, toujours après avoir placé la broche en sortie en manipulant le bit 7 de OED. En effet, Maxim IC nous rappelle les fondamentaux de SDCC **[10]** et en particulier la capacité du 8051 à adresser un bit unique (SBIT) sans manipuler le registre complet qui le contient (SFR pour *Special Function Register*). Cette fonctionnalité était limpide

lors de la programmation en assembleur, mais est rendue quelque peu obscure lors du passage au langage C avec des macros du type `__sbit __at(0xB0+7) PD7;` dans `include/fx2regs.h` de fx2lib pour faire appel à l'instruction assembleur SBIT spécifique au 8051.

Une fois ce programme compilé comme auparavant, cette fois l'exécutable est placé en mémoire non volatile EEPROM pour exécution lors de la mise sous tension par :

```
sudo fxload load_eeprom --device 04b4:8613 --ihex-path main.ihx -t FX2LP \
--control-byte 0xC2 -s Vend_Ax.hex
```

qui a besoin du *firmware* `Vend_Ax.hex` pour communiquer avec l'EEPROM et identifier le VID et PID du FX2LP en mode *bootloader* (`04B4:8613` tel qu'indiqué par `lsusb`). Notez qu'avec d'anciennes versions de `fxload`, la commande :

```
sudo fxload -D /dev/bus/usb/001/035 -I main.ihx \
-c 0xc2 -s Vend_Ax.hex -t fx2lp
```

était beaucoup plus pénible puisque le pseudofichier dans `/dev/bus/usb` change de nom à chaque mise hors tension puis sous tension du FX2LP avec un identifiant qui s'incrémente à chaque fois. Attention cependant, nous pouvons écrire une fois en mémoire non volatile EEPROM, mais pas une seconde fois, la solution à ce problème sera fournie plus bas (section 1.4). En attendant, nous conseillons de tester le bon fonctionnement du logiciel en RAM par `cycfx2prog`.

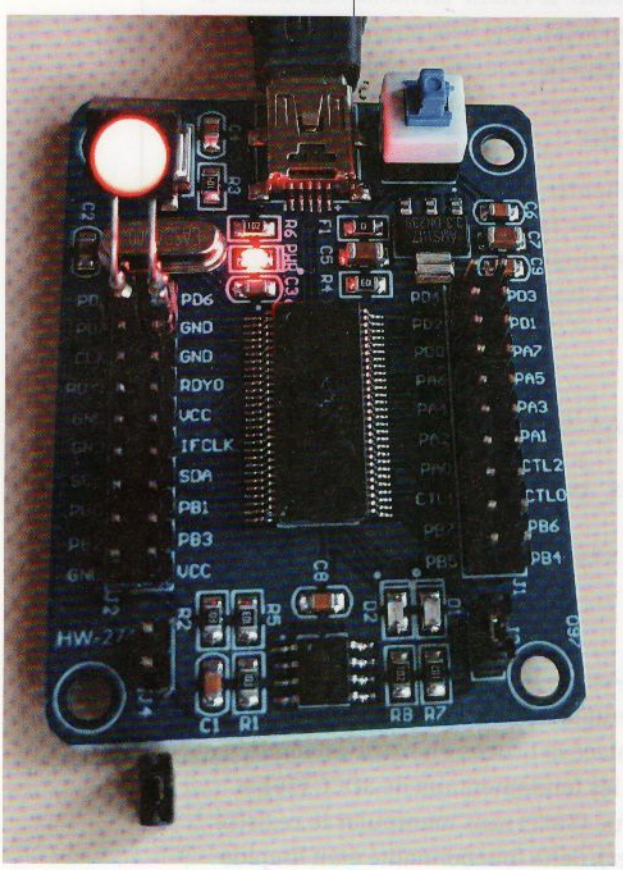


Figure 3 : Exemple de LED qui clignote sur PD7. Notez le jumper, en bas à gauche, qui a été retiré pour permettre l'identification de l'EEPROM et exécuter le code stocké en mémoire non volatile par fxload.

Le code source complet de cet exemple est disponible à **https://github.com/jmfriedt/max2771_fx2lp/tree/main/FX2LP/LED_blink**.

Ayant validé la compilation et la programmation du composant en faisant clignoter une LED, nous pouvons nous attaquer au cœur du sujet, la communication sur bus USB.

1.4 Reprogrammation de l'EEPROM

Nous avons mentionné qu'une fois l'EEPROM flashée, nous n'arrivions pas à y placer un second programme différent du premier. En effet, un premier transfert de programme depuis le PC vers l'EEPROM se solde par un succès avec le message :

```
FX2:  config = 0x42, disconnected, I2C = 100 KHz
Done.
```

mais au second essai, le message est cette fois :

```
WARNING: don't see a large enough EEPROM
```

qui indique un échec. Le problème vient des attentes de `fxload` qui ne sont plus respectées une fois l'EEPROM flashée une première fois, tel que nous allons l'expliciter.

D'après le schéma du circuit contenant le FX2LP **[9]**, le *jumper* J4 le plus éloigné des LED doit être mis en place pour passer le FX2LP en mode *bootloader* afin que la programmation de l'EEPROM soit possible. Ce faisant, la capacité du FX2LP à trouver une mémoire non volatile contenant un *firmware* valable est invalidée au démarrage, forçant le 8051 à passer en mode *bootloader*. Dans l'implémentation du circuit que nous exploitons, cet objectif est atteint en court-circuitant le bit de poids faible de l'adresse de l'EEPROM à la masse, donc en plaçant sa valeur à 0. Cependant, ceci implique que le *jumper* doit être retiré au moment de programmer l'EEPROM, puisque la *datasheet* du FX2LP explicite que l'adresse attendue pour le périphérique de stockage doit être 0x51 sur le bus I2C (dans la nomenclature qui ne conserve que les 7 bits de poids fort de l'adresse et omet de mentionner que les deux adresses transmises sur I2C sont `0xA2` ou `0xA3` pour une transaction en écriture ou lecture, respectivement). Dans la configuration proposée, le *jumper* impose un potentiel nul au bit de poids faible de l'EEPROM, définissant donc une adresse sur le bus I2C de `0x50` qui ne peut être reconnue lors de la programmation. Il faut donc absolument retirer le *jumper* avant de flasher l'EEPROM, faute de quoi le composant ne peut être détecté sur le bus I2C. Pour nous en convaincre, nous suivons les consignes de **[11]** à savoir :

1. Transférer en mémoire volatile le programme `Vend_Ax.hex` disponible par exemple dans les archives de `fxload` par `sudo cycfx2prog prg:Vend_Ax.hex run` (ou si on préfère rester avec `fxload`, par `sudo fxload load_ram --device 04b4:8613 --ihex-path Vend_Ax.hex -t FX2LP`).
2. Exécuter le programme qui lance l'ordre *Vendor Request* `0xA2` afin de lire le contenu de l'EEPROM selon :

```
import usb
import binascii
VID = 0x04B4
PID = 0x8613
dev = usb.core.find(idVendor = VID, idProduct = PID)
ret=dev.ctrl_transfer(0xC0,0xa9, 0, 0, 32)  # read EEPROM content
print(binascii.hexlify(ret))
```

3. Si nous laissons le *jumper* en place, la réponse `b'cdcdcdcdcdcd`... indique que l'EEPROM n'a pas été lue, son adresse ne correspond pas à celle attendue par `Vend_Ax`.

4. Si nous retirons le *jumper*, alors la réponse `b'c2b404138605a04203f20000` commence bien par `0xC2` tel que documenté en page 8 du manuel technique **[8]** pour indiquer « *During the power-up sequence, internal logic checks the I2C port for the connection of an EEPROM whose first byte is either 0xC0 or 0xC2. If found, it uses the VID/PID/DID values in the EEPROM in place of the internally stored values (0xC0), or it boot-loads the EEPROM contents into internal RAM (0xC2).* » suivi du VID et PID du composant en format *little endian*, donc « à l'envers » pour un lecteur occidental qui lit de gauche à droite : `b4041386` s'interprète comme `04b4:8613`.

D'après le code source de `fxload` à **https://github.com/mbed-ce/fxload/blob/master/src/ezusb.c#L674-L676**, celui-ci attend une réponse de 1 à la requête `GET_EEPROM_SIZE` (*Vendor Request* `0xA5` prise en charge par `Vend_Ax`) et sinon refuse de *reflasher* l'EEPROM. Or, émettre la commande `0xA5` vers un FX2LP exécutant `Vend_Ax` renvoie la valeur `0` et donc `fxload` refuse de continuer la programmation.

Par ailleurs, le premier octet de l'EEPROM contient `0xC2` ou `0xC0` selon la façon de configurer l'identifiant USB. L'alternative de retirer le *jumper* pour passer en mode *bootloader* échoue aussi puisque le FX2LP voit une EEPROM dont le premier octet contient `0xC2` et donc en exécute le code. La solution, fort peu élégante, que nous avons trouvée pour *reflasher* l'EEPROM du FX2LP est d'exécuter un programme depuis la RAM, donc chargé par `cycfx2prog`, pour écraser les premiers octets de l'EEPROM avec `0xFF`, et ainsi forcer le microcontrôleur avec le *jumper* retiré à exécuter son *bootloader* en l'absence de *firmware* valide. Pour ce faire, nous chargeons en RAM (`cycfx2prog`) le programme `Vend_Ax.hex` par `sudo cycfx2prog prg:Vend_Ax.hex run` et allons profiter de sa *Vendor Request* `0xA9` pour écrire dans l'EEPROM tel que décrit à **[11]**, et ainsi écraser le premier octet pour forcer le lancement du *bootloader* qui permettra de *reflasher* l'intégralité de l'EEPROM :

```
import usb
import binascii

VID = 0x04B4
PID = 0x8613
dev = usb.core.find(idVendor = VID, idProduct = PID)
ret=dev.ctrl_transfer(0xC0,0xa9, 0, 0, 64)  # read EEPROM content
print(binascii.hexlify(ret))
msg=bytearray([0xff,0xff,0xff,0xff]);       # write EEPROM (erase)
dev.ctrl_transfer(0x40 , 0xa9, 0x0, 0, msg)
ret=dev.ctrl_transfer(0xC0,0xa9, 0, 0, 64)  # read EEPROM content
print(binascii.hexlify(ret))
```

Le contenu de l'EEPROM est maintenant écrasé, et démarrer le FX2LP sans *jumper* (donc identification correcte de l'adresse I2C) permet de *reflasher* la mémoire non volatile.

Denis Bodor fait remarquer qu'une fonctionnalité identique est proposée par **http://www.triplespark.net/elec/periph/USB-FX2/eeprom/** qui contient `erase_eeprom` aux fonctionnalités identiques, quitte éventuellement à remplacer l'adresse I2C `0xA2` (« *assumes that A0 is tied to positive supply and A1,A2 of the EEPROM are tied to ground* ») par `0xA0` selon que le *jumper* ne

soit pas en place ou le soit. Ce programme efface tout le contenu de l'EEPROM au lieu de juste effacer le premier octet, seule condition nécessaire à refaire fonctionner `fxload` pour *reflasher* l'EEPROM déjà *flashée*.

1.5 Communication SPI commandée par USB

Afin d'apprendre de façon incrémentale à aborder le bus USB du point de vue de l'hôte (le PC sous GNU/Linux) et du *device* (le FX2LP), nous allons dans un premier temps nous appuyer sur le *firmware* précompilé de PocketSDR disponible à **https://github.com/tomojitakasu/PocketSDR/blob/master/FE_2CH/FW/v2.1/pocket_fw.hex** et *flashé* par `fxload` en EEPROM lorsque le microcontrôleur a été démarré avec le *jumper* associé à l'EEPROM en place, mais retiré au moment de la programmation (voir section 1.4). Une fois le *jumper* retiré, nous savons que le logiciel de PocketSDR est celui en cours d'exécution si `lsusb` indique au redémarrage (mise hors tension puis sous tension) :

```
Bus 001 Device 003: ID 04b4:1004 Cypress Semiconductor Corp. There
```

avec maintenant un couple `VID:PID` qui vaut `04b4:1004`, le nouveau PID étant défini dans le *firmware*. Il faut maintenant comprendre comment PocketSDR, dont le code source est disponible, enclenche des communications entre le FX2LP et le MAX2771 selon un protocole que la documentation technique de ce dernier qualifie abusivement de compatible SPI, abus de langage puisque MOSI et MISO sont confondus en un unique *SDATA* qui change d'impédance selon que la transaction se fasse en écriture ou en lecture (exactement la raison pour laquelle nous détestons I2C, dont la direction de la transaction ne peut être déduite de la trace acquise sur oscilloscope ou analyseur logique).

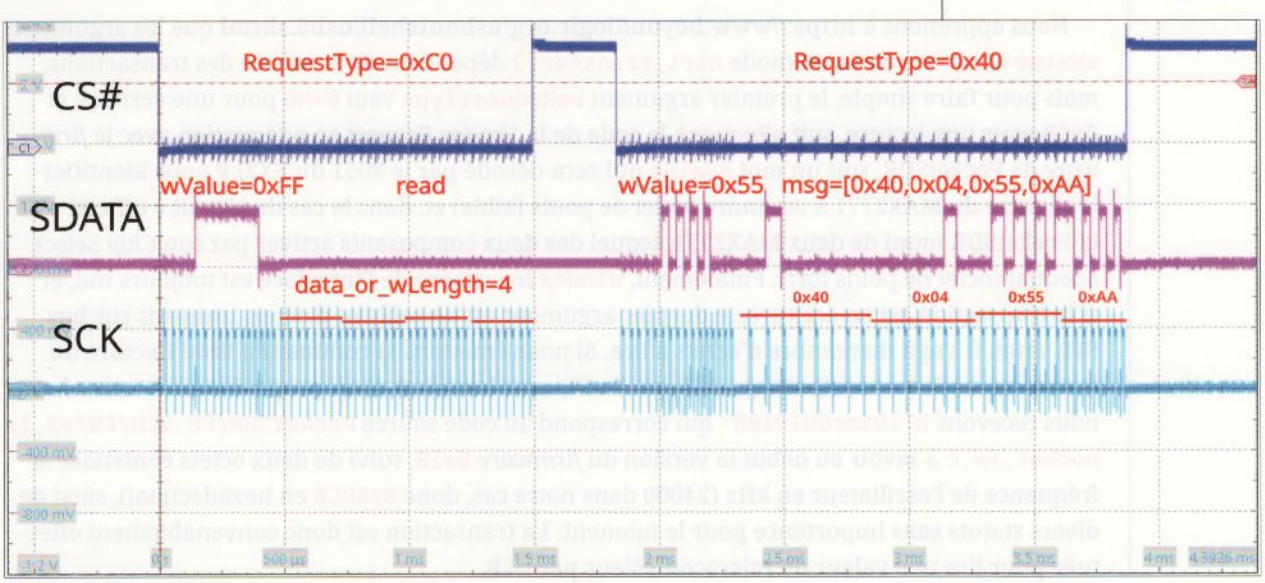


Figure 4 : Signaux produits par l'implémentation logicielle de SPI (bien que regroupant sur un même fil les signaux MOSI et MISO) en lecture et en écriture selon le code de la Vendor Request transmis. De haut en bas, la sélection du composant CS#, le bus de données SDATA et le bus d'horloge SCK.

Nous apprenons dans l'entête de `PocketSDR/FE_2CH/FW/v2.1/pocket_fw.c` qu'un certain nombre de *Vendor Requests* permettent, au même titre que les `ioctl()` dans un module noyau Linux, d'attribuer des opérations à des codes de commande arbitrairement sélectionnés. Dans le cas particulier de PocketSDR, la commande `0x40` renverra la version et le statut du *firmware*, `0x41` lit un registre du MAX2771 et `0x42` y écrit (Fig. 4).

Il faut donc apprendre à envoyer des *Vendor Requests* depuis GNU/Linux afin de se familiariser avec ces échanges implémentés dans le *firmware* précompilé de PocketSDR, en vue de s'assurer que notre implémentation du *firmware* compatible `sdcc` atteindra les mêmes objectifs, à savoir les transactions sur bus SPI. Le résultat est surprenamment facile à atteindre au moyen de la bibliothèque USB de Python 3, une fois qu'on a compris le sens des arguments :

```
import usb
import binascii

VID = 0x04B4
PID = 0x8613
dev = usb.core.find(idVendor = VID, idProduct = PID)
msg=bytearray([0x40,0x04,0x55,0xaa]);
# dev.ctrl_transfer(bmRequestType, bRequest, wValue, wIndex, data)
ret=dev.ctrl_transfer(bmRequestType=0xC0 ,bRequest=0x41, wValue=0xFF,
wIndex=0, data_or_wLength=4)
# 0x155 to activate second CS#, 0x55 to activate first CS#
dev.ctrl_transfer(0x40 , 0x42, 0x55, 0, msg) # write on SPI bus reg @ 0x55
                                             # int msg
# FX2LP firmware status
ret=dev.ctrl_transfer(0xC0,0x40, 0, 0, 10)  # read VR_STAT: returns 6 bytes
                                             # (EP0BCL=6;)
print(binascii.hexlify(ret))                 # b'105dc0010100'
```

Nous apprenons à **https://www.beyondlogic.org/usbnutshell/usb6.shtml** que les arguments `wValue` et `wIndex` de la méthode `ctrl_transfer()` dépendent de la nature des transactions, mais pour faire simple, le premier argument `bmRequestType` vaut `0x40` pour une écriture et `0xC0` pour une lecture, suit `bRequest` le code de la *Vendor Request* en adéquation avec le *firmware* de PocketSDR, suit un mot `wValue` qui sera décodé par le 8051 du FX2LP pour identifier le registre du MAX2771 à atteindre (octet de poids faible) et, dans le cas du circuit « officiel » de la PocketSDR muni de deux MAX2771, lequel des deux composants activer par son *Chip Select* adéquat (octet de poids fort). Finalement, `wIndex` le numéro de l'interface est toujours nul, et si la transaction est en écriture, le dernier argument est un tableau d'octets transmis sur bus SPI, sinon il s'agit du nombre d'octets à lire. Si nous émettons la commande `0x40` (lecture du statut) en mode lecture (`bmRequestType=0xC0`) avec le code Python proposé auparavant, alors nous recevons `b'105dc0010100'` qui correspond au code source `PocketSDR/FE_2CH/FW/v2.1/pocket_fw.c` à savoir au début la version du *firmware* `0x10`, suivi de deux octets contenant la fréquence de l'oscillateur en kHz (24000 dans notre cas, donc `0x5DC0` en hexadécimal), suivi de divers statuts sans importance pour le moment. La transaction est donc convenablement effectuée pour lire une valeur du microcontrôleur par USB.

Maintenant, pour déclencher des transactions sur USB, nous envoyons l'ordre (`bmRequestType`) `0x41` (lecture) ou `0x42` (écriture) suivi du numéro du registre du MAX2771 dans `wValue` et soit la séquence d'octets à écrire, soit le nombre d'octets à lire. Nous constatons à l'oscilloscope que les signaux SCLK, SDATA, et CS# se comportent comme prévu dans une liaison SPI (Fig. 4). Bien qu'il s'agisse d'une implémentation logicielle (*bitbang*) de SPI dans le 8051, la période et le rapport cyclique de SCLK, que l'on voit bien ne pas être constants sur la courbe du bas de Fig. 4, n'ont aucune importance puisqu'il s'agit d'un protocole synchrone dont seuls les fronts importent. Lorsque nous lisons un résultat, nous constatons dans le code source de `PocketSDR/FE_2CH/FW/v2.1/pocket_fw.c` que le cas `VR_REG_READ` se conclut par `EP0BCL=4;` donc le renvoi de 4 octets ou les 32 bits contenus dans chaque registre du MAX2771. Nous mettons en pratique par :

```
#!/usr/bin/env python3
import usb
import binascii
VID = 0x04B4
PID = 0x1004
dev = usb.core.find(idVendor = VID, idProduct = PID)
#PLL Fractional Division Ratio register 0x05
#default value 0x08000070
#                    ^^ reserved
#dec2hex(586329)
#ans =         .  08F259
msg=bytearray([0x08,0xF2,0x59,0x70]);
ret=dev.ctrl_transfer(0xC0 ,0x41, 0x05, wIndex=0, data_or_wLength=4)
print(binascii.hexlify(ret))
dev.ctrl_transfer(0x40 , 0x42, 0x05, 0, msg) # write on SPI bus reg @ 0x5
ret=dev.ctrl_transfer(0xC0 ,0x41, 0x05, wIndex=0, data_or_wLength=4)
print(binascii.hexlify(ret))
```

qui accède au registre 5 du MAX2771 ou la partie fractionnaire de la division de l'oscillateur dans la boucle à verrouillage de phase (PLL), en écriture et en lecture, pour vérifier que l'information a bien été stockée dans le registre.

1.6 Configuration de Vendor Requests par sdcc

Nous sommes convaincus de savoir échanger des messages *Vendor Requests* depuis GNU/Linux pour déclencher une action sur le FX2LP : ceci a été prouvé avec le *firmware* précompilé au moyen du compilateur propriétaire Keil. Nous devons maintenant reproduire ce comportement avec `sdcc`. Pour ce faire, nous partons de l'exemple `bulkloop/` de `fx2lib`. Nous y identifions la fonction `BOOL handle_vendorcommand(BYTE cmd) {...}` qui semble correspondre à nos besoins, et en particulier avec la gestion d'une commande nommée `VC_EPSTAT` identifiée par le code `0xB1` : nous avons déjà mentionné que tout comme `ioctl()` dans le noyau Linux, ces codes attribués aux commandes sont arbitraires et doivent être cohérents avec le code émis par l'application. Nous copions donc la séquence de *Vendor Requests* du *firmware* de PocketSDR depuis la fonction `BOOL handle_req(void) {...}` de

PocketSDR/FE_2CH/FW/v2.1/pocket_fw.c, en remplaçant les conditions imbriquées par une séquence switch/case un peu plus lisible. Par exemple, la commande VR_STAT de code 0x40 remplit les 6 premiers octets du tampon EP0BUF avec des valeurs permettant d'identifier la version du *firmware* ou la fréquence du quartz cadençant le MAX2771, et renvoie (EP0BCH = 0; EP0BCL = 6;) ce tampon à la fonction appelante. Côté Python, le pendant de cette requête est celle que nous avions vue auparavant, mais que nous comprenons n'attendre que 6 octets en retour :

```
import usb
import binascii
VID = 0x04B4
PID = 0x8613
dev = usb.core.find(idVendor = VID, idProduct = PID)
ret=dev.ctrl_transfer(0xC0,0x40, 0, 0, 6)  # read VR_STAT
print(binascii.hexlify(ret))
```

donc une requête en lecture (0xC0) de la *Vendor Request* 0x40 (VR_STAT) et la réponse est b'105dc0010100' pour indiquer 0x10 la version du *firmware*, correspondant aux EP0BUF[0] = VER_FW; dans le code source du *firmware* PocketSDR, puisque #define VER_FW 0x10, suivi de 0x5DC0 puisque dans PocketSDR :

```
EP0BUF[1]=MSB(F_TCXO); EP0BUF[2] = LSB(F_TCXO);
```

avec #define F_TCXO 24000. Suivent deux états de broches en entrée que nous ne contrôlons pas pour le moment. Le code source complet pour sdcc est consultable à **https://github.com/jmfriedt/max2771_fx2lp/blob/main/FX2LP/python_access_USB** qui bénéficie en partie de ce que Keil et fx2lib pour SDCC s'appuient tous deux sur des fichiers de constantes partageant la même nomenclature et donc facilement interchangeables.

Nous voici donc capables de recevoir des messages depuis le FX2LP grâce aux *Vendor Requests*, et nous continuons le plagiat du code de PocketSDR en copiant depuis PocketSDR/FE_2CH/FW/v2.1/pocket_fw.c l'implémentation logicielle (*bitbang*) du bus SPI, puisque la syntaxe de digitalRead(), digitalWrite() qui accèdent aux broches, mais aussi write_sclk(), write_sdata() et write_head(), est directement compatible avec SDCC, la bibliothèque fx2lib ayant le bon goût d'utiliser les mêmes constantes que le compilateur Keil tel que nous le constatons en comparant cypress/dscr.a51 de PocketSDR/FE_2CH/FW/v2.1 pour Keil et fx2lib/fw/dscr.a51 pour SDCC. Seul piège : PocketSDR définit une fonction delay() comme boucle de cyc itérations, alors que fx2lib propose une fonction du même nom, mais dont le prototype est void delay(WORD millis); et qui prend donc un argument en millisecondes (et non en cycles d'horloge).

On voit bien que la fonction fx2lib sera considérablement plus lente que celle de PocketSDR, et en effet nos observations sur le bus SPI cadencé par logiciel présentaient des variations très lentes de l'horloge et du bus de données. Une nouvelle fonction de délai avec un nom différent et effectuant bien un décompte rapide permet de retrouver presque à l'identique le comportement de PocketSDR grâce au programme compilé par SDCC proposé dans **https://github.com/jmfriedt/max2771_fx2lp/tree/main/FX2LP/python_access_USB**.

Nous en sommes au point où nous sommes capables de communiquer des ordres depuis le PC au FX2LP par un programme Python, que le 8051 interprète ces commandes et produise le motif adéquat sur le bus SPI, donc programme les registres du MAX2771 et en relise le contenu. Il ne reste maintenant « plus » qu'à capturer les octets placés sur le bus parallèle par le MAX2771 et cadencés par le signal `IFCLK` en vue de remplir la mémoire FIFO qui se vide périodiquement sur le bus USB dans une transaction *Bulk*. Quelques points de « détail » restent cependant à régler avant d'en arriver là.

1.7 Communication des données acquises du MAX2771 par FX2LP en USB Bulk

Toujours en partant de l'exemple `bulkloop/` de fx2lib, la dernière étape pour finaliser le portage du *firmware* PocketSDR vers `sdcc` en maîtrisant les transferts en mode *Bulk* sur USB est moins pénible qu'il n'y paraît, puisque de nouveau fx2lib exploite la même nomenclature que le compilateur Keil et donc il suffit de reprendre la fonction de configuration d'USB de PocketSDR nommée `void setup(void) {...}` qui configure tous les *endpoints* ainsi que `static void start_bulk(void) {...}` appelée en fin de `main()` (ainsi que `stop_bulk(void) {...}` qui peut être appelée par une *Vendor Request*) pour voir toutes les fonctions de communication. Nous éliminons dans un premier temps toutes les fonctions liées aux accès à la mémoire non volatile sur bus I2C : le pendant des fonctions Keil du type `EZUSB_WriteI2C()` et `EZUSB_ReadI2C()` nécessaires à `write_eeprom()` et `read_eeprom()` de PocketSDR existe dans fx2lib sous forme de `eeprom_read()` et `eeprom_write()`, mais ne seront pas nécessaires pour les premiers tests.

Parmi les autres subtilités de syntaxe, l'espace d'adressage en 16 bits du 8051, `xdata` de Keil, devient `__xdata` chez SDCC, et les mnémoniques assembleur inclus dans le code C sont préfixés de `__asm` suivi de l'instruction, par exemple `nop`, et se concluent avec `__endasm`. Évidemment, les vecteurs d'interruptions s'appellent différemment, par exemple `void ISR_Highspeed(void) interrupt 0 {...}` de Keil devient chez SDCC `void hispeed_isr(void) __interrupt (HISPEED_ISR) {...}`, mais comme l'exemple `bulkloop/` est fonctionnel, ce dernier point nous concerne peu.

À l'issue de cet exemple, nous sommes capables de compiler un *firmware* proposant toutes les fonctionnalités d'origine, cette fois compilé par `sdcc`, incluant les *Vendor Requests* pour communiquer les configurations sur bus SPI ainsi que le transfert de données acquises du MAX2771 en transactions USB *Bulk*.

1.8 Validation en finalisant (presque) le câblage du MAX2771 au FX2LP

Nous commençons à comprendre comment tout cela fonctionne, donc nous désirons maintenant effectuer la vraie communication entre le MAX2771 et le FX2LP en vue de récupérer des données. Pour ce faire, nous câblons la carte AliExpress EZ-USB vers la carte d'évaluation du MAX2771 au moyen de 5 fils reliant I0, I1, Q0 et Q1 à PB0 à PB3 respectivement et surtout DCLK (CLKOUT du MAX2771) à IFCLK/PE0 du FX2LP, ainsi que les broches de communication

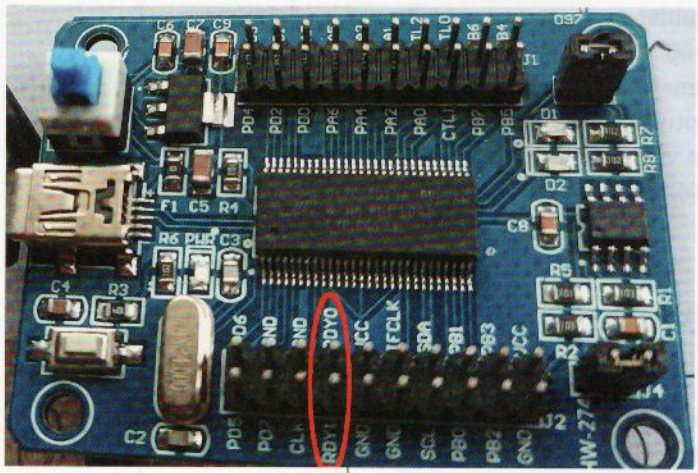


Figure 5 : La carte équipée du FX2LP prête à être utilisée, et pourtant impossible de la faire communiquer en USB Bulk, malgré un code compilé par Tomoji Takasu au moyen du compilateur propriétaire Keil auquel nous ne pouvons que faire confiance. Le problème vient d'une erreur de sérigraphie sur le circuit chinois qui induisait une configuration erronée de la direction de communication, tel que nous en a informés l'auteur japonais de PocketSDR.

SPI horloge, données, et activation (CS#) à PD2, PD3 et PD0 respectivement... et rien ne fonctionne. En effet, il faut aussi câbler les deux signaux qui définissent la direction de remplissage de la FIFO du FX2LP, soit en lecture soit en écriture depuis le PC, dans notre cas en suivant le schéma de **https://github.com/tomojitakasu/PocketSDR/tree/master/FE_2CH/HW/v2.3** en connectant RDY0/SLRD vers l'alimentation et RDY1/SLWR vers la masse. Nous relançons une mesure... et toujours rien. Impossible de recevoir la moindre trame USB depuis le FX2LP sur le PC.

Ici encore, Tomoji Takasu a la réponse puisqu'il a subi les mêmes déboires, qu'il a documentés à **https://blog.goo.ne.jp/osqzss/e/d86df04de96123fd5c73bbb6db6e8bc5** (à passer dans Google Translate pour les moins souples en japonais) : certaines cartes chinoises de développement du FX2LP ont inversé la sérigraphie de RDY0 et RDY1 qui définissent, en les polarisant à la masse ou à la tension d'alimentation, la direction de la communication. Notre carte (Fig. 5) est sujette à cette erreur, et inverser les signaux d'alimentation et masse entre RDY0 et RDY1 pour retrouver la configuration du schéma de la PocketSDR finit par permettre la communication. Baptiste Maréchal (SpacePNT, Neuchâtel) fait remarquer avec amusement que diverses illustrations de cette carte sur un même site Amazon sont incohérentes dans leur sérigraphie entre les diverses photographies qui font la promotion du produit, mais il fallait le savoir pour identifier l'erreur !

Une fois cette erreur corrigée, nous achevons de vérifier les fonctionnalités de communication en profitant de `PocketSDR/app/pocket_dump/pocket_dump` pour recevoir des données. Ce faisant, nous n'aurons pas (encore) appris à recevoir nous-mêmes des octets depuis la FIFO vers le PC, mais gardons cette compréhension pour la prochaine section. Par ailleurs, un point un peu « surprenant » est que, quelle que soit la configuration que nous proposons au MAX2771 par `PocketSDR/app/pocket_conf/pocket_conf`, le débit de communication reste toujours le même, puisque `pocket_dump` indique :

```
$ sudo app/pocket_dump/pocket_dump
  TIME(s)     T    CH1(Bytes)    T    CH2(Bytes)    RATE(Ks/s)
       5.3    I     126943232    I     126943232       23996.8
```

soit toujours une communication d'un réel (sans partie imaginaire, indiqué par I) au débit de 24 Méchantillons/s (que nous noterons désormais MS/s pour Msamples/s).

Un petit détour par la compréhension détaillée du transfert en mode *Bulk* côté *firmware* et lecture côté PC sous GNU/Linux nous permettrait presque de cacher un gros dysfonctionnement qui subsiste, mais que nous allons dévoiler plus tard et s'avérera bien plus intéressant qu'il n'y paraît.

1.9 Communication d'un motif connu en USB Bulk

Avant de vouloir lire un flux rapide de données, nous désirons déjà valider la communication microcontrôleur/PC sur bus USB par interface *Bulk* en envoyant un motif connu. Un auteur rencontré sur GitHub, Siddharth Deore **[12]**, a bien voulu fournir ses exemples, même s'il fallut une fois de plus se battre avec les évolutions de fx2lib pour faire fonctionner ces programmes. Cependant, cet exemple simple de communication *Bulk* depuis le FX2LP, disponible à **https://github.com/jmfriedt/max2771_fx2lp/tree/main/FX2LP/bulk_read_example**, est aussi l'occasion de voir comment gérer ce flux de données depuis un programme C exploitant *libusb*.

Du point de vue du microcontrôleur, l'exemple déclare l'*endpoint* 6 en interface *Bulk* et initialise les propriétés de la FIFO, que nous allons remplir manuellement dans cet exemple :

```
#define ALLOCATE_EXTERN
#include <fx2regs.h>
#include <fx2macros.h>
#include <delay.h>     // needed for SYNCDELAY4
#include <fx2ints.h>
#include <autovector.h>

static void initialize(void)
{ SETCPUFREQ(CLK_48M); // set the CPU clock to 48MHz
  SETIF48MHZ();        // set the slave FIFO interface to 48MHz
                       // IFCONFIG |= 0x40;*/
  // Set DYN_OUT and ENH_PKT bits, as recommended by the TRM.
  REVCTL = bmNOAUTOARM | bmSKIPCOMMIT; // REVCTL = 0x03;
  SYNCDELAY4;

  /* out endpoints do not come up armed */
  /* set NAKALL bit to NAK all transfers from host */
  EP6CFG = 0xe2; SYNCDELAY4; // 1110 0010 (bulk IN, 512 bytes,
                             // double-buffered)
  FIFORESET = 0x80; SYNCDELAY4; // NAK all requests from host.
  FIFORESET = 0x82; SYNCDELAY4; // Reset individual EP (2,4,6,8)
  FIFORESET = 0x84; SYNCDELAY4;
  FIFORESET = 0x86; SYNCDELAY4;
  FIFORESET = 0x88; SYNCDELAY4;
  FIFORESET = 0x00; SYNCDELAY4; // Resume normal operation.
}
```

La FIFO sera remplie du contenu lu sur les ports B et D : il peut donc être amusant de modifier périodiquement le statut d'un bit de ces ports, et comme il s'avère dans l'exemple précédent que la LED était connectée sur le port D7, nous déclenchons une interruption *timer* périodique qui change l'état de la broche PD7 qui doit aussi affecter l'affichage des données lues sur ce port et transmises par USB :

```
volatile __bit led_flag;
volatile char t0_counter;

void timer0_isr(void) __interrupt (TF0_ISR)
{ t0_counter++;
  if (t0_counter == 20)
     { led_flag = 1-led_flag;
       if (led_flag) { PD7 = 0; }
       else { PD7 = 1; }
       t0_counter = 0;
     }
}
```

Enfin, la fonction principale se contente d'initialiser les périphériques, et de boucler indéfiniment en testant si le tampon des FIFO est vide, et le cas échéant remplir avec le contenu des ports B et D, ou en commentaire pour se convaincre de la validité de la démarche, avec une séquence d'octets connue :

```
void main(void)
{ int i;
  initialize();
  led_flag=0;
  t0_counter=0;  // init timer0 vvv
  TMOD = 0x11;
  EA=1;          // enable interrupts
  ENABLE_TIMER0();
  TR0=1;         // start timer0

  OEB = 0x0; SYNCDELAY4; // set PORT-B to input
  OED = 0x0; SYNCDELAY4; // set PORT-D to input
  OED = (1 << 7);        // PD7 as output for blinking the LED !
  while (1)
    {if (!(EP2468STAT & bmEP6FULL)) // Wait for EP6 buffer to become non-full
       {for (i=0; i<512; i+=2)
          {EP6FIFOBUF[i] = IOB;     // fill buffer with port b and d
           EP6FIFOBUF[i + 1] = IOD;
           // EP6FIFOBUF[i] = 0x55; // testing with fixed values
           // EP6FIFOBUF[i + 1] = 0xAA;
            }
        // Arm the endpoint. Set BCH *before* BCL because BCL access
        // actually arms the endpoint.
```

```
        EP6BCH = 0x02; // commit 512 bytes
        EP6BCL = 0x00;
      }
    }
}
```

Ce programme est compilé et transféré en RAM du microcontrôleur du FX2LP pour exécution depuis le répertoire `FX2LP/bulk_read_example` de notre dépôt par :

```
make && sudo cycfx2prog prg:build/fifo_ep6.ihx run
```

Du point de vue de l'hôte sous GNU/Linux, la récupération des trames s'obtient en exploitant *libusb* (paquet `libusb-1.0-0-dev` sous Debian GNU/Linux) dont la configuration pour l'emplacement des fichiers d'entête et des bibliothèques peut s'obtenir avec `pkg-config`). Une fois la séquence d'incantations connue, le programme est simple et compact avec :

```
#include <stdio.h>
#include <stdlib.h>
#include <libusb-1.0/libusb.h> // ou <libusb.h> avec pkg-config --cflags
                               // libusb-1.0

#define N 512
#define tout_ms 1000
#define vid 0x04b4
#define pid 0x8613 // 0x1004;

int main()
{int i,j,xferred,res;
 unsigned char buf[N];
 libusb_context *ctx = NULL;   // initialize libusb
 libusb_init(&ctx);
 libusb_device_handle *hndl=libusb_open_device_with_vid_pid(ctx, vid, pid);
 libusb_claim_interface(hndl, 0);
 libusb_set_interface_alt_setting(hndl, 0, 1);
 while (1) {
   libusb_bulk_transfer(hndl, LIBUSB_ENDPOINT_IN | 6, buf, N, &xferred,
tout_ms);
   for (i=0; i<N; i+= 2)  // affichage du contenu de la FIFO en binaire
     {for (j=0;j<8;j++) printf("%d",((buf[i]>>(7-j))&1));
      printf(" ");
      for (j=0;j<8;j++) printf("%d",((buf[i+1]>>(7-j))&1));
      printf("\n");
     }
 libusb_release_interface(hndl, 0);
 libusb_close(hndl);
 libusb_exit(ctx);
}
```

dont les noms de fonctions semblent suffisamment explicites pour suivre naturellement la séquence de communications. Ce programme se compile avec GCC en pensant à se lier à `libusb` en complétant la commande de compilation avec `-lusb-1.0`.

Les mêmes fonctionnalités sont implémentées en Python 3 avec :

```
import usb.core
import usb.util
import binascii

N = 512
tout_ms = 1000
vid = 0x04b4
pid = 0x8613  # 0x1004

dev = usb.core.find(idVendor=vid, idProduct=pid)
dev.set_configuration()
usb.util.claim_interface(dev,0)
cfg = dev.get_active_configuration()
interface_number = cfg[(0, 0)].bInterfaceNumber
data = dev.read(0x86, N, tout_ms)  # 0x80 | 6
print(data)
```

Une fois le *firmware* flashé sur le FX2LP, l'exécution du programme Python sous GNU/Linux indique :

```
$ sudo ./bulk_read.py
array('B', [255, 127, 255, 127, 255, 127, 255, 127, 255, 127, 255,
127, 255 ...
```

alternant l'état des ports B et D (sur 8 bits), tandis que l'exécutable issu du programme C affiche :

```
$ sudo ./fx2lp_ep6_in_fifo
11111111 01111111
11111111 01111111
11111111 11111111
...
```

avec le 7e bit qui change d'état selon que PD7 soit à l'état haut ou bas sous contrôle de l'interruption *timer*.

Noter qu'en cas d'échec des transactions, il peut être judicieux de vérifier si le module noyau `usbtest` n'a pas été automatiquement chargé, et le retirer le cas échéant tel que décrit à **http://www.triplespark.net/elec/periph/USB-FX2/eeprom/** par `sudo rmmod usbtest`.

2. COMMUNICATION DES DONNÉES ACQUISES DU MAX2771 PAR FX2LP (POUR DE VRAI)

Nous avons identifié les différences de syntaxe entre compilateurs Keil et SDCC, identifié les erreurs de sérigraphie sur le circuit imprimé, tout devrait donc fonctionner. Nous avions conclu la section 1.8 en insinuant qu'un dysfonctionnement subsistait, un point de détail sans grande importance. En effet, lorsque nous configurons la carte avec notre *firmware* et lançons une acquisition par :

```
PocketSDR/app/pocket_dump/pocket_dump
```

la communication se fait toujours à 24 MS/s, même si nous tentons de configurer le convertisseur analogique numérique pour un débit différent, par exemple passer à 12 MS/s en modifiant uniquement le registre REFDIV de 3 (× 1) à 2 (/2). Pire, en modifiant cet unique registre avec une valeur qui semble cohérente, la communication est perdue. Sûrement un registre mal configuré quelque part... mais le problème s'est avéré à peine plus ardu. Il nous faut résoudre ce dernier problème pour affirmer que nous pouvons compiler le *firmware* de la PocketSDR fonctionnel au moyen de SDCC.

Nous connaissions les différents boutismes (*endianness*) des diverses architectures des processeurs – avec Intel en *little endian* qui place l'octet de poids faible d'un mot codé sur plusieurs octets à l'adresse la plus faible en mémoire, et Motorola qui en *big endian* place l'octet de poids fort à l'adresse la plus faible – et nous connaissions l'absence de définition de la nature signée ou non des entiers selon les déclinaisons des compilateurs C, mais nous découvrons maintenant que divers compilateurs pour une même architecture peuvent choisir des *endianness* différentes ! Tant que nous programmions le 8051 en assembleur, la question de représenter des grandeurs sur plusieurs octets ne se posait pas, puisque les registres du 8051 ne contiennent qu'un seul octet. Mais comme nous devons passer à la programmation du microcontrôleur en C au lieu de l'assembleur, nous devons être capables de représenter des nombres sur 2 octets (*short*) voire 4 octets (*long*), même pour une cible qui ne comporte que des registres codant des valeurs sur 8 bits. Et là, les choses se compliquent.

En effet, **[13]** enseigne que bien que ciblant une même architecture 8 bits, Keil est un compilateur *big endian* et SDCC est un compilateur *little endian*. Tant que les calculs sont gérés par un ordinateur exécutant un code compilé par un seul compilateur, l'organisation des données en mémoire est cohérente et il n'y a pas de problème. Cependant, en convertissant un code écrit pour Keil vers SDCC qui effectue des *casts* de tableaux d'octets (`char*`) vers un entier codé sur plus de 8 bits (`short` ou `int`, ou pour les utilisateurs de `stdint.h`, `int16_t` et `int32_t`), il faut penser à intervertir les octets qui seront sinon transférés dans le mauvais ordre vers le MAX2771. Ainsi, les lignes pour compilateur Keil :

```
else if (SETUPDAT[1] == VR_REG_READ) {
  *(uint32_t *)EP0BUF = read_reg(SETUPDAT[3], SETUPDAT[2]);
  EP0BCH = 0;
  EP0BCL = 4;
}
```

```
    else if (SETUPDAT[1] == VR_REG_WRITE) {
      EP0BCH = EP0BCL = 0;
      while (EP0CS & bmEPBUSY) ;
      write_reg(SETUPDAT[3], SETUPDAT[2], *(uint32_t *)EP0BUF);
    }
```

deviennent pour SDCC (on en a profité pour remplacer les `if ... else` imbriqués par un `switch ... case`) :

```
    case VR_REG_READ:
    { val32=read_reg(SETUPDAT[3], SETUPDAT[2]);
      *(uint32_t *)EP0BUF = bswap32(val32);
      EP0BCH = 0;
      EP0BCL = 4;
      return TRUE;break;
    }
```

DES DIVERSES CONVENTIONS DU CHAR EN FONCTION DES DÉCLINAISONS DE GCC

On pourrait croire que définir `char c;` en langage C est une instruction déterministe et reproductible. Il n'en est rien en l'absence du préfixe explicite `signed` ou `unsigned`, ou de l'utilisation de l'extension `stdint.h` qui permet de définir `uint8_t` pour un `unsigned char` et `int8_t` pour un `signed char` (cf. par exemple `/usr/msp430/include/stdint.h` pour le MSP430). En effet, sans ces précautions, le programme *a priori* trivial :

```
int main() {volatile char i=240;}
```

se compile avec l'option `-pedantic` pour indiquer les dépassements de capacité, vers un microcontrôleur AVR, un PC x86 ou un ARM sans système d'exploitation respectivement en :

```
$ avr-gcc -c -pedantic demo.c
demo.c: In function 'main':
demo.c:2:18: warning: overflow in implicit constant conversion [-Woverflow]
 {volatile char i=240;
                  ^~~
$ gcc -c -pedantic demo.c
demo.c: In function 'main':
demo.c:2:18: warning: overflow in conversion from 'int' to 'char' changes value
from '240' to '-16' [-Woverflow]
    2 | {volatile char i=240;
      |                  ^~~
$ arm-none-eabi-gcc -c -pedantic demo.c
```

```
case VR_REG_WRITE:
{ EP0BCH = EP0BCL = 0;
  while (EP0CS & bmEPBUSY) ;
  val32=*(uint32_t *)EP0BUF;
  val32=bswap32(val32);
  write_reg(SETUPDAT[3], SETUPDAT[2], val32);
  return TRUE;break;
}
```

avec :

```
#define bswap32(x) (((x) >> 24) | (((x) & 0x00FF0000) >> 8) \
                  | (((x) & 0x0000FF00) << 8) | ((x) << 24))
```

issu de **https://www.keil.com/dd/docs/c51/silabs/shared/si8051base/endian.h** qui se charge d'intervertir les octets (le fameux `htonl()` de `/usr/include/netinet/in.h` sous GNU/Linux).

donc les deux premiers compilateurs se plaignent que 240 dépasse la capacité de stockage d'un `signed char` (donc `signed` a été ajouté implicitement), mais `arm-none-eabi-` ne s'en plaint pas, laissant présager que là le `char` est implicitement `unsigned`, un gros soucis si une boucle teste le passage sous 0 pour cesser ses itérations. On peut compléter cet exemple par la recherche de la valeur minimum de `char` par le préprocesseur (`gcc -E`) :

```
$ gcc -E -dM demo.c  | grep CHAR_MIN
#define SCHAR_MIN (-SCHAR_MAX - 1)
#define CHAR_MIN SCHAR_MIN
$ avr-gcc -E -dM demo.c  | grep CHAR_MIN
#define SCHAR_MIN (-SCHAR_MAX - 1)
#define CHAR_MIN SCHAR_MIN
$ arm-none-eabi-gcc -E -dM demo.c  | grep CHAR_MIN
#define CHAR_MIN 0
```

démontrant donc que pour `gcc` (Intel) et `avr-gcc`, un `char` est signé avec une valeur minimale de -128, alors que pour `arm-none-eabi-gcc` la valeur minimum d'un `char` est 0 donc non signé. Ce dernier cas est peut-être lié à la norme imposée par ARM dans **https://developer.arm.com/documentation/dui0472/m/C-and-C---Implementation-Details/Basic-data-types-in-ARM-C-and-C--** qui indique :

```
char 8 1 (byte-aligned) 0 to 255 (unsigned) by default.
```

Sans vouloir donner de grain à moudre aux détracteurs du C, ce changement de signe du `char` entre déclinaisons du même compilateur nous a causé bien des soucis, en particulier dans les boucles du type `for (k=N;k>=0;k--)` qui deviennent infinies avec un `unsigned char` (toujours positif).

Il s'agit heureusement des deux seuls *casts* de tableaux d'octets vers un entier de plus de 8 bits qu'il faut corriger de la sorte, tel qu'on s'en convainc en recherchant `grep "int32_t\ \*" complete_fw.c` dans le code source du *firmware*.

Une fois ces dernières erreurs corrigées, nous avons un *firmware* complet compatible PocketSDR, mais compilable par `sdcc` sans dépendre d'un compilateur propriétaire, que Tomoji Takasu nous informe par ailleurs être limité à des exécutables de 4 KB pour sa version gratuite.

Nous avons mentionné en introduction que la carte d'évaluation du MAX2771 est munie d'un oscillateur à 16,368 MHz et que PocketSDR est configurée pour un oscillateur à 24 MHz. Cette information est renseignée dans le *firmware*, à **https://github.com/tomojitakasu/PocketSDR/blob/master/FE_2CH/FW/v2.1/pocket_fw.c#L27** pour la version originale pour Keil ou à **https://github.com/jmfriedt/max2771_fx2lp/blob/main/FX2LP/complete_fw/complete_fw.c#L28** pour notre version pour SDCC : cette constante `F_TCXO` sera remplacée par 16368 pour que le *firmware* ainsi recompilé fonctionne directement avec la carte d'évaluation munie de son oscillateur d'origine.

Ainsi, nous pourrons profiter de tous les exécutables fournis par PocketSDR pour tester le bon fonctionnement de la carte d'évaluation MAX2771 couplée au FX2LP. Pour ce faire, nous allons explorer quelques signaux satellitaires transmis dans la moitié supérieure de la bande L, autour de 1500–1600 MHz.

3. RÉSULTATS : IRIDIUM ET GPS

Nous allons acquérir, au rythme de quelques dizaines de mégaoctets par seconde, des gigaoctets de données dont nous voulons valider la pertinence, alors que nous ne sommes même pas certains du bon fonctionnement de la plateforme matérielle et de sa configuration logicielle. Il nous faut donc évaluer la difficulté à détecter le signal que nous espérons observer par rapport au bruit avant de nous lancer dans des mesures.

En effet, trois paramètres doivent être configurés pour déterminer les caractéristiques d'une acquisition :

- La fréquence centrale de l'acquisition, par programmation du facteur de multiplication de l'oscillateur de référence (24 MHz) par PLL pour asservir l'oscillateur commandé en tension (VCO). Cette porteuse n'a aucune importance dans l'analyse, car elle est éliminée lors du prétraitement analogique par mélange avec le signal à acquérir.
- La fréquence d'échantillonnage fs, définie par le rythme auquel le convertisseur analogique numérique (ADC) acquiert les données.
- Inclure une fréquence intermédiaire ou travailler directement en bande de base. La conséquence de ce choix est la production de signaux réels uniquement dans le premier cas, et complexes dans le second. En effet, dans le premier cas un unique mélangeur amène un signal réel proche de la bande de base, mais écarté de la fréquence intermédiaire, et c'est la transposition numérique (par traitement logiciel après conversion analogique numérique) depuis la fréquence intermédiaire f_{IF} vers la bande de base en multipliant par $\exp(j2\pi f_{IF} t)$

avec t=[0:N−1]/f_s le temps discret le long des N échantillons acquis à la fréquence d'échantillonnage fs, qui produira les complexes I et Q attendus. Le bénéfice du passage par la fréquence intermédiaire est de rejeter les bruits, notamment de l'électronique numérique, proches de 0 Hz, hors de la bande d'acquisition.

Les deux objectifs que nous nous fixons sont dans un premier temps d'observer le spectre des signaux des satellites Iridium, centré sur 1622 MHz, mais suffisamment puissant pour être bien visible, et dans un second temps GPS centré sur 1575,42 MHz, l'objectif ultime de ces développements, mais sous le bruit thermique donc difficile à déceler quand le bon fonctionnement du système électronique est en cours d'évaluation. En effet, rappelons que les 50 W émis par un satellite GPS avec ses antennes de 13 dBi de gain n'arrivent au sol après avoir traversé les 20000 km qui le séparent du récepteur qu'avec une puissance (équation de Friis de la conservation d'énergie sur la sphère sur laquelle se distribue la puissance émise) de

$$10\log_{10}(\underbrace{50\times 1000}_{\text{W}\to\text{mW}}) - 20\log_{10}(\underbrace{1575,42\cdot 10^6}_{\text{porteuse}}) - 20\log_{10}(\underbrace{20000\times 1000}_{\text{distance km}\to\text{m}}) + \underbrace{147,55}_{20\log_{10}(\frac{c}{4\pi})} + \underbrace{13}_{\text{gain}} = -122 \text{ dBm}$$

qui se compare avec l'intégration sur une bande passante de 2 MHz du plancher de bruit thermique à température ambiante

$$\underbrace{-174}_{\text{dBm/Hz}} + 10\log_{10}(\underbrace{2\cdot 10^6}_{\text{Hz}}) = -111 \text{ dBm}$$

donc un signal 11 dB sous le bruit thermique. Cette condition n'est pas la plus confortable pour valider un circuit inconnu avec une configuration inconnue.

3.1 Acquisition et analyse d'Iridium en temps réel

Nous avons discuté de la réception d'Iridium récemment **[14]**, mais avec un récepteur de radio logicielle généraliste fournissant nombre de bits de résolution, et donc une excellente quantification. Ici, nous désirons valider la détection, voir le décodage, d'Iridium avec des convertisseurs codant l'information sur 3 bits. L'issue de la tentative n'est pas évidente : en effectuant la corrélation entre un signal et la séquence pseudoaléatoire de N bits (« chips ») codant chaque bit de message émis par un satellite, la compression d'impulsion (intercorrélation) accumule lors de l'intégrale toute l'énergie distribuée dans les N chips dans un unique pic de corrélation qui voit donc sa quantification améliorée de log2(N) bits. Pour les 1023 chips de GPS, le gain est de l'ordre de 10 bits et les 3 bits initiaux deviennent 13 bits sur chaque pic de corrélation qui se répète chaque milliseconde (soit 1,023 Mchips/s divisé par N=1023). La situation n'est pas aussi favorable avec Iridium qui transmet un signal puissant, mais codé en phase sur deux états (BPSK) ou 4 états (QPSK) qu'il faut être capable d'identifier malgré la quantification médiocre et l'absence du gain de compression.

La première chose à voir est la gamme de fonctionnement de l'oscillateur commandé en tension, puisque la documentation technique en limite les caractéristiques à la bande utile, donc jusqu'à 1610 MHz, borne supérieure des fréquences nécessaires à décoder le système de

navigation par satellite russe GLONASS. Bien que les registres du MAX2771 laissent configurer une large gamme de valeurs et notamment de fréquences d'oscillateur local commandé en tension (VCO), rien ne garantit que le matériel puisse respecter ces demandes. Nous avons testé, en configurant grâce à `app/pocket_conf/pocket_conf` de PocketSDR et en connectant l'entrée du MAX2771 à un signal radiofréquence continu issu d'un synthétiseur, les diverses combinaisons de RDIV∈[0:1023] et NDIV∈[36:32767] afin de produire une fréquence de l'oscillateur local $f_{LO}=f_{Xtal}/RDIV \times NDIC$ avec f_{Xtal}=24 MHz afin de mesurer une raie décalée de 1 MHz de f_{LO} et constatons le bon fonctionnement jusqu'à NDIV=68 si RDIV=1 pour produire f_{LO}=1632 MHz, ou bien NDIV=546 avec RDIV=8 pour produire f_{LO}=1638 MHz, mais au-delà (NDIV=547) l'oscillateur local décroche. Ainsi, la fréquence centrale d'Iridium de 1622 MHz est bien compatible avec le MAX2771. Noter que la loi complète gouvernant la fréquence de l'oscillateur local au moyen d'une boucle à verrouillage de phase fractionnaire est

$$f_{LO} = \frac{f_{Xtal}}{RDIV} \times \left(NDIV + \frac{FDIV}{2^{20}} \right)$$

avec FDIV∈[36–32767] la partie fractionnaire du facteur multiplicatif de la fréquence. Pour chaque configuration, on pourra relire l'état des registres en lançant la commande `pocket_conf` sans argument : bien entendu, en l'absence du second MAX2771 qui équipe la PocketSDR mais est absent de la carte d'évaluation, la configuration et la lecture des registres du second composant seront incohérentes, mais sans conséquences.

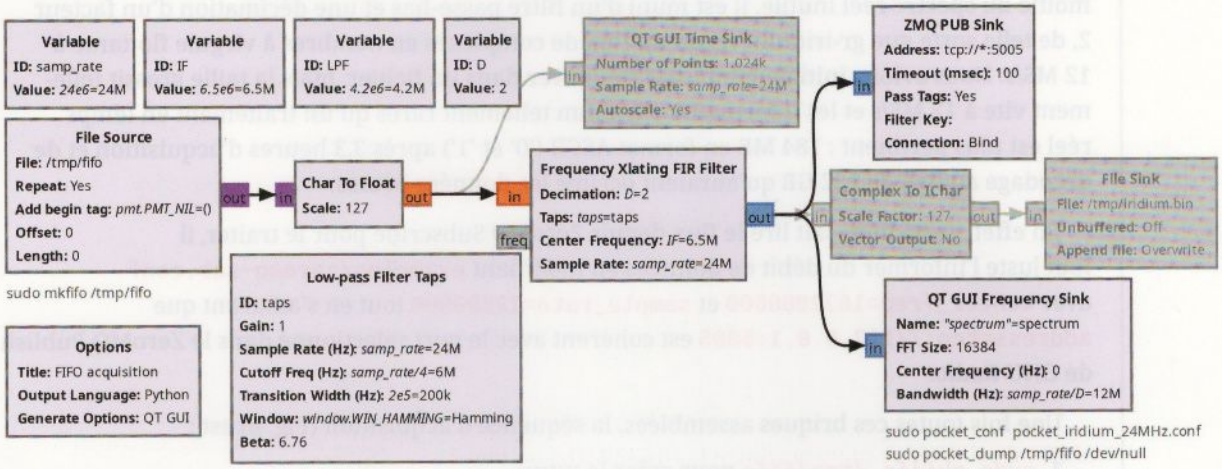
Le choix du débit de données sur le bus USB est déterminé par l'horloge qui cadence les ADC et donc le signal `IFCLK` qui rythme le remplissage de la FIFO du FX2LP. L'oscillateur principal (f_{Xtal}=24 MHz) qui cadence le FX2LP peut être multiplié ou divisé par 1, 2 ou 4 selon la valeur placée dans REFDIV lorsque ADCCLK=0, puis les registres `REFCLK_L_CNT` et `REFCLK_L_CNT` déterminent la cadence des données par

$$f_{ADC} = f_{Xtal}(REFDIV) \frac{L\_CNT}{4096 - M\_CNT + L\_CNT}$$

où f_{XTal}(REFDIV) est l'opération indexée par REFDIV sur f_{XTal} (/4, /2, × 1, × 2, × 4) et nous constatons que choisir L_CNT=2048 avec M_CNT=0 permet de diviser par 3, donc par exemple de produire 8 MS/s si REFDIV=3 pour multiplier par 1 f_{Xtal}.

À l'issue de ces explorations, la configuration finalement utilisée exploite une fréquence d'échantillonnage de 24 MHz (afin d'atteindre après transposition et décimation plus que les 10 MHz que couvre Iridium) et une fréquence intermédiaire de 6,5 MHz, choisie comme plus que les 5 MHz nécessaires à conserver 10 MHz efficaces en bande de base après transposition. Pour ce faire, NDIV=26925 et RDIV=400 avec INT_PLL=1 pour une PLL sans partie fractionnaire, tel que la fréquence de l'oscillateur local soit $24 \cdot 10^6/400 \times 26925 = 1615{,}5 \cdot 10^9$ Hz soit 1622–6,5 MHz avec 1622 MHz la fréquence centrale d'Iridium. Pour l'ADC, comme nous conservons la fréquence d'horloge, nous choisissons simplement `REFDIV=3`.

La seconde grande différence entre GPS et Iridium est qu'alors que le premier émet en continu, le second ne transmet que par impulsions brèves à des instants arbitraires dépendant de la position des satellites dans le ciel et des requêtes des terminaux au sol. Ainsi, enregistrer en aveugle quelques minutes d'Iridium pour se rendre compte que peu ou pas de signal est exploitable est frustrant : nous désirons visualiser



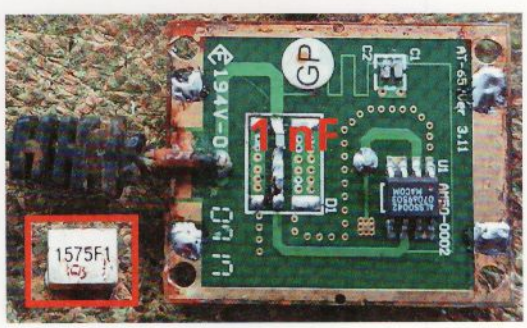


Figure 6 : Haut, chaîne de traitement GNU Radio prenant en entrée une FIFO sous forme de pipe nommée /tmp/fifo créée par mkfifo /tmp/fifo. Ce flux d'entiers réels codés sur 8 bits au rythme de 24 MS/s (défini dans le samp_rate) est converti en nombres à virgule flottante, transposition de fréquence par IF=6,5 MHz, filtrage et décimation, pour affichage dans l'analyseur de spectre sur une bande de 12 MHz et émission vers un ZeromMQ Publish dont on pensera à activer Pass Tags: Yes pour être cohérent avec gr-iridium. Ici, la socket est connectée au port 5005 de l'hôte local, mais les données pourraient très bien être transmises sur réseau pour un traitement déporté. Bas, antenne GPS L1 modifiée pour la réception des signaux Iridium en retirant son filtre passe-bande centré sur 1575,42 MHz pour le remplacer par un condensateur de 1 nF, donc une impédance de 0,1 Ω à cette fréquence, tout en évitant d'amener la composante DC d'alimentation de l'amplificateur au plus près de l'antenne sur la broche radiofréquence de l'amplificateur.

en temps réel le signal au moyen de l'analyseur de spectre (*Frequency QT GUI Sink*) de GNU Radio. Bien entendu, il n'existe pas d'interface GNU Radio pour le FX2LP, mais comme PocketSDR fournit `pocket_dump` capable d'écrire dans un fichier, il nous suffit de créer un *pipe* nommé **[16]** pour transférer les données acquises depuis le port USB vers un pseudofichier lu par le *File Source* de GNU Radio qui alimente l'analyseur de spectre. La chaîne de traitement un peu triviale *GNU Radio Companion* est proposée en Fig. 6 (haut, gauche), et sous réserve d'avoir retiré le filtre passe-bande (bas, gauche) d'une antenne GPS active **[14]** et avoir inséré un T de polarisation entre le MAX2771 et l'antenne pour alimenter son amplificateur **[14]**, nous observerons un spectre représentatif des signaux Iridium.

Par ailleurs, étant donné que gr-iridium de Sec et Schneider **[15]** ne connaît pas le concept de fréquence intermédiaire, nous profitons de l'affichage du spectre par GNU Radio, qui nécessite déjà d'éliminer cet écart à la porteuse nominale par un Xlating FIR Filter, pour transmettre le résultat de la transposition dans un flux ZeroMQ Publish. Comme le Xlating FIR Filter a rendu la

moitié du spectre réel inutile, il est muni d'un filtre passe-bas et une décimation d'un facteur 2, de telle sorte que gr-iridium reçoit un flux de complexes en nombres à virgule flottante à 12 MS/s. Nous avions initialement tenté de sauver dans un fichier, mais la taille grossit tellement vite à 12 MS/s et les informations Iridium tellement rares qu'un traitement en temps réel est plus pertinent : 184 MB en format ASCII ('0' et '1') après 3,3 heures d'acquisition et de décodage au lieu de 132 GB qu'auraient occupé les données brutes.

En effet, gr-iridium sait lire le flux depuis ZeroMQ Subscribe pour le traiter, il faut juste l'informer du débit de données en modifiant `examples/zeromq-sub.conf` avec `center_freq=1622000000` et `sample_rate=12000000` tout en s'assurant que `address=tcp://127.0.0.1:5005` est cohérent avec le port sélectionné dans le ZeroMQ Publish de GNU Radio.

Une fois toutes ces briques assemblées, la séquence d'acquisition (Fig. 6) est :

1. `sudo mkfifo /tmp/fifo` pour créer le tuyau.
2. `sudo app/pocket_conf/pocket_conf pocket_iridium_24MHz.conf` pour configurer le MAX2771.
3. `sudo app/pocket_conf/pocket_conf` pour vérifier que la configuration a bien été prise en compte.
4. `sudo app/pocket_dump/pocket_dump /tmp/fifo /dev/null` pour transmettre le flux IQ de l'unique MAX2771 vers le tuyau, l'autre flux partant à la poubelle.
5. GNU Radio lit depuis le tuyau pour transposer, décimer, afficher le spectre et reémettre vers ZeroMQ Publish par `python3 fifo_acq.py` avec le script Python produit par *GNU Radio Companion* depuis `fifoinout.grc`.
6. Enfin, `gr-iridium` reçoit le flux IQ pour en extraire les informations numériques et les sortir sur `stdout`. Un premier essai peut consister à attendre les trames *Iridium Ring Alert* (IRA) qui démontrent que des signaux de satellites sont bien reçus avec :

```
gr-iridium$ ./apps/iridium-extractor -D 4 --multi-frame ./examples/
zeromq-sub.conf | \
  python3 -u ../iridium-toolkit/iridium-parser.py --harder | grep IRA
```

mais comme il y a bien plus d'informations qui transitent que IRA, nous préférerons stocker ces bits dans un fichier pour post-traitement par :

```
gr-iridium$ ./apps/iridium-extractor  -D 4 --multi-frame ./examples/zeromq-
sub_12MSps.conf > \
   /tmp/240807iridium.bits
```

7. Nous traitons ces bits pour essayer de retrouver des phrases intelligibles avec :

```
iridium-toolkit$ iridium-parser.py -p 240807iridium.bits --harder --uw-ec >
240807iridium.parser
```

8. Nous pouvons finalement rechercher dans les phrases les trames de localisation et d'identification des avions selon le protocole ACARS :

```
iridium-toolkit$ reassembler.py -i 240807iridium.parser -m acars
```

ou produire un fichier KML contenant l'emplacement des faisceaux transmis dans les trames IRA par les satellites détectés :

```
iridium-toolkit$ grep ^IRA 240807iridium.parser | perl mkkml tracks >
240807iridium.kml
```

Le résultat de ces traitements est deux aéronefs détectés :

```
2024-08-07T14:02:03 [hdr: 0339010100000001] Dir:DL Mode:2 REG:F-GXLI
ACK:7 Label:_? (Demand mode) bID:F
2024-08-07T14:42:12                         Dir:DL Mode:2 REG:GFHFX
ACK:8 Label:_? (Demand mode) bID:Z
```

avec un petit avion effectivement identifié par **flightradar24.com** à ce moment entre Rome et Milan (Fig. 7), et un Beluga d'Airbus qui normalement ne devait pas voler à ce moment, mais a peut être effectué un test de communication depuis le sol. La carte des faisceaux (Fig. 8, page suivante) est cohérente avec une réception depuis Clermont-Ferrand (étoile jaune) selon une vue partiellement obstruée en direction de l'est/du sud-est.

Nous pouvons donc être à peu près confiants que nous avons bien reçu et décodé des trames Iridium. Rappelons que l'objectif n'est pas de réaliser un récepteur Iridium robuste, mais de valider le bon fonctionnement du MAX2771 sur un signal puissant, qui a tout de même permis de décoder nombre de trames numériques, un joli *hack* du MAX2771 au sens original du terme **[17]**.

3.2 Analyse de GPS en post-traitement

L'acquisition du signal GPS est validée dans un premier temps en détectant les signaux des satellites reçus – phase d'acquisition d'un récepteur qui ne connaît rien de son emplacement et de la géométrie de la constellation – en recherchant par

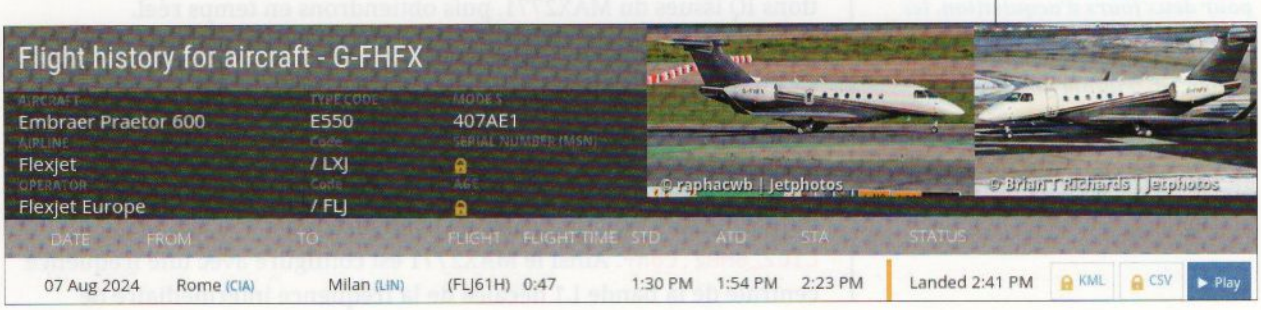


Figure 7 : Capture d'écran de FlightRadar24 validant l'analyse d'un message ACARS reçu par Iridium.

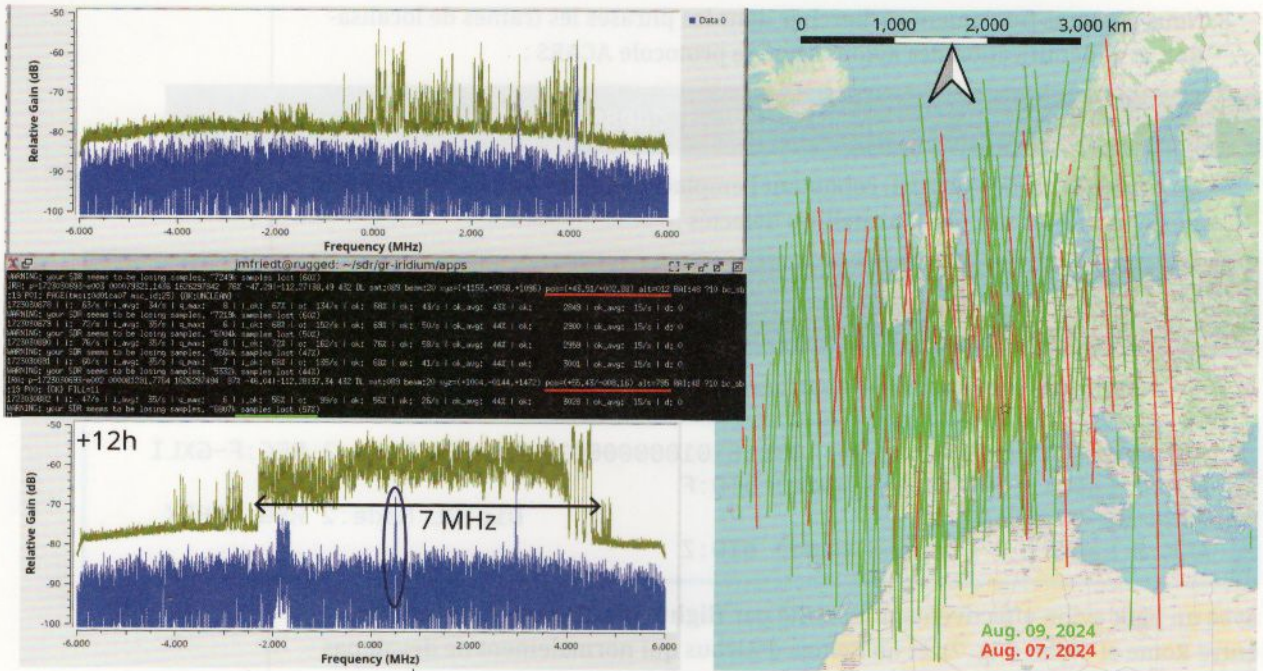


Figure 8 : Gauche, en haut, le spectre affiché par GNU Radio Companion après transposition de fréquence par le Xlating FIR Filter et décimation d'un facteur deux pour fournir 12 MHz de bande passante, suffisants pour couvrir tous les signaux d'Iridium en même temps. En haut en début d'acquisition, les canaux individuels de communication sont visibles, en bas après plus de 12 h d'acquisition, le spectre est devenu uniforme sur les plus de 7 MHz nécessaires à couvrir toutes les sous-bandes. Sur ce spectre, en ellipse bleue foncée un bref burst de communication capturé dans cette image. Droite, carte des faisceaux identifiés par la trame IRA (fond de carte, OpenStreetMap dans QGIS) pour deux jours d'acquisition, les 7 et 9 août 2024 pendant 12 h, pour valider la reproductibilité. Au milieu, gr-iridium nous informe de nombreuses trames perdues (souligné vert), mais les trames IRA sont tout de même détectées (souligné rouge).

corrélation tous les codes d'identification des satellites (*Gold Codes*) possibles pour tous les décalages Doppler possibles. Cependant, cette étape, prise en charge par `python/pocket_acq.py` de PocketSDR, doit aussi tenir compte d'un éventuel décalage de l'oscillateur local du récepteur : par défaut, ce programme ne recherche que les décalages Doppler dans la gamme ± 5 kHz correspondant uniquement à la vitesse de déplacement du satellite, mais il est prudent en cas d'échec du décodage d'étendre cette gamme (option `-d`, une valeur de 30000 pour 30 kHz est raisonnable pour un résonateur à quartz de qualité convenable). Une fois les satellites identifiés, garantie de la qualité du signal acquis, nous pourrons finaliser la chaîne de traitement en trouvant la solution optimale de positionnement du récepteur compte tenu du temps de vol observé pour au moins 4 satellites de la constellation : cette solution en position, vitesse et temps (PVT) sera obtenue au moyen de `gnss-sdr` que nous alimenterons avec le fichier des acquisitions IQ issues du MAX2771, puis obtiendrons en temps réel.

Dans un premier temps, le MAX2771 est configuré ; comme avec Iridium, nous profitons de :

```
app/pocket_conf/pocket_conf
```

mais cette fois avec un fichier de configuration `conf/pocket_L1L2_8MHz.conf`. Ainsi le MAX2771 est configuré avec une fréquence centrale de la bande L1 décalée de la fréquence intermédiaire de

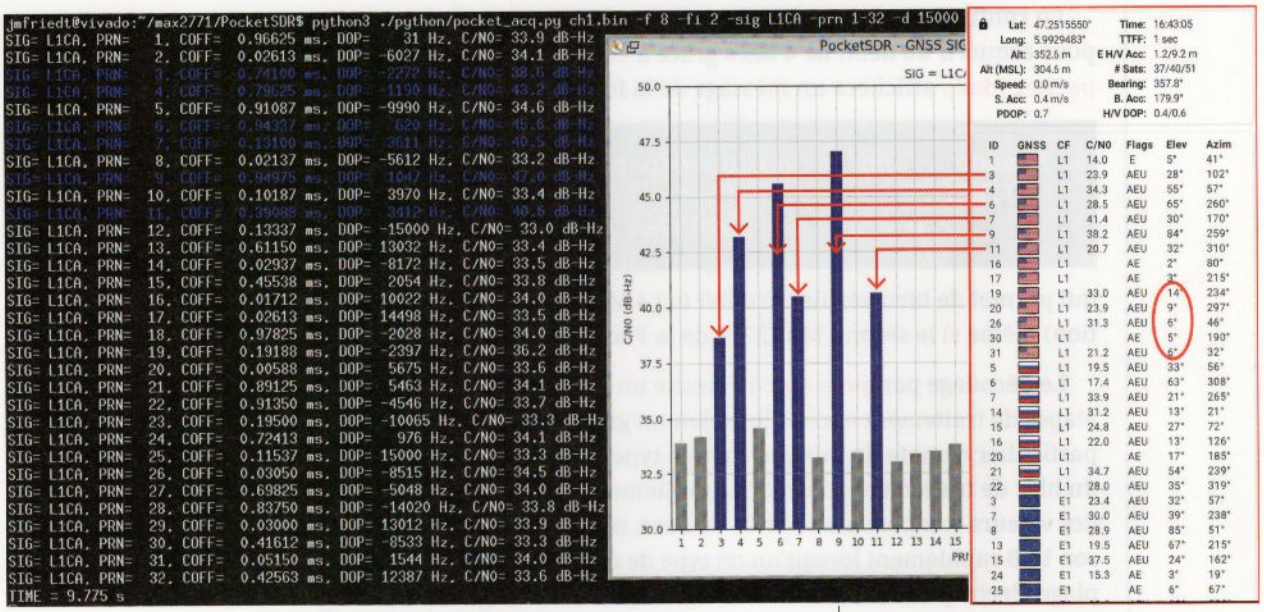


Figure 9 : Gauche, le résultat de pocket_acq.py de PocketSDR appliqué sur un fichier acquis par pocket_dump du même projet. Milieu : la sortie graphique de la phase d'acquisition des satellites, avec en abscisse le numéro du satellite (PRN) et en ordonnée le rapport signal à bruit. Droite, observation par téléphone portable sous Android exécutant GPSTest (cadre rouge), en parfaite cohérence avec le signal acquis par le MAX2771 (flèches rouges), à l'exception des satellites aux indices les plus élevés qui sont trop bas sur l'horizon pour être détectables (ellipse rouge).

2 MHz, l'échantillonnage à 8 MS/s, et la configuration du second MAX2771 pour la bande L2 simplement ignorée. Une fois la configuration relue et validée par `app/pocket_conf/pocket_conf` sans argument, nous acquérons les fichiers de mesures par `app/pocket_dump/pocket_dump -t 2 ch1.bin ch2.bin` avec `-t` pour indiquer que nous ne voulons enregistrer que deux secondes, et les arguments optionnels des noms de fichiers indiquent les deux voies de sortie. De nouveau en l'absence de second MAX2771, le fichier `ch2.bin` ne contiendra aucune valeur exploitable et seul `ch1.bin` sera exploitable. Afin d'économiser l'espace disque, `ch2.bin` pourra avantageusement être remplacé par `/dev/null` sans perte de fonctionnalité.

Afin de valider la pertinence des données acquises dans `ch1.bin`, nous allons tenter une acquisition des satellites, en corrélant séquentiellement les 32 codes pseudoaléatoires possibles pour tous les décalages Doppler possibles. Ce résultat s'obtient depuis le répertoire de PocketSDR par :

```
python3 ./python/pocket_acq.py ch1.bin -f 8 -fi 2 -sig L1CA -prn 1-32 -d 30000
```

pour indiquer que la fréquence d'échantillonnage est 8 MHz, fréquence intermédiaire de 2 MHz, impliquant implicitement que les données sont des réels (sans partie imaginaire), et que nous recherchons les satellites 1 à 32 de GPS avec un décalage de fréquence maximal de 30 kHz. Le résultat est illustré en Fig. 9 qui valide par ailleurs la cohérence de l'analyse avec l'observation d'un téléphone portable utilisé comme récepteur GNSS.

En configurant de la même façon le MAX2771 en mode IQ, sans fréquence intermédiaire, pour acquérir au débit de 4 MS/s grâce à `conf/pocket_L1L2_4MHz.conf` de PocketSDR, alors `pocket_dump` affichera un message de la forme :

```
$ sudo ./app/pocket_dump/pocket_dump -t 2 ch1.bin ch2.bin
  TIME(s)    T   CH1(Bytes)   T   CH2(Bytes)   RATE(Ks/s)
     2.0   IQ     15990784  IQ     15990784       3993.7
```

qui permet de bien vérifier le débit (ici 4 MS/s) et la nature (ici IQ, en accord avec la configuration) même si le second MAX2771 de la PocketSDR est absent.

Le décodage par `gnss-sdr` nécessite un fichier de configuration qui connecte les diverses étapes de traitement entre elles selon un graphique ordonnancé par GNU Radio, prenant en particulier compte du fait que chaque type de donnée en sortie corresponde au type en entrée du bloc de traitement suivant. Naturellement, un récepteur de radio logicielle voudrait traiter des valeurs complexes en bande de base, mais nous constatons que notre montage perd la liaison USB rapidement lorsqu'un tel type de donnée est transféré, probablement à cause d'un couplage électrique excessif entre les lignes portant les signaux d'horloge et de données au FX2LP (voir conclusion). Ayant donc abandonné l'option de traiter des complexes (type `ibyte` dans la nomenclature `gnss-sdr`) selon la configuration vue juste au-dessus (`pocket_L1L2_4MHz.conf`), il reste deux voies à explorer, le traitement en temps réel et le post-traitement de fichiers enregistrés selon la configuration `pocket_L1L2_8MHz.conf`.

La première solution est rapidement éliminée puisque le type FIFO qui permettrait de transmettre des données par un *pipe* nommé, type `Fifo_Signal_Source` du `SignalSource.implementation` dans la configuration de `gnss-sdr` ne sait pas traiter de nombre réel (sans partie imaginaire), et il ne reste donc que l'option de traiter un fichier contenant des acquisitions avec fréquence intermédiaire, et convaincre `gnss-sdr` de transposer le signal de cette fréquence intermédiaire avant d'en extraire les informations de temps de vol, donc de *peudo-range*, donc de solution PVT. Le fichier de configuration commence par le classique :

```
[GNSS-SDR]
GNSS-SDR.internal_fs_sps=8000000
```

qui indique que les informations sont acquises à 8 MS/s. La source venant d'un fichier contenant des données sur 8 bits est renseignée par :

```
SignalSource.implementation=File_Signal_Source
SignalSource.filename=ch1.bin
SignalSource.item_type=byte
SignalSource.sampling_frequency=8000000
```

qui est intuitif. L'étape suivante est la plus complexe puisque nous devons convertir les entiers en nombres à virgule flottante complexes après avoir transposé de la fréquence intermédiaire de 2 MHz grâce au *Frequency Xlating FIR Filter* de GNU Radio, à savoir un mélange avec un oscillateur local, un filtre et une décimation :

```
SignalConditioner.implementation=Signal_Conditioner
DataTypeAdapter.implementation=Byte_To_Short
InputFilter.implementation=Freq_Xlating_Fir_Filter
InputFilter.input_item_type=short
InputFilter.output_item_type=gr_complex
```

commence par convertir les données d'entrée de 8 à 16 bits en vue d'alimenter le *Frequency Xlating FIR Filter* qui sortira des complexes en virgule flottante. Les propriétés du filtre sont déterminées par sa bande passante et sa bande de coupure (Fig. 10), toujours normalisées à la fréquence de Nyquist (demi-fréquence d'échantillonnage fs) :

```
InputFilter.taps_item_type=float
InputFilter.number_of_taps=5
InputFilter.number_of_bands=2
InputFilter.band1_begin=0.0
InputFilter.band1_end=0.40
InputFilter.band2_begin=0.50
InputFilter.band2_end=1.0
InputFilter.ampl1_begin=1.0
InputFilter.ampl1_end=1.0
InputFilter.ampl2_begin=0.0
InputFilter.ampl2_end=0.0
InputFilter.band1_error=1.0
InputFilter.band2_error=1.0
InputFilter.filter_type=bandpass
InputFilter.grid_density=16
InputFilter.sampling_frequency=8000000
InputFilter.IF=2000000
```

donc 40 % de la bande (allant de 0 à la demi-fréquence d'échantillonnage) est passante, et la coupure commence à 50 % de cette bande, toujours représentée en fréquence normalisée, donc en attribuant la valeur 1 à la fréquence de Nyquist égale à la demi-fréquence d'échantillonnage. Le nombre de coefficients du filtre (`taps`) est de 5. Finalement :

```
Resampler.implementation=Pass_Through
Resampler.sample_freq_in=8000000
Resampler.sample_freq_out=8000000
Resampler.item_type=gr_complex
```

ne décime pas le flux, mais le transmet aux phases d'acquisition et de traitement *tracking* pour trouver la solution PVT issue du signal L1 de la constellation GPS en exploitant au maximum 12 canaux, donc potentiellement les signaux de 12 satellites. Ce nombre de canaux est limité par la puissance de calcul de l'ordinateur pour un traitement en temps réel, et en pratique par le nombre de satellites visibles depuis un site donné :

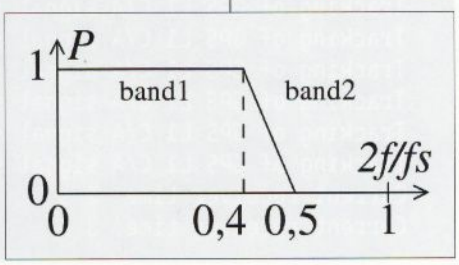


Figure 10 : Gabarit du filtre passe-bas. Notez l'abscisse graduée en fréquence normalisée, comme dans tout système échantillonné en temps discret.

```
Channel.signal=1C
Channels.in_acquisition=1
Channels_1C.count=12
Acquisition_1C.implementation=GPS_L1_CA_PCPS_Acquisition
Acquisition_1C.item_type=gr_complex
Acquisition_1C.doppler_max=30000
Acquisition_1C.doppler_step=250
cracking_1C.implementation=GPS_L1_CA_DLL_PLL_Tracking
cracking_1C.item_type=gr_complex
Tracking_1C.early_late_space_chips=0.5
Tracking_1C.pll_bw_hz=25.0;
Tracking_1C.dll_bw_hz=3.0;
Tracking_1C.dump=false;
```

qui comme auparavant tente de compenser un décalage Doppler jusqu'à 30 kHz, par pas de 250 Hz choisis comme une valeur petite devant l'inverse de la durée de chaque bit qui est 1 ms, donc petit devant 1 kHz.

Le reste n'est que classique pour conclure la recherche des solutions PVT et est imposé d'après la documentation de **https://gnss-sdr.org/docs/sp-blocks/** :

```
TelemetryDecoder_1C.implementation=GPS_L1_CA_Telemetry_Decoder
PVT.implementation=RTKLIB_PVT
PVT.positioning_mode=PPP_Static
PVT.output_rate_ms=1000
```

À l'issue de l'exécution par `gnss-sdr -c fichier.conf` avec le fichier de configuration contenu dans `fichier.conf`, nous avons le plaisir d'obtenir :

```
Initializing GNSS-SDR v0.0.19.git-main-87fcfd237 ... Please wait.
RF Channels: 1
Processing file /tmp/ch1.bin, which contains 3137273856 samples (3137273856 bytes)
GNSS signal recorded time to be processed: 392.059 [s]
Current receiver time: 1 s
Tracking of GPS L1 C/A signal started on channel 0 for satellite GPS PRN 01 (Block IIF)
Tracking of GPS L1 C/A signal started on channel 1 for satellite GPS PRN 13 (Block IIR)
Tracking of GPS L1 C/A signal started on channel 2 for satellite GPS PRN 14 (Block III)
Tracking of GPS L1 C/A signal started on channel 3 for satellite GPS PRN 15 (Block IIR-M)
Tracking of GPS L1 C/A signal started on channel 4 for satellite GPS PRN 16 (Block IIR)
Tracking of GPS L1 C/A signal started on channel 5 for satellite GPS PRN 17 (Block IIR-M)
Tracking of GPS L1 C/A signal started on channel 6 for satellite GPS PRN 18 (Block III)
Tracking of GPS L1 C/A signal started on channel 7 for satellite GPS PRN 19 (Block IIR)
Tracking of GPS L1 C/A signal started on channel 8 for satellite GPS PRN 20 (Block IIR)
Tracking of GPS L1 C/A signal started on channel 9 for satellite GPS PRN 21 (Block IIR)
Tracking of GPS L1 C/A signal started on channel 10 for satellite GPS PRN 22 (Block IIR)
Tracking of GPS L1 C/A signal started on channel 11 for satellite GPS PRN 23 (Block III)
Current receiver time: 2 s
Current receiver time: 3 s
```

```
...
Current receiver time: 13 s
GPS L1 C/A tracking bit synchronization locked in channel 8 for satellite GPS PRN 20
(Block IIR)
GPS L1 C/A tracking bit synchronization locked in channel 1 for satellite GPS PRN 13
(Block IIR)
...
Current receiver time: 22 s
New GPS NAV message received in channel 8: subframe 2 from satellite GPS PRN 20 (Block
IIR) with CN0=45 dB-Hz
New GPS NAV message received in channel 1: subframe 2 from satellite GPS PRN 13 (Block
IIR) with CN0=44 dB-Hz
...
New GPS NAV message received in channel 1: subframe 1 from satellite GPS PRN 13 (Block
IIR) with
CN0=44 dB-Hz
GPS L1 C/A tracking bit synchronization locked in channel 10 for satellite GPS PRN 07
(Block IIR-M)
First position fix at 2024-Jul-22 17:57:18.100000 UTC is Lat = 47.2517 [deg], Long =
5.99328 [deg],
Height= 364.788 [m]
Current receiver time: 1 min 17 s
Position at 2024-Jul-22 17:57:19.000000 UTC using 4 observations is Lat = 47.251622 [deg],
Long = 5.993225 [deg], Height = 361.46 [m]
Velocity: East: 0.32 [m/s], North: -0.04 [m/s], Up = -0.05 [m/s]
Current receiver time: 1 min 18 s
Position at 2024-Jul-22 17:57:20.000000 UTC using 4 observations is Lat = 47.251622 [deg],
Long = 5.993205 [deg], Height = 360.35 [m]
...
```

dont nous n'avons conservé que les principales étapes pour décrire le sens des messages. Les informations commençant par `Tracking` n'indiquent nullement la présence d'un signal exploitable dans l'enregistrement, mais uniquement que la signature (code pseudoaléatoire PRN) d'un certain satellite (GPS PRN XX) est en cours de recherche dans le signal traité, assigné au canal « `channel` ». Le point crucial est l'apparition de `tracking bit synchronization locked` qui indique que le signal d'un satellite a été identifié et pourra être analysé en vue de décoder le message de navigation : le numéro du satellite correspond au canal qui lui avait été attribué. Enfin, toutes les informations nécessaires à placer le satellite dans l'espace sont acquises lors de `New GPS NAV message received`, même si en pratique il faudra décoder les 5 « *sub-frames* » successives puisque par exemple la première phrase indique les corrections à appliquer aux horloges atomiques embarquées, les deux suivantes les éphémérides du satellite, puis la date et les conditions ionosphériques et enfin le statut de la constellation. Nous constaterons que le succès est souvent au rendez-vous après le décodage d'une « *sub-frame* » 3 (même si ici ce fut après une *sub-frame* 1), sous réserve d'avoir décodé les messages de navigation d'au moins quatre satellites (pour résoudre l'équation à 4 inconnues que sont la position et le temps) pour fournir le résultat tant attendu de `First position fix` qui se répétera ensuite tant que suffisamment de satellites sont visibles. Comme chaque *frame* dure 30 secondes, il faut attendre au minimum cette durée une fois le premier message de navigation obtenu.

3.3 Analyse de GPS en temps réel

Pour conclure cette étude, il peut sembler désirable d'obtenir la position GPS en temps réel et non en post-traitement d'un fichier enregistré qui est nécessairement contraint en espace et donc en durée. Cependant, la source FIFO de `gnss-sdr` n'accepte que des valeurs complexes, et en l'état actuel du montage spaghetti, le transfert de complexes induit une corruption du flux USB au bout de quelques secondes à dizaines de secondes. Nous avons donc abordé le problème de la façon suivante :

- création d'un *pipe* nommé `sudo mkfifo /tmp/fifo` pour acquérir les données (appartenant à `root` pour que `pocket_dump` lancé en `sudo` puisse y écrire) ;
- `sudo pocket_dump /tmp/fifo /dev/null` pour alimenter le *pipe* avec les données réelles acquises à 8 MS/s avec une fréquence intermédiaire de 2 MHz, le second canal partant dans le vide de `/dev/null` puisqu'un seul MAX2771 équipe la carte d'évaluation ;
- une chaîne de traitement GNU Radio se charge de transposer en fréquence (*Frequency Xlating FIR Filter*) et en profite pour filtrer et décimer le flux résultant dont la majorité des composantes spectrales est devenue inutile (Fig. 11, gauche). Le débit de sortie est 2 MS/s complexes ;
- ayant constaté qu'un fichier de sortie vers un autre *pipe* nommé communique mal avec `gnss-sdr`, nous avons opté pour une sortie de GNU Radio sous forme de *socket* ZeroMQ Publish, puisque `gnss-sdr` propose l'interface *ZeroMQ Subscribe*.

Pour atteindre ce résultat, les modifications au fichier de configuration de `gnss-sdr` sont la définition de la source et retirer la transposition de fréquence dans `gnss-sdr` puisque GNU Radio s'en est déjà chargé en amont :

```
GNSS-SDR.internal_fs_sps=2000000
SignalSource.implementation=ZMQ_Signal_Source
SignalSource.endpoint=tcp://127.0.0.1:5555
SignalSource.sample_type=gr_complex
SignalSource.sampling_frequency=2000000
SignalConditioner.implementation=Pass_Through
Channels_1C.count=12
Channels.in_acquisition=1
Channel.signal=1C
```

et le reste est identique à la configuration précédente.

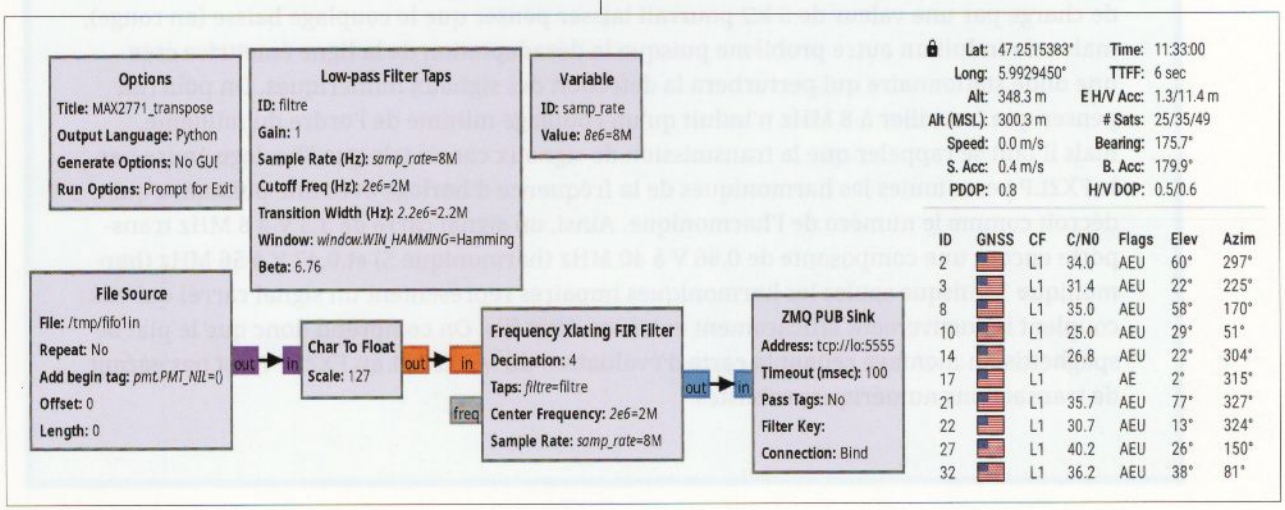
```
Tracking of GPS L1 C/A signal started on channel 0 for satellite GPS PRN 01
(Block IIF)
Current receiver time: 1 min 49 s
New GPS NAV message received in channel 9: subframe 1 from satellite GPS PRN
21 (Block IIR) with CN0=42 dB-Hz
New GPS NAV message received in channel 5: subframe 1 from satellite GPS PRN
02 (Block IIR) with CN0=43 dB-Hz
New GPS NAV message received in channel 6: subframe 1 from satellite GPS PRN
08 (Block IIF) with CN0=44 dB-Hz
```

```
New GPS NAV message received in channel 4: subframe 1 from satellite GPS PRN
32 (Block IIF) with CN0=43 dB-Hz
First position fix at 2024-Jul-26 09:31:48.120000 UTC is Lat = 47 [deg],
Long = 6 [deg], Height= 3.8e+02 [m]
Current receiver time: 1 min 50 s
The RINEX Navigation file header has been updated with UTC and IONO info.
Position at 2024-Jul-26 09:31:49.000000 UTC using 4 observations is Lat =
47.251620 [deg], Long = 5.993221 [deg],
Height = 366.06 [m]
Velocity: East: 0.91 [m/s], North: 0.65 [m/s], Up = 3.82 [m/s]
Current receiver time: 1 min 51 s
Loss of lock in channel 11!
Tracking of GPS L1 C/A signal started on channel 11 for satellite GPS PRN 19
(Block IIR)
Position at 2024-Jul-26 09:31:49.989988 UTC using 4 observations is Lat =
47.251560 [deg], Long = 5.993090 [deg],
Height = 311.77 [m]
Velocity: East: -0.83 [m/s], North: -1.32 [m/s], Up = -2.92 [m/s]
```

qui démontre la convergence de la solution en moins de deux minutes, en accord avec les observations de GPSTest sur téléphone mobile Android utilisé comme récepteur de référence (Fig. 11, droite).

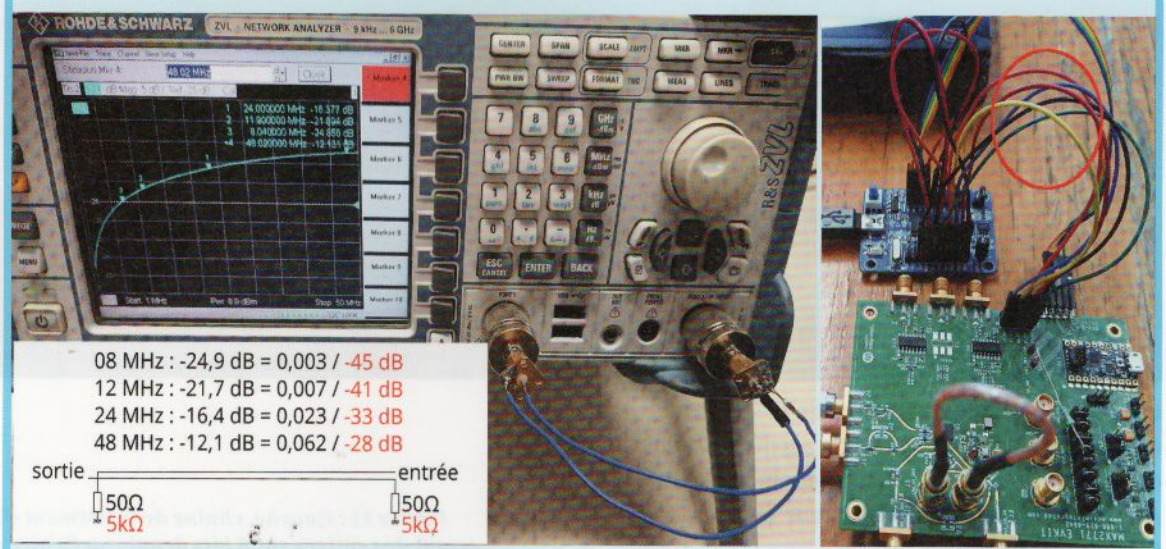
Une vidéo de cette séquence de traitements, fournissant notamment la séquence de commandes et la dynamique d'acquisition que nous ne pouvons reproduire dans ces pages statiques, se trouve à **https://www.youtube.com/watch?v=B5UcFnkbXIk**.

Figure 11 : Gauche, chaîne de traitement GNU Radio lisant des entiers sur 8 bits depuis un fichier qui est en réalité une FIFO (pipe nommé), suivi de la conversion vers un nombre flottant avec une homothétie pour ramener le résultat entre -1 et 1, transposition de fréquence et communication du résultat par socket ZeroMQ Publish, la sortie dans un fichier connecté à un autre pipe n'ayant pas donné de bons résultats. Droite, capture d'écran de GPSTest exécuté sur téléphone portable utilisé comme récepteur de référence, démontrant la cohérence avec les satellites exploités par gnss-sdr pour obtenir la solution (PRN 2, 8, 21 et 32).



COUPLAGE ENTRE FILS ADJACENTS

La mesure à l'analyseur de réseau du couplage entre deux fils adjacents illustre le problème du potentiel induit par un signal sur le fil portant le signal voisin. Ce problème est d'autant plus important que la fréquence de communication augmente.



Un analyseur de réseau mesure par son paramètre de transmission S_{21} le ratio du potentiel d'entrée V_i au potentiel de sortie V_o (ce sont bien des tensions et non des puissances). La figure ci-dessus reporte en échelle logarithmique décibels (dB) ainsi que la conversion en échelle linéaire $V_o/V_i=10^{dB/10}$ pour avoir le rapport des tensions entre le signal transmis sur un fil de données et le couplage sur le fil voisin. Nous avons reporté en noir la mesure lorsque chaque fil est terminé à la masse par une résistance 50 Ω visant à adapter l'impédance, mais maximisant le courant dans le fil et donc le couplage. Remplacer la résistance de charge par une valeur de 5 kΩ pourrait laisser penser que le couplage baisse (en rouge), mais cela induit un autre problème puisque la désadaptation de la ligne émettrice crée une onde stationnaire qui perturbera la détection des signaux numériques. On pourrait penser que travailler à 8 MHz n'induit qu'un couplage minime de l'ordre du millième, mais il faut se rappeler que la transmission de signaux carrés tels que l'horloge émise par le FX2LP porte toutes les harmoniques de la fréquence d'horloge avec une puissance qui décroît comme le numéro de l'harmonique. Ainsi, un signal carré de 3,3 V à 8 MHz transporte encore une composante de 0,66 V à 40 MHz (harmonique 5) et 0,47 V à 56 MHz (harmonique 7 puisque seules les harmoniques impaires représentent un signal carré) qui eux couplent inductivement efficacement entre les deux fils. On comprend donc que le plat de spaghettis du montage reliant la carte d'évaluation du MAX2771 au FX2LP n'est pas garant de transactions numériques robustes.

CONCLUSION

Nous avons identifié le protocole de la carte d'évaluation du MAX2771, compris que seule la conversion de commandes USB vers SPI est prise en charge, mais pas le transfert de données haut débit, appris à configurer le FX2LP comme interface de transfert entre les mots formés de bits en parallèle et une horloge vers USB, et acquis ces données sur PC. Cependant, le montage spaghetti avec les longs fils de communication portant des signaux à plusieurs MHz voire dizaine de MHz ne peut garantir une transaction robuste, et un circuit dédié s'avère nécessaire pour tirer pleinement parti des performances du MAX2771.

Tout comme le PocketSDR qui a inspiré cette étude, nous visons à exploiter les signaux acquis simultanément par plusieurs MAX2771 pour évaluer la direction d'arrivée d'un signal et éventuellement annuler une source de leurrage ou de brouillage. Le circuit dédié à ces développements, et bien d'autres feront l'objet de l'article qui poursuivra cette description.

Tous les codes sources sont disponibles à **https://github.com/jmfriedt/max2771_fx2lp/** et en particulier dans le sous-répertoire `Maxim_EvalBoard` pour cet article. Les ouvrages de la bibliographie ont été obtenus sur *Library Genesis*. JMF

RÉFÉRENCES

[0] Tous les codes cités dans cet article sont disponibles à **https://github.com/jmfriedt/max2771_fx2lp/** y compris ce **https://github.com/jmfriedt/max2771_fx2lp/blob/main/Maxim_EvalBoard/max2771.py**

[1] J.-M. Friedt, É. Carry, « Enregistrement de trames GPS – développement sur microcontrôleur 8051/8052 sous GNU/Linux », GNU/Linux Magazine France, 81 (février 2006).

[2] J.-M Friedt, « Exploitation des signaux de référence de navigation par satellite pour un positionnement centimétrique : RTKLib fait appel à Centipède et l'IGN pour afficher dans QGis », Hackable 48 (mai/juin 2023) - **https://connect.ed-diamond.com/hackable/hk-048/exploitation-des-signaux-de-reference-de-navigation-par-satellite-pour-un-positionnement-centimetrique-rtklib-fait-appel-a-centipede-et-l-ign-pour-afficher-dans-qgis**

[3] Analog Devices Inc., Software Development: MAX2771EVkit GUI, 1.0.0 à **https://www.analog.com/en/resources/evaluation-hardware-and-software/evaluation-boards-kits/max2771evkit.html**

[4] J.-M. Friedt, W. Feng, « Anti-leurrage et anti-brouillage de GPS par réseau d'antennes », MISC 110 (juillet-août 2020) - **https://connect.ed-diamond.com/MISC/misc-110/anti-leurrage-et-anti-brouillage-de-gps-par-reseau-d-antennes**

[5] Les projets **https://fpga4u.epfl.ch/wiki/FX2.html** et **https://fpga4u.epfl.ch/wiki/FPGA4RF.html** datent de 2011 mais restent d'actualité.

[6] **https://sdcc.sourceforge.net/** mis à jour le 6 mars 2024, aussi disponible comme paquet Debian du même nom.

[7] Codes sources du firmware de l'analyseur logique sigrok à **https://sigrok.org/download/source/sigrok-firmware-fx2lafw/**

[8] Cypress, CY7C68013 EZ-USB FX2 USB Microcontroller – High-Speed USB Peripheral Controller (2002) à **https://www.keil.com/dd/docs/datashts/cypress/cy7c68xxx_ds.pdf**

[9] **https://saturn.ffzg.hr/rot13/index.cgi/U2CY7C68013-56.pdf?action=attachments_download;page_name=fx2lp;id=20140817120222-0-10210**

[10] Using the Free SDCC C Compiler to Develop Firmware for the DS89C430/450 Family of Microcontrollers à **https://www.analog.com/en/resources/app-notes/using-the-free-sdcc-c-compiler-to-develop-firmware-for-the-ds89c430450-family-of-microcontrollers.html**

[11] « Reading Serial EEPROM which is Connected to the FX2 » (2008) à **https://community.infineon.com/t5/Knowledge-Base-Articles/Reading-Serial-EEPROM-which-is-Connected-to-the-FX2/ta-p/254704**

[12] **https://github.com/siddharthdeore/fx2lp_usb_dev**

[13] Keil est big endian selon **https://groups.google.com/g/comp.arch.embedded/c/Cabbhrl9_oM/m/3yOfgiqm7_YJ** tandis que SDCC est little endian tel que mentionné à **https://community.silabs.com/s/article/common-pitfalls-when-using-sdcc**

[14] J.-M Friedt, « À l'écoute des messages transmis par satellite en orbite basse : Iridium », MISC Hors-Série 29 (2024) -- **https://connect.ed-diamond.com/misc/mischs-029/a-l-ecoute-des-messages-transmis-par-satellite-en-orbite-basse-iridium**

[15] **https://github.com/muccc/gr-iridium** présenté à Sec et Scheider, Iridium Hacking, Chaos Communication Camp (2015) à **https://av.tib.eu/media/38121**

[16] J.-M Friedt, « Échanges de données pour un traitement distribué : communication par réseau ou entre langages », GNU/Linux Magazine France 267 (janv.-févr. 2024) - **https://connect.ed-diamond.com/gnu-linux-magazine/glmf-267/echanges-de-donnees-pour-un-traitement-distribue-communication-par-reseau-ou-entre-langages**

[17] Comme les lecteurs de ce journal le savent bien, et les journalistes de France Info ou les auteurs de **https://www.orangecyberdefense.com/fr/solutions/services-manages/micro-soc-poste-de-travail/edr** ne le savent pas, un *hack* est le détournement d'une technologie de son objectif initial, et aucunement un acte malveillant *a priori*. Tel que l'indique A. Guitton dans « Hackers. Au cœur de la résistance numérique » (Éd. au Diable Vauvert, 2013), le *hacker* est une personne s'attachant à « comprendre le fonctionnement d'un mécanisme, afin de pouvoir bidouiller pour le détourner de son fonctionnement originel » ou de façon répétée dans le début de son ouvrage, « comprendre, bidouiller, détourner ». La définition au début de R. des Bois, « Lève-toi et code : Confessions d'un hacker » (La Martinière, 2018) aurait aussi pu être en relation avec l'introduction de cet article en affirmant « voir quelque chose de cassé et ne pas pouvoir s'empêcher de ne rien faire, soit tu l'exploites, soit tu le répares, mais impossible d'ignorer ce dysfonctionnement et le laisser comme tel. » Quel dommage que l'auteur ne se souvienne pas de cette définition dans la suite de son discours, ouvrage sans intérêt après cette courte mention pertinente, dont le protagoniste se réduit à acheter sur le Web des failles de sécurité pour les exploiter contre des utilisateurs crédules et incompétents, sans faire preuve de créativité technique.