ÉLECTRONIQUE | EMBARQUÉ | RADIO | IOT

HACKABLE

L'EMBARQUÉ À SA SOURCE

N° 50
SEPT. / OCT. 2023

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 - 50 - F: 14,90 € - RD

CPPAP : K92470

RETROBREW / Z80

Assembleur Z80 : Commençons à développer pour notre ordinateur 8 bits sur platine à essais p.52

LEGO / NFC / CRYPTO

Explorons et prenons le contrôle du lecteur USB et des figurines NFC du jeu LEGO Dimensions p.82

Domotique / Raspberry Pi

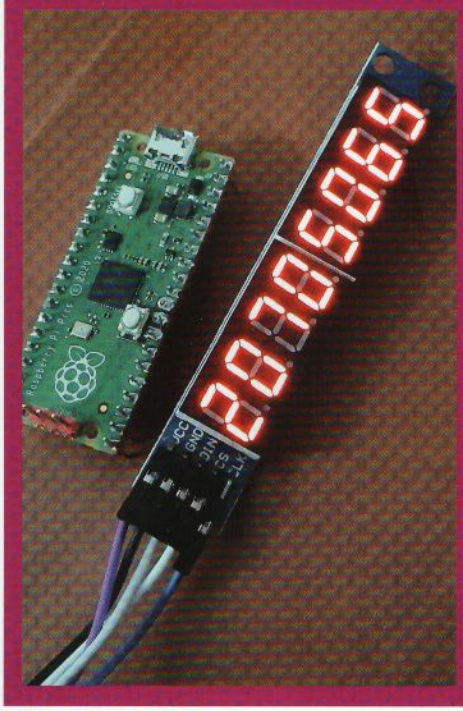
Maison de vacances, Airbnb, gîte, hébergement, local associatif...

CONTRÔLEZ L'ACCÈS D'UNE RÉSIDENCE À DISTANCE p.66

- Déverrouillage de la porte par Wi-Fi
- Contrôle de la fourniture d'eau par SMS
- Pilotage de l'alimentation électrique

PI PICO / AUTHENTIFICATION

Créez votre Authenticator TOTP et renforcez votre sécurité avec des mots de passe à usage unique p.22

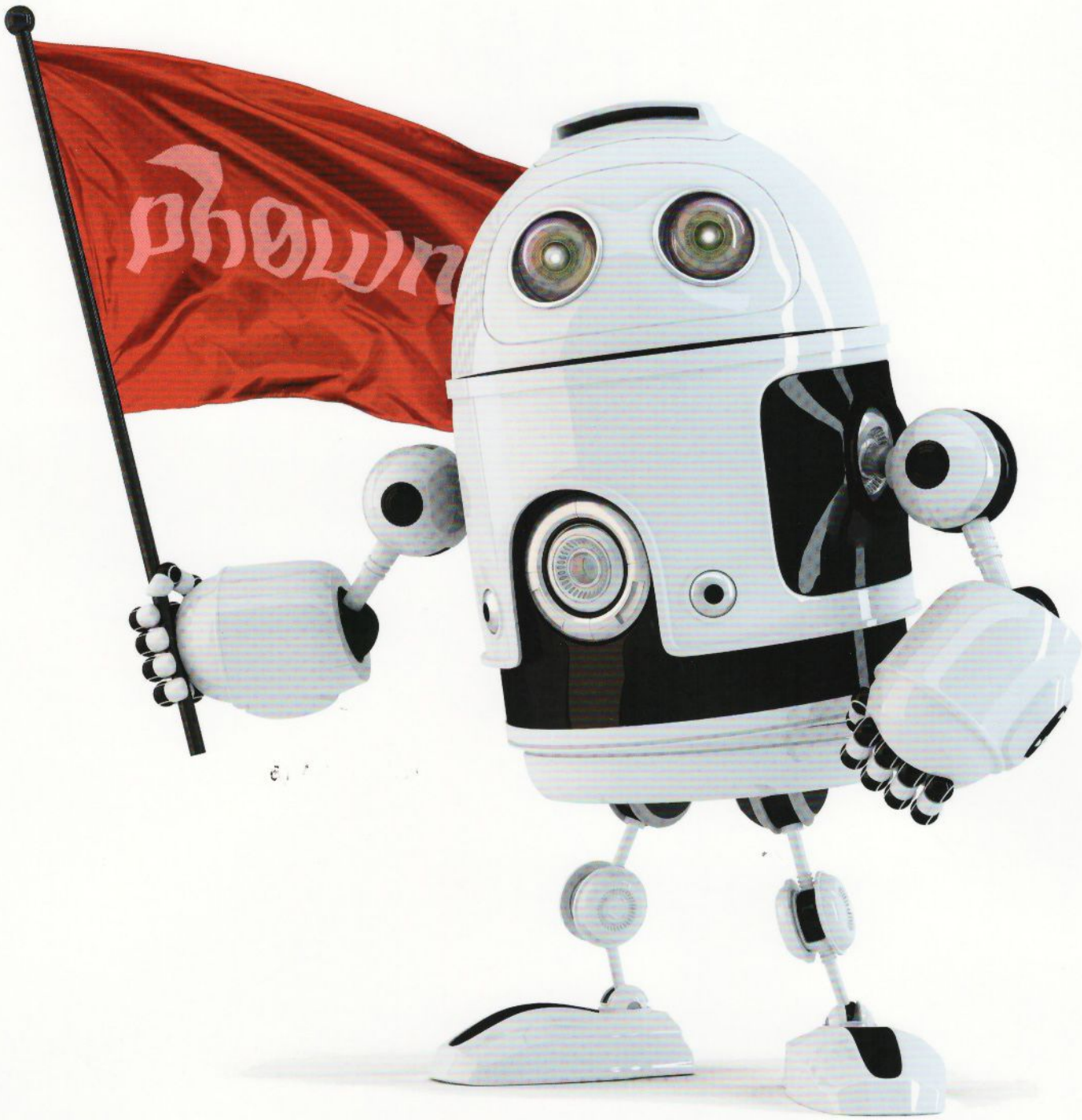


Z180 / REVERSE

Découvrez le banking ou comment accéder à 1 Mio de RAM avec seulement 16 bits d'adresses p.110

RÉFLEXION / HUMEUR

Ne laissez plus vos projets dériver inéluctablement de l'électronique vers l'informatique ! p.04



Ph0wn CTF

Workshops and Capture The Flag for Smart Dervices
Sophia Antipolis, France
November 24-25, 2023
https://ph0wn.org

ÉDITO

Heu... Arduino ? Vous faites quoi, là ?

Récemment, les nouvelles cartes Arduino UNO R4 Minima et UNO R4 WiFi ont fait leur apparition et sont très souvent présentées (ne serait-ce qu'en raison de leur nom) comme les successeurs de la bonne vieille UNO Rev3 (ou R3). La belle affaire, me direz-vous, la UNO avait bien besoin d'une mise à jour et, en effet, le changement est violent. Au revoir l'AVR 8 bits obsolète et bonjour le microcontrôleur Renesas RA4M1 (ARM Cortex-M4) !

Mais avec un prix officiel de 18 € pour la R4 Minima, il est difficile de ne pas faire la comparaison avec la Raspberry Pi Pico et son RP2040 (double cœur ARM Cortex-M0+) avec dix fois plus de RAM et 8 fois plus de flash pour presque trois fois moins cher (et je ne parle même pas du fantastique PIO intégré au RP2040).

L'approche est, à mon sens, assez étrange, d'autant que rien n'est fait pour éviter les confusions. La broche A0 des R4, par exemple, traditionnellement liée à l'ADC, car faisant partie des entrées analogiques A0 à A5, peut ici être aussi une sortie analogique, avec une réelle conversion via un DAC et non une PWM. Mais pourquoi rendre les choses aussi déroutantes sachant que c'est, en plus, la même fonction, `analogWrite()`, qui est utilisée ? Idem pour la relation entre GPIO et labels des broches, puisqu'elle est différente entre R4 Minima et R4 WiFi... Et enfin, pour ajouter au questionnement, la carte R4 WiFi utilise le même microcontrôleur Renesas, mais se voit équipée d'un module Espressif ESP32-S3, alors que l'Arduino Nano 33 IoT utilise un Atmel SAMD21 (ARM Cortex-M0+) secondé par un u-blox NINA-W102 qui est aussi un composant sur base ESP32... Ah, et je vois que vient tout juste d'arriver l'Arduino Nano ESP32 qui, je vous le donne en mille, est construit autour d'un ESP32-S3 (à 18 € !). Et, cerise sur le gâteau, il ne faut pas oublier l'Arduino Nano RP2040 Connect et son RP2010 équipé du même u-blox NINA-W102, à trois fois le prix d'une Pico W.

En tout, ce n'est pas moins d'une cinquantaine de cartes Arduino qui sont listées sur le *store* officiel, la moins chère étant l'Arduino Nano Every (j'en découvre l'existence) et son poussif ATMega4809 avec 48 Kio de flash et 6 Kio de SRAM, pour le prix de deux (voire trois) Pi Pico ou huit clones Wemos D1 Mini (ESP8266)...

Tout ceci, cette avalanche de diversité, me laisse dubitatif. Sachant l'écosystème Arduino grandement orienté vers l'éducation, on se demande vraiment comment le débutant pourrait s'y retrouver...

Denis Bodor

Hackable Magazine
est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND
STIMULANT DE MATIÈRE GRISE

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

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

Suivez-nous sur Twitter
@hackablemag

Alsace

SOMMAIRE

À PROPOS DE HACKABLE...

HACKS, HACKERS & HACKABLE

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

RÉSISTONS À L'INFORMATISATION GALOPANTE DE L'ÉLECTRONIQUE !

Yann Guidon
[yg@ygdes.com – vieux grincheux]

Si vous lisez Hackable, c'est probablement parce que vous aimez tenir dans vos mains un objet qui est le résultat de votre travail, un symbole physique et tangible de votre ingéniosité, une récompense pour vos investissements, vos efforts et votre patience. Mais vous devrez lutter contre un ennemi bien plus puissant et dévastateur que l'obsolescence programmée, la crise économique, les délais de livraison, les composants douteux à des prix improbables, les aléas du traçage des colis ou le livreur qui ne sonne même pas à la porte. Cet ennemi est plus ancien et plus internalisé que la conjoncture socio-économique industrielle, et son unique solution pourrait vous faire passer pour un luddite ou un excentrique. Il ne tient qu'à vous de décider, et cette question cruciale n'est pas posée au bac philo : voulez-vous vraiment vous amuser, et accessoirement, à quel point est-il acceptable de se laisser enfermer dans un monde virtuel ?

– Résistons à l'informatisation galopante de l'électronique ! –

Quoi de mieux pour commencer cet appel à la prise de conscience qu'une citation de notre vénéré maître des lieux :

« Tiens, je vais relancer le projet Z80 homebrew. »

« Oh, une carte indus Z180 sur eBay ! »

« La MMU du Z180 est merdique, le 68008 est mieux. »

« Et si j'écrivais un script de build pour le cross-compiler m68k ? »

« Tiens, le cross-compiler sera utile pour FUZIX d'Alan Cox. »

« Oooh, FUZIX est porté sur la Pico/RP2040 ! »

Vous remarquerez que de fil en aiguille, non seulement le projet initial n'est pas terminé (la *sunk cost fallacy* est un tout autre sujet de discussion, voir aussi *« Histoire de side projects »* à **https://www.commitstrip.com/fr/2014/11/25/west-side-project-story/**), mais surtout l'attention s'est lentement déportée d'une question matérielle à un investissement logiciel. Comme cela reste « de l'embarqué », la pente glissante citée plus haut est toujours pertinente dans le cadre de ce magazine. Mais où est l'odeur du napa^Wflux le matin ? Où est le résultat palpable ? Quand pourra-t-on dire que « ce sera fini » ? Qu'en restera-t-il au final ?

1. TOUT EST INFORMATIQUE, ET INVERSEMENT

Si je me suis permis cette citation, ce n'est pas pour jeter la première pierre à notre dévoué rédacteur en chef, bien au contraire, puisque je suis aussi coupable de ce genre de glissement, et certainement de façon bien pire ! Et je suis malheureusement bien loin d'être le seul. Je me rends compte de cette « informatisation galopante de l'électronique » depuis au moins vingt ans et, contrairement à Denis, j'ai fait des promesses et même dû livrer des produits, ce qui m'a exposé encore plus aux paradoxes que cela implique.

Livrer un client est probablement l'une des plus belles satisfactions dans le domaine de l'électronique, c'est même un accomplissement. On se sent d'abord valorisé d'avoir eu raison de bout en bout (ou presque) dans un projet, d'avoir surmonté les inévitables aléas, appris de nouveaux « trucs », résolu les contraintes, combattu nos doutes et d'avoir « fait du bon boulot », qui va s'intégrer dans un « tout » plus grand, et pour lequel on reçoit (normalement) une récompense pécuniaire, en plus de la fierté et la reconnaissance. Mais surtout, le deuxième effet est une *libération* : le défi étant terminé, on peut enfin se reposer, profiter un peu de la récompense, passer à autre chose. Après une courte période de relâchement et de batifolage, qui nous permet de faire le point et de regarder autour de nous, il est courant de replonger dans autre chose et de *choisir* un nouvel objectif raisonnable. C'est pour cela que je me présente comme électronicien, même si je passe plus de 99 % de mon temps sur mon ordinateur.

En revanche, avec l'informatique, « on sait quand ça commence, mais jamais si ça finira ». Il n'y a pas ce soulagement d'avoir obtenu un résultat, car c'est *toujours perfectible*, d'une façon ou d'une autre, même si le résultat désiré est éventuellement atteint, et on ne choisit pas forcément nos objectifs : ils sont souvent subis par des contraintes externes liées à la plateforme. Par exemple, les développeurs sous Windows sont obligés de mettre à jour leurs outils toutes les quelques années, donc leurs programmes, leur propre code, pour se conformer aux nouvelles lubies de Microsoft (qui aime couper

les branches sur lesquelles sa communauté de développeurs s'est installée). Cette marche forcée contre l'obsolescence logicielle programmée est aussi devenue courante dans le monde des gadgets connectés ou des jeux en ligne qui sont jetés à la poubelle dès que leur créateur coupe l'accès à ses serveurs (mais c'est un autre débat).

Le monde des Logiciels Libres est un peu moins tyrannique à ce niveau, mais le même phénomène de mise à jour perpétuelle existe, bien que plus subtil. Ce n'est pas le *tonneau des Danaïdes*, puisque je peux toujours compiler du code écrit en C il y a 20 ans (comme ce bon vieux GDUPS **[gdups]**), mais GNU/Linux a considérablement évolué depuis, les distributions et tout l'environnement sont métamorphosés (pour le meilleur ou pour le pire *tousse*systemd*tousse*). On peut s'en sortir et la compatibilité étant une des valeurs fondamentales de POSIX, il existe une solution, qui demandera de se plonger dans les archives des Saintes Écritures (comme les *manpages* ou *StackOverflow*). La dissonance entre les principes simplistes d'antan (l'idée de base d'UNIX était simple) et la fuite en avant technologique obligent à faire de nombreux compromis. Tout cela n'est pas un choix, c'est du tir sur cible mouvante et on apprend au passage des choses dont on n'a pas vraiment besoin, sans rapport direct avec la problématique d'origine.

Mon insistance pour réinventer la roue vient de cette habitude d'éviter au maximum les dépendances : plus elles sont nombreuses, plus elles nous feront perdre du temps dans l'avenir, puisqu'elles casseront peut-être nos efforts au hasard, ici et là, et forceront un jour à abandonner le travail déjà réalisé, à cause d'un *bug* ou d'une fonctionnalité changée ou supprimée par quelqu'un d'autre, sans se préoccuper de notre avis. Un système autocontenu, indépendant, avec le moins possible d'outils externes va durer plus longtemps qu'un ordinateur sous Windows, par exemple (et c'est ce qui a contribué au succès des cartes Raspberry Pi). La pérennité est justement ce que peut apporter un système électronique correctement conçu, mais même les plus simples gadgets se retrouvent « infectés » par du code qui ne peut être écrit, compilé et flashé que par un outil très spécifique tournant sur une plateforme particulière (car son créateur ne veut pas ou ne peut pas porter et tester sur toutes les plateformes possibles, ou parce qu'il est déjà acquis à un écosystème). L'informatique a rendu l'électronique éphémère...

2. L'AUTRE « NO CODE » ET LE CHEMIN DE LA PÉRENNITÉ

Car l'informatique est éphémère par essence, alors que l'objet électronique dure et persiste (au risque de polluer) ce qui permet de vraiment le posséder (ou du moins, c'est ce que l'on en attend). Il y a forcément des contre-exemples, mais à moins de travailler sur un système en COBOL (ce qui n'est généralement pas fait pour le plaisir), votre prochain *commit* sur *GitHub* restera pertinent pendant un an, trois ans voire dix ans, alors qu'un appareil que vous aurez réalisé restera sur votre bureau (ou dans vos cartons) bien plus longtemps, témoin de vos capacités (*et de votre résistance*, hahaha), qu'il soit utile ou décoratif, fonctionnel ou en pièces détachées. Pensez à tout ce qui l'empêcherait de marcher ou réduirait ses fonctionnalités ou son évolutivité, une fois que les piles usées seront changées, et les condensateurs remplacés. Si l'objet utilise un microcontrôleur ou (pire) un

FPGA, c'est mal parti, même si vous disposez du code source.

Or, tous les projets n'ont pas besoin d'une telle complexité. À la fin d'un atelier d'initiation à l'électronique que j'avais animé pour un groupe d'enfants, l'un d'eux a justement partagé sa remarque enthousiaste :

« On peut faire clignoter une LED sans utiliser d'Arduino ! »

Et nous n'avions abordé qu'une seule des très nombreuses façons de le faire à partir de composants de base. L'atelier n'avait pas pour objectif de former de futurs ingénieurs, mais de faire découvrir des concepts fondamentaux (comme les deux Lois de l'Électronique*), d'attiser la curiosité et de faire entrevoir la richesse qui se cache dans les objets de notre quotidien moderne, pour les démystifier. Un smartphone n'est pas magique !

Pourtant, de nos jours, la tentation est forte de prendre les dernières avancées pour acquises, alors que nous baignons dans la facilité apparente de la technologie, sans en réaliser les héritages profonds, les implications et leur fragilité. Les plus jeunes aiment le confort et le potentiel des systèmes dernier cri alors que les autres pensent (à juste titre) que « c'est trop compliqué ».

Mais faire clignoter une LED ? Il n'y a rien de magique ou d'insondable. C'est le B-A-BA et le premier exercice lors de la prise en mains d'une nouvelle plateforme embarquée. Évidemment, une platine Arduino n'a pas pour unique finalité de juste faire clignoter une LED, c'est capable de tellement plus. Pourtant, vous connaissez le proverbe : « quand on a un marteau, tout problème devient un clou » et je ne compte plus le nombre de projets où j'ai vu une grappe de platines Arduino, déployées pour des tâches extrêmement basiques, en batterie, « parce que ça marche, c'est rapide et ce n'est pas cher » au lieu de réfléchir à l'architecture et l'extension du système (« Il vous faut un hub USB à *combien de ports* ?! »).

Avant Arduino, il y avait aussi le « péril » posé par le NE555 (« *plus personne ne sait câbler un bistable* »), mais celui-ci ne nécessite pas tout un environnement informatique pour le programmer et sa structure ne sera jamais mise à jour. De l'autre côté, les *startups* nous promettent un avenir « no code » où nous pourrons programmer sans écrire de code (ce qui est ridicule, car *tout est code*, même les environnements graphiques à base de blocs comme celui de Scratch), alors que le « no code » idéal serait justement de *ne pas avoir besoin de code*, de programme ou d'algorithme.

Évidemment, la programmation est inévitable dès que la complexité dépasse le clignotement d'une LED, et elle sera toujours nécessaire, mais il est aussi important de savoir *reconnaître quand s'en passer* puisque dans de nombreux cas, intégrer un microcontrôleur ou un ordinateur dans un appareil ne résout pas tout et peut même compliquer énormément un système, ou pire : nous en rendre esclaves.

L'exemple le plus flagrant est le mirage de la domotique, qui promet depuis des décennies un avenir merveilleux où tout se fera à notre place, si l'on accepte de croire qu'un gadget électronique (souvent très propriétaire) sera plus durable, fiable, pratique et moins cher qu'un simple interrupteur mécanique qui ne coûte que quelques euros, ne

* Quand on branche, ça fonctionne mieux. Quand on branche dans le bon sens, ça fonctionne encore mieux !

nécessite aucune batterie, aucune mise à jour, aucun abonnement et n'a aucun risque de piratage... Les ampoules dites « connectées » incompatibles entre différents fabricants, les différents protocoles qui ne sont pas interopérables, les systèmes complexes et fermés « briqués » lorsque le fabricant disparaît au bout de quelques années, tout cela ne semble pourtant pas entrer dans les consciences des consommateurs. Ils continuent d'en acheter, juste pour voir, car c'est abordable donc au pire, le risque est faible.

Il y a certainement des cas où des « solutions domotiques » peuvent avoir un intérêt, en particulier les personnes handicapées (mais ce n'est pas un marché très populaire). Cependant, pour le commun des mortels, cela reste « une solution à la recherche d'un problème ». Une maison n'est pas une usine. Un interrupteur, une ampoule, je ne vois pas où est le problème, pour moi c'est ça le vrai « no code » et ça fonctionne très bien depuis cent ans.

3. SURVIE EN MILIEU INFORMATIQUE

Pourtant, nous savons bien que tout est maintenant numérique et il est impossible d'en échapper, à moins de rester attentifs à toutes les alternatives. Cela demande aussi un investissement considérable et intenable, ce qui d'ailleurs justifie l'abonnement à ce magazine pour externaliser la veille technologique. Et n'oublions pas que **tout choix reste un compromis** en fonction d'un cahier des charges, des compétences et des moyens disponibles. La mode du moment est une distraction, ce qui compte est l'application.

Souvent, un simple automatisme n'a même pas besoin d'un microcontrôleur et, étant d'une certaine génération, j'avais appris la logique pneumatique au lycée, ainsi que son abstraction en GRAFCET (*Graphe Fonctionnel de Commande des Étapes et Transitions*) qui permet ensuite sa traduction (manuelle ou logicielle). J'ai aussi pu toucher à un automate programmable PB15 en classe de technologie, il y a presque trente ans. De nos jours, les lycéens apprennent le Python, donc ils sont opérationnels sur des plateformes sophistiquées comme Rasperry Pi ou micro:bit, mais rarement plus basique : lorsqu'ils devront faire clignoter une LED, certains penseront avoir besoin de tout un système d'exploitation derrière.

Plus récemment, j'ai aussi *re*découvert la logique à relais (évoquée ici **[hk25]**) qui peut avoir des avantages très intéressants en termes de fiabilité et de réparabilité, à condition de bien garder la documentation du système. D'un côté, en utilisant des relais à bobine alternative 220 Vac, il n'est plus nécessaire de se préoccuper d'une alimentation basse tension qui grillera au bout de quelques années, et en cas de défaillance, il suffit de remplacer un ou deux relais sur un rail DIN, un de ces standards increvables. D'un autre côté, cela prend évidemment un peu plus de place et la consommation statique n'est pas négligeable (on peut aussi parler de fonction de dégivrage intégrée).

Mais cette solution a été la meilleure pour un projet visant deux ou trois décennies de longévité, probablement plus. J'ai identifié que le système correspondait à une machine à états finis à quatre états et c'était réglé avec juste trois relais, simplement en arrêtant de penser de façon algorithmique. J'aurais pu utiliser un PLC (automate programmable) industriel, mais les

modèles actuels sont typiquement programmés par un ordinateur, évidemment sous Windows, et une telle configuration a une durée de vie de 5 ans en moyenne, sans compter les aléas de stockage de la machine, du renouvellement de la licence du logiciel ou la préservation du code source en *environnement d'entreprise* (où les personnes vont et viennent en créant une sorte d'amnésie institutionnelle). La technologie se heurte donc à la culture et aux habitudes des humains, et perdure tant que ces derniers y prêtent encore attention. Et c'est justement parce que le fabricant avait « égaré » les schémas d'origine (vieux de trente ans) que j'ai été appelé à la rescousse !

De plus, les objets technologiques courants ont une « garantie constructeur ou importateur » durant typiquement un an, parfois trois, donc il est impossible de se projeter dans le futur lointain. Le consumérisme actuel a permis une baisse phénoménale des prix, mais les appareils d'antan, jusque dans les années 1980, proportionnellement plus chers, étaient aussi destinés à durer très, très longtemps (sinon personne n'aurait osé *investir* dedans). Lorsque le schéma d'un appareil (comme un récepteur radio ou un poste de télévision) n'était pas collé à l'intérieur du boîtier, on pouvait acheter ou consulter de gros recueils de schémas comprenant aussi des points de tests pour la calibration ou le dépannage. Aujourd'hui, on assiste à une prise de conscience et les législateurs (aux USA comme en Europe) comme les hobbyistes (voir la figure 1) commencent à réclamer le droit de ne pas devoir jeter un appareil qui a juste un petit souci.

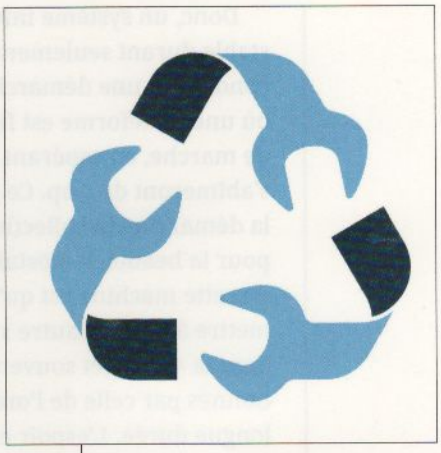


Figure 1 : Ce qui est vieux redevient nouveau. Il fut un temps où réparer les objets était naturel, aujourd'hui c'est une lutte innovante. Source : Yves Parent à https://github.com/Solidifyconceptdevelopment/Repair-symbol.

C'est aussi pour cela que malgré certains composants un peu exotiques, les oscilloscopes Tektronix de mon âge sont bien plus réparables (et donc prisés par les amateurs) que des générations plus récentes de la même marque. D'ailleurs, lors de son acquisition il y a 20 ans, j'avais scanné le manuel de référence de mon fidèle 475 et ces fichiers sont aujourd'hui quasiment inaccessibles (où sont les CD et sont-ils encore lisibles ?) alors que l'appareil trône toujours parmi ses congénères sur mon espace de travail. Les fichiers passent, les circuits restent, et prions pour que **archive.org** survive éternellement.

N.B. : Croyez-vous aux coïncidences ? Il n'a pas fallu longtemps pour que l'expérience rejoigne la pratique. « Comme par hasard » quelques jours après avoir terminé cet article, c'est le 2465 qui a claqué son RIFA. Il n'a suffit que de quelques minutes de recherche en ligne pour trouver le Service Manual complet de ce formidable modèle, ainsi que des vidéos et un blog décrivant justement cette panne. Je peux donc réparer mon matériel, seul, sans crainte.

Donc, un système informatique est par nature fragile, stable durant seulement quelques années. Cela peut conduire à une démarche de « survivalisme informatique » où une plateforme est figée dans le temps, archivée en état de marche, en espérant que ni les batteries ni les piles ne s'abîmeront de trop. Ce genre d'archivage contraste avec la démarche du collectionneur qui entretient la machine pour la beauté, la nostalgie ou la spéculation : la valeur de cette machine est qu'elle est vitale pour maintenir ou mettre à jour un autre système obsolète plus important, dont la *survie* (et souvent les enjeux financiers) sont conditionnés par celle de l'ordinateur placé en hibernation longue durée. L'espoir est qu'il suffira de le rebrancher en cas de besoin, pour retrouver un environnement intact qui minimisera le temps d'intervention, évitant le *reverse-engineering* et une nouvelle réinvention de la roue. Mais ne venez pas me parler de virtualisation : dans les cas qui nous concernent, le logiciel est souvent trop fortement lié au matériel et aux interfaces obsolètes !

Oubliez le port USB et la mise à jour des pilotes de périphériques, je vous parle d'un temps où cela n'existait pas. Une époque où il était normal d'accéder directement aux broches des ports d'entrées-sorties d'un PC, avec des instructions IN et OUT à des adresses précises, pour contrôler leur état. Le port série avec son connecteur DB9, et le port d'imprimante parallèle en DB25 étaient des espaces de bricolage incroyables, ceux qui ont bricolé leur programmateur PIC type JDM me comprendront. Un autre exemple parmi tant d'autres : je continue de stocker des PIC16HV dans leur boîtier à fenêtre pour effacement aux UV. Cette dernière opération nécessite un simple appareil (une boîte avec quelques tubes inactiniques), mais la programmation passait par un adaptateur spécifique pour le port DB25 (incompatible évidemment avec les dongles USB).

Avant l'an 2000, les séquences d'instructions étaient précisément codées sur PC pour générer des impulsions de durées spécifiques, ce qui déclencherait des exceptions ingérables dans un système moderne : l'interface matérielle n'existe plus, les délais implicites ne sont plus respectés ou les broches du *SouthBridge* ne sont plus connectées à un port physique. Et non seulement les opérations sur les ports d'E/S sont presque impossibles à émuler dans un OS actuel, mais en plus les logiciels d'antan (compilateurs et programmateurs de PAL, flasheurs d'EPROM et j'en passe) travaillaient en DOS 16 bits, ce qui n'est plus supporté par Windows depuis bien des années.

Et au niveau matériel, la situation s'est aussi dégradée, plus ou moins visiblement. Par exemple, un port RS232 fonctionne aujourd'hui avec des niveaux de 3 V ou 5 V (pour économiser quelques composants) et le vénérable MAX232 ne peut pas dépasser +/-10 V alors que d'anciens appareils attendent des niveaux de 12 V *au moins*. Les ports DB9 et DB25 ne sont pas les seules victimes de l'évolution forcée des PC : le port miniDIN5 du clavier et de la souris était tout à fait *hackable* aussi, sans parler du port ISA, qui fut supplanté par le port PCI (lui-même jeté aux oubliettes). Aujourd'hui, la barre est malheureusement placée très haut si l'on désire se connecter à un port PCI Express.

Le logiciel est fortement lié à la machine, qui elle-même est bloquée dans une époque. Quelles autres interfaces seront supprimées, empêchant l'interfaçage de nos programmes avec l'extérieur dans le futur ? Comment survivrons-nous aux prochains élagages technologiques sans préserver les machines actuelles dans des archives ? Comment

garantir la pérennité à très long terme d'un projet lorsque l'industrie avance à marche forcée et rend tout obsolète en quelques années, sans que nous passions pour des ennemis du progrès ?

J'avais cru trouver une solution en me concentrant sur les ordinateurs de type PC industriels, car ceux-ci sont plus modulaires et proposent plus d'interfaces utiles, au-delà du port USB, HDMI et Ethernet (quand ils sont encore disponibles, comme si le Wi-Fi résolvait tout). Cela m'a mené à une collection de cartes PC104 et PICMG qui fut stoppée il y a dix ans par l'arrivée de la carte à la framboise. Le Rapsberry Pi a changé la donne avec son connecteur d'extension GPIO très puissant, mais au final, c'est un retour à la case départ puisque les efforts sont de nouveau concentrés sur le logiciel.

4. L'APPEL DES SIRÈNES MÉTAVIRTUELLES

C'est donc un fait que l'informatique est inévitable dans tous les domaines de notre vie actuelle, même pour la conception électronique :

- Beaucoup de livres, si vous en aviez, sont obsolètes ou numérisés donc au début d'un nouveau projet, c'est normal de consulter toutes les ressources disponibles d'Internet.
- Les sites des fabricants, la disponibilité et les prix des différents distributeurs, les autres projets ayant utilisé des composants envisagés, cela passe aussi par votre navigateur.
- Vous pouvez maintenant simuler un circuit analogique ou numérique relativement fidèlement avec SPICE ou même directement dans votre navigateur avec CircuitJS (surnommé Falstad **[hk33]**). C'est même devenu indispensable pour saisir, tester et partager des idées grâce à ses fonctionnalités inégalées d'exportation et d'échange des données.
- Saisir le schéma, placer les composants et router les pistes se déroule sur votre ordinateur avec des logiciels que vous avez choisis, installés, configurés et pris en mains, ce qui est un autre investissement préalable. Et si vous ne l'aviez pas fait, vous devez maintenant le réaliser en même temps que le projet lui-même. Quelle démarche prendra le dessus sur l'autre ?
- Une fois votre fichier GERBER généré, il est soumis à un fabricant sur un autre continent par une interface web. Des sociétés proposent même des services de *sourcing* des composants, de soudure et de test. L'interaction par Internet continue avec la gestion de la livraison, suivie plus ou moins en temps réel sur le site du transporteur.

Et jusqu'à ce moment-là, vous n'avez peut-être pas encore touché à un seul composant concret. Mais toutes les étapes de cette liste ne sont que des améliorations d'un processus qui fut « analogique » il y a encore vingt ou trente ans, où l'ordinateur apporte de l'efficacité, de la précision (pour les composants montés en surface), de la répétabilité, de la gestion de la complexité…

D'ailleurs, rien ne nous empêche de faire des prototypes sur des platines prétrouées ou sans soudure, de sortir les décalcomanies de pastilles et les rouleaux d'adhésif pour dessiner directement sur le cuivre, que l'on grave à l'acide chaud dans un bac : c'est juste plus pénible, plus salissant, plus cher, moins fiable. Il est toujours possible et même

recommandé de tester à la main la valeur de chaque composant et chaque tension, ce qui devient de plus en plus difficile à mesure que les circuits intégrés englobent de plus en plus de fonctions.

Parmi les puces complexes, les microcontrôleurs ajoutent leurs couches logicielles, de programmation et de configuration (qui est déjà une forme de métaprogrammation). Les plateformes comme Arduino ont un peu simplifié l'usage de ces circuits dont le manuel fait quelques centaines de pages en moyenne, au moyen de quelques couches d'abstraction. Soit, mais au moins la puce existe, car il existe un monde où on sort difficilement son nez de son ordinateur.

Les FPGA sont un ordre de grandeur plus compliqué encore. D'abord, il faut utiliser plusieurs langages par exemple, même si le principal reste celui de la description des circuits (souvent Verilog ou VHDL). Mais les fichiers de configuration spécifiques aux constructeurs, ainsi que la gestion des données qui seront traitées par le circuit peuvent créer une sorte de Tour de Babel. Au moins, une puce FPGA de base seule ne coûte pas très cher, les coûts augmentent avec les circuits autour. Un kit de démarrage ou de développement peut coûter entre 50 et 300 € selon les fonctionnalités requises, donc il y a une chance pour que le projet aboutisse (après avoir pris un peu de poussière sur une étagère).

Je sais bien que nous, les *serial makers*, aimons les projets pour ce qu'ils nous apportent durant la conception, surtout l'excitation de l'exploration et de la découverte, donc la mise en pratique n'est pas toujours au rendez-vous. Un certain nombre de projets FPGA resteront au stade de triturage dans l'ordinateur un certain temps, pour se familiariser avec toutes les nuances d'une plateforme spécifique, avant qu'un résultat utile ne se matérialise un jour.

5. COINCÉ ENTRE DEUX MONDES

Mais tout cela n'est rien comparé à un milieu encore plus élitiste que les FPGA : celui de la conception des circuits intégrés. Comme j'en ai parlé dans ces colonnes **[hk36]**, ce qui était autrefois la chasse gardée de quelques universités et de quelques grosses sociétés est devenu aujourd'hui le terrain de jeu de nombreux amateurs qui ont attendu, semble-t-il, très longtemps pour pouvoir *enfin* jouer dans la cour des grands (sous le patronage conditionnel de Google). Mais ils ne sont pas soumis aux mêmes contraintes, en particulier celles du marché et de la rentabilité, et en combinant tout ce que nous avons vu dans ces dernières pages, cela conduit à une situation suffisamment paradoxale et délirante pour m'inciter à écrire cet article.

Revenons un peu en arrière. J'ai déjà pu constater que peu d'électroniciens sont assez doués pour écrire de *bons logiciels* (et je me compte dans cette catégorie). Bien sûr, écrire du code, et même du bon code, n'est pas si dur que ça, mais un logiciel est bien plus que des morceaux de code assemblés, c'est toute une « façon de faire ». Ce n'est pas la même démarche. Corollairement, de l'autre côté du spectre, un développeur logiciel talentueux et expérimenté n'est pas le plus qualifié pour mener à bien un projet électronique. Il n'en est pas forcément incapable, mais créer un système électronique fiable nécessite un investissement de temps et d'apprentissage tel qu'il ne serait plus

un développeur informatique, mais bel et bien un électronicien.

La double compétence est rare. Le résultat est que beaucoup d'outils d'EDA (automatisation de conception électronique) sont des cauchemars :

- soit parce qu'ils sont écrits avec les pieds par des débutants ou étudiants qui savent ce qu'ils veulent, mais ne savent pas comment faire, découvrent les technologies durant la réalisation, et créent une interface absconse en dépit de beaucoup d'usages et coutumes informatiques ;
- soit parce que le développeur se laisse aller à utiliser tout son savoir et projette ses talents informatiques d'une façon inadaptée à l'électronique.

Bien que ce soit particulièrement évident dans l'univers des amateurs, ceci n'est évidemment pas une dichotomie absolue et ces deux mondes se rencontrent justement lorsqu'il faut concevoir un circuit intégré. De nombreuses sociétés ont fait fortune en créant des logiciels à la fois très utilisables et parfaitement adaptés : les outils de Mentor Graphics m'ont fortement impressionné à la fin des années 90. Ce n'est donc pas une fatalité.

Pour faire progresser la situation, il faut non seulement de la double compétence, mais aussi suffisamment de personnes motivées et focalisées sur un but commun. C'était assez facile il y a quarante ans à l'époque de Mead-Conway, avec les contraintes techniques assez génériques, permettant à un étudiant de concevoir son petit circuit durant un semestre. Aujourd'hui, ce n'est plus le cas et la complexité de méthodes, des outils, des technologies et de leurs contraintes empêche une personne seule de maîtriser intégralement la chaîne de conception, de la définition du circuit jusqu'à l'intégration de la puce dans un appareil. Beaucoup de simplifications superficielles sont donc introduites, chacune apportant son lot de complexité derrière le rideau.

Bien sûr, Google a donné un coup de pied stratégique pour que les industriels prennent conscience non seulement de l'intérêt des petites gens pour le domaine, mais aussi de l'intérêt pour les entreprises de s'ouvrir un peu, car la première à divulguer son PDK (les fichiers de définitions de sa sauce secrète) aura un avantage stratégique (car tout le monde l'utilisera et il deviendra un *standard de fait*). Skywater et Global Foundries ont pour l'instant compris ce qu'ils avaient à gagner et les « amateurs » ont maintenant de vraies données à éplucher, remplaçant finalement les couches d'abstraction placées dans le domaine public par quelques universités (qui ne voulaient pas contrevenir à leur NDA). C'est la fin d'une ère très gênante pour tout le monde, où l'autocensure était implicitement de mise.

Cette deuxième révolution déclenchée par Google, 45 ans après celle de Carver-Mead, apporte son lot de nouveaux projets et d'outils ouverts. Un exemple frappant est TinyTapeout et **wokwi.com** : ce sont des « bacs à sable » inspirés par l'environnement Arduino, qui permettent d'avoir une petite idée de comment une puce fonctionne et est conçue. Quasiment aucune puce financée par Google, contenant un circuit choisi au hasard par une loterie, n'aura d'utilisation réelle à cause des nombreuses limitations : nombre de circuits livrés, nombre de portes logiques et surtout le faible nombre et la vitesse des broches d'entrées/sorties. Donc, il n'est pas possible d'explorer la vitesse et des paramètres subtils affectant la performance d'un circuit numérique, par exemple.

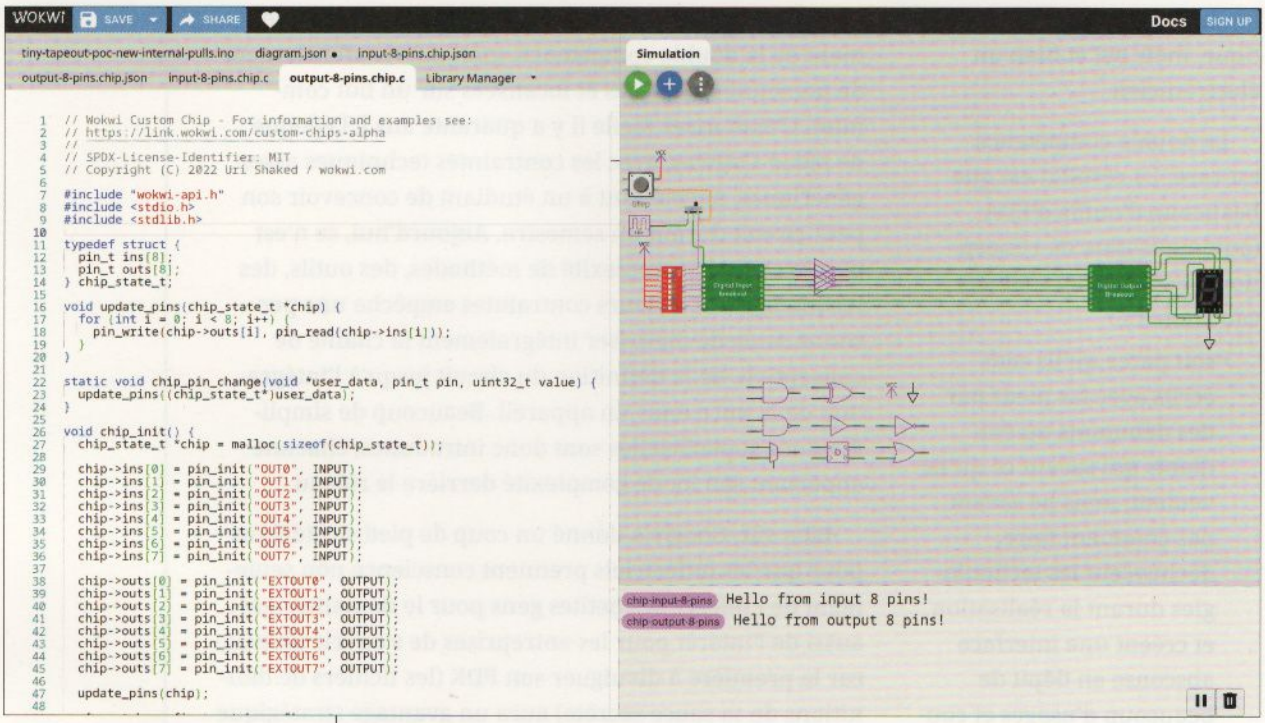


Figure 2 : Copie d'écran de l'interface Wokwi permettant la conception de petits circuits intégrés numériques en s'inspirant de l'interface Arduino.

TinyTapeout permet de réaliser un décodeur pour afficheur à 7 segments et il y a d'innombrables autres façons d'en réaliser moins cher. Le résultat est éventuellement un *jouet pour nerd* qui trônera fièrement sur un bureau, mais ne résoudra aucun problème (puisque n'importe quel FPGA est plus puissant et moins pénible) et sera à la limite de l'utilisable en pratique (même si certaines limitations pourraient se relâcher dans le futur).

Un des aspects ennuyeux de ces nouveaux outils pour débutants est qu'en essayant de cacher la complexité sous-jacente, on éclipse aussi tous les outils, les standards et les méthodes de conception modernes issues de dizaines d'années de développement. C'était frustrant à l'arrivée d'Arduino : on n'écrit plus du code source en C(++), mais on écrit un « sketch » dans « le langage Arduino ». Le *rebranding* est passé comme une lettre à la poste : en évitant de parler de « trucs de geeks », on rend la plateforme tout de suite attrayante, ce qui a inspiré d'autres projets. Rien que Wokwi, sur la figure 2, utilise trois langages en surface (ino, JSON et C) et aucun n'est spécifique au monde des circuits numériques. Verilog reste disponible, caché derrière des couches de métaprogrammation, alors que VHDL aurait toute sa place justement comme langage unifié, établi, solide et standardisé depuis 1987.

J'ai finalement compris que ce qui compte pour les humains n'est pas l'effort mais sa perception. Lorsque l'on débute, on ne sait pas tout ce dont on aura besoin

plus tard et on tend naturellement vers la facilité et la rapidité, sans réaliser qu'on s'engage sur un chemin inutilement compliqué, paradoxalement. Une rapide recherche sur Internet permet de répondre à la plupart des questions techniques pointues, c'est la beauté du confort moderne qui fait de nous des consommateurs peu motivés à s'investir.

6. RÉINVENTONS ENCORE LA ROUE !

Pourquoi utiliser ce qui existe déjà alors que l'on peut tout simplement le réinventer ? Il y a du mérite à l'idée d'attirer les utilisateurs d'Arduino, en leur proposant un langage qu'ils connaissent déjà, même si le fichier source est entouré de formats *ad hoc* et non standardisés, donc non portables. Après tout, même dans l'industrie, chacun se fait sa propre sauce, car c'est souvent plus rapide à bricoler (« on verra plus tard pour la maintenance, l'important est de livrer à temps »).

D'autre part, un développeur seul et capable de mettre au point une interface telle que Wokwi n'est peut-être pas coutumier de tous les standards, méthodes et contraintes du milieu de la microélectronique. C'est vrai que c'est une sous-culture à part entière avec ses propres problématiques de brevets, de *copyrights*, de secrets industriels et surtout, de gros sous.

Mais souvent, lorsque la roue est réinventée, j'ai constaté que ce n'est pas autant un syndrome NIH (*Not Invented Here*) qu'une ignorance de ce qui existe déjà et *pourquoi* cela fut mis au point. L'exemple le plus courant est les langages de description du matériel (HDL en anglais) : Verilog fut créé en premier, à une époque à chaque fabricant avait son propre langage interne pour la description des composants, et ses utilisateurs ont vite été dépassés par ses limitations. Le gouvernement des USA a demandé une standardisation des HDL et le VHDL a été construit sur la base déjà établie et puissante d'Ada (adopté quelques années plus tôt).

Aujourd'hui pourtant, Verilog est souvent préféré « parce qu'il est simple » et VHDL est considéré comme « trop obscur et compliqué » (c'est ce qui arrive quand on veut tout faire avec un seul outil, son *Language Reference Manual* est long et souvent abscons). Et inexorablement, les utilisateurs de Verilog rencontrent tôt ou tard, de nouveau, les limites de leur langage et vont bricoler un truc pour contourner telle ou telle contrainte. Au final, on obtient SystemVerilog qui réutilise beaucoup de concepts dont VHDL disposait depuis quinze ans, sans pour autant égaler la puissance de ce dernier. Peu importe, si SystemVerilog n'est pas assez puissant, il suffit de l'interfacer avec du code en C (en utilisant la *Direct Programming Interface*), avec tous les *bugs* que cela risque d'injecter. Et si le C ne suffit pas, générons du Verilog avec Python (comme Migen) ou tout autre langage de haut niveau. On peut alors réaliser des fonctions déjà fournies par défaut par VHDL depuis 1987 et continuer d'ignorer GHDL qui est déjà opérationnel depuis quinze ans. Nous retrouvons un phénomène similaire à « l'arduinisation » : l'impression de facilité mène paradoxalement aux difficultés.

De plus, la multiplication des langages (motivée par la familiarité avec ces derniers) à l'intérieur d'un projet est rapidement compensée par la complexité et surtout les différences subtiles entre les notions et paradigmes de chaque langage : ce que l'un peut faire n'est pas forcément accessible à l'autre et vice versa. C'est le même phénomène qui risque de se produire de nouveau, lorsque des débutants vont apprendre des méthodes créées par d'autres débutants et perpétuer une

nouvelle sous-culture, recréant un système qui convergera vers ce qui existe déjà depuis longtemps, mais mal architecturé et avec un nom plus à la mode, qui ne provoquera pas de blocage mental à son évocation. De même, avant la déferlante d'Arduino, on parlait simplement de « programmation de microcontrôleur » ou d'AVR, et il en est toujours ainsi sous le capot, mais c'est avec la nouvelle étiquette et une attitude plus « noob-friendly ».

De mon côté, j'ai choisi la voie du « tout VHDL » pour minimiser le nombre de langages. Au lieu de TCL, j'utilise simplement `bash` pour scripter les compilations par GHDL et lancer les exécutables. Les extensions en C sont possibles, mais la révision VHDL'93 fournit déjà quasiment tout le nécessaire depuis trente ans.

7. TEST. TEST EARLY. TEST OFTEN !

Avec le mouvement des Logiciels Libres qui a su s'imposer depuis deux décennies, tous les geeks sont aujourd'hui des informaticiens. Donc, ils abordent chaque nouveau projet selon cette perspective, appliquent la même approche, réutilisant tous les outils à leur disposition et on peut dire qu'il y en a beaucoup, dont de nombreux qui sont utiles pour des projets microélectroniques. Mais les outils ne suffisent pas.

Encore une fois, on retombe dans une problématique de culture et de méthodologie, et les contraintes de la conception d'un circuit électronique ne sont pas les mêmes que pour générer un `.rpm` ou un `.deb`. Imaginez un instant que les signatures SHA n'existaient pas et que les dépôts de paquets de votre distribution vous livraient, au hasard, un fichier plus ou moins subtilement corrompu, avec disons 50 % de chance. La détection de certaines erreurs pourrait obliger à faire fonctionner le programme longtemps, mais quelle est la durée d'un test qui garantit une parfaite intégrité ? La réponse est bien sûr *infinie* puisque cela touche à l'indécidabilité et au problème de l'arrêt de Turing.

Je me permets cette allégorie pour insister sur la différence d'objectifs et de contraintes, donc de moyens et de méthodes, entre la théorie des programmes informatiques, et la variabilité et la faillibilité des systèmes physiques. Ces derniers sont testés un à un pour vérifier leur conformité à un cahier des charges. Une puce électronique, à peine sortie de la fonderie, a intrinsèquement des défauts, parfois indétectables.

Les développeurs logiciels utilisent bien sûr des techniques comme l'intégration continue ou la QA (*Quality Assurance*) mais peu de projets l'implémentent réellement, complètement, et d'autres se contentent d'un *bug tracker*. En dehors de projets critiques comme GCC et du *kernel* Linux, je n'ai jamais vu la démarche de *vérification en amont* mise en avant, à part peut-être lorsqu'il est trop tard et un *bug* se manifeste, donc la validation est souvent mise dans le même sac que les outils de déverminage. Les langages permettant la vérification formelle existent mais sont souvent considérés comme appartenant à une niche, c'est moins *fun* à utiliser qu'un langage à la mode.

Les tests unitaires, ces programmes externes qui testent chaque fonction d'un programme cible, afin de vérifier leur conformité et leur comportement, ne sont pas assez généralisés dans le monde informatique. Souvent, une fonction est écrite, testée dans le programme final lui-même pour voir, et on passe à la suite. Ça fonctionne la plupart du temps, mais ce n'est pas un luxe que peuvent se permettre les concepteurs

de circuits intégrés, surtout à une époque où un jeu de masques coûte des millions de dollars. Si les défauts physiques sont inévitables et aléatoires, il est hors de question que *toutes* les puces qui sortent de l'usine passent à la poubelle à cause d'une erreur obscure dans un fichier de code source (les *errata* d'Intel tels que le *F00F bug* ou *FDIV* devraient nous servir de leçon).

Donc, le code source du circuit électronique doit être rigoureusement, exhaustivement vérifié, à toutes les étapes de la conception pour corriger les erreurs le plus tôt possible. On appelle cela la démarche « *Design for Testing* » (ou DFT) que certains limitent à simplement ajouter un port de contrôle et d'observation (une chaîne JTAG). Pourtant, la DFT n'est pas un produit et va bien au-delà de l'ajout d'un BIST (*Built-In Self-Test*) ou l'utilisation d'un ATPG (*Automatic Test Pattern Generator*), comme on ajouterait l'option `-Wpedantic` et `-Warray-bounds` à une invocation de GCC.

Pour reprendre l'allégorie précédente, la démarche DFT est la différence entre « je sais que ça marche » et « voici la preuve que ça marche ». Comme le problème de l'arrêt l'a montré, l'indécidabilité d'un test global oblige à diviser le système pour valider individuellement chaque élément simple de sa structure. En pratique, chaque sous-circuit doit être testé et vérifié, aussi bien durant la conception (au moment de la simulation) que lorsqu'il sera intégré sur la puce. Aucune erreur n'est permise.

Nous en revenons aux tests unitaires, qui reformulent le comportement attendu du circuit, et comparent avec ce que ce dernier fait réellement. Typiquement, pour chaque fichier source HDL, un autre fichier *doit* générer des vecteurs de test et vérifier le résultat. C'est précieux pour vérifier les régressions, mais aussi pour réfléchir *dès le début du projet* à une méthode de test pour le circuit final (il est facile de concevoir un circuit que l'ATPG ne pourra pas couvrir totalement, il faut relancer ce dernier très souvent pour détecter rapidement si la conception emprunte une mauvaise voie). C'est une habitude saine à prendre, et d'autant plus évidente en VHDL, car le même langage permet aussi bien de décrire un circuit que son comportement, le tout simulé avec un seul logiciel, sans perte de sémantique.

Les tests unitaires, scriptables et automatisés prennent du temps supplémentaire de développement, mais ils en économisent plus tard en réduisant les incertitudes. J'ai repris cette idée pour mes développements en C où, à la fin d'un fichier de fonctions, j'ajoute un `main()` activé par un `#define` : en plus de valider les fonctions (qui pourraient faillir lors du portage vers une autre architecture), cela documente aussi le code en fournissant des exemples fonctionnels dont l'utilisateur peut s'inspirer. De plus, cela s'intègre facilement dans les tests de régression ou l'intégration continue.

Je ne retrouve pas cette habitude salvatrice dans Wokwi ou la plupart des outils de CAO électronique amateurs. C'est normal, dans un sens, puisque jusque-là, les applications étaient surtout pour les microcontrôleurs ou les FPGA dont les puces sont vendues sans défaut (vérifiez quand même si vous achetez en Chine).

8. TOUT ÇA POUR ÇA ?

Le panorama que je viens de dresser était latent depuis très longtemps et je suis aux premières loges depuis ce fumeux projet F-CPU en 1998. Mais depuis quelques années, les choses évoluent brusquement et beaucoup de geeks pensent que

leur rêve de silicium va enfin se réaliser, sans se rendre compte du décalage entre leurs habitudes de développement et la sous-culture de la microélectronique. La confusion est potentiellement renforcée par l'apparition des « outils pour débutants » jouant sur « l'arduinisation ».

La démarche de Google, bien que discutable sur certains détails, est tout à fait louable et assez claire : d'une part, Google désire favoriser la mise au point d'outils « relativement libres » pour la conception de (ses) puces, d'autre part il accumule beaucoup d'expériences en testant simultanément beaucoup plus d'approches que sa propre équipe n'en serait capable, grâce à un vivier de personnes relativement compétentes (et « embauchables » si vient le moment). Cette approche de *crowdsourcing* est assez rentable, même si je ne connais pas le budget réel. On peut en avoir un ordre de grandeur puisqu'un circuit de base revient à environ 10000 $ chez **efabless.com**, soit le salaire d'un ingénieur Google, sans avoir à payer les chasseurs de têtes ou la DRH.

En échange, les gagnants de la loterie reçoivent (après avoir payé la douane et autres frais) des exemplaires de leur petit circuit pour jouer avec. L'idée est d'expérimenter, se permettre d'échouer, itérer, améliorer : le premier lot a eu un joli souci à cause d'une mauvaise option de configuration d'un outil, mais peu importe puisque le pipeline de conception est amorcé et le but est d'apprendre des erreurs. Cependant, les limitations actuelles des entrées-sorties de la puce (peu de broches et faible vitesse) empêchent d'exploiter pleinement toutes les capacités offertes par une technologie 130 nm totalement exposée par un PDK ouvert. Donc, les résultats sont de petits circuits tels qu'on en ferait dans un cours d'introduction à l'université, sans prétention, le concepteur est fier si un des circuits qu'il reçoit fonctionne, et la boucle recommence.

Tout cela réduit considérablement le coût de l'apprentissage pratique, du moins pour la partie de la conception des outils de CAO et des circuits intégrés. Les puces peuvent être livrées sur un circuit imprimé déjà conçu, car tout le monde n'est pas capable d'en réaliser un : c'est une très bonne idée, mais alors les concepteurs amateurs ne sont pas exposés aux contraintes d'interfaçage avec le monde physique. Car si on fabrique une puce, ce n'est pas pour l'exposer sur un mur, mais bien pour qu'elle *serve* à quelque chose, non ?

Au final, quelqu'un rêve de fabriquer sa propre puce parce qu'elle résoudrait un problème ou apporterait quelque chose qui n'existe pas encore, avec de meilleures performances si possible. C'est particulièrement frappant dans le monde des microprocesseurs qui est en perpétuelle expansion : les idées d'architectures continuent de foisonner et l'envie est forte de les réaliser. Si les FPGA sont possibles et abordables de nos jours, fondre sa propre architecture dans du silicium est le *nec plus ultra*. Malheureusement, peu de projets ouverts ou libres aboutissent vraiment (j'avais cité OpenRISC et LEON il y a trois ans).

Prenons l'exemple d'un SoC libre étendant un jeu d'instructions établi, projet dirigé par un libriste très expérimenté en collaboration avec une université française. Grâce à quelques collaborations et un financement, une puce a été fondue.

Et c'est tout.

Cela s'est passé durant la période du COVID, mais elle est terminée et je ne suis même pas sûr qu'un exemplaire de la puce ait été mis sous tension ou monté sur une carte. Il n'y a rien à

montrer, pas de prototype à afficher, pas de preuve physique, aucun code exécuté sur une implémentation tangible. Pas de `hello world!` ni même de *blink* en deux ans. Tous ces efforts semblent être retombés comme un soufflé.

On peut argumenter que « c'était juste un essai pour voir », que le design n'était pas complet, mais alors pourquoi ne pas aller jusqu'au bout de ce qui est déjà disponible ? Pourquoi avoir fait tous ces efforts sans vérifier le résultat, pourquoi renoncer si près du but à apprendre des erreurs ? Où est la démarche itérative, sans le *feedback* des tests ?

Évidemment, le développement continue : puisqu'il s'agit d'un microprocesseur, non seulement le code HDL doit être développé, mais aussi le compilateur et des programmes, des librairies, et tout ce qui va avec. C'est beaucoup pour une équipe de quelques personnes, même si elles sont un peu rémunérées. Donc, que fait un informaticien ? Il fait de l'informatique, pendant que la première puce, l'aboutissement symbolique pour tant de personnes, prend la poussière quelque part. Les concepts continuent de se développer, les logiciels s'améliorent, mais rien n'a été *testé en vrai*, puisqu'« en théorie ça marche » et au pire, on pourrait le vérifier en connectant une sonde JTAG pour balayer la *scanchain* interne. On attend la deuxième puce plus complète.

Je suis effaré que des milliers d'euros aient été dépensés ainsi. Pourtant, je n'ai de leçon à donner à personne si l'on considère les fortunes que j'ai investies en composants qui ne serviront peut-être pas. Mais je sais que si je recevais une puce que j'ai conçue, la première chose que je ferais serait de la loger sur un support pour la tester. Ce support ferait partie d'un système déjà conçu en amont, en même temps que la puce, j'aurais même testé le testeur avec un FPGA pour simuler la puce. Le temps de fabrication d'une puce est suffisamment long pour en profiter pour concevoir et valider le banc de test physique.

Donc, ce n'est pas l'allocation de l'argent, mais des efforts qui me chagrine. Cela met simplement en lumière une différence de méthode, d'approche et de démarche, une fuite en avant qui trouve sa source dans l'informatisation des mentalités. L'électronique se virtualise et se dématérialise, ce qui nous affranchit des considérations telles que la validation et la vérification, ou juste obtenir l'aboutissement libérateur d'un résultat.

9. UNE PERTE SÈCHE

Se réfugier dans l'informatique ou se retrouver piégé(e) dedans appauvrit notre vie en général.

Il est clair qu'un ordinateur n'a besoin que de courant et d'une connexion à Internet pour nous être utile, ce qui prend beaucoup moins de place et d'investissement matériel qu'un bureau encombré de composants, d'outils et d'appareils de test. Mais la finalité est bien différente : l'intérêt est de pouvoir fabriquer nos propres gadgets et donc de nous les approprier, à l'heure où même nos données sur le « cloud » nous échappent.

La démarche informatique impose la rigueur et affûte l'esprit, la logique, la déduction et la créativité, mais réaliser un objet réel et complexe comme un système électronique fait appel à des compétences beaucoup plus larges que l'on attribue souvent aux ingénieurs diplômés, sans oublier l'amusement et le plaisir de prendre contrôle sur notre

environnement ou de résoudre des problèmes qui touchent chacun. Créer un objet qui nous aide au quotidien ou répond à une demande sérieuse (économique, sociale, artistique, industrielle, scientifique...) est tellement plus gratifiant que de changer la configuration de son interface graphique, et ce n'est pas par hasard.

Cette gratification ne nous est plus accessible lorsque nous sommes emportés dans un tourbillon de dépendances informatiques interminables, qu'elles soient psychologiques, logicielles ou liées à la plateforme. L'attrait des idées nouvelles, les possibilités prometteuses, l'environnement informatique en perpétuelle et inexorable évolution, tout cela nous maintient dans un cercle vicieux que la pandémie a révélé.

CONCLUSION

Que dire de plus ? Vous connaissez les symptômes et le syndrome, vous avez donc le choix. L'électronique aura toujours besoin de l'informatique pour avancer et vous seul(e) êtes à même de décider quel niveau de dépendance est acceptable, quelle durée de pérennité est requise, quels moyens sont disponibles à court et à long terme...

L'informatique est et doit rester un outil, pas une fin en soi, ce que nous oublions à mesure que nous l'utilisons.

Les projets professionnels sont relativement épargnés par ce syndrome, car un résultat direct et tangible est attendu, les échéances sont souvent bien définies, les budgets sont alloués et les solutions anticipées. Le piège est pour les projets amateurs, aux contours mal définis, sans cahier des charges ou objectif fixe. Il est si dur de résister à l'appel d'une idée séduisante, je sais.

Rejoignez la résistance, elle n'est pas futile, *faites* quelque chose de concret ! **YG**

RÉFÉRENCES

[gdups] Guidon, Yann *« GDUPS ou la chasse aux fichiers dupliqués » GLMF n°61* pp. 56-65, mai 2004, code actuellement disponible à **https://hackaday.io/project/19176-gdups**

[hk25] Guidon, Yann *« La fabuleuse histoire des calculateurs numériques à l'ère électromécanique » Hackable n°25* pp. 72-80, juillet 2018, **https://connect.ed-diamond.com/Hackable/hk-025/la-fabuleuse-histoire-des-calculateurs-numeriques-a-l-ere-electromecanique**

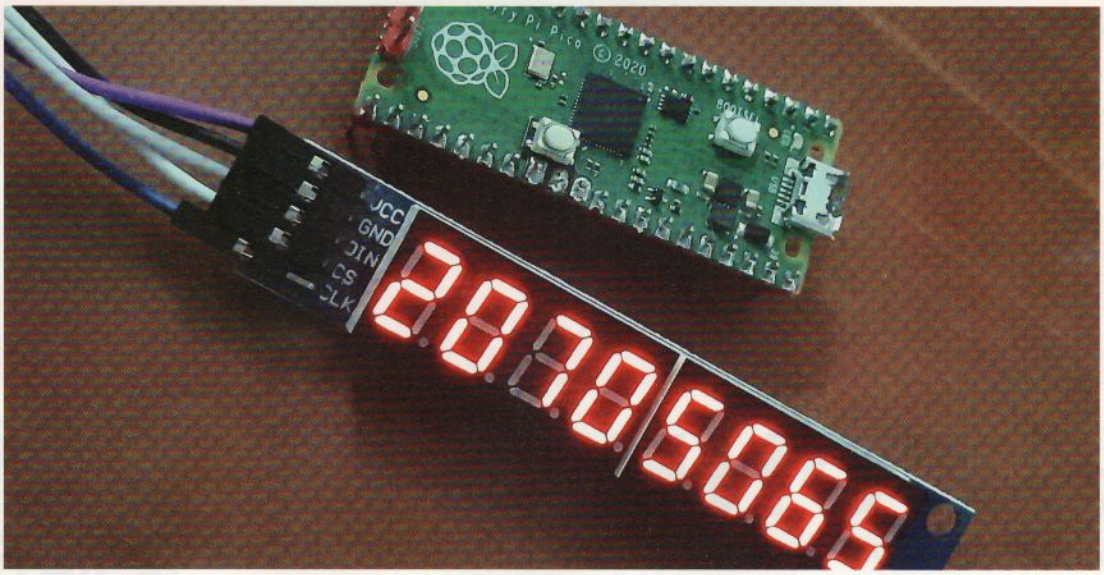
[hk33] Guidon, Yann *« Circuitjs simule des circuits électroniques dans votre navigateur » Hackable n°33* pp. 6-15, avril 2020, **https://connect.ed-diamond.com/Hackable/hk-033/circuitjs-simule-des-circuits-electroniques-dans-votre-navigateur**

[hk36] Guidon, Yann *« Une brève histoire des ASIC libres » Hackable n°36* pp. 42-66, janvier 2021, **https://connect.ed-diamond.com/Hackable/hk-036/une-breve-histoire-des-asic-libres**

CRÉEZ VOTRE AUTHENTICATOR 2FA AVEC UNE CARTE RASPBERRY PI PICO

Denis Bodor

Les vols de comptes, en particulier à cause de l'utilisation de mots de passe trop faibles, sont monnaie courante et la situation ne risque pas de s'améliorer dans le futur. Une solution possible pour se protéger de cette menace consiste à utiliser une authentification multifacteurs. Le nom de compte et le mot de passe seuls ne sont alors plus suffisants pour se connecter, et un code temporaire à usage unique vient renforcer la sécurité. Il existe des matériels et logiciels (sur smartphone) dédiés à cela, mais pourquoi ne pas créer le sien avec une petite touche personnelle ?



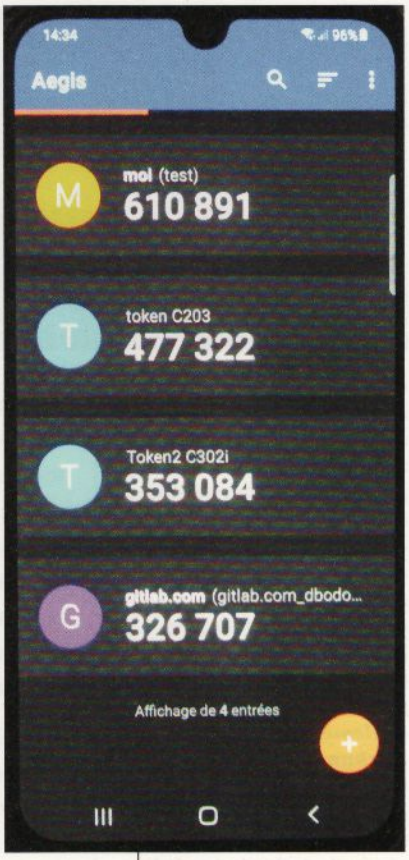
– Créez votre Authenticator 2FA avec une carte Raspberry Pi Pico –

Le principe de l'authentification à deux facteurs, ou *2-Factor Authentication* (abrégé 2FA), est relativement simple en soi. Votre mot de passe est un facteur et on en ajoute un autre. Il faut donc deux informations secrètes pour vous authentifier et non une seule. Ceci peut être un code, envoyé par SMS sur votre mobile, une clé de sécurité type U2F/FIDO2 comme une YubiKey, ou n'importe quel mécanisme permettant de prouver qui vous êtes, en plus d'un simple mot de passe. L'une de ces solutions repose sur un standard appelé TOTP (*Time-based One-Time Password*) lui-même construit sur HOTP (*HMAC-based One-Time Password*), les deux étant respectivement couverts par les RFC-4226 et RFC-6238.

Voilà, d'entrée de jeu, un gros paquet de jargon technique. En pratique, les choses sont beaucoup plus simples : vous disposez d'une application ou d'un petit périphérique qui dispense un code, changeant à intervalle régulier, que vous ajoutez après avoir saisi un identifiant et un mot de passe. Si le code est valide, vous vous connectez, et si ce n'est pas le cas, la connexion sera refusée. Quelqu'un souhaitant « pirater » votre compte, qu'il s'agisse d'un service en ligne, une connexion à un serveur ou quelque chose de local, devra disposer de ce « générateur ». Bien entendu, les codes n'étant valides que durant une plage de temps très réduite, réutiliser un code qu'il aura vu passer ne lui servira à rien.

De nombreux services en lignes permettent de configurer une authentification à deux facteurs et parmi tous les choix possibles, le standard TOTP est celui que l'on trouve le plus fréquemment (après U2F/FIDO2 ou des mécanismes plus spécifiques comme la confirmation via un code mail ou SMS). Un excellent exemple est *Google Authenticator*, une application Android et iOS, bien entendu utilisable avec les services Google (en méthode alternative d'authentification), mais il en existe tout un tas d'autres. Petit problème cependant, cette application n'est plus *open source* depuis quelques années **[1]** et le dépôt GitHub est d'ailleurs archivé. Heureusement, des alternatives existent comme FreeOTP **[2]**, basé sur la dernière version sous licence Apache de *Google Authenticator*, mais aussi et surtout *Aegis Authenticator* **[3]** sous licence GPLv3, les deux étant disponibles en version source, mais également directement installables via les *stores* habituels (Google Play, App Store, F-Droid, etc.).

Une autre solution possible est totalement matérielle, sous la forme d'un petit périphérique de la taille d'une clé USB, comme celui que nous avons exploré dans le numéro 46 **[4]** de chez *Token2*, similaire aux produits *ExcelSecu* que l'on trouve sur AliExpress, par exemple. Le principal avantage de ce type de solutions matérielles est la sécurité accrue puisque, comme nous allons le voir, l'élément secret au cœur de l'algorithme utilisé ne



Aegis Authenticator est une application Android open source alternative au maintenant propriétaire Google Authenticator, permettant de générer et gérer des codes TOTP. Celle-ci est installable via le Play Store et F-Droid, mais également disponible en version source depuis GitHub.

Ceci est un token C203 de chez Token2, une solution TOTP autonome se présentant sous la forme d'un simple porte-clés. Le secret intégré est configuré d'usine et ne peut être modifié. De plus, différentes sécurités qui protègent cette information (anti-tamper) sont en place, mais impliquent également l'impossibilité de changer la pile intégrée lorsque celle-ci finira inéluctablement par arriver au bout de sa vie et ainsi rendre le produit inutilisable.

peut être extrait de ce genre de produit sans le détruire (dispositif *anti-tamper*), contrairement à une application pour smartphone qui offre une surface d'attaque plus importante. En revanche, plusieurs points négatifs accompagnent ce type d'implémentation matérielle, dont la durée de vie limitée du produit dépendant totalement de l'accu intégré, la taille du code à usage unique généralement limité à 6 chiffres ou encore, pour certains modèles d'entrée de gamme, l'absence de possibilité d'ajuster la configuration à ses besoins ou son niveau de paranoïa (secret partagé, algorithme de hachage cryptographique utilisé, etc.).

L'objectif du présent projet n'est pas de simplement reproduire quelque chose qui existe déjà dans le commerce et est, relativement au gain de sécurité, assez économique. Nous allons, au contraire, viser un objectif permettant d'avoir la souplesse de l'application smartphone, tout en ayant l'indépendance et l'autonomie des *tokens* TOTP. **Attention** cependant, ceci est entièrement expérimental dans le sens où cela fonctionne, certes, mais n'est clairement pas plus sécurisé qu'une application Android et très loin des bénéfices d'une solution comme celle proposée par Token2 ou ExcelSecu. Nous utiliserons ici une carte Raspberry Pi Pico ne disposant d'absolument aucune fonction cryptographique matérielle et le secret qui y sera stocké pourra sans trop de difficulté être récupéré, si l'attaquant à un accès physique au montage. Ne prenez pas cela à la légère, dérober le secret sans que l'utilisateur ne le sache est bien pire que de simplement lui voler le périphérique, et donc de rendre la compromission évidente.

C'est peut-être un point que je revisiterai dans l'avenir en utilisant, par exemple, un ESP32-S2 disposant des fonctionnalités nécessaires (voir **[5]**) ou en utilisant un composant dédié, un *secure element*, comme l'ATECC508B de Microchip. Cette dernière option est peu probable étant donné que l'accès à la documentation complète, exactement comme avec d'autres fabricants de composants de ce type, est conditionné par un NDA ou *Non-Disclosure Agreement*, puisque la sécurité par obscurantisme a encore de beaux jours devant elle, apparemment (je vous recommande d'ailleurs le très intéressant, mais vieux, billet à ce sujet sur le blog des créateurs du Trezor Wallet **[6]**).

1. QU'ALLONS-NOUS CONSTRUIRE ?

Notre objectif ici sera de créer un montage autonome (comprendre « non connecté » et non « sur batterie ») fonctionnant comme un croisement entre l'application *Google Authenticator* ou *Aegis Authenticator*, et un *token* TOTP. Celui-ci doit être configurable pour

supporter différents algorithmes de hachage (SHA-1, SHA-256, SHA-512), avoir un secret partagé modifiable ou encore permettre une mise à jour de l'horloge, conserver cette configuration entre les redémarrages et disposer d'une console via une interface série pour la configuration et l'ajustement des paramètres. Tout ceci sera basé sur une carte Raspberry Pi Pico équipée d'un convertisseur USB/série (je n'ai pas envie de m'amuser avec l'interface USB/CDC soulevant un certain nombre de problématiques), d'un module i²c RTC DS3231 équipé d'une pile bouton et d'un afficheur à LED de 8 fois 7 segments contrôlé par un MAX7219 interfacé en SPI.

Le but n'est pas d'avoir quelque chose de portable, bien au contraire. L'idée, une fois la preuve de concept validée, est de changer d'échelle et d'avoir un dispositif mural avec d'énormes afficheurs LED 7 segments (idéalement 10 chiffres) présentant de manière continue le code à usage unique. Un simple coup d'œil à cet afficheur permettra, à terme, de connaître et saisir le code en question lors d'une connexion à un service ou un autre configuré dans ce sens. Il est parfaitement clair que quiconque voyant cet afficheur sera en mesure de fournir le code en complément du mot de passe. Le dispositif n'a pas pour objet d'être visible en dehors d'un environnement privé ou professionnel avec un accès restreint.

Ce projet, qui est par définition abouti au moment où je rédige cet article, couvre un grand nombre de problématiques intéressantes, tant au niveau de la programmation du RP2040 de la Pico que de l'utilisation de fonctions cryptographiques, de la gestion des dates et heures, des accès concurrents aux variables ou encore des petites subtilités concernant la manipulation de la mémoire flash.

Le code formant la base du *firmware* est à la fois inspiré d'un autre de mes projets, *TinyTOTP* **[7]**, faisant peu ou prou la même chose (et plus encore), mais sur PC en reposant sur OpenSSL et de différents éléments externes sous licence MIT, dont un code de Francesco Pantano **[8]** retravaillé de manière conséquente et un support pour RTC DS3231 développé par Bryan Nielsen **[9]**. Le tout sera probablement disponible sur mon GitLab **[10]** au moment où vous lirez ceci. À toutes fins utiles, précisons que la partie stockage de configuration se base sur l'un de mes articles précédent, paru dans le numéro 42, et détaillant comment utiliser une partie de la flash d'une Pico pour émuler une zone EEPROM comme celle dont dispose certains microcontrôleurs Atmel AVR des Arduino (« Arduini », du coup ?).

2. TOTP, HOTP, HMAC ET SHA

Avant d'entrer dans le détail de TOTP et donc également de HOTP, il est important de s'arrêter un instant sur la notion de code d'authentification de message (MAC en anglais) et celle de fonction de hachage.

2.1 Hachage

Commençons donc par cette dernière puisqu'elle forme la base de tout le système. Une fonction de hachage prend en entrée un volume variable de données de grande taille et retourne une donnée de taille fixe et aux caractéristiques précises (comme un ensemble de caractères donnés, par exemple). Un excellent exemple d'utilisation ce type de fonctions est la commande Unix `md5sum` :

```
$ echo -n coucou > test

$ md5sum test
721a9b52bfceacc503c056e3b9b93cfa  test

$ dd if=/dev/urandom of=test2 bs=1M count=1
1+0 enregistrements lus
1+0 enregistrements écrits
1048576 bytes (1,0 MB, 1,0 MiB) copied, 0,00915319 s, 115 MB/s

$ md5sum test2
0884008191a7265668bb97192a2caf2c  test2
```

Nous utilisons deux fois la commande, tout d'abord sur un fichier texte contenant le mot « coucou » puis sur un fichier contenant des données aléatoires issues de `/dev/urandom`. Le premier fichier fait 6 octets et le second 1 Mio, mais le résultat est une chaîne de caractères représentant une valeur hexadécimale sur 32 positions, soit 16 octets dans les deux cas. Les deux valeurs de hachage ont une même taille qui est fixe et répondent à des caractéristiques identiques, elles sont composées uniquement de chiffres de « 0 » à « 9 » et de lettres de « a » à « f ». `md5sum` utilise la fonction de hachage MD5 (pour *Message Digest 5*).

Mais plus important encore, les deux valeurs de hachage sont différentes et si nous modifions le contenu du fichier `test`, même très légèrement en y ajoutant simplement « x », la valeur de hachage change totalement :

```
$ echo -n "x" >> test

$ md5sum test
3c0dc4e5ff01824cc39f85a4bdb64de5  test
```

Autre point important, l'opération inverse consistant à retrouver les données à partir de la valeur de hachage n'est pas possible. Pas plus que de trouver ou composer un second jeu de données qui produira la même valeur de hachage qu'un autre (notion de collision). On parle alors de fonctions de hachage cryptographique et celles-ci sont massivement utilisées pour vérifier l'intégrité des données, comme base pour la signature électronique ou encore pour les codes d'authentification.

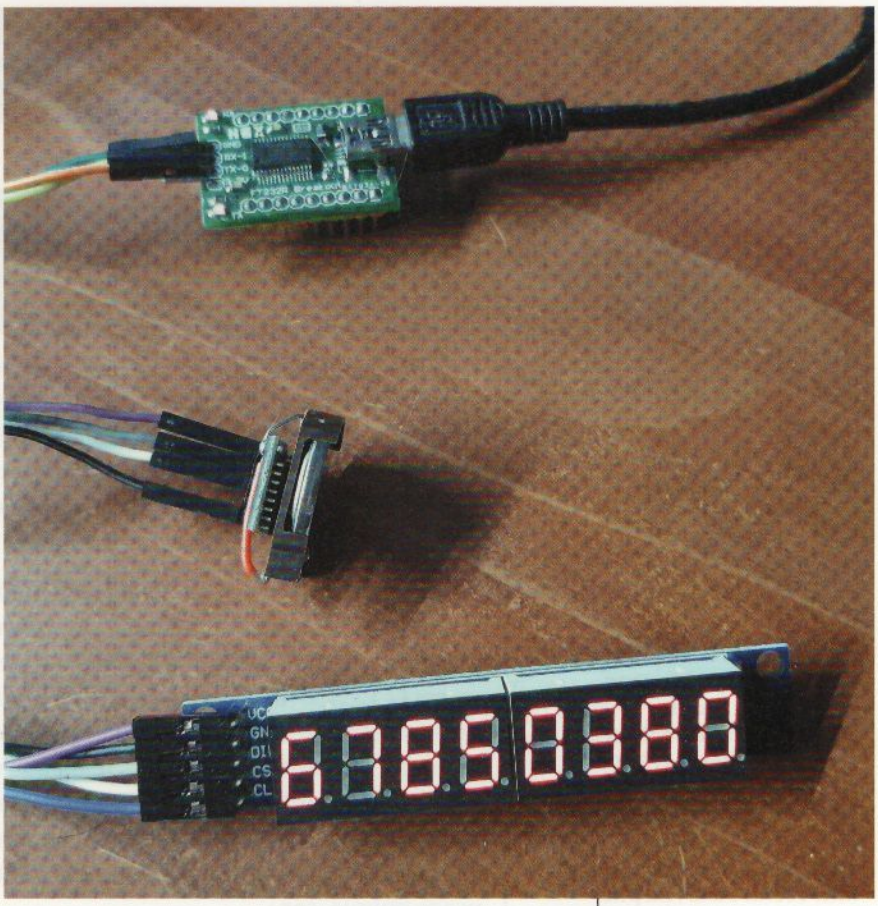
Notez que MD5 est aujourd'hui obsolète et depuis 2011 (RFC6151) n'est plus considéré comme une fonction de hachage cryptographique puisqu'il n'est pas résistant aux collisions. Le plus démonstratif exemple est celui de Marc Stevens, mettant à disposition **[11]** deux fichiers de 64 octets, possédant deux octets de différence et donnant exactement la même sortie avec `md5sum`. MD5 est depuis remplacé, plus ou moins partout, par des fonctions de hachage conçues par la NSA appelée SHA pour *Secure Hash Algorithm* regroupant SHA-1, la famille SHA-2 (SHA-224, SHA-256, SHA-384 et SHA-512) et plus récemment la famille SHA-3 (SHA3-224, SHA3-256, SHA3-384 et SHA3-512). Des commandes sont également disponibles sur de nombreux systèmes pour remplacer `md5sum` : `sha1sum`, `sha224sum`, `sha256sum`, `sha384sum`, `sha512sum`...

2.2 HMAC

Une valeur de hachage permet de s'assurer, par exemple, de l'intégrité des données, mais ne donnera absolument aucune indication quant à son authenticité. En d'autres termes, une valeur de hachage confirmera que les données sont les bonnes, mais pas qu'elles proviennent effectivement d'une origine spécifique. Pour régler ce problème, et valider les deux informations, il est possible de combiner une fonction de hachage cryptographique avec un secret prenant la forme d'une clé.

On parle alors de HMAC, pour *Hash-based Message Authentication Code* ou code d'authentification basé sur un hachage. Différents algorithmes existent permettant de combiner une clé, typiquement une série d'octets, avec un message via une fonction de hachage et, bien entendu, un certain nombre d'entre eux sont standardisés et reposent sur des fonctions de hachage cryptographique qui le sont tout autant : HMAC-MD5 (obsolète donc), HMAC-SHA-1, HMAC-SHA-256 ou encore HMAC-SHA-512.

Un HMAC-SHA-1 ressemblera à une valeur de hachage SHA-1, soit une série de 20 octets représentés en notation hexadécimale sur 40 positions, mais il ne s'agit pas simplement d'une donnée permettant de vérifier les données associées. En utilisant le secret partagé entre les deux intervenants, produire à un bout comme à l'autre un HMAC-SHA-1 avec les mêmes données et la même clé donnera le même résultat. Si les données changent ou si la clé n'est pas la bonne, le résultat sera différent et on en déduira que soit les données sont corrompues, soit que l'un des deux intervenants n'est pas celui qu'il prétend être puisqu'il ne dispose pas de la bonne clé.



Voici les trois éléments dont nous avons besoin pour créer notre Authenticator TOTP : un afficheur, un module RTC et un convertisseur USB/série pour la configuration.

Comme pour `md5sum` ou plus exactement `sha1sum`, nous pouvons rapidement faire la démonstration de l'utilisation d'un HMAC-SHA-1 par exemple, en ligne de commande grâce à OpenSSL :

```
$ echo -n "coucou" | openssl dgst -sha1 -hmac "12345678"
(stdin)= 2b8b41981d6f804b57df88faaecb94c6f2c07a93
```

Ou, pour la version hexa :

```
$ echo -n "coucou" | openssl dgst -sha1 \
  -mac hmac -macopt hexkey:3132333435363738
(stdin)= 2b8b41981d6f804b57df88faaecb94c6f2c07a93
```

Notre donnée de base, ou message, est toujours « coucou », mais nous spécifions dans les options d'`openssl` que nous souhaitons produire un HMAC en utilisant la fonction de hachage SHA-1 et en utilisant le secret « 12345678 ». Le destinataire de notre message pourra alors faire de même en connaissant le secret partagé et s'assurer que c'est bien moi qui ai envoyé le message, et que celui-ci est bien « coucou ».

Imaginons un instant ce qu'il est possible de faire avec ce genre de choses dans un contexte d'authentification multifacteur. Partons du principe que moi, comme le serveur ou service partageons le fameux secret (« 12345678 ») et qu'à chaque tentative de connexion un tel message authentifié doit être présenté. Bien sûr, si nous ne changeons pas le message, le HMAC sera toujours le même, ça n'a pas d'intérêt. Mais que se passe-t-il si, lui comme moi, maintenons à jour un compteur dont la valeur remplace « coucou » et qui s'incrémente à chaque authentification réussie ? Nous obtenons une solution d'authentification à mot de passe à usage unique, car :

- un attaquant espionnant la communication ne peut déduire ni le secret ni la valeur du compteur à partir du HMAC ;
- le message vient forcément de moi, sinon le HMAC sera différent, car la clé invalide ;
- la validation, et donc l'authentification seront uniques, car le compteur augmente toujours et le message change à chaque fois.

Vous avez donc là le début de ce que permettent des choses comme *Aegis Authenticator* ou un produit comme ceux de Token2. Mais juste le début...

Ce module RTC DS3231 est initialement prévu pour équiper une Raspberry Pi en s'enfichant directement sur les broches 1, 3, 5 et 9 (7 étant non utilisé), mais pourra facilement être adapté à n'importe quel montage à base de microcontrôleur. La pile soudée est un problème, car une fois épuisée, il sera nécessaire de bricoler un peu pour réanimer le module.

2.3 HOTP

Le mécanisme hypothétique que je viens d'expliquer pourrait parfaitement fonctionner à condition de saisir votre *login*, votre mot de passe et quelque chose comme `3bc0ca897a41cd6952f6b0c3489e640252955b99`, sans

faire la moindre faute de frappe. Ce n'est pas très réaliste. Heureusement pour nous, quelqu'un de plus intelligent a également eu cette idée qui est, depuis, devenue un standard défini par la RFC-4226 **[12]**.

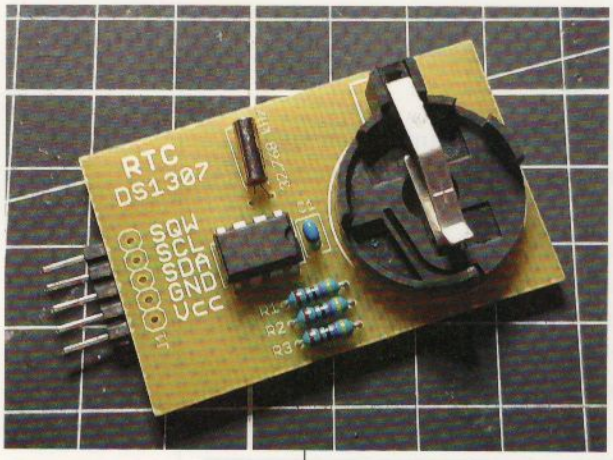
HOTP, pour *HMAC-based One-Time Password* est l'algorithme permettant de non seulement produire un HMAC (SHA-1 dans la RFC) en combinant le secret partagé avec la valeur du compteur (en *big-endian*), mais en en découlant ensuite un code de 4 à 10 chiffres (6 par défaut et 8 recommandé) en notation décimale, facile à mémoriser et à saisir par un humain. Pour obtenir ce code, on commence par extraire les 4 derniers bits (bits de poids faible sur les 160 pour SHA-1) du HMAC. Ceux-ci sont ensuite utilisés comme index pour sélectionner 31 bits du HMAC (31 pour éviter que le bit de poids le plus fort ne serve comme bit de signe et que la valeur ne puisse être négative). Cette première opération ressemble à ceci une fois implémentée en C :

```
uint32_t DT(uint8_t *digest, int tthmac)
{
    uint64_t offset;
    uint32_t bin_code;

    offset   = digest[tthmac-1] & 0x0f;

    bin_code = (digest[offset]  & 0x7f) << 24
        | (digest[offset+1] & 0xff) << 16
        | (digest[offset+2] & 0xff) <<  8
        | (digest[offset+3] & 0xff);

    return bin_code;
}
```

Avec `digest` un pointeur vers le HMAC sous la forme d'un tableau de `uint8_t` et `tthmac` la taille du HMAC en octets (20 pour SHA-1, 32 pour SHA-256 et 64 pour SHA-512). Mais ce n'est pas tout, ce `bin_code` de 31 bits (ou 32 mais forcément positif), est ensuite utilisé pour créer le fameux code qui est le reste de la division de `bin_code` par 10 puissance le nombre de chiffres à obtenir (opération modulo). Là encore en C, ceci deviendrait :

```
uint32_t mod_hotp(uint32_t bin_code, int digits)
{
    int power = pow(10, digits);
    uint32_t otp = bin_code % power;

    return otp;
}
```

En résumé, à ce stade nous avons un HMAC du compteur et du secret, une troncature pour obtenir 31 bits et un modulo pour avoir un code de la taille demandée (entre 4 et 10). Si vous avez compris ces étapes, vous avez compris la majeure partie de ce que raconte la RFC-4226 !



La RTC DS1307 est un classique Arduino, mais celle-ci ne fonctionnera pas avec une carte Raspberry Pi Pico. En effet, la puce DS1307 ne peut être alimentée avec une tension inférieure à 4,5 V et la Pico n'est pas tolérante aux 5 V.

2.4 TOTP

Et si là, vous vous dites « mais je n'ai pas besoin d'avoir un compteur, puisque le nombre de secondes écoulées depuis une certaine date en est un qui fonctionne tout seul », bravo, vous venez d'inventer TOTP ou *Time-based One-Time Password* et pouvez vous attacher à la rédaction d'une RFC en tout point identique à la RFC-6238.

Cette dernière, bien plus courte que la RFC-4226 (8 pages au lieu de 26), décrit tout simplement comment utiliser HOTP, mais en remplaçant le compteur par le nombre de secondes écoulées depuis le 1er janvier 1970 à minuit (également appelée heure Unix ou heure Posix) sur 64 bits (sinon il y aura un problème le 19 janvier 2038 à 03:14:08 UTC. Pour vous faire peur, c'est par ici **[13]**), mais aussi comment éviter d'avoir un nouveau code à chaque seconde, en introduisant la notion de *time-step*. En arrondissant l'heure Unix à un intervalle déterminé (30 secondes comme recommandé par la RFC), ceci laisse le temps à l'utilisateur de saisir le code avant qu'il ne change et, dans le même temps, assure une sécurité suffisante pour éviter que quelqu'un réutilise ledit code passé la période en question.

Un autre point qu'il faut adresser lorsqu'on utilise un repère temporel est la désynchronisation des horloges entre plusieurs systèmes. Arrondir à un multiple de 30 secondes ne règle pas le problème et même si les deux systèmes ne sont désynchronisés que de quelques secondes, un code valable pour l'un ne l'est pas forcément pour l'autre. Pour régler cela, la RFC indique qu'il suffit de considérer comme valide un certain nombre de codes après ou avant le code courant et ainsi offrir un peu plus de souplesse. Mieux encore, il devient possible, après validation d'un des codes, de déduire la dérive d'horloge et donc d'utiliser cette information pour avertir l'utilisateur, ajuster automatiquement l'horloge ou déclencher une synchronisation par un autre biais.

Enfin, alors que la RFC-4226 ne parle que de HMAC-SHA-1, la RFC-6238 précise clairement qu'il est possible (*MAY*) d'utiliser HMAC-SHA-256 ou HMAC-SHA-512, et ce, avec le même mécanisme de troncature et de modulo que pour HMAC-SHA-1.

3. PASSONS À LA PRATIQUE

La théorie c'est bien amusant, surtout avec des petits bouts de codes, mais le but du jeu est d'avoir quelque chose de fonctionnel et pratique (peut-être même de réellement utilisable, qui sait). L'objectif est simple, nous allons créer, sur la base d'une carte Raspberry Pi Pico, un montage qui affiche un code TOTP. Ceci implique plusieurs choses, à commencer par le matériel nécessaire :

- Une carte Raspberry Pi Pico parfaitement standard.
- Un module RTC pouvant fonctionner en 3,3 V et s'interfaçant très classiquement en i²c. Celui utilisé ici repose sur un Maxim DS3231 et est initialement prévu pour une Raspberry Pi (grand modèle). Ce genre de module se trouve pour une paire d'euros sur les sites habituels et intègre généralement une pile soudée qui, en fin de vie, demande alors un peu de bricolage pour réanimer le module (il existe des versions avec des piles amovibles, mais c'est plus cher).
- Une solution pour afficher entre 4 et 10 chiffres entre 0 et 9. Typiquement, comme notre projet est censé être fixe, inutile de s'embêter avec quelque chose d'économe en énergie et l'option « LED » vient alors immédiatement à l'esprit. Cherchez simplement quelque chose comme « *7 segment display 8 digit* » sur un site tel qu'AliExpress et vous serez submergé d'offres proposant des modules tout faits avec 8 chiffres, pilotés en SPI par un MAX7219 pour moins d'un euro. Vu le prix, prévoyez de suite de jouer du fer à (des)souder pour aligner correctement les deux afficheurs 7 segments à 4 chiffres l'un par rapport à l'autre (c'est à peine visible éteint, mais sous tension, on ne voit que ça). Notez qu'il existe aussi des versions utilisant un registre à décalage 74HC595, étrangement plus cher, mais disponible en d'autres couleurs que rouge. Ceci n'est pas pris en charge par mon code pour le moment.
- Un module USB/série permettant de faire dialoguer le montage avec un PC pour obtenir des informations et surtout configurer les différents paramètres comme la date/heure, le secret, le nombre de chiffres du code, le type de HMAC, etc. On pourrait se passer temporairement ou définitivement de ce composant en utilisant l'USB OTG du RP2040 ou en n'ayant recours à l'interface série que ponctuellement (pour le pénible changement d'heure saisonnier, par exemple). Je préfère cependant la version UART pure et dure, en particulier avec un projet aussi complexe impliquant l'utilisation des deux cœurs ARM du RP2040.

Deux cœurs ? Oui, c'est ici l'occasion d'expérimenter quelque chose de trop souvent laissé de côté. Le microcontrôleur RP2040 dispose d'un processeur ARM avec deux cœurs Cortex-M0+ à 133 MHz et 264 Kio de SRAM intégrés et le fait d'avoir, en parallèle, une fonction périodique pour la génération du code et une gestion de la communication série est une opportunité parfaite de diviser le travail et d'exploiter au mieux le matériel.

L'architecture générale du code est assez simple, nous allons classiquement avoir une boucle principale exécutée juste après la phase de configuration/initialisation des composants. Celle-ci se chargera d'agir en fonction de l'état d'une paire de variables et, en particulier, d'un « drapeau » déclenchant l'accès à la RTC et la génération du code TOTP. Ce drapeau sera levé, à intervalle régulier, via un *timer* appelant une fonction *callback*. Tout ceci se passera sur le premier cœur, alors que sur le second tournera une autre boucle infinie, chargée de scruter les caractères provenant du port série et de traiter les demandes de l'utilisateur. Personnellement, je suis encore et toujours émerveillé (et enchanté) qu'on puisse envisager ce type de structure avec une carte fabriquée en Europe continentale, vendue seulement 5 € et disposant d'un SDK léger, souple et compréhensible, contrairement à d'autres (*tousse* STM32CubeMX *tousse*).

3.1 HMAC, SHA et HOTP

PicoTOTP, c'est le nom que j'ai donné à ce projet **[14]**, n'est pas ma première expérience dans l'univers de l'authentification deux facteurs, et TOTP en particulier. Il s'agit, en effet, d'un descendant d'un autre projet nommé TinyTOTP. TinyTOTP **[7]** est un outil fonctionnant sur plateforme GNU/Linux ou [Free|Open|Net]BSD. Il se base initialement un code de Francesco Pantano (« c_otp » **[8]**) et se veut être un outil en ligne de commande permettant de générer et valider des codes d'authentification TOTP. TinyTOTP repose initialement sur OpenSSL pour ce qui relève des fonctions cryptographiques (HMAC) puisqu'il s'agit de quelque chose d'omniprésent sur les systèmes Unix *open source*. Cependant, l'idée germant de porter ce code vers un microcontrôleur 32 bits, j'ai créé une branche spécifique du projet (« nossl ») se détachant de cette dépendance à OpenSSL et celle-ci peut être vue comme une version préliminaire à ce dont il est question ici.

La principale difficulté qu'on rencontre lorsqu'on n'a plus accès aux facilités offertes par OpenSSL est de trouver des implémentations fiables des algorithmes dont on a besoin. Car, voyez-vous, il est excessivement risqué de s'improviser spécialiste en sécurité et de jouer les apprentis sorciers en voulant implémenter, à partir de rien, ces algorithmes. Fort heureusement, un certain nombre d'implémentations éprouvées existent et utilisent des licences permettant de les intégrer sans problème dans un nouveau projet. C'est le cas, par exemple, de l'implémentation de Simon Tatham pour PuTTY, qu'on retrouve un peu partout dans de très nombreux dépôts GitHub/GitLab, et c'est bien entendu, celle que j'ai choisi d'utiliser, puisque ne reposant que sur les fonctions classiques de la libc.

Le code de Francesco Pantano portant sur la génération HOTP/TOTP a donc été adapté, trituré et révisé pour devenir tout d'abord TinyTOTP, puis TinyTOTP branche « nossl » et enfin un élément important de PicoTOTP. Et tout cela commence par la fonction chargée de générer les HMAC :

```
uint8_t
*hmac(unsigned char *key, int klen, uint64_t interval, int tthmac)
{
  uint8_t *ret = NULL;
  switch (tthmac) {
  case SHA256_DIGEST_SIZE:
     if (!(ret = (uint8_t *)malloc(SHA256_DIGEST_SIZE))) {
         exit(1);
     }
     hmac_sha256(key, klen, (unsigned char *)&interval, sizeof(interval), ret);
     return(ret);
  case SHA512_DIGEST_SIZE:
     if (!(ret = (uint8_t *)malloc(SHA512_DIGEST_SIZE))) {
         exit(1);
     }
     hmac_sha512(key, klen, (unsigned char *)&interval, sizeof(interval), ret);
     return(ret);
```

```
  default:
     if (!(ret = (uint8_t *)malloc(SHA1_DIGEST_SIZE))) {
         exit(1);
     }
     hmac_sha1(key, klen, (unsigned char *)&interval, sizeof(interval), ret);
     return(ret);
  }
}
```

Les fonctions `hmac_sha256()`, `hmac_sha512()` et `hmac_sha1()` proviennent toutes du code de PuTTY, intégré au projet sous la forme de fichiers `sha256.c`, `sha256.h`, `sha512.c`, `sha512.h`, `sha.c` et `sha.h`. Aucune modification nécessaire, c'est du C bien propre comme on l'aime. Remontons alors d'un cran avec la fonction générant effectivement le code HOTP :

```
uint32_t
HOTP(uint8_t *key, size_t klen, uint64_t interval, int digits, int tthmac)
{
    uint8_t *digest;
    uint32_t result;
    uint32_t dbc;

    // conversion en big endian
    interval = ((interval & 0x00000000ffffffff) << 32) |
        ((interval & 0xffffffff00000000) >> 32);
    interval = ((interval & 0x0000ffff0000ffff) << 16) |
        ((interval & 0xffff0000ffff0000) >> 16);
    interval = ((interval & 0x00ff00ff00ff00ff) <<  8) |
        ((interval & 0xff00ff00ff00ff00) >>  8);

    // calcul HMAC
    digest = (uint8_t *)hmac(key, klen, interval, tthmac);

    // troncature
    dbc = DT(digest, tthmac);

    // modulo
    result = dbc % (int)pow(10, digits);

    if(digest)
        free(digest);
    return result;
}
```

`DT()` a été vu précédemment, et le modulo est directement intégré (plutôt qu'un appel inutile à une fonction). Le `uint32_t` est le code HOTP résultant de l'opération, avec `key` un pointeur vers les `uint8_t` du secret, `klen` sa taille, `interval` le compteur, `digits` le nombre de chiffres en sortie et `tthmac` le type de HMAC utilisé, avec :

```
#define   SHA1_DIGEST_SIZE  20
#define SHA256_DIGEST_SIZE  32
#define SHA512_DIGEST_SIZE  64
```

Une seconde ! HOTP ?! Oui, rappelez-vous, TOTP n'est rien d'autre que HOTP, mais avec `interval` correspondant au nombre de secondes écoulées depuis le 01/01/1970 à minuit. La fonction nous permettant de faire du TOTP est donc la même, mais Francesco a préféré créer une fonction dédiée pour éviter la confusion (oui, on pourrait aussi utiliser l'attribut `alias`, mais je trouve que cela rendrait le code moins lisible) :

```
uint32_t
TOTP(uint8_t *key, size_t klen, uint64_t time, int digits, int tthmac)
{
    uint32_t totp;

    totp = HOTP(key, klen, time, digits, tthmac);
    return totp;
}
```

Vous remarquerez que, partout, nous utilisons le nombre de chiffres du code à obtenir en argument alors que, matériellement, nous en avons une quantité fixe (8). Ceci est fait dans le but de rendre (ou laisser) le code générique et adaptable, en plus de permettre, éventuellement, l'utilisation d'une partie seulement des afficheurs 7 segments.

Tout ceci est placé dans des fichiers sources trouvant place dans un sous-répertoire `libs/` du projet et, dans notre `main.c`, nous finalisons le tout en créant la *vraie* fonction de génération TOTP :

```
uint32_t
totp(uint8_t *k, size_t keylen, int digits, int tthmac)
{
    int status;
    struct tm datetime;

    if ((status = readDS3231Time(&datetime))) {
        printf("Error getting RTC time, %s\n",
           ds3231ErrorString(status));
    }

    time_t t = floor(((unsigned long long)mktime(&datetime)) / VALIDITY);

    return TOTP(k, keylen, t, digits, tthmac);
}
```

C'est ici que nous utilisons la date/heure de l'heure Unix tirée de la RTC, arrondie au *time-step* précisé par la macro `VALIDITY` (qui vaut `30`). Et voilà qui nous amène donc directement à la partie concernant l'utilisation du DS3231.

3.2 Gestion de la RTC

La partie RTC utilise le code de Bryan Nielsen sans trop de modifications. Il est d'ailleurs assez étonnant de constater la rareté de ce genre de support alors même que, du côté d'Arduino, il existe une profusion de bibliothèques permettant de gérer DS1307, DS1338, DS3231, etc., et ce, de manière très complète (avec alarme et NVRAM). L'intégration dans le projet se fait sans aucune difficulté en copiant simplement `ds3231.c` et `ds3231.h` dans notre `libs/` et en incluant le *header file* dans notre `main.c`. Ceci nous donne accès aux fonctions d'initialisation, de lecture et d'écriture dans les registres du DS3231.

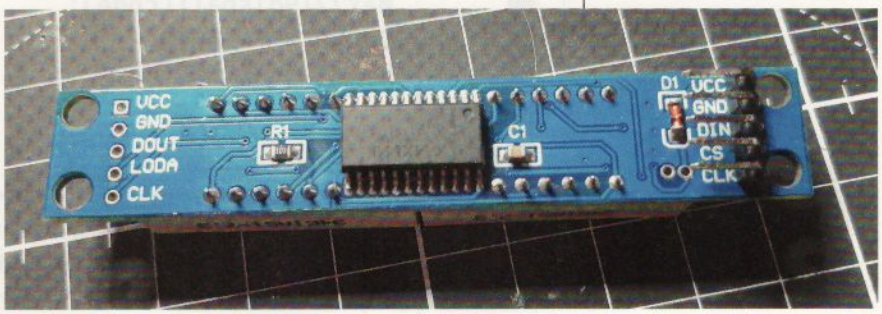
Quelques lignes sont alors suffisantes pour prendre en charge le composant :

```
if ((status = initDS3231())) {
    printf("Error occurred during DS3231 initialization. %s\n",
        ds3231ErrorString(status));
}
printf("DS3231 initialized.\n");
```

Lire la date et l'heure dans le DS3231 est tout aussi facile, comme le montre la fonction `totp()` présentée ci-devant. Mais nous allons rencontrer un problème d'accès concurrent au matériel. En effet, nous allons mettre en place un *timer* avec `add_repeating_timer_ms()` et en argument, une période de répétition et une fonction *callback* ressemblant à :

```
bool
gentotp_callback(struct repeating_timer *t)
{
    gentotp = true;
    return true;
}
```

La variable globale `gentotp`, déclarée avec `volatile bool gentotp = false;` sera ensuite utilisée dans la boucle principale pour savoir s'il faut rafraîchir l'afficheur et donc faire un nouvel appel à `totp()`. Mais, en parallèle, nous aurons sur le second cœur une interface utilisateur permettant entre autres de configurer la date/heure dans le DS3231. Faire le pari qu'un accès simultané au composant par les deux morceaux de code est peu probable serait une erreur grossière. Nous avons là un cas typique d'accès concurrent à une ressource et il existe une solution tout aussi typique : un mutex.



Ce type de module d'affichage se prête parfaitement à notre réalisation avec ses 8 chiffres qui sont, précisément, le nombre recommandé dans les RFC.

Il existe de multiples façons de piloter des afficheurs LED 7 segments et l'une d'elles consiste à mettre en œuvre un MAX7219 interfacé en SPI. Ce genre de modules se trouvent pour environ 1 €/pièce sur tous les sites habituels (eBay, AliExpress, BangGood, Amazon, etc.).

Un mutex, pour *MUTual EXclusion*, est une primitive de synchronisation qu'on peut facilement illustrer dans la vie réelle. Imaginez une ressource comme des toilettes dans une station-service et une clé pour y accéder. Si vous voulez utiliser les toilettes, vous demandez la clé et si elle n'est pas déjà empruntée par quelqu'un, vous la recevez, empêchant quiconque d'autre d'utiliser les lieux en même temps que vous. Lorsque vous avez fini vos petites affaires, vous rendez la clé (après vous être lavé les mains) et les toilettes sont à nouveau disponibles pour quelqu'un d'autre.

Dans notre code, les toilettes représentent l'accès à la RTC et le SDK Pico met à notre disposition quelques fonctions intéressantes. Dans un premier temps, nous avons besoin du mutex lui-même :

```
mutex_t mutex;
```

Qui sera initialisé dans `main()` avec :

```
mutex_init(&mutex);
```

Dès lors, notre boucle principale dans `main()` ressemblera à ceci :

```
uint32_t owner_out;

[...]

while(1) {
  if (gentotp == true) {
    if (mutex_try_enter(&mutex, &owner_out)) {
      code = totp(config.key, config.keylen,
         config.digits, config.hmac);
      max7219dispall(code);
      gentotp = false;
      mutex_exit(&mutex);
    }
  }
}
```

`mutex_try_enter()` retourne *VRAI* si la tentative d'avoir la propriété du mutex réussit. De ce fait, l'exécution du corps de l'`if` implique que nous nous sommes approprié le mutex et pouvons utiliser `totp()` et donc accéder à la

RTC, afficher le code, mettre à jour `gentotp` et rendre le mutex. Si ce n'est pas le cas, parce qu'un autre morceau de code possède la propriété du mutex, ce n'est pas grave, nous ferons autant de tours dans la boucle que nécessaire jusqu'à ce que le mutex soit disponible.

Par ailleurs, dans la gestion de l'interface série utilisateur que nous verrons plus loin, nous retrouvons le même genre d'approche :

```
mutex_enter_blocking(&mutex);
if ((status = setDS3231Time(gmtime(&tmpt)))) {
    printf("Error setting time, %s\n",
        ds3231ErrorString(status));
}
mutex_exit(&mutex);
```

Ici, l'appel est bloquant et l'accès à la RTC, en écriture cette fois, ne pourra se faire qu'une fois le mutex libéré. Là encore, ce n'est pas un problème, si la RTC est accédée par `main()`, nous attendons simplement que l'affichage soit fait pour enregistré la date/heure.

3.3 Gestion de l'affichage

Vous avez sans doute remarqué l'appel à `max7219dispall(code)` dans la boucle principale et cette fonction provient directement d'un exemple livré avec le SDK Raspberry Pi Pico (`pico-examples/spi/max7219_8x7seg_spi/max7219_8x7seg_spi.c`). Le fait que cet exemple corresponde précisément au circuit utilisé sur le module en ma possession (qui est un reste du projet lié à Home Assistant dans un précédent numéro **[15]**) est un pur hasard mais une véritable aubaine. Le support du MAX7219 est sommaire, mais amplement suffisant ici avec une initialisation sensiblement modifiée et prenant en compte un paramètre réglant l'intensité des LED (entre 0 et 15) :

```
    max7219init(0);
    max7219clear();
```

La fonction `max7219dispall()` précédemment utilisée est relativement simple et se contente d'écrire les registres correspondant aux 8 chiffres dans une boucle :

```
void max7219dispall(int32_t num)
{
  int digit = 0;
  while (digit < 8) {
    max7219writereg_all(CMD_DIGIT0 + digit,
       num % 10);
    num /= 10;
    digit++;
  }
}
```

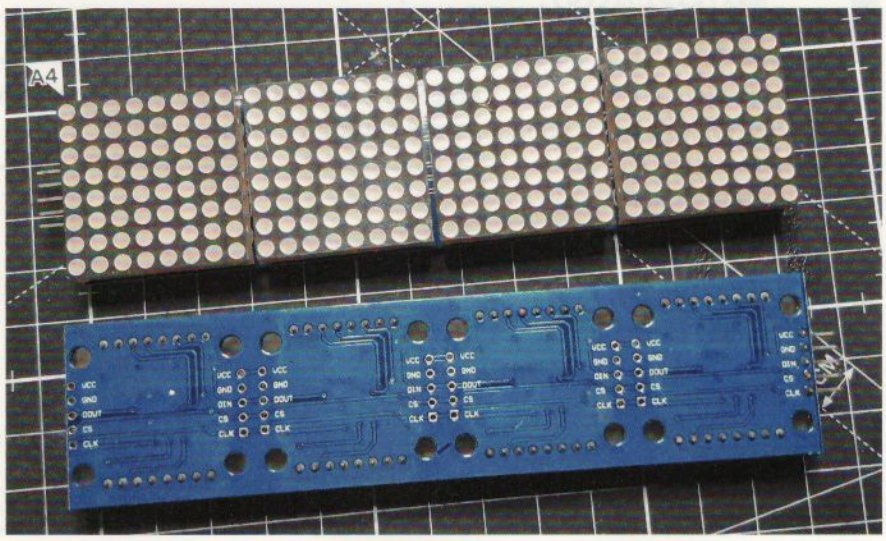
À ce stade du projet, c'est suffisant, mais ce code fera l'objet d'une réécriture pour supporter davantage de fonctionnalités, voire de modules d'affichage. L'une des pistes que je compte suivre est, par exemple, le fait d'utiliser les points des afficheurs 7 segment comme indicateur du temps de validité du code, à l'instar de l'application *Aegis Authenticator* ou des produits Token2.

3.4 Utilisation de la flash pour la configuration

Il est temps à présent de se pencher sur la partie que j'ai trouvée la plus intéressante à développer et qui demandera encore quelques affinages futurs : la configuration. Comme spécifié précédemment, l'idée n'est pas d'avoir un montage statique avec des paramètres ou un secret stockés « en dur » dans le code, mais de permettre une (re)configuration via l'interface série. Bien entendu, ceci ne présente que peu d'intérêt si la configuration ne survit pas à une coupure d'alimentation. Une option envisageable aurait été de stocker ces informations dans la mémoire non volatile (NVRAM) de la RTC, mais la DS3231 n'en dispose pas. Seule la DS1307 semble posséder un espace mémoire maintenu par la pile bouton, mais ce composant ne fonctionne qu'avec un VCC de 4,5V **minimum** et 56 octets ne sont, de plus, pas suffisants pour nos besoins.

Nous voulons stocker dans la configuration le type de HMAC, le nombre de chiffres du code TOTP à obtenir, mais aussi et surtout le secret et sa taille en octets. Si nous choisissons arbitrairement une taille maximum du secret à 64 octets (512 bits), nous avons besoin de quelque chose comme 80 octets en tout devant survivre à une coupure ou un *reset* (davantage en augmentant la taille maximum du secret).

Le RP2040 ne disposant pas d'une zone EEPROM comme les AVR des Arduino,



On retrouve également le MAX7219 dans une configuration permettant de piloter des matrices de LED 8×8. Là, nous avons un MAX7219 par matrice et pouvons afficher autre chose que des chiffres et quelques lettres vaguement lisibles. Le contrecoup est de devoir gérer des polices de caractères et le rendu dans les matrices, ce qui n'est pas forcément souhaitable ou amusant ici.

nous devons stocker cela en flash. C'est là quelque chose que nous avons déjà abordé en explorant la plateforme Raspberry Pi Pico dans le numéro 42 **[16]**. L'astuce consiste à modifier l'organisation mémoire utilisée par la chaîne de compilation de manière à se ménager un petit espace, en flash, qui n'est pas utilisable pour le programme. Ceci se fait en utilisant un script personnalisé pour l'éditeur de liens et un argument spécifique dans notre `CMakeLists.txt` :

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

Le fichier `memmap_custom.ld`, placé dans le répertoire du projet, est copié du SDK et modifié ainsi :

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

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

ENTRY(_entry_point)

SECTIONS
{
[...]
```

Nous nous ménageons donc un espace de 4096 octets en fin de flash pour y stocker ce que bon nous semble. Dans le précédent article sur le sujet, nous n'avons pas été confrontés au moindre problème, et pour cause : nous n'avions utilisé qu'un seul cœur. Ici, les choses sont un peu plus intéressantes. Je ne vais pas paraphraser ici ce qui a été détaillé dans le précédent article pour passer directement à la pratique. Avant toute chose, nous avons besoin des symboles provenant du script de l'éditeur de liens :

```
extern uint32_t __EEPROM_START__[];
uint32_t start_of_eeprom = (uint32_t)__EEPROM_START__;
extern uint32_t __EEPROM_LENGTH__[];
uint32_t eeprom_length = (uint32_t)__EEPROM_LENGTH__;
```

Ceci fait, nous avons accès aux données en flash et pouvons créer une structure pour stocker notre configuration :

```
typedef struct {
        int hmac;
        int digits;
        uint8_t key[MAXKEYLEN];
        size_t keylen;
        unsigned int checksum;
} config_t;
```

Nous reviendrons dans un instant sur ce membre `checksum`, mais avant, jetons un œil à la fonction qui récupère les données de la flash :

```
void
loadconfig()
{
    uint8_t *eeprom;
    eeprom = (uint8_t *)start_of_eeprom;
    memcpy(&config, eeprom, sizeof(config));
}
```

Et celle qui les enregistre :

```
void
saveconfig()
{
    uint8_t *eeprom;
    eeprom = (uint8_t *)start_of_eeprom;
    uint8_t *data;

    printf("saving config...\n");

    if ((data = malloc(eeprom_length)) == NULL) {
        printf("malloc() error!!!\n");
        while(1) { ; }
    }
    memset(data, 0, eeprom_length);
    memcpy(data, &config, sizeof(config));

    uint32_t ints = save_and_disable_interrupts();
    flash_range_erase((uint32_t)eeprom-XIP_BASE, eeprom_length);
    flash_range_program((uint32_t)eeprom-XIP_BASE,
        (const uint8_t *)data, eeprom_length);
    restore_interrupts(ints);

    free(data);

    printf("done.\n");
}
```

Il s'agit là principalement de copies de données en prenant en compte que l'adresse des données en flash est relative à l'espace d'adressage tel que vu par le programme, et surtout qu'il est capital de suspendre des interruptions avant écriture. Nous n'avons que 80 octets à écrire, mais la flash n'est pas accédée comme la SRAM en écriture. Il faut effacer la zone puis l'écrire, et ce, en utilisant des multiples de 4096 et 256 octets. Techniquement, il n'est pas nécessaire ici d'allouer et d'écrire les 4 Kio après copie des données puisqu'une écriture de 256 octets suffirait. Cependant, tout réécrire simplifie le code et nous permet de changer la structure à l'avenir sans rencontrer de problème (parce qu'on va forcément oublier ce point). On peut envisager de supporter plusieurs configurations par exemple, ce qui serait assez amusant (en multipliant des afficheurs).

Au démarrage, il nous suffit de lire la configuration et c'est là qu'intervient le fameux `checksum`. En effet, comment savoir si les données sont programmées ou non la première fois ? On pourrait envisager une configuration par défaut, mais ceci supposerait de flasher deux fois la Pico (une fois avec la configuration qui se copie en flash et une seconde fois, sans cette étape). Ma solution est de tout simplement ajouter une somme de contrôle (pour l'instant uniquement sur le secret) en partant du principe que, si elle est invalide, la configuration ne sera pas utilisée, le code TOTP pas calculé et que l'utilisateur devra utiliser la console série pour configurer (ou reconfigurer) le montage.

Notre *checksum* est un CRC32b relativement courant :

```
unsigned int
checksum(uint8_t *array, size_t len)
{
  int i, j;
  unsigned int byte, crc, mask;

  crc = 0xFFFFFFFF;
  for (i = 0; i < len; i++) {
    byte = array[i];
    crc = crc ^ byte;
    for (j = 7; j >= 0; j--) {
      mask = -(crc & 1);
      crc = (crc >> 1) ^ (0xEDB88320 & mask);
    }
  }
  return ~crc;
}
```

Et le début de notre `main()` se verra étoffé de :

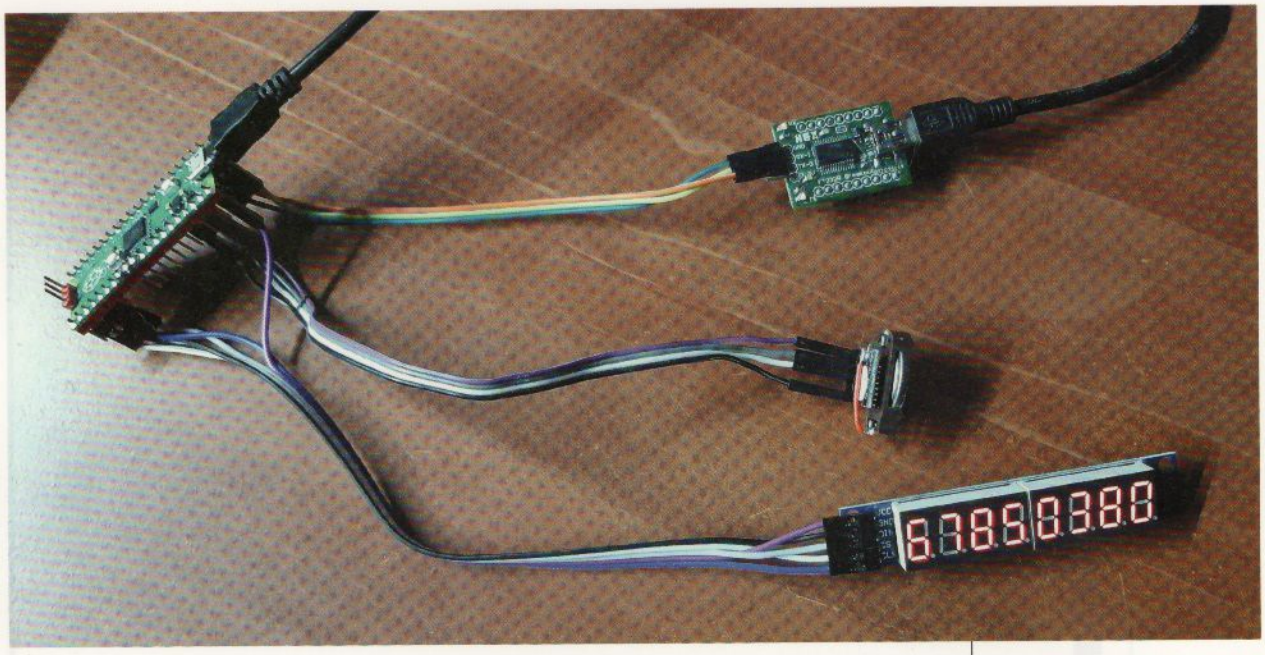
```
printf("Loading configuration...");
loadconfig();
printf("done.\n");

if (config.keylen > 64 ||
  config.checksum != checksum(config.key, config.keylen)) {
    printf("\n\nEEPROM configuration is invalid!\n");
```

```
} else {
  if ((status = readDS3231Time(&datetime))) {
    printf("Error getting RTC time, %s\n", ds3231ErrorString(status));
  }

  add_repeating_timer_ms(10000, gentotp_callback, NULL, &timer);

  code = totp(config.key, config.keylen, config.digits, config.hmac);
  max7219dispall(code);
}
```

Mais ce n'est pas tout. Nous avons, ou dans le déroulé de l'article « allons avoir », un **gros** problème parfaitement exposé dans une discussion sur le forum Raspberry Pi **[17]** : « *if the 2nd core is active and you do anything to the flash, it will likely crash* ». Il ne suffit pas, en effet, de désactiver les interruptions pendant l'effacement et l'écriture, il faut que cet accès soit exclusif. Or, contrairement à l'article du numéro 42, nous avons ici deux codes fonctionnant en parallèle et, effectivement, tenter d'utiliser `saveconfig()` plante totalement l'exécution (sauf, étrangement, si les données sont identiques).

La technique consiste donc à n'écrire en flash qu'après avoir arrêté l'autre cœur et pour ce faire, j'ai décidé d'utiliser un autre *flag* de manière à transférer cette opération de la boucle traitant les commandes utilisateur vers celle présente dans `main()`. Ainsi, si l'utilisateur désire enregistrer la configuration, nous commençons par permettre l'arrêt du cœur avec `multicore_lockout_victim_init()` avant de passer `saveflag` à `true`. C'est ensuite dans `main()` que tout se joue, avec une nouvelle version du `while()` infini :

```
while(1) {
  if (gentotp == true) {
    if (mutex_try_enter(&mutex, &owner_out)) {
      code = totp(config.key, config.keylen, config.digits, config.hmac);
      max7219dispall(code);
      gentotp = false;
      mutex_exit(&mutex);
    }
  }
  if (saveflag) {
    // arrêt timer
    cancel_repeating_timer(&timer);
    // calcul checksun
    config.checksum = checksum(config.key, config.keylen);
    // arrêt du second core
    multicore_lockout_start_blocking();
    // écriture flash
    saveconfig();
    // reboot
    watchdog_enable(250, 1);
    while(1) { ; }
  }
}
```



Le montage terminé est fonctionnel, mais il ne s'agit que d'un POC (Proof Of Concept) visant à affiner le concept. L'objectif final sera d'avoir un affichage beaucoup plus gros et un assemblage robuste pour une installation murale.

`multicore_lockout_start_blocking()` ne pourra être utilisé que sur un cœur ayant accepté de se faire arrêter (via `multicore_lockout_victim_init()`) et nous embrayons directement sur l'écriture de la configuration avant de provoquer un *reset* et donc relire et valider la configuration. Pour cela, il suffit d'activer le *watchdog* puis de ne rien faire pendant plus de 250 ms. Cette valeur arbitraire est choisie afin de laisser le temps aux messages d'arriver sur la console série, c'est purement cosmétique.

Ce problème réglé, nous pouvons enfin nous pencher que la partie la plus interactive...

3.5 Console série

Le code permettant la gestion des échanges avec l'utilisateur sera stocké dans `console.c` et la fonction contenant la boucle principale, `console()`, sera appelée depuis `main()` juste avant le `while()` qui vient d'être listé, avec `multicore_launch_core1(console)`. Cette fonction se résumera à un `while(1)` dont le corps est un énorme ensemble d'`if`/`else` et de `switch`/`case`. Nous divisons le jeu de commandes en deux catégories, celles permettant l'affichage d'informations, composées d'une unique lettre sans argument et celles modifiant la configuration active.

La libc utilisée par la chaîne de compilation ARM, et donc le SDK Pico, ne met pas à disposition de fonction permettant de traiter des entrées. La première chose à faire est donc de créer une fonction `getline()` pour créer une chaîne de caractères à partir des `char` obtenus, un à un, depuis l'UART :

```
ssize_t getline(char *buf, size_t size)
{
    size_t count;
    int c = 0;

    if (size == 0)
        return 0;

    for (count = 0; c != '\r' && count < size - 1; count++) {
        c = getchar();
        if (c == EOF) {
            if (count == 0)
                return -1;
            break;
        }
        buf[count] = (char)c;
    }

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

`buf` sera déclaré dans `console()` et possédera une taille fixe permettant de supporter toutes les commandes que nous avons à créer (y compris la configuration du secret) :

```
#define BUFFSIZE    256

[...]

void console() {
    char buffer[BUFFSIZE] = { 0 };
    int status;
[...]

    while(1) {
        if (getline(buffer, BUFFSIZE) > 0) {
            if (buffer[strlen(buffer) - 1] == 0x0d)
                buffer[strlen(buffer) - 1] = 0;
```

S'en suit un simple test via `strlen(buffer)`, si c'est un unique caractère, alors c'est une commande de consultation, sinon c'est une modification avec argument. Les commandes d'une lettre sont l'affichage de la date (`d`), la même chose mais en heure Unix (`t`), l'affichage du code TOTP (`c`), du HMAC utilisé (`h`), du nombre de chiffres du code (`n`), du secret (`k`), et de l'aide (`?`). À cela s'ajoute la réinitialisation de la configuration (`z`), l'enregistrement en flash avec redémarrage (`s`) et le redémarrage seul (`r`). Tout ceci prend la forme d'un simple `switch`/`case` ne présentant aucune difficulté particulière.

C'est du côté de la modification de la configuration que les choses deviennent plus intéressantes puisqu'il faut gérer les arguments. Ces commandes utilisent les mêmes lettres que pour la consultation, mais sont complétées d'informations directement accolées (`n` affiche le nombre

de chiffres, mais n6 change cette valeur à 6). La plupart des commandes ne posent pas de souci particulier puisqu'il suffit de détecter un simple chiffre ASCII qui, déduit de 48, nous donne directement la valeur à utiliser. D'autres ont besoin d'une conversion de chaîne vers un entier, comme t pour définir date/heure en temps Unix :

```
case 't':
  tmpt = (unsigned long)strtoul(buffer+1, &endptr, 10);
  if (endptr == buffer+1) {
    fprintf(stderr, "You must specify a valid number.\n");
  } else {
    mutex_enter_blocking(&mutex);
    if ((status = setDS3231Time(gmtime(&tmpt)))) {
      printf("Error setting time, %s\n",
          ds3231ErrorString(status));
    }
  mutex_exit(&mutex);
  }
  break;
```

Pour le secret (commande k), les choses se compliquent un peu, car nous devons convertir la chaîne en un tableau de valeur 8 bits. Pour cela, nous créons une fonction spécifique :

```
size_t hex2array(const char *line, uint8_t *array, size_t maxlen)
{
  size_t alen = 0;
  uint32_t temp;
  int indx = 0;
  char buf[5] = { 0 };

  while (line[indx] && alen < maxlen) {
    if (line[indx] == '\t' || line[indx] == ' ') {
      indx++;
      continue;
    }

    // traitement par paire
    if (isxdigit((int)line[indx])) {
      buf[strlen(buf) + 1] = 0x00;
      buf[strlen(buf)] = line[indx];
    } else {
      return 0;
    }

   if (strlen(buf) >= 2) {
      sscanf(buf, "%"PRIx32"", &temp);
      array[alen] = (uint8_t)(temp & 0xff);
      *buf = 0;
      alen++;
```

```
      if (alen > maxlen)
        return 0;
    }

    indx++;
  }

  // il reste un caractère
  if (strlen(buf) > 0)
    return 0;

  return alen;
}
```

La fonction pourrait être plus simple, mais nous choisissons de supporter la présence d'espaces et de tabulations dans la chaîne afin de faciliter la saisie manuelle (même si un copier-coller est plus que probable). Nous traitons alors les caractères reçus par groupes de deux que nous vérifions avec `isxdigit()` pour nous assurer qu'il s'agit bien de notation hexadécimale, avant de convertir avec `sscanf()`. À la moindre erreur, la fonction retourne `0` qui indique la taille du tableau et donc du secret final.

Mais la partie la plus pénible est sans le moindre doute le traitement d'une date et heure fournie dans un format humainement intelligible. Pénible, jusqu'à ce qu'on se rappelle l'existence de `strptime()` et on peut alors composer cela ainsi :

```
case 'd':
  if (strlen(buffer+1) != 19) {
    printf("Bad date/time format! Use 'dYYYY-mm-dd HH:MM:SS'.\n");
  } else {
    memset(&datetime, 0, sizeof(struct tm));
    if (strptime(buffer+1, "%Y-%m-%d %H:%M:%S", &datetime) == NULL) {
      printf("Conversions failed! Use 'dYYYY-mm-dd HH:MM:SS'.\n");
    } else {
      tmpt = mktime(&datetime);
      mutex_enter_blocking(&mutex);
      if ((status = setDS3231Time(gmtime(&tmpt)))) {
        printf("Error setting time, %s\n", ds3231ErrorString(status));
      }
      mutex_exit(&mutex);
    }
  }
  break;
```

Et on se heurte alors à un problème, puisque le compilateur précise immédiatement que nous avons là une déclaration implicite de la fonction, alors même que nous avons correctement inclus `time.h` (via `libs/ds3231.h`). La page de manuel de la fonction précise qu'il faut définir la macro `_XOPEN_SOURCE`, mais là encore, ce n'est pas suffisant. En réalité, vous devez en définir deux en tout début de source :

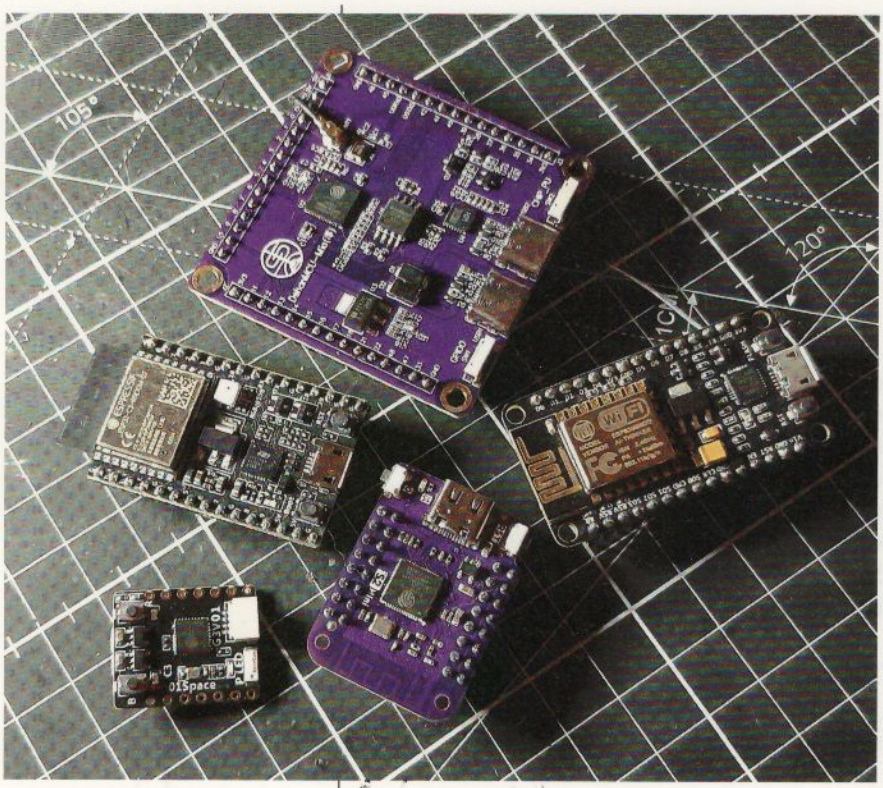
```
#define __USE_XOPEN
#define _GNU_SOURCE
```

Et pour finir le tout, il ne nous reste plus qu'à ajouter une fonction pour établir une configuration par défaut :

```
void
setdefconf()
{
  uint8_t k[64] = {
      0x9a, 0x32, 0xec, 0xb1, 0xd5, 0x0a, 0xae, 0x25,
      0xff, 0xbe, 0x8d, 0x16, 0x79, 0x06, 0x48, 0x19,
      0x9e, 0x0a, 0xdc, 0x0c, 0x4c, 0x31, 0xb8, 0xf0,
      0x39, 0x38, 0xfc, 0xc2, 0x7c, 0x02, 0x68, 0x89,
      0xd2, 0xd6, 0x4d, 0xf0, 0x63, 0xc4, 0x24, 0xd6,
      0x7b, 0x8f, 0xc4, 0x86, 0x9b, 0x19, 0x9a, 0x2b,
      0x45, 0xcc, 0xf5, 0x8f, 0x07, 0xa4, 0x86, 0x47,
      0x57, 0xeb, 0x6b, 0xa1, 0xe1, 0x90, 0x01, 0x6c
  };
  printf("sizeof config = %zu\n", sizeof(config));

  config.hmac = SHA512_DIGEST_SIZE;
  config.digits = 8;
  config.keylen = 64;
  memcpy(config.key, k, 64);
}
```

Et une autre pour afficher le code TOTP en console avec une légère mise en forme :

```
void
dispwithgap(uint32_t code, int digits, int dispgap)
{
  char dispcode[32];

  if (!dispgap) {
    printf("%0*lu", digits, code);
    return;
  }

  snprintf(dispcode, digits+1, "%0*lu", digits, code);
  for (int i=0; i<strlen(dispcode); i++) {
    printf("%c", dispcode[i]);
    if (((i + 1) % dispgap) == 0)
      printf(" ");
  }
}
```



Différents modèles de microcontrôleurs Espressif disposent de fonctionnalités cryptographiques qui renforceraient grandement la sécurité en rendant inaccessible le secret en lecture, une fois inscrit en mémoire. De plus, le calcul HMAC est confié directement au matériel, simplifiant grandement le code. C'est une option autour de laquelle je risque d'expérimenter dans l'avenir...

CONCLUSION

Comme me l'a fait remarquer une personne qui se reconnaîtra sur Twitter, l'un des principes de base de TOTP est aussi d'avoir l'*authenticateur* sur soi, comme on a son mot de passe dans sa tête (en principe). Mais on peut également voir au-delà du concept initial en se détachant de la notion de « prouver qui on est » pour plutôt aller vers le « prouver **où** on est », et ce, indépendamment de la méthode de connexion (Wi-Fi, Ethernet, 4G, etc.). Ainsi, avec un tel afficheur dans un lieu sécurisé, il est parfaitement possible de restreindre l'accès à un service et le rendre dépendant d'une présence physique à cet endroit donné. Le tout, en permettant à certaines personnes de tout de même y accéder de n'importe où, en utilisant, par exemple, *Aegis Authenticator* sur leur smartphone, avec une configuration identique. Eh oui, ceci revient plus ou moins à avoir un périphérique de type Token2 C302-i rangé dans un tiroir sur place, mais sans les disputes à propos de « qui l'a mal rangé où » et sans le risque de vol, de disparition ou de perte.

Bien entendu, cela ne règle pas le principal problème de ce type de création « maison », à savoir le manque de sécurité matérielle. J'ai choisi de mettre à disposition une commande permettant d'afficher le secret via la liaison série pour une raison très simple : si vous avez accès au montage, rien ne vous empêche de lire la flash et d'extraire le secret pour l'utiliser par ailleurs. Ceci n'est absolument pas sûr et une version « industrielle » devra trouver une solution au problème. Une option possible consiste à migrer du microcontrôleur RP2040 vers quelque chose offrant davantage de fonctionnalités, en particulier côté cryptographie. Les ESP32-S2 et S3 (Xtensa), ainsi que C3, C6 et H2 (RISC-V) offrent ce type de chose en mettant à disposition non seulement des fonctions cryptographiques matérielles (dont HMAC SHA-256), mais en permettant de stocker les clés de façon sécurisée. D'autres constructeurs proposent

également ce type de spécifications pour leurs microcontrôleurs, dont les STM32.

Ensuite, dans les évolutions possibles du projet, nous avons également pas mal de largesse. Certaines d'entre elles seront même probablement implémentées au moment où vous lirez cet article, puisque le développement n'est aucunement arrêté à ce stade. Parmi les possibilités envisageables, nous avons le fait d'ajouter des commandes et d'étoffer la configuration, en incluant par exemple la luminosité de l'afficheur. Il est également possible d'utiliser les points des afficheurs comme indicateur temporel de validité du code TOTP. Mais, par-dessus tout, une évolution majeure pourrait être le support de plusieurs configurations et non d'une seule. Ainsi, le montage serait complété d'autres afficheurs à LED et la configuration divisée en plusieurs structures sous la forme d'une liste chaînée. On pourrait alors voir s'afficher un groupe de 8 chiffres par configuration, avec un secret et un HMAC différent pour chacun d'eux.

Et enfin, cette réalisation m'ayant donné envie d'explorer les fonctionnalités cryptographiques malheureusement absentes du RP2040, je pense probable qu'un portage vers ESP32 (ou peut-être STM32) voie le jour, à un moment ou un autre. La disponibilité du Wi-Fi pourrait permettre, de plus, une parfaite synchronisation en utilisant un protocole comme NTP pour corriger d'éventuelles dérives d'horloge. À suivre, donc... **DB**

RÉFÉRENCES

[1] **https://github.com/google/google-authenticator/wiki**

[2] **https://freeotp.github.io/**

[3] **https://github.com/beemdevelopment/Aegis**

[4] **https://connect.ed-diamond.com/hackable/hk-046/creons-un-outil-de-configuration-pour-un-token-totp**

[5] **https://docs.espressif.com/projects/esp-idf/en/latest/esp32s2/api-reference/peripherals/hmac.html**

[6] **https://blog.trezor.io/introducing-tropic-square-why-transparency-matters-a895dab12dd3**

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

[8] **https://github.com/fmount/c_otp**

[9] **https://github.com/bnielsen1965/pi-pico-c-rtc**

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

[11] **https://marc-stevens.nl/research/md5-1block-collision/**

[12] **https://www.ietf.org/rfc/rfc4226.txt**

[13] **https://rollmops.ninja/**

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

[15] **https://connect.ed-diamond.com/hackable/hk-047/home-assistant-quelques-changements-ajouts-et-ajustements**

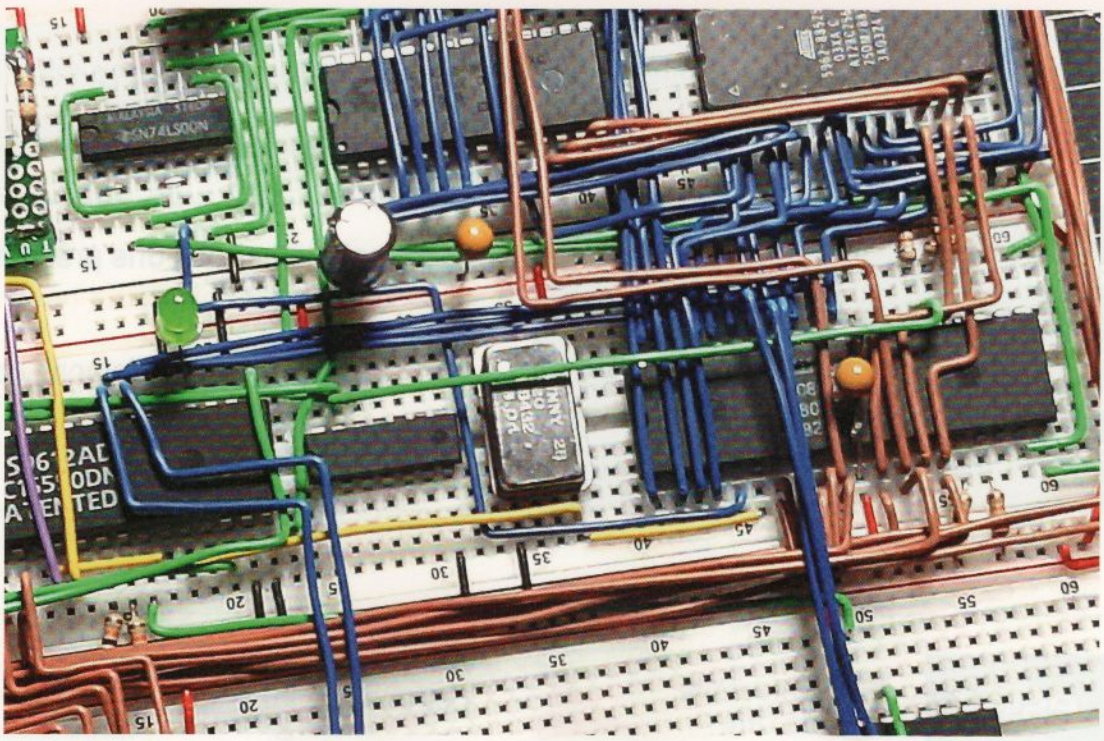
[16] **https://connect.ed-diamond.com/hackable/hk-042/raspberry-pi-pico-emuler-une-eeprom-interne-pour-le-stockage-de-donnees**

[17] **https://forums.raspberrypi.com/viewtopic.php?t=310689**

Z80 SUR PLATINE : APRÈS LE HARD, LE SOFT !

Denis Bodor

Dans le précédent article, nous avons fait le tour des différents éléments matériels qui composent notre ordinateur 8 bits, pour l'instant assez minimaliste, construit sur platine à essais autour d'un processeur Zilog Z80. Si vous avez suivi les indications que j'ai détaillées, vous devez, en principe, avoir quelque chose de fonctionnel, et il est temps de vérifier tout cela avec un peu de code.



Avant toute chose, l'une des deux implémentations en ma possession a subi une légère évolution que j'avais brièvement évoquée. En effet, ma version platine à essais utilise à présent un unique oscillateur à quartz de 1,8432 MHz, similaire à celui de 1 MHz utilisé pour le processeur. Ceci permet de légèrement simplifier le montage puisque nous n'avons plus besoin d'un quartz dédié à l'UART ni des deux résistances et condensateurs céramiques. Le Z80 gagne également en rapidité avec 843,2 kHz de plus et l'ensemble est parfaitement stable, du moins avec une platine à essais de qualité. L'impact sur le ou les codes que nous allons développer est minime, car rien ne change concernant l'UART, seule la gestion des délais sera sensiblement différente puisque le Z80 fonctionne plus rapidement. Le contrecoup de ce changement est donc purement « évolutif », car s'il nous vient l'envie de rebasculer le Z80 sur un oscillateur de 1 MHz pour une raison ou une autre, l'UART sera difficilement configurable. Ma version « plaque pastillée » n'a, en revanche, pas changé puisqu'il aurait fallu dessouder les composants et revoir l'agencement. Encore une fois, ceci ne change rien, et vous n'êtes absolument pas obligé de faire de même de votre côté.

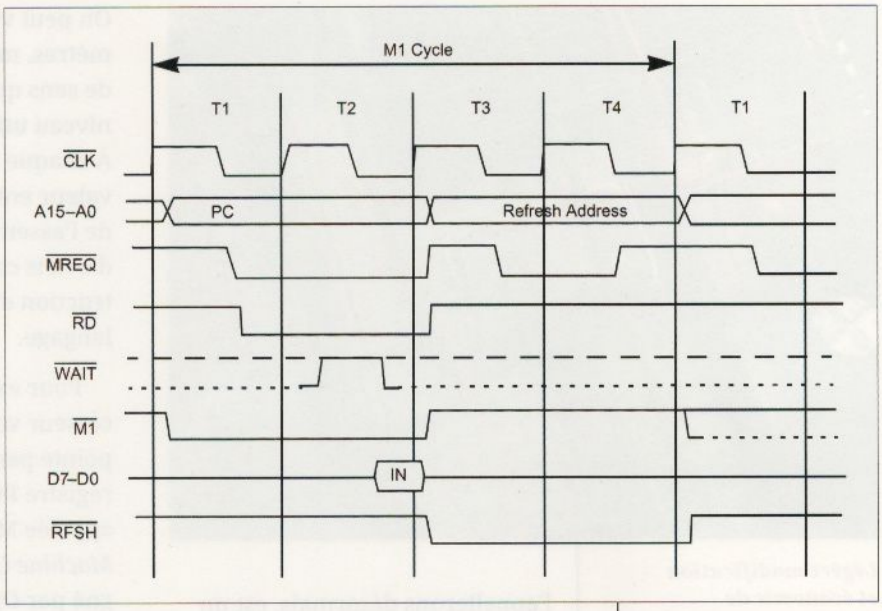
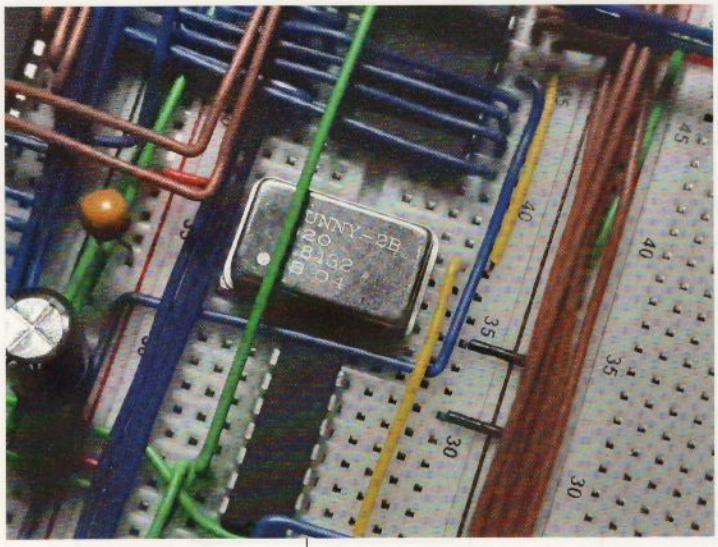


Diagramme inclus dans le manuel du Zilog Z80, présentant le cycle M1 et les différents changements d'état des broches/signaux du processeur en fonction des cycles d'horloge.

1. DONNER DU CODE BINAIRE AU PROCESSEUR

Comme décrit dans l'article précédent, notre ordinateur utilise un espace mémoire dont les adresses sont composées de 16 bits. Le bit le plus haut et donc de poids le plus fort est utilisé pour activer soit l'EEPROM 28C256, soit la RAM 62256, respectivement de 32 Kio chacune. Notre espace d'adressage se divise donc en deux avec d'un côté `0x0000` à `0x7FFF` pour l'EEPROM (ou ROM puisque du point de vue du Z80, celle-ci n'est accessible qu'en lecture seule) et de l'autre 32 Kio de RAM de `0x8000` à `0xFFFF`. Lorsque le processeur est mis sous tension et/ou réinitialisé, il va automatiquement présenter l'adresse `0x0000` sur le bus pour procéder à une lecture des données. Ceci parce qu'à ce moment précis, le compteur ordinal ou pointeur d'instruction (*program counter* ou PC en anglais) est initialisé à `0x0000`. Le PC, comme nous



Légère modification et économie de composants par rapport au montage initial : cette version utilise un unique oscillateur 1,8432 MHz cadençant à la fois le Z80 et l'UART PC16550DN.

l'appellerons désormais, est un registre du processeur contenant toujours l'adresse de la prochaine instruction à exécuter, et est incrémenté après chaque lecture (*fetch*).

Entrons un instant un peu plus en détail dans le fonctionnement du Z80 pour bien comprendre comment une instruction est exécutée. Et pour cela, commençons par préciser clairement ce qu'est une instruction. Si nous voulons, par exemple, placer la valeur `0x41` dans le registre *A* (l'accumulateur) du Z80, nous utiliserons l'instruction assembleur suivante :

```
ld a,#0x41
```

L'instruction en question, qui est l'ensemble de la ligne, se compose d'un *opcode* pour *operation code*, qui deviendra un octet après assemblage (ici, `0x3e`) et d'un opérande spécifiant ce sur quoi porte l'opération. On peut voir cela comme un ou des paramètres, même si ce terme n'a réellement de sens que pour les langages de plus haut niveau utilisant fonctions et arguments. À chaque *opcode* correspond donc une valeur entre `0x00` et `0xff` et c'est le travail de l'assembleur de générer un ensemble d'octets correspondant à chaque instruction d'un programme écrit dans ce langage.

Pour exécuter une instruction, le processeur va tout d'abord lire l'*opcode* pointé par l'adresse contenue dans le registre PC. Cette première phase est appelée M1 dans le jargon du Z80, pour *Machine Cycle 1*, parfois également désigné par *Opcode Fetch*. Puis, en fonction de l'*opcode* en question, qui a été lu et placé dans un registre interne, peut s'en suivre aucune, une ou plusieurs lectures mémoire (M2 ou *Memory Read*) et/ou écritures mémoire (M3 ou *Memory Write*).

Ces cycles machine ne correspondent pas aux cycles d'horloge du système qui sont, quant à eux, désignés par la lettre T suivie d'un numéro à l'intérieur de chaque cycle machine. Un cycle d'horloge est un cycle complet du signal en entrée sur la broche 6 du Z80, non un simple changement d'état (mais deux).

Ainsi, M1 se découpe en T0, T1, T2 et T3 où chaque cycle déclenche un certain nombre d'actions physiques : le contenu de PC est présenté sur le bus d'adresses en même temps que la broche /M1 est mise à la masse, la broche /MREQ passe ensuite à la masse, puis /RD, les données présentes sur le bus de données sont prisent en compte, /M1 repasse à la tension d'alimentation, puis /RD, puis /MREQ et arrive ensuite le mécanisme de rafraîchissement pour la mémoire dynamique (qu'elle soit utilisée ou non), en passant /RFSH à la

masse, puis en présentant l'adresse mémoire à rafraîchir sur les lignes A0 à A15 et en basculant brièvement /MREQ à la masse avant de conclure par un passage de /RFSH à Vcc. À noter que le Z80 dispose d'une sortie /M1, en broche 27, passant à la masse à chaque fois que le processeur se trouve dans le cycle M1. Ceci permet de visualiser clairement l'état du Z80 et d'éventuellement utiliser ce signal pour d'autres usages. Le « *User Manual* » du Z80 **[1]** présente un diagramme résumant tout cela en page 9. Pour des informations plus détaillées que le manuel, je recommande la lecture du blog d'Andre Weissflog (floooh) et en particulier un billet qu'il a rédigé à propos de son émulateur Z80 **[2]**.

Ainsi, après cette légère digression technique, ce que nous avons à faire pour programmer notre ordinateur 8 bits est simplement de lui mettre la première instruction d'une série composant un programme, à l'adresse pointée par le registre PC au *reset*. Cette adresse est en ROM à `0x0000` et pour ce faire, nous allons devoir écrire une description textuelle des instructions, qui sera traduite ensuite en code binaire à inscrire dans la ROM. Autrement dit, nous devons écrire un code assembleur et l'assembler avec un outil dédié. Oui, on peut techniquement aussi le faire à la main, avec un crayon et un bout de papier, mais nous sommes en 2023, que diable !

2. SDCC : SMALL DEVICE C COMPILER

Pour créer notre premier programme, nous avons besoin d'un assembleur et, vous l'aurez compris, celui-ci est intimement lié au processeur utilisé. Il en existe plusieurs pour le Z80 et même si globalement la syntaxe est similaire entre chacun d'eux, puisqu'après tout il ne s'agit que de traduire un jeu d'*opcodes* et d'opérandes définis par le constructeur du processeur, il existe quelques différences empêchant de simplement assembler, sans modification, un programme prévu pour un assembleur avec un autre.

Pour le Z80, nous avons énormément de choix, *Z80 assembler* (alias `z80asm`), *Z80-ASM* de Petr Kulhavy, *Pasmo*, *CRASM*... ou encore l'assembleur de SDCC (`sdasz80`). Étant donné que l'objectif, à terme, est de programmer dans un langage de plus haut niveau, du C en l'occurrence, j'ai directement opté pour SDCC après quelques essais avec `z80asm` **[3]**. Comme nous le verrons plus tard, lorsqu'on développe pour une plateforme comme le Z80, il n'est pas rare de mélanger C et assembleur. Il est donc judicieux, à mon sens, de préférer un unique ensemble d'outils homogènes plutôt que de créer des redondances inutiles.

SDCC est un projet existant de longue date et toujours activement en développement, la dernière version 4.3.0 date d'ailleurs de juin de cette année. C'est une suite (ou chaîne) de compilation complète comprenant le compilateur, l'assembleur, l'éditeur de liens et un ensemble d'outils complémentaires (simulateur, *debugger*, *binutils*, etc.) permettant de produire du code binaire à destination des familles de processeurs Intel MCS51, Motorola HC08, Padauk, STMicroelectronics STM8 ou encore Zilog Z80, incluant Z80 bien sûr, mais aussi Z180, SM83, Rabbit 2000/2000A/3000A, etc.

SDCC est disponible pour la totalité des systèmes d'exploitation, *open source* ou non, et est fréquemment directement intégré sous la forme de paquets très faciles à installer. Une autre option, en particulier si vous souhaitez profiter de la dernière version stable, consiste à recompiler le tout. C'est

relativement facile, sur un système de type Unix *open source* du moins, et ne nécessitera aucune modification du système (installation) pour être utilisable. Un simple `./configure --prefix $HOME/kkpart` suivi de `make && make install`, après désarchivage des sources, vous permettra d'obtenir tous les outils, directement dans un sous-répertoire `bin/` de `$HOME/kkpart` avec tous les fichiers connexes (*headers* et bibliothèques), et tout fonctionnera à merveille en appelant ces commandes depuis cet endroit.

3. PREMIER PROGRAMME

Pour ce premier « essai », notre objectif sera d'aller immédiatement plus loin que la simple série de `NOP` (*No OPeration*), instruction ne faisant rien, ou la boucle basique infinie. Je pars du principe que vous n'avez pas forcément un analyseur logique à disposition, permettant de capturer les états du bus d'adresses et autres signaux de gestion, et donc qu'il nous faut un autre moyen de s'assurer que le programme fait bien ce qu'il doit faire. Cela tombe bien, nous avons une interface série à disposition puisque nous avons intégré l'UART PC16550DN dès le départ.

Cependant, pour utiliser ce périphérique, nous devons d'abord le configurer et ceci passe par l'initialisation de certains de ses registres afin de choisir la vitesse de communication, le format des données et l'éventuelle utilisation de signaux de contrôle de flux. Nous avons vu dans l'article précédent que l'accès à la ROM ou la RAM est signalé via la broche /MREQ (*Memory Request*) passant à la masse. Il en va de même pour l'accès aux périphériques d'entrée/sortie, via la broche /IORQ (*Input/Output Request*). À ce moment, une adresse est présentée sur les lignes du bus d'adresses et via les signaux /RD ou /WR, le processeur lit ou écrit des données via les broches D0 à D7.

Le PC16550DN, pour sa part, dispose de seulement trois lignes d'adresses, A0 à A2, et nous avons directement connecté son entrée /CS2 au /IORQ du Z80. Ceci signifie donc que dès que le processeur souhaite procéder à une entrée/sortie sur un périphérique, l'UART sera activé, que ce soit en lecture (via son /RD) ou en écriture (via son /WR). Nous verrons, plus loin dans la série, comment gérer plusieurs périphériques (*spoiler alert* : de la même façon que nous distinguons ROM et RAM), mais pour l'heure, ceci signifie que lire ou écrire à l'adresse d'entrée/sortie 0x00, par exemple, aura le même effet que de le faire à l'adresse 0x08, 0x10 ou encore 0x58, car l'UART ne voit que les trois bits de poids le plus faible sur le bus d'adresses. Ce n'est pas un problème, mais c'est important de le savoir pour plus tard.

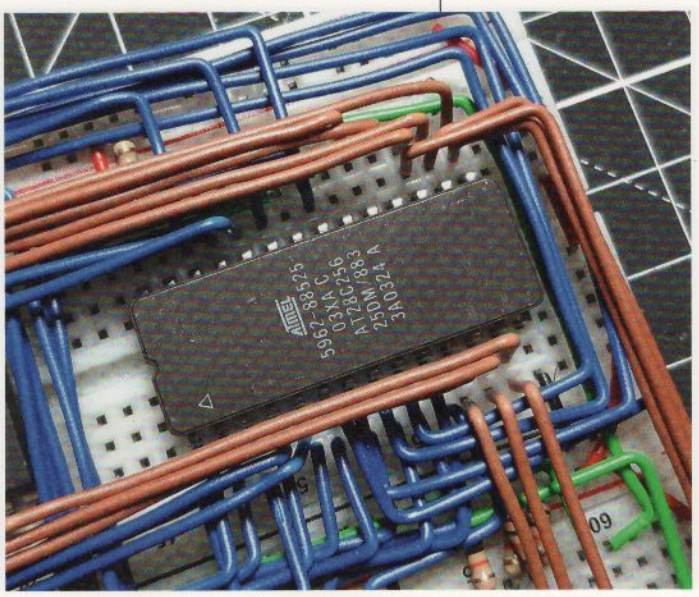
Qui dit 3 bits d'adresses dit 8 emplacements adressables dans le PC16550DN et donc trois registres pour y lire ou écrire des valeurs 8 bits. Attention, ceci n'est pas une vérité universelle, il existe des périphériques avec 3 lignes d'adresses n'ayant, par exemple, que 5 ou 6 registres. Les registres du PC16550DN sont les suivants :

- `0x00` RBR (*Receiver Buffer Register*) en lecture seule et THR (*Transmitter Holding Register*) en écriture seule. Ce sont respectivement les registres où nous lisons un octet reçu ou écrivons un octet à envoyer. Notez que dans l'absolu, il ne s'agit que d'un registre, mais se comportant de deux façons différentes selon l'opération (lire ou écrire) qui est faite, et de ce fait est représenté par deux registres à la même adresse dans la documentation du composant **[4]**.

- `0x01` IER (*Interrupt Enable Register*) : permet de choisir quels événements auront un impact sur l'état de broche INTR du composant, qui peut être reliée à la broche /INT du Z80. Ceci n'est pas important pour l'instant, nous verrons cela lorsqu'il sera question de traiter des interruptions (IRQ). Par défaut, au *reset*, aucune interruption n'est configurée (cf. page 13 de la documentation pour les valeurs par défaut des registres).
- `0x02` IIR (*Interrupt Ident. Register*) en lecture seule et FCR (*FIFO Control Register*) en écriture seule. Encore un registre qui, selon l'opération, aura un comportement et un nom différent. En lecture, il permet de savoir quel événement a déclenché une interruption, mais en écriture, son utilisation est totalement différente, puisqu'il permet de configurer une mémoire tampon (*buffer*) FIFO (*First-In First-Out*) de 16 octets, interne au composant. Ce *buffer* stocke les données le temps qu'elles soient transmises ou lues. Écrire dans ce registre permet d'activer le *buffer*, l'effacer, configurer la génération d'une interruption, etc.
- `0x03` LCR (*Line Control Register*) : ce registre permet de configurer les caractéristiques du format de données utilisé et en particulier, le nombre de bits de données et de stop ainsi que la parité (paire ou impaire), si elle est utilisée.
- `0x04` MCR (*MODEM Control Register*) : permet de régler les paramètres de communication avec un MODEM et les signaux qu'il utilise, en plus des simples lignes de transfert de données (RX/TX) DTR, RTS, DCD, CTS, DSR, etc. Ceci n'est plus très pertinent aujourd'hui.
- `0x05` LSR (*Line Status Register*) : fournit des informations sur l'état des transferts de données, dont la disponibilité du registre THR pour l'envoi ou la présence de données à lire dans RBR. C'est aussi ici que seront signalées d'éventuelles erreurs.
- `0x06` MSR (*MODEM Status Register*) : est lié à MCR et permet de s'informer de l'état des lignes de contrôle (CTS, DCD, etc.) pour les échanges avec un MODEM. Encore une fois, les communications analogiques via MODEM étant presque totalement inexistantes aujourd'hui, ce registre n'est plus important.
- `0x07` SCR (*Scratchpad Register*) : ce registre ne fait strictement rien concernant l'UART, c'est un simple emplacement de stockage de 8 bits pour le développeur.

Cela fait beaucoup pour seulement huit registres, je vous le concède, mais ce n'est pas tout. Le registre LCR dispose d'un bit (7) particulier appelé DLAB (*Divisor Latch Access Bit*) changeant l'utilité (et les noms) des registres aux adresses `0x00` et `0x01`.

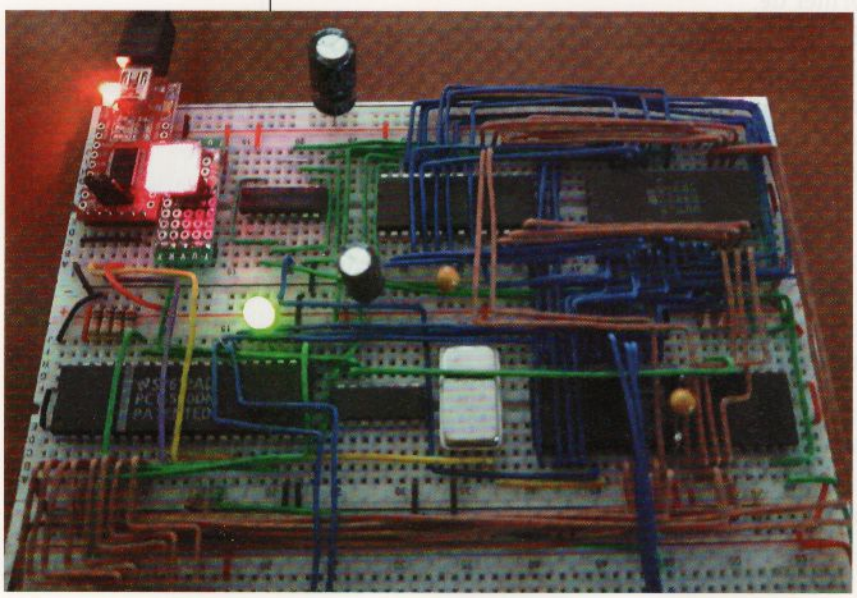
Le fait d'avoir un code en EEPROM démarré automatiquement au reset est quelque chose qui est toujours d'actualité sur les machines récentes. On appelle cependant cela aujourd'hui un firmware UEFI et il est stocké dans une flash série.



Si DLAB vaut 1, ces deux adresses correspondent respectivement aux registres DLL (*Divisor Latch LS*) et DLM (*Divisor Latch MS*). Il permet de stocker une valeur sur 16 bits divisée en 8 bits de poids fort (MS pour *Most Significant byte*) et 8 bits de poids faible (LS pour *Least Significant byte*). Cette valeur est un diviseur qui permet au PC16550DN de définir une vitesse de communication (*baud rate*) en fonction de la fréquence de la source d'horloge utilisée.

Nous utilisons un oscillateur (ou un quartz) à 1,8432 MHz, et cette fréquence est tout d'abord divisée par 16, puis par la valeur sur 16 bits, spécifiée via ces deux registres. Si, par exemple, vous souhaitez une communication à 2400 bps, il suffit de faire le calcul : 1 843 200 Hz / 16 / 2400 = 48. Nous pouvons alors placer 0x30 (48) dans DLL et 0x00 dans DLM, et le port série de votre ordinateur communiquera à cette vitesse. Si vous ne voulez pas faire les calculs, la documentation du PC16550DN fournie même un jeu de valeurs en page 15 allant de 50 bps à 56 000 bps avec un oscillateur à 1,8432 MHz. En ce qui nous concerne, le débit courant de 9600 bps fera parfaitement l'affaire et le diviseur sera donc 12.

Une des règles implicites de la construction d'un ordinateur 8 bits « retro-brew » est d'impérativement garnir la réalisation de quelques LED. Plus il y en a, mieux c'est... tant qu'il reste de la place sur les platines à essais.



Bien entendu, pour arriver à faire cela, il faut que DLL et DLM soit accessibles, et non RBR/THR et IER aux mêmes adresses. Nous devons donc placer le bit DLAB à 1 dans LCR, et le mettre à 0 une fois les deux valeurs inscrites. Et ça tombe très bien, puisque nous devons de toute façon toucher à LCR pour configurer le format des données. Nous avons déjà défini que la communication se fera à 9600 bps et à cela s'ajoute maintenant qu'il y aura 8 bits de données, deux 2 de stop et une parité impaire, généralement désignée en 8O2 par opposition au classique 8N1 (8 bits de data, 1 bit de stop et pas de parité).

Les bits de LCR sont :

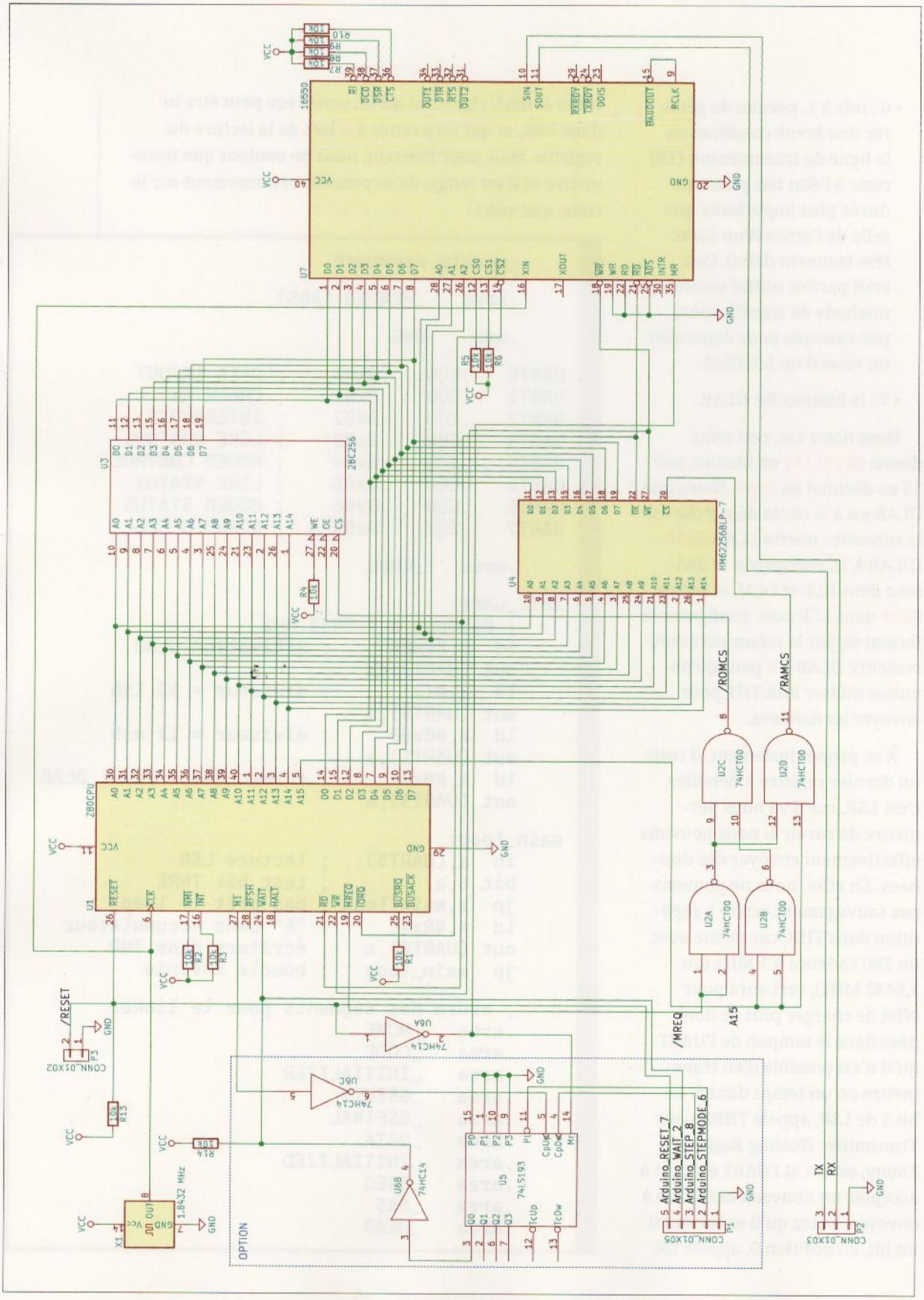
- 0 et 1 : nombre de bits de données avec `00` = 5 bits, `01` = 6 bits, `10` = 7 bits et `11` = 8 bits.
- 2 : nombre de bits de stop avec `0` = 1 bit et `1` = 2 bits.
- 3 : parité avec `0` = non et `1` = oui.
- 4 : parité paire/impaire avec `0` = impaire et `1` = paire.
- 5 : permet de forcer le bit de parité à 1 ou 0. Ceci est rarement utilisé et est désigné dans la documentation de l'UART par les termes *sticky parity*, également appelé par ailleurs parité *mark/space* (synonymes de *high/low*, état haut/bas, en communication série).

- 6 : mis à 1, permet de générer une *break condition* où la ligne de transmission (TX) reste à l'état bas pour une durée plus importante que celle de l'envoi d'un caractère (souvent deux). Ceci était parfois utilisé comme méthode de signalisation, par exemple pour demander un *reset* d'un MODEM.
- 7 : le fameux bit DLAB.

Dans notre cas, ceci nous donne `00001111` en binaire, soit 15 en décimal ou `0x0f`. Notez que DLAB est à `0`, car la démarche est la suivante : mettre LCR à `0x80` (DLAB à `1`), enregistrer le diviseur dans DLL et DLM, et écrire `0x0f` dans LCR pour configurer le format et, par la même occasion, remettre DLAB à `0` pour qu'on puisse utiliser RBR/THR pour envoyer les données.

À ce propos justement, il reste un dernier registre à détailler, c'est LSR, car il va nous permettre de savoir si nous pouvons effectivement envoyer des données. En effet, nous ne pouvons pas sauvagement écrire à répétition dans THR, car même avec un Z80 cadencé à 1 MHz (ou 1,8432 MHz), ceci aura pour effet de charger plus de données dans le tampon de l'UART qu'il n'est possible d'en transmettre en un temps donné. Le bit 5 de LSR, appelé THRE pour *Transmitter Holding Register Empty*, est à `1` si l'UART est prêt à accepter un nouveau caractère à envoyer. Notez qu'il existe aussi un bit, en position 0, appelé DR (*Data Ready*), signalant qu'un octet reçu peut être lu dans RBR, et qui sera remis à `0` lors de la lecture du registre. Mais pour l'instant, nous ne voulons que transmettre et il est temps de se pencher sérieusement sur le code, que voici :

```
    .module uarttest

    .area   _HEADER (ABS)

    .org    0x0

UART0   .EQU    0x00    ; DATA IN/OUT
UART1   .EQU    0x01    ; CHECK RX
UART2   .EQU    0x02    ; INTERRUPTS
UART3   .EQU    0x03    ; LINE CONTROL
UART4   .EQU    0x04    ; MODEM CONTROL
UART5   .EQU    0x05    ; LINE STATUS
UART6   .EQU    0x06    ; MODEM STATUS
UART7   .EQU    0x07    ; SCRATCH REG

    .area   _CODE

init_uart:
    ; 9600bps à 1.8432 MHz
    ld  a,#0x80     ; activation DLAB
    out (UART3),a
    ld  a,#12       ; diviseur = 12 lsb
    out (UART0),a
    ld  a,#0x00     ; diviseur = 12 msb
    out (UART1),a
    ld  a,#0x0f     ; format 8O2 + reset DLAB
    out (UART3),a

main_loop:
    in  a,(UART5)   ; lecture LSR
    bit 5,a         ; test bit THRE
    jp  z,main_loop ; pas prêt -> loop
    ld  a,#0x41     ; "A" dans accumulateur
    out (UART0),a   ; écriture dans THR
    jp  main_loop   ; boucle infinie

    ; ordre des segments pour le linker
    .area   _HOME
    .area   _CODE
    .area   _INITIALIZER
    .area   _GSINIT
    .area   _GSFINAL
    .area   _DATA
    .area   _INITIALIZED
    .area   _BSEG
    .area   _BSS
    .area   _HEAP
```


16550
U7
Z80CPU
U1
28C256
U3
HM62256BLP-7
U4
74HCT00
U2A
U2B
U2C
U2D
74HC14
U6A
U6B
U6C
U6B
74LS193
U5
1.8432 MHz
X1
/RESET
/ROMCS
/RAMCS
/MREQ
A15
TX
RX
OPTION
Arduino_RESET_7
Arduino_WAIT_2
Arduino_STEP_8
Arduino_STEPMODE_6
CONN_01X02
CONN_01X05
CONN_01X03
VCC
GND

Parlons tout d'abord des différentes occurrences de `.area` permettant d'indiquer à l'assembleur et surtout, ensuite, à l'éditeur de liens, comment est organisé notre code et où les différentes parties doivent aller. Dans le cas présent, avec un programme aussi simple, la plupart de ces **segments** sont vides et n'ont de sens que pour des développements futurs où nous utiliserons le langage C, avec utilisation de la RAM, un tas (*heap*), une pile (*stack*) et des variables globales à initialiser. Car, en effet, ce premier code n'utilise que la ROM et débute à l'adresse `0x0000` tel que précisé par la directive `.org`.

Pour nous simplifier la vie et ne pas avoir besoin d'utiliser des adresses, nous utilisons `.EQU` pour définir des symboles ayant pour valeur l'adresse des différents registres de l'UART. Notez que tout ce qui suit un point-virgule n'est pas interprété par l'assembleur, ce sont de simples commentaires.

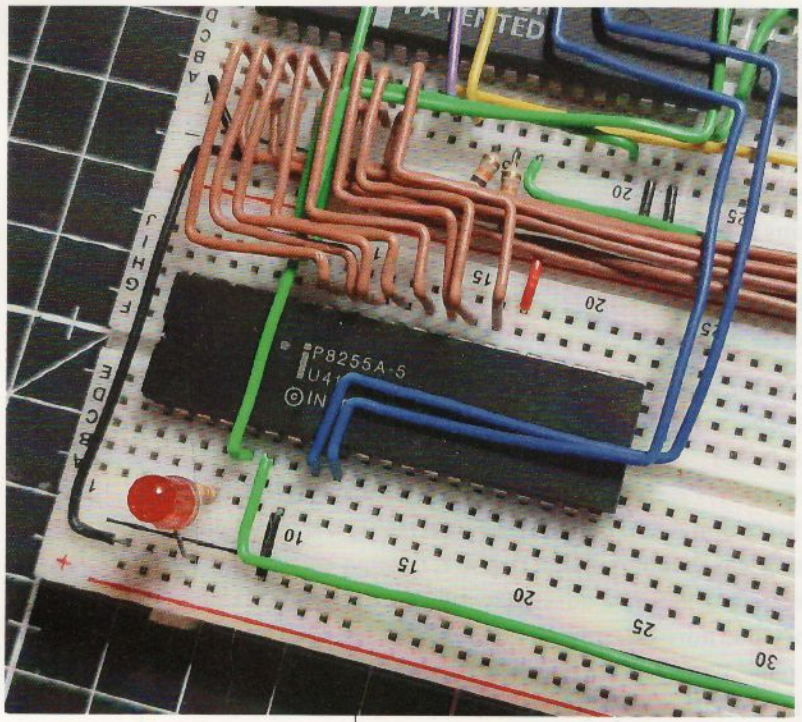
`init_uart:` et `main_loop:` nous permet de définir des étiquettes (ou labels) qui sont utilisés comme des points de référence dans le code, correspondant respectivement aux adresses où se trouvent les instructions `ld a,#0x80` et `in a,(UART5)`.

Ce qui nous amène donc à parler des cinq instructions que nous utilisons, mais avant cela, des modes d'adressage et/ou, autrement dit, de la façon de désigner des adresses et valeurs en assembleur. Le symbole `#` placé devant une valeur précise qu'il s'agit d'une **valeur littérale**. C'est le cas de `#0x80`, par exemple, indiquant que nous plaçons **littéralement** la valeur `0x80` dans le registre *A*. En l'absence de ce symbole, l'assembleur SDCC nous rappellera immédiatement à l'ordre, car il n'existe pas d'instruction `ld a,` prenant autre chose qu'un registre, un littéral ou une adresse mémoire en opérande.

Pour spécifier une adresse, justement, nous utilisons les parenthèses. Ainsi, si nous voulions placer l'octet se trouvant à l'adresse `0x0123` de la mémoire dans le registre *A*, nous utiliserions `ld a,(0x0123)`. On parle alors d'**adressage direct** et ceci fonctionne avec une adresse littérale (le `#` n'est ici pas nécessaire) ou le contenu d'un registre, comme `ld a,(lh)` (**adressage implicite** ou adressage indirect par registre). Là, c'est l'octet à l'adresse contenue dans le registre (16 bits) *HL* qui est placé dans *A*. L'adressage direct est utilisé pour l'instruction `out (UART3),a` par exemple, puisque nous plaçons l'octet contenu dans *A* à l'adresse du port désignée par `UART3`. Il en va de même pour `in a,(UART5)` nous permettant de faire l'opération inverse.

Après cette mise au point, finalement, il ne nous reste plus qu'à voir les sauts pour couvrir l'ensemble de ce code. Nous en avons deux ici, le plus simple étant le `jp main_loop` de fin, qui se contente de revenir au label en question. Personnellement, je préfère voir cela comme un changement du registre PC plutôt que comme un « saut », car c'est effectivement ce que fait cette ligne : copier l'adresse désignée par l'étiquette `main_loop` dans le pointeur d'instruction (PC) et poursuivre l'exécution à cet endroit.

`jp z,main_loop` est un saut conditionnel avec une mécanique assez simple. Dans le Z80 existe un registre un peu particulier appelé *F*, pour *flag* (drapeau). Ce registre n'est pas manipulable directement et 6 des 8 bits qui le composent changent en fonction de l'état du processeur à un moment donné. Ainsi, le bit 6 appelé *Z* est le *Zero Flag*, qui passe à 1 si le résultat d'une opération arithmétique ou logique donne `0x00` dans le registre *A*, si une comparaison entre le contenu du registre *A* et celui à une adresse mémoire pointée par *HL* sont identiques, ou encore si un bit spécifique est à 0 dans le registre spécifié.



Ceci est un Intel 8255 ou PPI pour Programmable Peripheral Interface. Initialement développé pour accompagner le i8080, ce composant fournit pas moins de 24 entrées/sorties parallèles ou, ce qu'on appelle aujourd'hui des GPIO. Nous aborderons le sujet prochainement, patience...

Et c'est précisément ainsi que nous testons la disponibilité de l'UART en émission. Nous chargeons dans *A* le contenu du registre LSR avec `in a,(UART5)`, puis nous testons le sixième bit (THRE) indiquant que le registre THR est vide avec `bit 5,a`. Si THRE est a 0, indiquant que THR n'est **pas** vide, alors le *flag* Z sera à 1 et notre instruction `jp z,main_loop` aura pour effet de mettre l'adresse de `main_loop` dans le pointeur d'instruction. Nous bouclons donc jusqu'à ce que le bit THRE de LSR soit à 1, indiquant que nous pouvons y placer l'octet à émettre. Chose que nous faisons immédiatement ensuite avant de revenir à `main_loop`.

4. ASSEMBLAGE, ÉCRITURE DE L'EEPROM ET EXÉCUTION

Ce code est placé dans un fichier `uarttest.s` et nous pouvons alors utiliser les outils SDCC pour produire notre binaire. Pour cela, nous commençons par assembler notre programme pour obtenir un fichier objet :

```
$ sdasz80 -plosgw uarttest.rel uarttest.s
```

Notez que le résultat, le fichier `uarttest.rel`, n'est pas binaire comme c'est le cas pour les `.o` produits par GCC ou Clang/LLVM, mais un fichier texte. Les différentes options utilisées permettent surtout de produire nombre de fichiers annexes pour avoir davantage d'information sur l'opération :

- `-o` : précise le nom du fichier objet `.rel` ;
- `-l` : crée un fichier listing `uarttest.lst` ;
- `-s` : crée un fichier de description des symboles `uarttest.sym` ;
- `-p` : désactive la pagination dans les listings ;
- `-w` : format large pour le listing de la table de symboles ;
- `-g` : les symboles indéfinis sont globaux.

Le fichier `uarttest.lst` est particulièrement intéressant puisqu'il vous permet de voir directement la correspondance entre votre code assembleur et sa traduction en binaire. `uarttest.sym` quant à lui vous permettra de connaître la taille de chaque symbole et celle des segments utilisés.

Une fois ce fichier objet obtenu, il est temps de passer à l'édition de liens avec :

```
$ sdcc --verbose -mz80 --code-loc 0x0000\
--data-loc 0x8000 --no-std-crt0 \
-o uarttest.ihx uarttest.rel

sdcc: Calling linker...
sdcc: sdldz80 -nf uarttest.lk
```

En réalité, c'est `sdcc` qui invoque le *linker* (`sdldz80`) pour nous après avoir généré son script (`uarttest.lk`). Ici, nous avons un simple code assembleur et n'avons pas besoin des routines standard (*crt0*) permettant de passer la main à la fonction `main()` d'un code C, d'où le `--no-std-crt0`. Mais nous précisons l'architecture (`-mz80`) ainsi que les adresses de départ pour le code (en ROM) et les données (en RAM). Nous verrons cela plus en détail dans le prochain article où la RAM entrera effectivement en jeu. Le fichier `uarttest.ihx` que nous obtenons en sortie est une représentation hexadécimale du code binaire, au format Intel HEX. Nous utilisons alors `sdobjcopy` pour traduire cela en binaire pour notre EEPROM en profitant de l'occasion pour ajouter du bourrage (*padding*) pour arriver aux 32 768 octets nécessaires à la programmation du Z8C256 :

```
$ sdobjcopy -Iihex -Obinary --gap-fill 0x00 \
  --pad-to 0x8000 uarttest.ihx uarttest_padded.bin
```

Le fichier `uarttest_padded.bin` contient notre programme à enregistrer en ROM et on retrouve la correspondance avec les octets du `uarttest.lst` :

```
$ hexdump uarttest_padded.bin
0000000 803e 03d3 0c3e 00d3 003e 01d3 0f3e 03d3
0000010 05db 6fcb 10ca 3e00 d341 c300 0010 0000
0000020 0000 0000 0000 0000 0000 0000 0000 0000
*
0008000
```

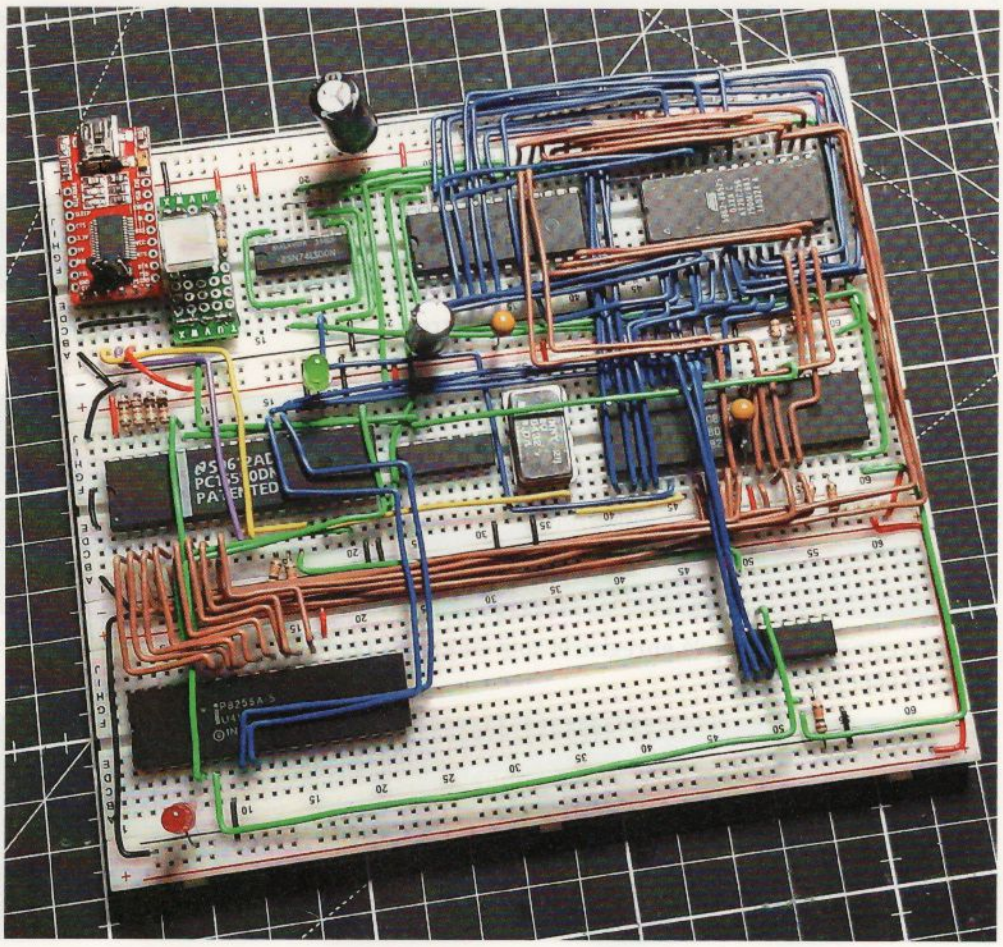
Il ne reste plus alors qu'à enregistrer tout cela avec, par exemple, `minipro` pour le programmeur TL866CS/TL866II+ :

```
$ minipro -p AT28C256 -w uarttest_padded.bin
Found TL866II+ 04.2.132 (0x284)
Erasing... 0.02Sec OK
Writing Code...  6.71Sec  OK
Reading Code...  0.48Sec  OK
Verification OK
```

On placera ensuite l'EEPROM sur la platine à essais et, si tout est correctement connecté, après mise sous tension et *reset*, une tripotée de « A » devrait arriver sur l'émulateur de terminal (Minicom, par exemple) configuré en 9600 8O2. Si ce n'est pas le cas, c'est que quelque chose ne fonctionne pas au niveau des branchements. Vérifiez chaque connexion, une à une, visuellement et avec un multimètre en testeur de continuité. Généralement, et je parle d'expérience, le premier assemblage n'est jamais parfait et une simple erreur d'inattention est suffisante à provoquer un problème empêchant le bon déroulement du programme.

5. LA SUITE AU PROCHAIN NUMÉRO

Ce premier volet logiciel est une simple mise en jambe visant à vérifier le fonctionnement de l'ordinateur et de son UART. Il sera encore question d'assembleur dans la prochaine partie, mais avec une optique totalement différente : préparer le terrain à l'exécution d'un code en C. Je l'ai rapidement évoqué ici, un code C ne s'exécute pas



Mon implémentation du projet sur platine à essais est sensiblement en avance sur le contenu des articles. Depuis la dernière fois, une platine supplémentaire a été ajoutée et les bus étendus jusqu'à celle-ci, où se trouve un second périphérique. Le composant DIP 16 en bas à droite est un 74LS138, un décodeur 3 vers 8 permettant d'activer soit l'UART soit le 8255.

aussi simplement que les quelques bouts d'assembleur que nous venons de voir. Il y a tout un travail à faire avant de pouvoir passer la main au `main()`, en particulier au niveau de la RAM et de l'initialisation des variables statiques. Ceci devrait nous emmener vers des considérations qui sont également valides lorsque vous développez pour Arduino, sur une Pi ou même un PC GNU/Linux ou autre.

D'ici là, et si votre montage fonctionne relativement bien du premier coup, n'hésitez pas à essayer deux ou trois choses en assembleur en vous assistant, par exemple, de la table des *opcodes* du Z80 disponibles en ligne sur le site de *Deep Toaster* **[5]** et, bien entendu, de la page dédiée sur *Rosetta Code* **[6]**. Attention cependant, tous les assembleurs Z80 n'utilisent pas la même syntaxe que celle de SDCC et un certain travail d'adaptation sera donc nécessaire (mais ceci est un excellent exercice pour mieux comprendre et apprendre). DB

RÉFÉRENCES

[1] https://www.zilog.com/docs/z80/um0080.pdf

[2] https://floooh.github.io/2021/12/06/z80-instruction-timing.html

[3] https://savannah.nongnu.org/projects/z80asm/

[4] https://www.scs.stanford.edu/10wi-cs140/pintos/specs/pc16550d.pdf

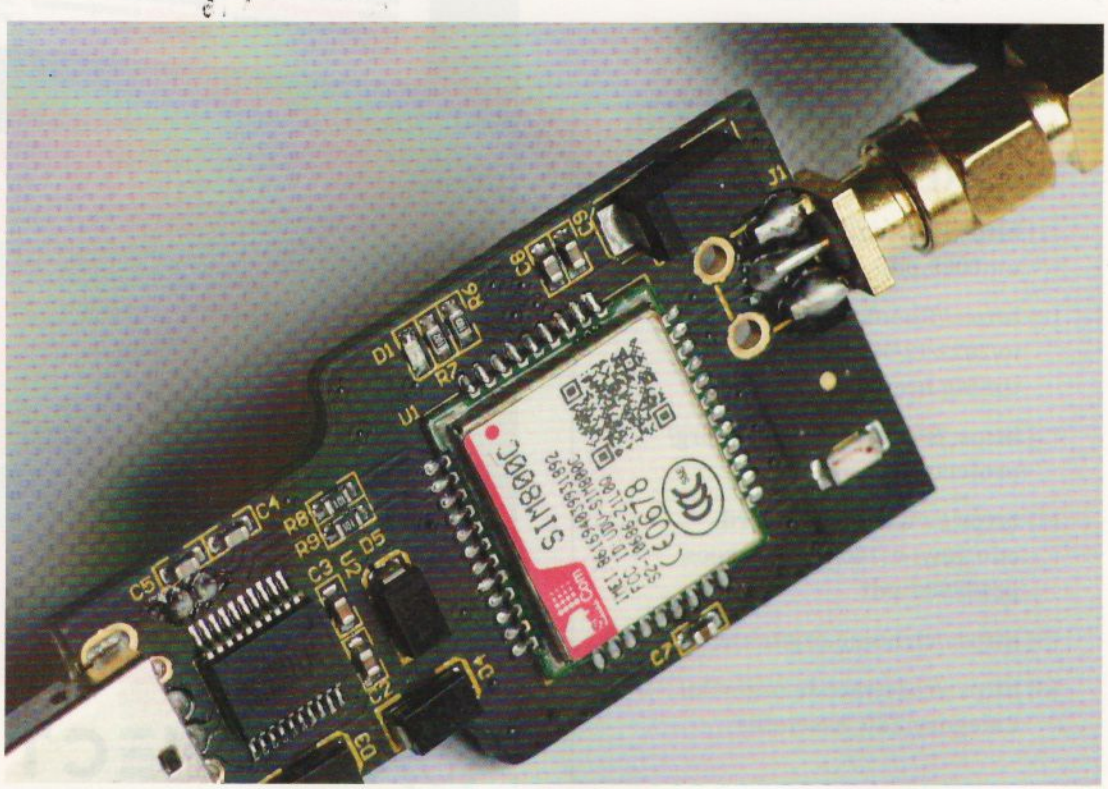
[5] https://clrhome.org/table/

[6] https://rosettacode.org/wiki/Category:Z80_Assembly

BOX AIRBNB, DOMOTIQUE AVEC DES SMS

Antoine Lucas
[Consultant freelance Antoine & Associés]

Vous prêtez votre appartement, mais lorsque le voyageur l'a quitté, vous constatez que la chasse d'eau fuyait avec un chauffage à fond. Nous allons programmer un Raspberry Pi pour éteindre l'eau et l'électricité à distance en fonction d'une messagerie SMS. Cerise sur le gâteau : la porte s'ouvrira si on se connecte au réseau Wi-Fi du Raspberry Pi.



– Box Airbnb, domotique avec des SMS –

Avec ce système, fini les problèmes de redirection de port et d'abonnement à un DynDNS ! Le numéro de téléphone est le seul point d'entrée pour la domotique. L'objectif est de faire des économies d'eau et d'énergie, avec un boîtier pilotable par SMS. Ce boîtier a deux fonctions : couper l'eau et le courant d'un appartement sur réception d'un SMS, et ouvrir la porte d'entrée sur sélection d'un réseau Wi-Fi et de son mot de passe. De plus, la connexion au réseau ne coûte qu'un abonnement téléphonique basique à 2 euros par mois.

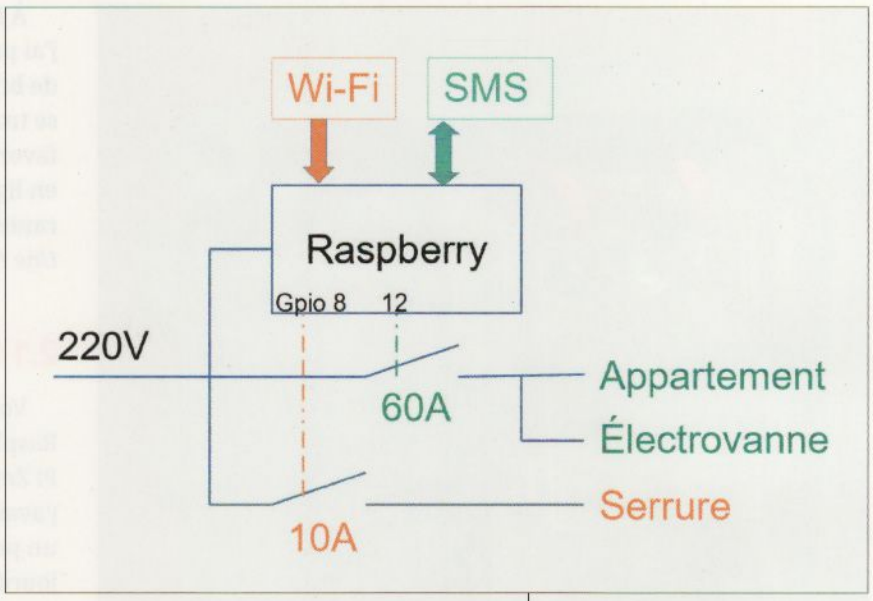


Figure 1 : Schéma des interactions du Raspberry Pi.

1. PRINCIPE

Le Raspberry Pi présente deux connexions réseau :

- une carte SIM pour l'envoi et la réception de SMS, à l'aide d'un dongle USB ;
- une carte Wi-Fi pour l'ouverture de la porte.

Le Raspberry Pi contrôle les deux relais suivants (interrupteurs) :

- l'alimentation électrique de toute la maison (ou au moins les équipements de chauffage) à l'exception du circuit électrique du Raspberry Pi lui-même ;
- une gâche électrique (verrou de l'appartement).

L'utilisateur enverra les messages SMS suivants :

- `OFF/ON` pour l'interrupteur général ;
- `password nouveauPassword` pour changer le mot de passe du Wi-Fi.

La connexion au réseau Wi-Fi permet d'ouvrir la porte.

2. MATÉRIEL

Je vais présenter la panoplie complète de ce projet qui peut être subdivisé en deux parties :

- ouverture de porte via un réseau Wi-Fi (composants nécessaires : Raspberry Pi, relais 10 A et serrure électronique) ;
- domotique via la messagerie SMS (composants nécessaires : Raspberry Pi, carte 2G, relais 60 A et éventuellement l'électrovanne).

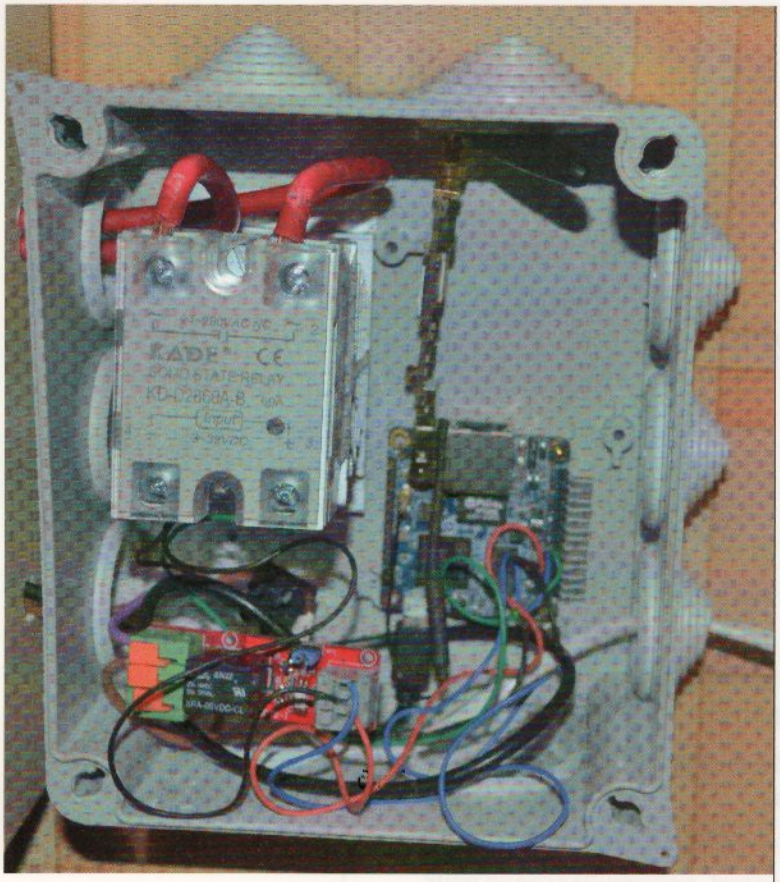
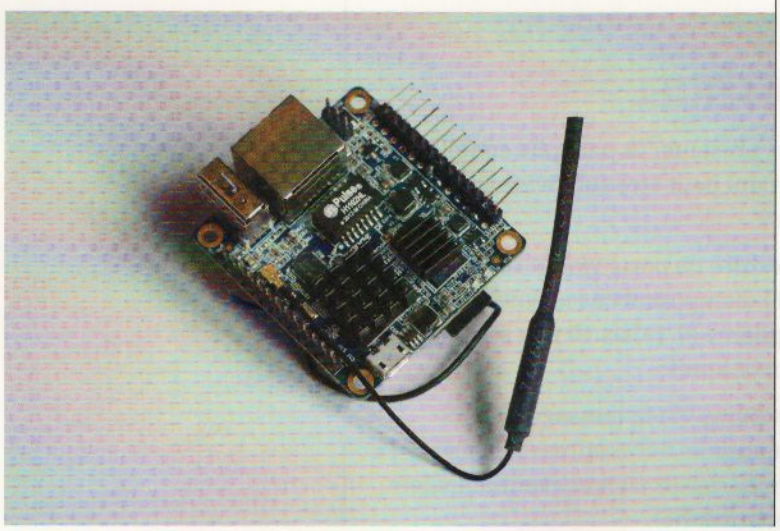


Figure 2 : L'ensemble de l'électronique.

Figure 3 : Orange Pi Zero.



À l'exception de la serrure que j'ai prise chez une grande enseigne de bricolage, tous les composants se trouvent chez mon fournisseur favori, un géant chinois du commerce en ligne dont le nom évoque les quarante voleurs du conte des *Mille et Une Nuits*.

2.1 Raspberry Pi

Vous aurez besoin d'une carte Raspberry Pi, et j'ai choisi l'Orange Pi Zero, avec 512 Mo de RAM que j'avais acheté avant la COVID à un prix bien plus attractif qu'aujourd'hui. N'importe quelle carte compatible avec GNU/Linux, disposant d'un port USB, de quelques GPIO programmables et du Wi-Fi conviendra. Prenez également une carte micro SD : je vous conseille de prendre une carte qui permet de nombreux cycles de lecture et d'écriture, je la choisis parmi les gammes « endurantes » adaptées aux *dashcams*.

Cette carte reste d'actualité, avec une consommation électrique aussi réduite que sa taille (un carré de 4,5 cm de côté). Aucune puissance CPU ou mémoire n'est requise pour ce projet.

2.2 Carte 2G GSM

À l'heure de la 5G, cela peut sembler anachronique de parler de SMS et de connexion GSM (2G), mais voici la carte que je vous propose, avec une puce SIM800C qui est reconnue par GNU/Linux sans aucune modification. C'est sur cette carte que vous brancherez votre carte SIM.

Son coût est légèrement inférieur à 10 euros. Une autre solution consiste à prendre une carte CPU qui intègre la connectivité 2G (ou 3G, 4G) comme certaines cartes Orange Pi « IoT ». Faites attention à ce que tous les *drivers* existent bien sous GNU/Linux, votre serviteur a été déçu par l'une d'elle, qui est toute option sur Android, mais pas sur GNU/Linux. Notez bien que ce n'est pas très durable, car les réseaux 2G et 3G vont s'éteindre entre 2025 et 2029.

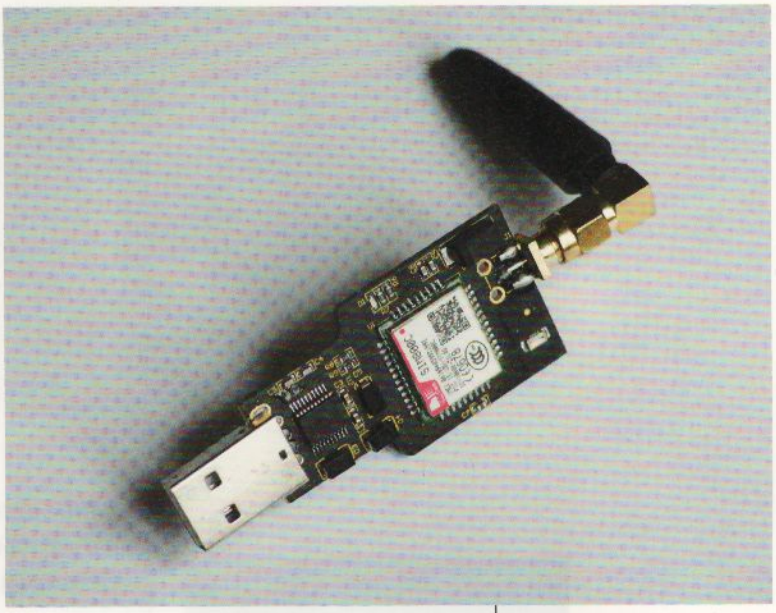


Figure 4 : Carte USB 2G GSM avec un emplacement pour une carte micro SIM.

2.3 Relais 10 A ou 20 A

Il s'agit des relais classiques que l'on utilise pour se familiariser à la domotique, idéal pour allumer une lampe ou la télévision. Nous l'utilisons pour ouvrir un circuit 12 V afin de contrôler la serrure. Un relais 10 A convient bien (souvent de petite taille et de couleur bleue). J'ai choisi cette version 20 A/14 VDC pour ses jolies couleurs vert et orange et aussi, car il y a un bornier bien pratique. Son prix est généralement de 2 ou 3 euros.

Le fonctionnement d'un relais classique est simple (en noir sur la figure 5) : on met du courant dans une bobine et cela actionne un interrupteur. Dans le cas de ces petits relais adaptés pour Arduino, il y a une troisième borne alimentée en +5 V et la commande en 3 V actionne un optocoupleur qui envoie le 5 V dans la bobine.

Figure 5 : Relais pour Arduino ou Raspberry. En noir, un relais électromécanique 5V classique.

2.4 Relais SSR 60 A

Il nous faut un relais pour couper le courant d'un appartement entier (220 V avec un fort ampérage). J'ai choisi un relais d'une capacité de 60 A pour du courant alternatif 220 V, avec une commande en courant continu de 0 à 10 V. Un petit appartement a un abonnement de 6 KVA, soit 45 A environ, il pourra être coupé par un relais de 60 ampères. Il est à noter que ces relais sont souvent vendus avec un radiateur passif que je vous recommande également d'acheter. Ces relais SSR pour *Solid State Relay* ou contacteur statique en français coûtent environ 10 euros.

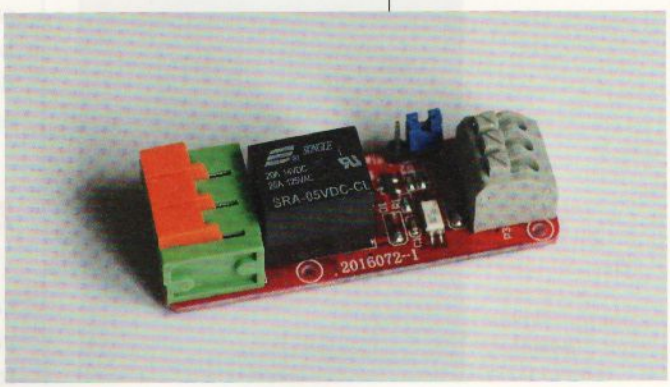


Figure 6 : Relais SSR, la commande se fait sur les bornes du bas 3 et 4 (0 ou 3 V continu, donné par le Raspberry Pi), et cela ferme le contact sur les bornes du haut (1 et 2) pour ce modèle qui est normalement ouvert.

La plupart des relais SSR sont « normalement ouverts », mais je vous recommande de fouiller un peu pour en avoir un « normalement fermé » (NC en anglais, pour *Normally Closed*). Cela permettra d'avoir le courant en cas de *bug* informatique : vous débranchez votre Raspberry Pi.

Là encore, le monde des plombiers et celui des électriciens ne s'entendent pas : une vanne ouverte laisse passer l'eau, mais un interrupteur ouvert coupe le courant électrique.

Une autre solution serait d'utiliser un contacteur industriel de 60 A, mais le prix est beaucoup plus élevé et il peut générer parfois un peu de bruit.

Faites-vous aider par un électricien professionnel avant de toucher au tableau électrique d'un logement.

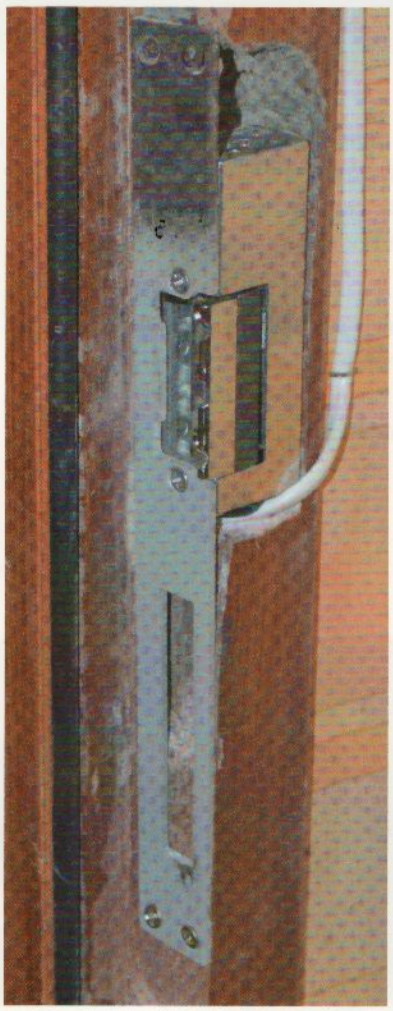


Figure 7 : La serrure électronique : je ne vous recommanderais pas mon serrurier...

2.5 Serrure électronique

Pour ouvrir la porte de l'appartement, nous avons besoin d'une composante mécanique. Pour une rénovation, les serrures en applique sont plus adaptées que la gâche installée chez moi en figure 7. Prévoyez un budget d'environ 15 euros.

Veillez à choisir une serrure « normalement fermée », ce qui signifie que lorsque le courant est coupé, la serrure reste bloquée. D'une manière générale, dans ce projet, veillez à prendre tous les composants « normalement fermés ». Laissez quand même la technologie du XXe siècle pour ouvrir : la clé depuis l'extérieur et la poignée pour sortir, ce sera apprécié en cas de panne électrique (Figure 8).

2.6 Électrovanne

L'électrovanne est utilisée pour couper l'eau de l'appartement lorsqu'il n'est pas occupé. Assurez-vous de choisir une électrovanne « normalement fermée », c'est-à-dire qu'il n'y a pas d'eau s'il n'y a

plus de courant électrique. Certains modèles fonctionnent avec une tension de 12 V (offrant une sécurité supplémentaire en cas de fuite d'eau), tandis que d'autres utilisent une tension de 230 V si vous souhaitez éviter l'utilisation d'un transformateur. Prévoyez également un budget d'environ 15 euros.

Pour la dimension de l'électrovanne, il faut se rappeler que les plombiers mesurent le diamètre des tuyaux en millimètres, mais les dimensions des raccords en pouces de l'unité impériale anglaise. Le plombier commande donc des raccords mâles 3/4 pouces pour des tuyaux de cuivre 16 mm, le mélange des deux unités sur le même objet ne le gêne même pas. C'est un peu comme si on mettait un lecteur CD 12 cm dans un emplacement 3 pouces, n'est-ce pas ? Il faut prendre la vanne qui a la même dimension que le raccord sur lequel vous la visserez. À priori, 1 pouce ou 3/4 de pouce pour l'arrivée générale d'un appartement, 1/2 pour des sections plus petites.

Pour l'outillage, oubliez les clés de 24 mm ou autre outillage un peu précis : le plombier utilise une paire de pinces multiprise

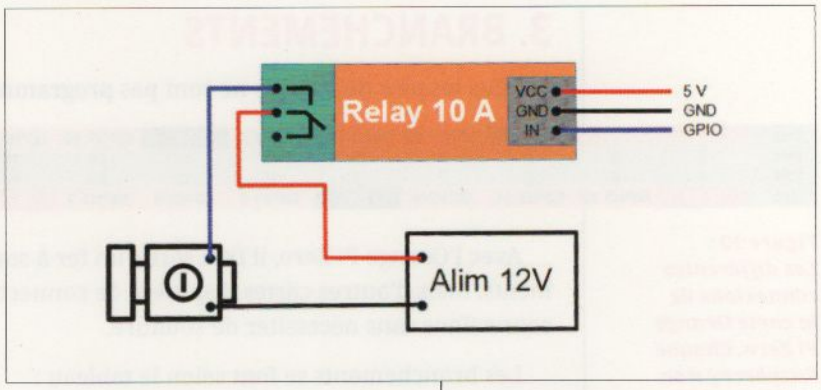


Figure 8 : Le câblage de la serrure. Lorsque le GPIO est sur 1 dans GNU/Linux, il vaut 3 V à l'entrée du relais et le courant passe vers la serrure.

Alligator qui font toutes les dimensions, même anglaises, en serrant bien fort.

Vous noterez que le bornier est à cosse (cosse clip pour les connaisseurs, pour la serrure aussi, c'est un connecteur à cosse, mais à œillet), cela s'achète dans les boutiques d'accessoires automobiles, c'est super ce projet : vous visitez plein de boutiques différentes ! C'est une source de *bug* supplémentaire, car si vous ne savez pas bien sertir les cosses comme moi, la vanne ou la serrure ne fonctionnera pas comme prévu.

Figure 9 : Une électrovanne 1/2 pouces. Il suffit de la brancher derrière un transformateur 12 V pour qu'elle s'ouvre avec le courant, et coupe l'arrivée d'eau quand il n'y a plus d'électricité.

3. BRANCHEMENTS

Tous les *pins* de la carte ne sont pas programmables, il faut suivre son schéma :

Use	5V	5V	0V	GPIO 198	GPIO 199	GPIO 7	0V	GPIO 19	GPIO 18	0V	GPIO 2	GPIO 13	GPIO 10
PIN	2	4	6	8	10	12	14	16	18	20	22	24	26
PIN	1	3	5	7	9	11	13	15	17	19	21	23	25
Use	3.3V	GPIO 12	GPIO 11	GPIO 6	0V	GPIO 1	GPIO 0	GPIO 3	3.3V	GPIO 15	GPIO 16	GPIO 14	0V

Figure 10 : Les différentes connexions de la carte Orange Pi Zero. Chaque Raspberry a sa table spécifique.

Avec l'Orange Pi Zero, il faut sortir un fer à souder pour faire les branchements, mais d'autres cartes disposent de connecteurs Dupont, ce qui facilite les connexions sans nécessiter de soudure.

Les branchements se font selon le tableau :

Pin Orange Pi	Destination
2 (5 V)	Relais 10 A pin VCC (+)
6 (GND)	Relais 10 A pin GND (-)
8 (GPIO 198)	Relais 10 A pin IN (S)
12 (GPIO 7)	Relais 60 A pin 3 (+)
14 (GND)	Relais 60 A pin 4 (-)

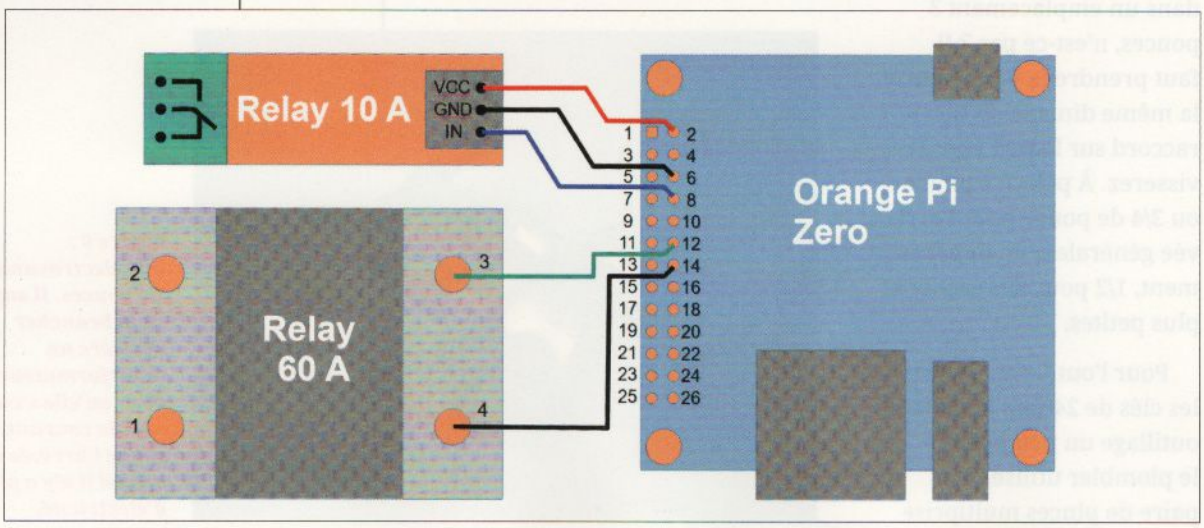
4. INSTALLATION

L'ensemble du code est disponible sur mon *repository* GitHub **[1]**.

4.1 Préparation du système

L'Orange Pi est livré sans système d'exploitation, vous devrez l'installer sur une carte micro SD qui fera office de disque dur du micro-ordinateur.

Figure 11 : Le schéma de câblage de l'électronique.



Il faut partir d'une image Orange Pi, vous la trouvez sur **http://www.orangepi.org** rubrique ***Service & Download*** **[2]**.

J'ai téléchargé l'image d'Ubuntu Bionic, pour l'installer sur la carte micro SD (en mode *admin*) :

```
1: tar xzvf Orangepizerolts_2.0.8_ubuntu_bionic_server_linux5.4.27.tar.gz
2: dd if=Orangepizerolts_2.0.8_ubuntu_bionic_server_linux5.4.27.img of=/dev/sdb
```

Avec `/dev/sdb` à ajuster en fonction de votre système, il s'agit du chemin vers la carte micro SD (vous la branchez sur un PC GNU/Linux avec un adaptateur USB ou un adaptateur de carte SD).

`dd` est une commande qui copie bloc à bloc, vous noterez que l'on ne copie pas dans `/dev/sdb1`, ce qui écraserait la première partition du deuxième disque de type « sd », mais sur le support complet (la table de partition est incluse dans la copie).

Une fois le support préparé, il faut le glisser dans l'emplacement adéquat de votre carte Orange Pi.

Vous pourrez alors démarrer la carte Orange Pi Zero, la relier à votre box internet par un câble réseau et vous y connecter par *ssh* en `root` avec le mot de passe `orangepi`. La récupération de l'adresse IP de l'Orange Pi se fait par exemple en se connectant sur la page de configuration de la box internet.

Une première étape consiste à installer l'utilitaire `gpio` permettant de modifier les *pins* de la carte Orange Pi avec un script shell :

```
1: git clone https://github.com/orangepi-xunlong/wiringOP
2: cd wiringOP
3: sed -i 's/orangepizero-lts/orangepizerolts/g' build
4: ./build
5: cp gpio/gpio /usr/bin/
```

Il faut prévoir un script au démarrage (sur le paramètre `@reboot` de la *crontab* par exemple, en *root* tapez `crontab -e` pour éditer la table) pour mettre les deux `gpio` concernés en mode `out` :

```
@reboot gpio -1 mode 8 out
@reboot gpio -1 mode 12 out
```

Comme pour tout utilitaire UNIX, la commande `gpio -help` ou `man gpio` vous donnera un premier aperçu de ses fonctions. Les principales :

- `gpio readall` vous affiche un tableau avec l'état complet de tous les GPIO ;
- `gpio mode xxx type` change le mode du GPIO numéro xxx. Il y a 7 types possibles : `in out pwm clock up down tri`. Nous utiliserons le mode out ;
- `gpio read xxx` lit la valeur du GPIO xxx ;
- `gpio write xxx value` change la valeur du GPIO xxx.

Nous utiliserons l'option `-l` pour spécifier le *pin* physique plutôt que le numéro de GPIO.

4.2 Connexion Wi-Fi

La première étape consiste à configurer la carte Wi-Fi en mode « point d'accès ». Selon la carte utilisée, l'interface réseau peut être nommée `wlan0`, `wlp1s0` ou autre. Sur la (très ancienne) image fournie par Orange Pi, c'est le nommage simple des interfaces réseau : `wlan/eth`. Mais sur un *kernel* plus récent, vous avez plein d'autres possibilités pour nommer les interfaces réseau **[3]**. Voici un exemple de commande pour mettre la carte Wi-Fi en mode point d'accès :

```
1: CON_NAME=hotspot
2: nmcli con add type wifi ifname wlan0 con-name $CON_NAME autoconnect yes
ssid Appartement_XXX
3: nmcli con modify $CON_NAME 802-11-wireless.mode ap 802-11-wireless.band
bg ipv4.method manual ipv4.address "192.168.10.1" wifi-sec.key-mgmt wpa-psk
wifi-sec.psk mypassword
4: nmcli con up $CON_NAME
```

`nmcli` est l'outil de ligne de commande du NetworkManager. Vous aurez à customiser `Appartement_XXX` et `mypassword` pour le nom du réseau et son mot de passe.

Pour recevoir une notification de connexion, vous pouvez utiliser une astuce en détournant l'utilisation principale du DHCP, car il permet d'appeler un script lorsqu'une demande de DHCP est reçue.

```
1: apt-get update
2: apt-get install isc-dhcp-server
```

Puis modifier le fichier `/etc/dhcp/dhcpd.conf` :

```
subnet 192.168.10.0 netmask 255.255.255.0 {
     range 192.168.10.10 192.168.10.100;
}
on commit {
 execute("/script/onDhcpRun.sh", "commit", "","");
}
```

Le script d'ouverture de la porte, on met le *pin* 8 à `1`, pour fermer le circuit contrôlé par le relais 10 A `/script/onDhcpRun.sh` :

```
gpio -1 write 8 1
sleep 10
gpio -1 write 8 0
```

Cela ouvre le loquet pendant 10 secondes.

Il vous suffit d'autoriser l'exécution de ce script et de redémarrer le service pour que les changements soient pris en compte :

```
1: chmod 'a+x' /script/onDhcpRun.sh
2: service isc-dhcp-server restart
```

4.3 Domotique par SMS

Pour la domotique par SMS, nous utiliserons l'utilitaire *Gammu*, qui permet d'envoyer et de recevoir des SMS à l'aide de scripts shell. Vous pouvez l'installer en utilisant la commande suivante :

```
1: apt-get install gammu gammu-smsd
```

Il faut configurer le serveur Gammu en modifiant le fichier `/etc/gammu-smsdrc` :

- dans la section `[gammu]`, il faut indiquer le port de la carte GSM, avec celle que je vous ai proposée :

```
port = /dev/ttyUSB0
```

- par défaut, Gammu prend 0000 comme code PIN, s'il en faut un autre (1234 chez Free je crois, ou si vous l'avez modifié pour rajouter quelques sources de *bugs* farfelus), il faut l'ajouter dans la section `[smsd]` :

```
PIN = 1234
```

Avec ces modifications, vous pouvez déjà envoyer des messages avec la commande :

```
gammu-smsd-inject TEXT +33612345678 -text "Bonjour lecteurs de Hackable"
```

Vous l'avez compris, à la place de +33612345678, vous mettez le numéro de téléphone du destinataire.

À chaque réception de message, Gammu peut lancer un script qu'il faut indiquer dans ce même fichier de configuration. Il faut ajouter une ligne dans la section `[smsd]` :

```
RunOnReceive=/script/parseSms.sh
```

Pour traiter chaque message, j'ai utilisé un script shell, car même si la syntaxe écorche parfois les yeux, la succession de commandes UNIX est bien pratique.

Pour vous aider dans la lecture de ce script, il faut savoir que Gammu lit automatiquement les SMS et les ajoute dans un répertoire `/var/spool/gammu/inbox` avec les informations de date, d'heure et du numéro de téléphone dans le nom du fichier, quelque chose comme `IN20230610_160718_00_+33612345678_00.txt`. Le contenu du message est dans le fichier (sans aucune autre informations ni *headers*). Le fonctionnement du script consiste à se déplacer dans le répertoire *inbox* de Gammu, qui ne contient qu'un fichier, vérifier les *patterns* « `on`, `off`, `password` », puis supprimer ce fichier. Pour aller plus loin :

- la commande `awk '{print $2}' fichier(s)` écrit le deuxième mot de chaque ligne des fichiers ;
- la commande `ls | awk -F"_" '{print $4}'` liste les fichiers (un seul) du répertoire courant, sépare chaque ligne avec le séparateur `_` et affiche le 4e mot, dans notre cas ce devrait être le numéro de téléphone ;

- la commande `var=`command`` met dans la variable `var` la sortie standard de l'exécution de la commande ;
- la commande `grep -i ON *` affiche les lignes qui contienne `ON` dans le texte, sans prendre en compte les majuscules ou minuscules. Cette commande renvoie un entier qui est 0 s'il y a eu au moins une ligne affichée (le `return(0)` du *main* ou le `exit(0)` d'un programme C), 1 sinon. Ce code de retour peut être récupéré dans la variable bash `$?` (ça ne s'invente pas) ;
- la commande `[ $var == 0 ]` avec les crochets est la commande test du bash (`man test` pour plus d'informations). Ici, elle vérifie que la variable `var` a pour valeur 0 et renvoie un booléen utilisé par la boucle `if`.

Le script `/script/parseSms.sh` s'écrit de cette façon :

```
#!/bin/bash
## lecture d'un message
cd /var/spool/gammu/inbox

## On récupère le numéro de téléphone par le nom du fichier :
## IN20230610_160718_00_+33612345678_00.txt
tel=`ls | awk -F"_" '{print $4}' `

command="Inconnu"

grep -i ON *
retVal=$?

if [ $retVal == 0 ]
then
    command="ON"
    gpio -1 write 12 1
fi

grep -i OFF *
retVal=$?

if [ $retVal == 0 ]
then
    command="OFF"
    gpio -1 write 12 0
fi

grep -i PASSWORD *
retVal=$?
```

```
if [ $retVal == 0 ]
then
    command="PASSWORD"
    PASSWD=`awk '{print $2}' * `
    CON_NAME=hotspot

    nmcli con down $CON_NAME
    nmcli con modify $CON_NAME wifi-sec.psk "${PASSWD}"
    nmcli con up $CON_NAME
fi

## suppression du message
rm *

## on envoie le SMS
msg="Receive message ${command}"
gammu-smsd-inject TEXT $tel -text "$msg"
```

Il vous suffit d'autoriser l'exécution de ce script et de redémarrer le service pour que les changements soient pris en compte :

```
1: chmod 'a+x' /script/parseSms.sh
2: service gammu-smsd restart
```

CONCLUSION

La complexité de ce projet n'est sans doute pas – vous l'avez remarqué – sur la programmation ou l'informatique, mais sur le fait qu'il faut à la fois être électricien, serrurier, plombier et soudeur de carte électronique. En cas de pépin, on peut avoir le combo complet « *chérie, j'ai loupé ma plomberie : on n'a plus d'eau, plus d'électricité et la porte d'entrée ne ferme plus, mais mon Raspberry Pi marche nickel* ». Entraînez-vous avant sur chaque sous-partie, ou faites-vous aider avant de faire le projet dans son ensemble. Si vous utilisez les mêmes composants que moi, vous aurez peut-être le *bug* « perte de la connexion GSM au bout d'une semaine, même avec un redémarrage du Raspberry Pi ». Je n'ai pas encore trouvé l'origine : *bug* informatique, hardware, transformateur insuffisant, perturbations sur le réseau électrique... tout est possible.

Pour réaliser ce projet, il est tout à fait possible d'utiliser une électronique plus simple, je pense au microcontrôleur avec une puce ESP32, qui a une consommation électrique réduite et un coût d'achat inférieur. En effet, les cartes à microcontrôleurs ESP32 comme celle en figure 12 intègrent le Wi-Fi et la connectivité GSM, et peuvent être une alternative intéressante pour ce projet. Ces cartes sont disponibles à un prix abordable, généralement moins de 20 euros. La partie « serrure Wi-Fi » se fait très facilement avec un NodeMCU à 4 euros.

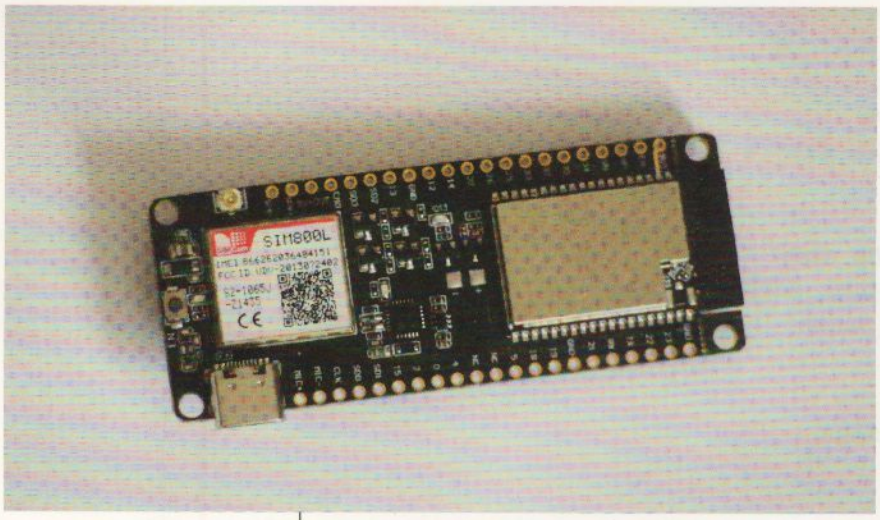


Figure 12 : Une carte ESP32 avec un module SIM800L, alternative à l'Orange Pi et sa carte 2G.

Un article ultérieur pourra se pencher sur cette approche et ses avantages.

Pour étendre les fonctionnalités et avoir une domotique plus complète, il n'est pas nécessaire de réinventer la roue. HomeAssistant est une plateforme qui offre une large gamme de fonctionnalités pour la domotique. Cependant, il peut être difficile d'accéder à distance à HomeAssistant avec une clé 3G en raison de certaines limitations telles que l'absence de redirection de port en IPv4 et la disponibilité rare des services de DynDNS en IPv6. Le plus facile est de passer par l'infrastructure d'un tiers (souvent payant) qui fournit un *cloud* ou un VPN.

Une solution possible pour contourner ces limitations consiste à envoyer l'adresse IPv6 par SMS et à modifier le client Android de HomeAssistant afin qu'il référence le numéro de téléphone de la box domotique plutôt qu'un nom de domaine : un DNS par SMS. Cela permettrait d'établir une connexion directe à la box domotique via le réseau mobile, sans dépendre d'une infrastructure externe. **AL**

RÉFÉRENCES

[1] GitHub, projet Box Airbnb, **https://github.com/antoinelucas64/BoxAirbnb**

[2] Orange Pi, téléchargement de l'image Orange Pi Zero, **http://www.orangepi.org/html/hardWare/computerAndMicrocontrollers/service-and-support/Orange-Pi-Zero.html**

[3] Freedesktop, Predictable Network Interface Names, **https://www.freedesktop.org/wiki/Software/systemd/PredictableNetworkInterfaceNames**

JOUONS AUX LEGO... AVEC DES TAGS NFC

Denis Bodor

LEGO Dimensions est un jeu vidéo sorti en 2015 et reposant sur un concept original, également utilisé par d'autres éditeurs à la même époque, comme Nintendo avec Amiibo ou Disney Interactive Studios avec Disney Infinity. L'idée de base consiste à utiliser des objets ou figurines équipés d'un tag NFC et permettant des interactions entre le jeu vidéo et le monde physique/réel. Ce type de jeux semble aujourd'hui passé de mode, mais le matériel reste très intéressant et surtout, réutilisable dans un autre contexte.

– Jouons aux LEGO... avec des tags NFC –

Avant toute chose, précisons que l'objectif n'est aucunement ici de chercher à profiter illégalement du jeu ou de ses extensions. Le lecteur souhaitant se consacrer à ce genre d'activités pourra trouver sans peine des outils et même des applications Android, diffusées sous le manteau ou ouvertement. Il existe même des dizaines de vidéos sur YouTube expliquant comment procéder sans avoir la moindre approche technique. Non, ce qui nous occupe ici est de comprendre comment a été conçu le produit et comment pouvoir éventuellement s'en resservir pour d'autres usages.

1. LEGO DIMENSIONS

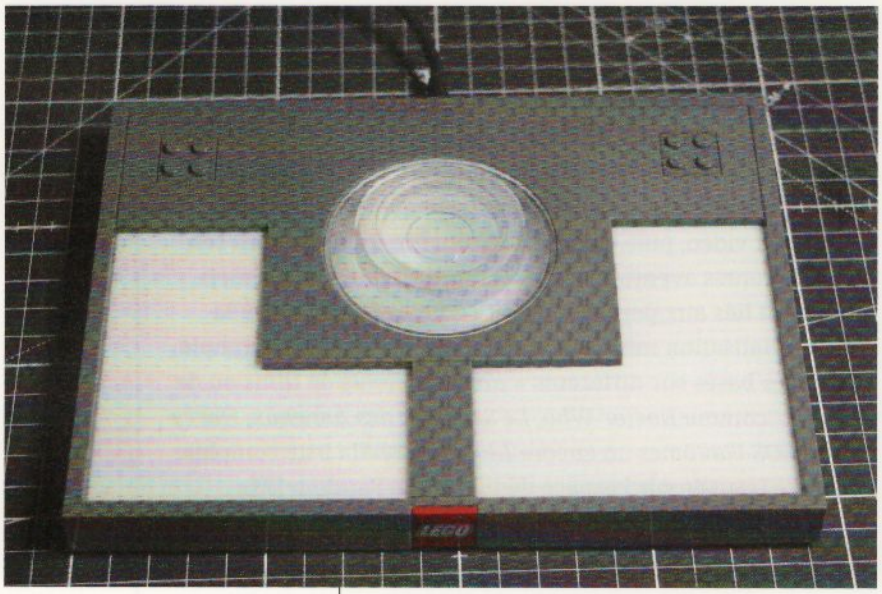
LEGO Dimensions est, ou « était » puisqu'il n'y a plus aucune nouveauté ou extension depuis 2017 **[1]**, un jeu vidéo de *Warner Bros. Games*, pour Sony PS3/PS4, Microsoft XBox One/360 et Nintendo Wii U. La version de base du jeu, ou *Starter Pack*, se compose du jeu lui-même (CD/DVD), une base *LEGO Toy Pad* (ou « portail » dans le jeu) se connectant en USB à la console, de 4 « objets » NFC (3 *minifigures*, Batman, Gandalf et Cool-Tag, plus 1 véhicule, la Batmobile), et d'un ensemble de briques LEGO pour construire physiquement le portail et les autres éléments.

Le principe du jeu est relativement simple puisque les objets posés sur le Toy Pad apparaissent automatiquement dans le jeu vidéo, puis en disparaissent lorsqu'ils en sont retirés. Différentes aventures et contenus ludiques complémentaires sont liés aux personnages et l'éditeur a fait suite à la commercialisation initiale en proposant des « packs » complémentaires basés sur différents « univers » issus de films ou de séries TV comme *Doctor Who*, *Le Seigneur des Anneaux*, *Harry Potter*, *SOS Fantômes* ou encore *Les Simpson* (la liste complète peut être trouvée sur l'espace dédié du site Fandom **[2]**).

D'un point de vue technique, le Toy Pad se comporte comme un lecteur NFC disposant de 3 zones distinctes. Une au centre et deux autres, sur chaque côté, permettant de placer à chaque fois jusqu'à 3 minifigures ou véhicules. Les trois zones sont rétroéclairées à l'aide de LED RGB permettant au jeu de signifier au joueur qu'il doit placer un objet à un emplacement particulier. De plus, les opérations NFC ne se limitent pas à la lecture des *tags* puisqu'il est possible, dans le jeu, de faire évoluer les véhicules (ou gadgets) et de les décliner en trois versions ou *rebuilds*. À chaque fois, le joueur reconstruit le modèle puis le place sur le périphérique pour être reprogrammé.

Les minifigures ou véhicules sont des supports en plastique contenant un *tag* NFC NTAG213 compatible NFC Forum Type 2 et permettant d'y enficher soit un personnage LEGO, soit un véhicule ou gadget qu'on aura construit avec les briques fournies. Les *tags* des minifigures se distinguent de ceux des objets par le fait qu'ils peuvent être mis à jour par le Toy Pad pour refléter l'évolution dans le jeu. Encore non utilisés, ils sont donc « vierges » ou du moins initialisés avec un contenu neutre et générique, mais il est important de remarquer que ceux-ci ne pourront pas être effacés pour être réutilisés par ailleurs. La structure même des NTAG213 (tout comme des NTAG215 et NTAG216) propose un mécanisme de verrouillage interdisant l'écriture sur tout ou partie du *tag* et celui-ci est activé ici par défaut.

Notez qu'il n'est, bien entendu, pas possible d'utiliser des NTAG213 achetés par ailleurs et de les « formater » pour qu'ils fonctionnent comme ceux de l'éditeur du jeu (avec le



Le Toy Pad est censé être décoré d'un ensemble de briques LEGO et sa construction fait partie du jeu lui-même. Cependant, même en dehors de tout hack et pour une utilisation « normale » avec une console, la déclinaison « sobre » est beaucoup plus pratique.

Toy Pad, du moins). Ceci n'est absolument pas une limitation physique ou technique, les *tags* sont parfaitement standard, mais un mécanisme logiciel propre au jeu. Ceci signifie également qu'il n'est pas possible de faire fonctionner le Toy Pad comme un lecteur NFC standard et donc de lire d'autres *tags*, standard ou non avec les outils courants (comme ceux reposant sur la LibNFC). Une connexion à un ordinateur fera apparaître le matériel comme un périphérique USB HID ne disposant d'aucun pilote dans les systèmes d'exploitation courants, si ce n'est le support HID de base.

2. LES DIFFÉRENTES FACETTES DU RECYCLAGE

Le *Starter Pack* LEGO Dimensions n'est plus commercialisé depuis quelque temps déjà, pas plus que les extensions qui ont suivi. Il reste cependant possible de trouver assez facilement tout cela sur eBay, neuf comme d'occasion. Les prix sont très variables en fonction de ce qui compose les annonces et un aspect « collectionneur » commence à se faire jour. On notera au passage que certaines personnes revendent le jeu seul (sans Toy Pad ou figurines) sans même savoir qu'il sera parfaitement inutilisable pour l'acheteur. D'autres vendent uniquement le Toy Pad, des *tags* sans personnage ou briques, ou les personnages du *Starter Pack* sans *tag* et/ou sans jeu. C'est un peu la jungle, mais en cherchant un peu, on arrive à mettre la main sur un ensemble Toy Pad plus quelques *tags* pour moins de 20 €.

Que peut-on faire avec ce matériel clairement passé de mode aujourd'hui ? Jouer est une option, à condition d'avoir tous les éléments à disposition (Pad + *tags* + DVD). Les consoles concernées sont encore d'usage (même si en fin de vie comme les PS3) et le caractère ludique n'a pas pris une ride (l'avantage des jeux vidéo LEGO). Tout simplement jouer reste donc une option parfaitement valable.

Côté bidouille en revanche, le Toy Pad est intimement lié aux *tags* d'origine et ne sera que d'un usage limité sans en avoir sous la main ou arriver à composer ses propres *tags* compatibles, sans pour

autant dériver vers le piratage. Le cahier des charges est donc clairement borné :

- comprendre comment communique le Toy Pad, comment est structuré le contenu des *tags* et la nature de la communication entre Toy Pad et *tags* ;
- pouvoir utiliser le Toy Pad connecté à un ordinateur via USB en faisant fonctionner un code « maison » ;
- utiliser les *tags* existants avec le Toy Pad mais sans lien avec le jeu ;
- créer des *tags* « hors collection » susceptibles d'être utilisés avec notre code « maison » mais parfaitement **incompatibles** avec le jeu.

Peu après la sortie du jeu, un certain nombre de passionnés, dont votre dévoué serviteur, se sont penchés sur le sujet, cherchant à comprendre le fonctionnement de l'ensemble. C'était il y a longtemps et l'on retrouve aujourd'hui différents dépôts GitHub, relativement anciens, proposant différentes implémentations ou tentatives purement autodidactiques plus ou moins abouties. Parmi les noms et pseudos, on retrouve, sans surprise, une bonne partie de la communauté gravitant, de près ou de loin, autour du Proxmark3 et en particulier du *fork* d'Iceman **[3]**. Avec le recul et en reprenant le sujet en main aujourd'hui, une grande partie du flou technique de l'époque a été dissipée et les zones d'ombre qui paraissait impénétrables à l'époque sont totalement documentées grâce au talent de ces personnes très prolifiques, pour certaines anonymes. Tout ceci pour dire que ce qui va suivre n'est aucunement un travail totalement personnel, et que même si j'ai apporté de l'eau au moulin fut un temps, je ne saurais être crédité d'une quelconque découverte décisive.

L'application finale, une fois le matériel maîtrisé, sera laissée à la discrétion du lecteur. Il est parfaitement envisageable de mettre en œuvre le Toy Pad et ses *tags* pour créer un nouveau jeu totalement différent de l'original, concevoir une interface multimédia (lecture audio), gérer sa domotique ou pourquoi pas, fabriquer une sorte de *Stream Deck* LEGO compatible avec OBS.

Mais pour arriver à accomplir de telles choses, encore faut-il savoir exactement ce qui se passe avec le matériel et comment fonctionnent les protections mises en place. Plusieurs aspects doivent être abordés :

- la communication du Toy Pad avec le PC pour contrôler les LED et obtenir des informations sur les *tags* placés ou retirés ;
- le contenu des *tags* qui, en partie, ne sont pas lisibles sans authentification préalable ;
- et la communication des *tags* avec le Toy Pad, puisque ce dernier se charge de l'authentification.

Les deux derniers points sont liés, puisqu'une authentification est en place pour les *tags*, mais celle-ci est interne à la communication entre le Pad et les *tags*. Si nous voulons pouvoir créer des *tags* lisibles par le Pad, il faut que cette authentification fonctionne afin de pouvoir la reproduire et la configurer pour des *tags* neufs. En effet, les minifigures d'occasion coûtent au minimum 2 € et sont partiellement verrouillées en écriture (même en cas d'authentification réussie), alors qu'un rouleau de 100 étiquettes autocollantes rondes NTAG213 se trouve entre 15 € et 30 € (AliExpress, Amazon, ShopNFC, etc.).

Nous avons beaucoup de choses à voir, car l'ensemble de l'architecture est relativement complexe par nature. Je ne m'attarderai donc pas outre mesure sur des points

de détail comme la structure standard d'un code en C, le `Makefile`, l'installation des bibliothèques sur le système, leur initialisation, la gestion des signaux (SIGINT et SIGTERM) ou encore une partie des fonctions utilitaires permettant de convertir les données entre différents formats (d'un tableau de 4 octets vers `uint32_t` en fonction de l'*endianness*). Il est beaucoup plus instructif, dans l'espace imparti pour l'article, de traiter des aspects cryptographiques et des logiques utilisées, qui sont particulièrement intéressants.

3. PARLER AU TOY PAD

Comme dit précédemment, le Toy Pad est un matériel USB (`0e6f:0241`) qui se présente comme un périphérique HID (*Human Interface Device*). Cette classe de périphériques USB est généralement celle des claviers, souris et contrôleurs de jeu, mais également d'un ensemble de gadgets et autre périphériques comme des onduleurs ou des clés de sécurité U2F/FIDO2 (type YubiKey). L'avantage de l'USB HID dans ce dernier cas est évident, plutôt que de devoir développer le code permettant de communiquer au plus bas niveau (USB), on bénéficie d'une interface standard abstraite utilisant un format de données standardisé, mais aussi d'une description de ce format pour chaque périphérique. Lorsqu'un périphérique HID est connecté à un hôte, il met à disposition une « description » qui détaille qui il est, ce qu'il est capable de faire et comment il est possible de communiquer avec lui. Ce *HID report descriptor* fournit donc toutes les informations nécessaires et facilite l'implémentation, il suffit d'envoyer les « rapports HID » dans le format décrit et le périphérique répondra avec des « rapports HID » formatés de la même manière.

La gestion de la communication USB HID est dépendante du système et des pilotes que vous utilisez. Ainsi, un code développé pour le support HID de macOS ne fonctionnera pas sous GNU/Linux et inversement. Il en va de même pour Windows, les héritiers de BSD ou tout autre système prenant en charge ce protocole. Sous GNU/Linux, nous pouvons utiliser *hiddev* ou *hidraw* selon le niveau de gestion souhaité de la part du noyau, mais le code en question ne sera pas portable sous, par exemple, FreeBSD qui utilise une API totalement différente (et relativement plus complexe, je sais, j'ai essayé).

Le jeu, lors de sa commercialisation initiale, était livré avec ces trois personnages ou « minifigures » issues d'univers totalement différents. C'était d'ailleurs la base même du scénario : des univers ou dimensions qui entrent en collision.

Heureusement, pour faciliter les choses et rendre les développements facilement transposables d'un système à un autre, il existe une bibliothèque offrant un niveau d'abstraction très sympathique : HIDAPI **[4]**. Celle-ci est généralement disponible avec toutes les distributions GNU/Linux, mais également pour Windows, FreeBSD et macOS (et OpenBSD + NetBSD, même si le site officiel n'en parle pas). Sous GNU/Linux, cette bibliothèque peut reposer sur la LibUSB ou sur le *hidraw* du noyau (`pkg-config` propose les deux déclinaisons).

La bibliothèque HIDAPI ne vous dispense cependant pas totalement de comprendre comment fonctionne la communication HID, mais rend tout cela plus simple. Pour communiquer avec un périphérique HID, on utilise un rapport d'un certain format, identifié par un numéro, indiquant une taille pour les données et les données elles-mêmes. Obtenir la description n'est pas difficile, il suffit d'utiliser l'outil dédié du système :

```
$ sudo usbhid-dump
002:017:000:DESCRIPTOR         1687792273.050298
06 00 FF 09 01 A1 01 19 01 29 20 15 00 26 FF 00
75 08 95 20 81 00 19 01 29 20 91 00 C0
```

Heu... OK. Et maintenant ? Maintenant, il faut décoder tout cela et vous avez le choix entre l'outil officiel de l'USB Forum qui date de Mathusalem **[5]**, un outil quelconque trouvé sur GitHub ou, tout simplement, la fantastique page **[6]** proposée par Frank Zhao sur son site **https://eleccelerator.com/** et qui nous dit :

```
0x06, 0x00, 0xFF,  // Usage Page (Vendor Defined 0xFF00)
0x09, 0x01,        // Usage (0x01)
0xA1, 0x01,        // Collection (Application)
0x19, 0x01,        //   Usage Minimum (0x01)
0x29, 0x20,        //   Usage Maximum (0x20)
0x15, 0x00,        //   Logical Minimum (0)
0x26, 0xFF, 0x00,  //   Logical Maximum (255)
0x75, 0x08,        //   Report Size (8)
0x95, 0x20,        //   Report Count (32)
0x81, 0x00,        //   Input (Data,Array,Abs,No Wrap,
                   //   Linear,Preferred State,No Null Position)
0x19, 0x01,        //   Usage Minimum (0x01)
0x29, 0x20,        //   Usage Maximum (0x20)
0x91, 0x00,        //   Output (Data,Array,Abs,No Wrap,Linear,
                   //   Preferred State,No Null Position,Non-volatile)
0xC0,              // End Collection
```

Le périphérique communique avec un protocole non standard (comprendre « pas comme une souris ou un clavier générique ») et avec des messages de 32 fois 8 bits. Fait confirmé par la littérature et le code des différentes personnes ayant exploré le Toy Pad. La page GitHub de HIDAPI présente un exemple simple permettant de communiquer avec un périphérique USB HID via une paire de fonctions après avoir initialisé la bibliothèque (`hid_init()`) et ouvert la connexion en fournissant les ID USB du périphérique concerné (`hid_open(0x0e6f, 0x0241, NULL)`). Ceci sera la base de notre implémentation.

Nous savons déjà que les messages font 32 octets, mais s'ajoute à cela une spécificité de HIDAPI : le premier octet d'un message envoyé spécifie le numéro (ou ID) du rapport (ici, `0x00`). Nous avons donc en réalité 33 octets à manipuler lors d'une communication hôte vers périphérique. À cela s'ajoute une caractéristique propre au Toy Pad (comme tout ce qui se trouve dans les données elles-mêmes) : les messages sont numérotés et cette valeur est présente à la position 3 des données. Il nous faut donc maintenir en permanence un compteur et l'incrémenter à chaque envoi.

La structure d'un message, en faisant abstraction de l'ajout HIDAPI pour le numéro de rapport, ressemble à ceci :

```
octet 0 : valeur magique (0x55)
octet 1 : taille des informations embarquées (payload)
octet 2 : commande
octet 3 : compteur de messages
octets 4...x : informations (payload)
octet : somme de contrôle
octets : bourrage (0x00) pour arriver à 32 octets
```

La somme de contrôle est relativement simple à calculer puisqu'il s'agit littéralement de la somme de toutes les valeurs du *payload* (ou « charge utile »), c'est-à-dire le message sans les 4 premiers octets qui constituent l'entête, à chaque fois réduite à 8 bits. Sur la base de ces informations, nous pouvons écrire une fonction intermédiaire procédant aux calculs et ajustements à notre place, avant d'envoyer le tout au périphérique avec la fonction `hid_write()` de HIDAPI :

```
void send_message(hid_device *handle,
   unsigned char id, unsigned char *buf, size_t buflen)
{
    int checksum=0;
    unsigned char *sendbuf;

    // numéro message
    buf[3] = messagenumber;

    // calcul checksum
    for(i=0; i<2+buf[1]; i++) {
        checksum+=buf[i];
        checksum = checksum & 0xff;
    }
    buf[2+buf[1]]=checksum;

    // allocation de 32 octet + 1 pour l'ID
    if ((sendbuf = malloc(buflen+1)) == NULL)
        err(EXIT_FAILURE, "send_message() malloc Error");
```

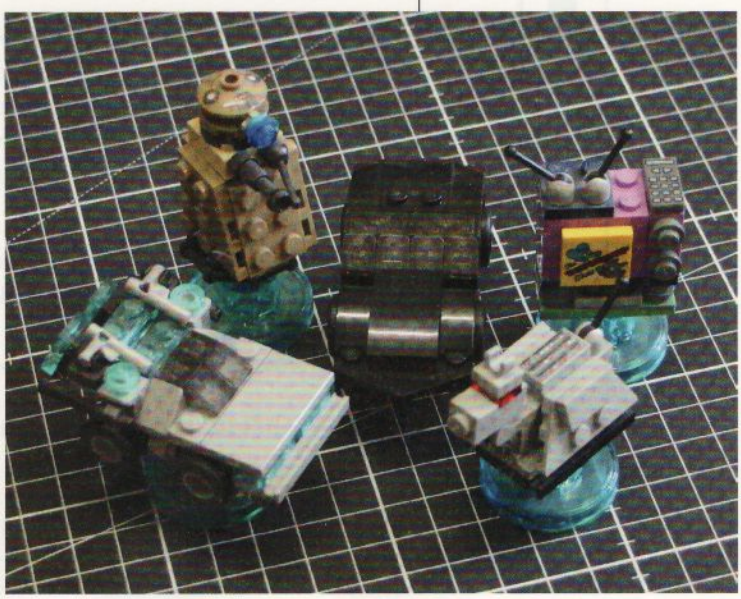
```
    memcpy(sendbuf+1, buf, buflen);
    // Le premier octet est l'ID du rapport
    sendbuf[0] = id;

    if (hid_write(handle, buf, buflen+1) != buflen+1)
        warnx("hid_write() error");

    free(sendbuf);

    messagenumber++;
}
```

À présent que nous disposons d'une fonction prenant en charge les spécificités du Toy Pad, nous pouvons nous pencher sur l'initialisation de la communication. En effet, celui-ci ne répondra à nos demandes et ne fournira d'informations que s'il est correctement initialisé avec un message particulier. Le message en question est :

```
unsigned char initmsg[] =
{ 0x55, 0x0F, 0xB0, 0x01, 0x28, 0x63, 0x29, 0x20,
   0x4C, 0x45, 0x47, 0x4F, 0x20, 0x32, 0x30, 0x31,
   0x34, 0xF7, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
   0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };
```

On y retrouve les différents éléments de structure que nous venons de voir et il est assez amusant de constater que le *payload* d'initialisation est en réalité la simple chaîne de caractère « © *LEGO 2014* ». Ceci fait, le protocole tel qu'utilisé entre une console de jeu et le périphérique poursuit normalement par une configuration du générateur de nombres pseudo-aléatoires (PRNG) suivi d'une demande d'authentification utilisant cette valeur de configuration et un chiffrement TEA (*Tiny Encryption Algorithm*). L'objectif de l'opération est, pour le jeu, de s'assurer que le Toy Pad est celui qu'il dit être. On remarquera ici plusieurs choses. Premièrement, cette authentification ne nous est pas nécessaire, l'ensemble fonctionne parfaitement en sautant cette étape. Ensuite, on peut être surpris que ce type de mécanisme soit en place, car, comme nous le verrons plus loin, le chiffrement est utilisé par ailleurs lors des communications, avec la même clé de chiffrement et le même algorithme. Ceci semble donc fortement redondant.

Les véhicules ou gadgets ne sont pas des personnages jouables dans l'univers LEGO Dimensions, mais des accessoires utilisés par les personnages, susceptibles d'évoluer et d'être améliorés au cours du jeu.



Puisque nous avons créé une fonction permettant de facilement envoyer nos messages, faisons de même pour la réception. Car effectivement, à chaque message, il y a une réponse et nous faisons donc du *polling* en appelant à répétition :

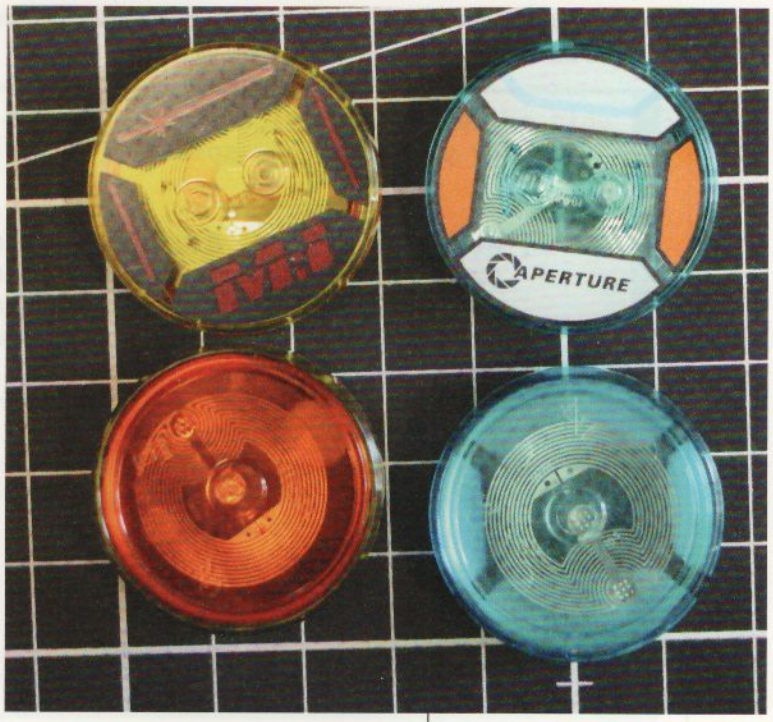
```
int query_report(hid_device *handle,
   unsigned char id, unsigned char *buf, size_t buflen)
{
    unsigned char *recbuf;
    int res;

    if ((recbuf = malloc(buflen)) == NULL) {
        err(EXIT_FAILURE, "query_report() malloc Error");
    }

    res = hid_read_timeout(handle, recbuf, buflen, 1000);

    // timeout (erreur silencieuse)
    if (res == 0)
        return(-1);

    // autre erreur
    if (res != buflen) {
        warnx("hid_read() error");
        return(-1);
    }

    memcpy(buf, recbuf, buflen);

    free(recbuf);

    return(0);
}
```

Notez l'utilisation de `hid_read_timeout()` et non `hid_read()`, nous permettant de spécifier un temps limite en millisecondes et donc d'éviter les appels bloquants. Plus loin dans le code, nous allons boucler pour être à l'écoute des événements (ajouts et retraits de *tags*) et ceci nous permet de plus facilement capter un *SIGINT* et de libérer les ressources proprement avant de quitter notre programme.

Grâce à cette nouvelle fonction, nous pouvons donc débuter la communication avec quelque chose comme (je vous fais grâce des vérifications des valeurs retournées à chaque appel, mais elles existent) :

```
hid_init();
handle = hid_open(LEGOVID, LEGOPID, NULL);

printf("initmsg\n");
send_message(handle, 0x00, initmsg, 32);
query_report(handle, 0x00, getbuf, 32);
```

Une fois l'initialisation faite, nous pouvons enfin faire faire quelque chose au périphérique. Commençons par un peu de lumière...



Les socles des figurines referment des tags NFC NTAG213. Les différentes couleurs de plastiques dénotent les générations, ou « vagues », de diffusion des contenus supplémentaires entre 2015 et 2017.

3.1 Contrôler les LED

Vous l'aurez compris, communiquer avec le Toy Pad revient à simplement envoyer des messages ayant la structure décrite précédemment et de s'enquérir des réponses. Le *payload* peut être des données ou des arguments à transmettre, mais l'élément le plus important est l'octet représentant la commande en position 2. Nous avons fait déjà connaissance avec `0xb0` pour l'initialisation et évoqué `0xb1` (*seed* PRNG) et `0xb3` (challenge). Mais il existe, vraisemblablement, au moins deux autres catégories de commandes :

- `0xc0`, `0xc2`, `0xc3`, `0xc6`, `0xc7` et `0xc8` pour contrôler les LED ;
- et `0xd2`, `0xd3` et `0xd4` pour gérer les opérations sur les *tags*.

La gestion des LED est étonnamment riche avec différentes commandes permettant d'obtenir des effets directement gérés par le Pad et sans autres interventions de l'hôte :

- `0xc0` : changement immédiat de couleur ;
- `0xc2` : changement immédiat de couleur avec transition optionnelle ;
- `0xc3` : changement de couleur en flashant ;
- `0xc6` : transition douce des couleurs des trois zones ;
- `0xc7` : clignotement des zones avec différentes fréquences et couleurs ;
- `0xc8` : changement immédiat de couleur des trois zones avec des couleurs individuelles.

Nous n'allons pas utiliser ici toutes ces variations qui reviennent plus ou moins toutes à faire la même chose. Nul doute qu'elles soient certainement là pour une bonne raison, mais pour une utilisation « de base », `0xc2` me semble bien suffisant. Vous pourrez trouver un descriptif complet des arguments de ces commandes dans le dépôt GitHub de *woodenphone* **[7]**.

Plutôt que de nous amuser à composer à chaque fois un message, écrivons une fonction dédiée au contrôle des LED :

```
void setcolor(hid_device *handle,
   u_char panel, u_char speed, u_char r, u_char g, u_char b)
{
    unsigned char msg[] =
     { 0x55, 0x08, 0xc2, 0x00, 0x00, 0x00, 0x00, 0x00,
       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
       0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00 };

    msg[3] = 0;     // num msg, send_message() s'en charge
    msg[4] = panel; // zone lumineuse
    msg[5] = speed; // temps de transition, 0x00 = immédiat
    msg[6] = 0x01;  // nbr pulsations, 0x00, 0xc8 et plus loop
    msg[7] = r;     // rouge
    msg[8] = g;     // vert
    msg[9] = b;     // bleu

    send_message(handle, 0x00, msg, CMD_SIZE);
}
```

`CMD_SIZE` est une macro définie à 32, la taille des messages, et le reste est relativement clair. Avec cette fonction, nous pouvons, après l'initialisation, allumer le Toy Pad pour signifier sa disponibilité (un petit effet « disco » ne fait jamais de mal, en particulier un samedi soir) :

```
    usleep(DELAY);
    setcolor(handle, PAD_CENTER, 0, 255, 60, 0);
    query_report(handle, 0x00, getbuf, 32);
    usleep(DELAY);
    setcolor(handle, PAD_CENTER, 0, 0, 0, 0);
    query_report(handle, 0x00, getbuf, 32);
    usleep(DELAY);
    setcolor(handle, PAD_RIGHT, 0, 255, 60, 0);
    query_report(handle, 0x00, getbuf, 32);
    usleep(DELAY);
    setcolor(handle, PAD_RIGHT, 0, 0, 0, 0);
    query_report(handle, 0x00, getbuf, 32);
    usleep(DELAY);
    setcolor(handle, PAD_LEFT, 0, 255, 60, 0);
    query_report(handle, 0x00, getbuf, 32);
    usleep(DELAY);
    setcolor(handle, PAD_LEFT, 0, 0, 0, 0);
    query_report(handle, 0x00, getbuf, 32);
    usleep(DELAY);
    setcolor(handle, PAD_ALL, 10, 64, 64, 32);
    query_report(handle, 0x00, getbuf, 32);
```

Notez les PAD_CENTER, PAD_RIGHT, PAD_LEFT et PAD_ALL, définis ainsi :

```
#define PAD_ALL             0x00
#define PAD_CENTER          0x01
#define PAD_LEFT            0x02
#define PAD_RIGHT           0x03
```

Ces valeurs seront également utilisées lorsque le Toy Pad nous signalera l'apparition (et la disparition) d'un *tag*. Et à ce propos, justement...

3.2 Événements, requêtes et réponses

Pour l'instant, toutes les commandes que nous avons vues sont embarquées dans des messages débutant systématiquement par l'octet 0x55 qui est désigné, dans les codes d'un peu tout le monde, comme étant un *magic number*. Nous ne l'avons pas encore vu, puisque nous n'avons rien fait des réponses du Toy Pad, mais les messages que nous obtenons en retour débutent également par cette valeur... Pour l'instant !

Les messages 0x55 sont des réponses à des requêtes que nous envoyons, mais lorsqu'on pose un *tag* sur l'une des surfaces, nous obtenons, après un appel à notre query_report(), un message différent. Celui-ci se compose ainsi :

```
octet 0 : valeur magique (0x56)
octet 1 : taille du payload
octet 2 : emplacement sur le Pad
octet 3 : ??
octet 4 : index
octet 5 : événement
octet 6 à 12 : UID du tag
octet 13 : somme de contrôle
octet 14 à 31 : bourrage (0x00) pour arriver à 32 octets
```

L'*index* est le numéro du *tag* maintenu par le Toy Pad. Ceci permet ensuite de s'adresser directement au *tag*, pour le lire ou l'écrire, sans avoir à l'identifier par un autre moyen. L'événement peut être :

```
#define TAG_PLACED    0x00 // placé
#define TAG_REMOVED   0x01 // retiré
```

La somme de contrôle est calculée exactement comme précédemment et nous pouvons donc vérifier l'intégrité du message qui nous arrive. Surveiller l'arrivée de ces messages n'est pas difficile et se résume à une simple boucle while() conditionnée par une variable globale (gobyebye). Via des appels à sigaction(), nous avons installé un *handler* pour les signaux *SIGINT* et *SIGTERM* juste après l'initialisation du Pad :

```
void termination_handler(int sig)
{
    printf("\nCaught signal %d\n", sig);
    gobyebye=1;
}
```

Ainsi, un simple CTRL+C nous sort de la boucle et nous pouvons alors éteindre les LED, libérer les ressources et conclure proprement le programme. Ceci n'est possible que parce que nous utilisons `hid_read_timeout()` et non `hid_read()` (bloquant).

En plaçant la minifigure *Doctor Who* sur le périphérique, celui-ci nous envoie : `56 0b 03 00 00 00 04 bc 42 7a 5c 49 80 05` suivi de 19 `00` sans importance (bourrage/padding). Ce même *tag*, lu sur un smartphone avec l'application *NFC TagInfo* ou *NXP TagInfo* nous confirme l'UID *04BC427A5C4980*. En retirant la figurine, nous obtenons un message en tous points identique, si ce n'est dans l'octet 5 qui passe de `0x00` à `0x01` avec, là encore, l'UID du *tag*.

Cet identifiant unique de 7 octets, propre à chaque NTAG21x peut déjà être utilisé pour différentes applications. En effet, non seulement nous pouvons détecter l'arrivée et le départ des minifigures et donc réagir en conséquence, mais ceci fonctionne également avec un NTAG213 neuf. Plus amusant encore, même un *tag* MIFARE Classic 1k sera détecté. L'UID n'est pas tout à fait correct puisqu'il est transmis sur 7 octets (`432EA94F000000`) alors qu'un tel *tag* n'en utilise que 4 (`432EA94F`), mais on peut estimer que cela fonctionne. Le champ des possibilités est donc déjà important, mais il est possible d'aller plus loin.

Le contenu « sensible » des *tags* est protégé par un mécanisme d'authentification propre aux NTAG21x, pour y accéder nous devons posséder la clé adéquate, mais le Toy Pad, lui, peut le faire à notre place. Deux cas de figure distincts se présentent selon qu'il s'agisse d'un *tag* de véhicule/gadget ou d'une minifigure.

Dans le premier cas, le Pad accède au contenu du *tag* et nous remonte simplement les informations qui s'y trouvent. Ceci peut être accompli en utilisant la commande `0xd2` structurée ainsi :

```
octet 0 : valeur magique (0x55)
octet 1 : taille du payload
octet 2 : commande 0xd2
octet 3 : compteur de messages
octet 4 : index du tag tel que rapporté par un message 0x56
octet 5 : numéro de la première page du tag à retourner
octet 6 : somme de contrôle
octet 7 à 31 : bourrage (0x00) pour arriver à 32 octets
```

La notion de « page » est propre aux *tags* NTAG21x donc les données sont organisées sous forme de groupes de 4 octets désignés ainsi. Un NTAG213 comporte 180 octets organisés en 45 pages (540 octets et 135 pages pour un NTAG215 et 924 et 231 pour un NTAG216). Certaines de ces pages sont réservées aux informations du constructeur (UID, etc.), à la configuration et à l'authentification, mais la majeure partie est destinée au stockage d'informations utilisateur (144 octets sur les 180 disponibles, dans le cas d'un NTAG213).

La commande `0xd2` prend en argument le numéro de la première page dont nous souhaitons obtenir le contenu, mais c'est en réalité un groupe de 4 pages, 16 octets donc, qui est retourné (c'est ainsi que fonctionne une lecture sur un NTAG21x). Comme nous connaissons le format de la réponse, nous pouvons créer une structure qui nous permettra de facilement accéder aux éléments retournés :

```
typedef struct {
    u_char type;         // 0x55
    u_char size;         // taille Payload
    u_char counter;      // compteur message
    u_char index;        // index du tag
    u_char page0[4];     // page + 0
    u_char page1[4];     // page + 1
    u_char page2[4];     // page + 2
    u_char page3[4];     // page + 3
    u_char checksum;     // somme de contrôle
    u_char padding[11]; // bourrage
} D2response;
```

Il nous suffira alors de *caster* la réponse obtenue via notre fonction `query_report()`, ou plus exactement de faire pointer toutes les structures sur les mêmes données avec quelque chose comme :

```
// buffer for incomming messages
unsigned char getbuf[CMD_SIZE] = { 0 };
TagEvent *tagevent = (TagEvent *)getbuf;
D2response *d2resp = (D2response *)getbuf;
D4response *d4resp = (D4response *)getbuf;
```

Les données qui nous intéressent se trouvent à partir de la page 0x24 (36) et nous avons là 16 octets qui caractérisent le *tag* posé sur le Pad. Une réponse à notre commande `0xd2` pourra, par exemple, être `55 12 0a 00 06 04 00 00 00 00 00 00 00 01 00 00 00 00 00 00 7c`, qui nettoyée des informations secondaires, se résumera à :

- page 0x24 : `06 04 00 00`, l'identifiant du véhicule/gadget en *big endian*, correspondant donc à `0x00000406` ou 1030 en décimal (c'est le *TARDIS* de *Doctor Who*) ;
- page 0x25 : `00 00 00 00`, aucune idée (peut-être `0x0000000000000406`, sur 64 bits en couplant avec les 4 octets de la page 0x24) ;
- page 0x26 : `00 01 00 00`, une valeur qui semble identifier un *tag* véhicule/gadget plutôt qu'une minifigure ;
- page 0x27 : `00 00 00 00`, même raisonnement que pour l'identifiant, peut-être les 4 premiers octets d'une valeur 64 bits.

En jonglant simplement avec l'ordre des octets via quelques fonctions utilitaires, nous pouvons donc facilement identifier non seulement qu'il s'agit d'un véhicule/gadget, mais aussi, et surtout duquel il s'agit. En fouillant sur GitHub, on trouve sans problème des listes complètes sous un format ou un autre (souvent en JSON) qui nous permettent de créer un tableau :

```
static const char *legoVehicleStr[] = {
/* 000 */ "Police Car",
/* 001 */ "Aerial Squad Car",
/* 002 */ "Missile Striker",

/* 003 */ "Gravity Sprinter",
/* 004 */ "Street Shredder",
/* 005 */ "Sky Clobberer",

/* 006 */ "Batmobile",
/* 007 */ "Batblaster",
/* 008 */ "Sonic Batray",

/* 009 */ "Benny's Spaceship",
/* 010 */ "Lasercraft",
/* 011 */ "The Annihilator",

/* 012 */ "DeLorean Time Machine",
/* 013 */ "Ultra Time Machine",
/* 014 */ "Electric Time Machine",

/* 015 */ "Hoverboard",
/* 016 */ "Cyclone Board",
/* 017 */ "Ultimate Hoverjet",
[...]
```

Il nous suffira alors de chercher l'identifiant, déduit de 1000, pour afficher la chaîne correspondante dès la réponse reçue, avec un morceau de code comme :

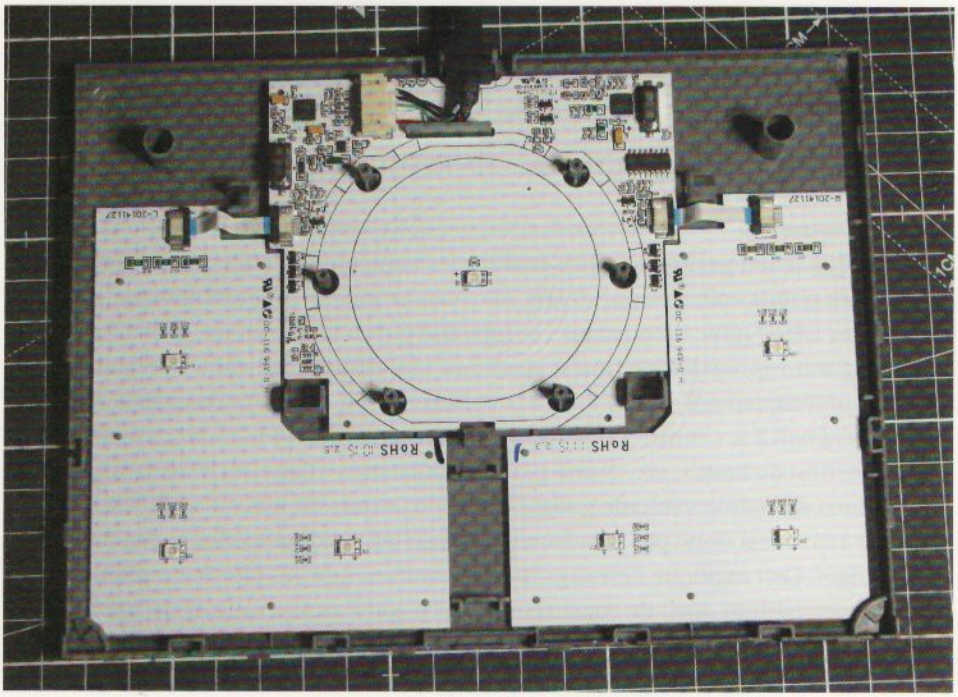
```
printf("Reply to D2 cmd (message %02X): ", D2query_number);
printf("%02X%02X   ", d2resp->page0[0], d2resp->page0[1]);
tagRef = d2resp->page0[1] << 8 | d2resp->page0[0];
if (tagRef>=1000) {
  printf("%04X = %u   ", tagRef, tagRef);
  tagRefIdx = tagRef-1000;
  if (tagRefIdx < tagsNamesLen) {
    printf("This is : " BOLDGREEN "%s" RESET, legoVehicleStr[tagRefIdx]);
    if (tagRefIdx > 154) tagRefIdx++;
    switch(tagRefIdx % 3) {
    case 0:
      printf("  (basic build)\n");
      break;
    case 1:
      printf("  (rebuild 1 of: %s)\n", legoVehicleStr[tagRefIdx-1]);
      break;
    case 2:
      printf("  (rebuild 2 of: %s)\n", legoVehicleStr[tagRefIdx-2]);
    }
```

– Jouons aux LEGO... avec des tags NFC –

```
    } else {
      printf("VEHICLE NOT IN MY LIST !\n");
    }
  } else {
    printf("NOT A VALID TAG ID/REF (<1000)\n");
  }
```

Notez que le jeu permet de faire évoluer les véhicules au cours de l'aventure, grâce à des points collectés par divers moyens. Le joueur est alors invité à utiliser les briques LEGO pour reconstruire la figurine puis à la placer sur le Toy Pad pour que le *tag* soit reprogrammé (mis à jour). Les identifiants des véhicules/gadgets se suivent numériquement sous la forme : « construction de base », « reconstruction 1 », « reconstruction 2 », « construction de base », « reconstruction 1 », « reconstruction 2 », « construction de base », etc. À une petite nuance près ! Cette série de groupes de 3 variations du même véhicule/gadget est interrompue au niveau de l'identifiant 1154. Le 1155 n'est donc pas la seconde reconstruction d'un véhicule, mais un nouveau gadget. Ceci explique l'étrange incrémentation que j'ai placée en plein milieu du code, permettant ainsi de conserver l'astuce consistant à utiliser un modulo pour retrouver la construction de base. Pourquoi ce trou ? Mystère. Tout aussi étrange que le fait que certains identifiants ne sont associés à aucune désignation. S'agirait-il d'extensions n'ayant jamais vu le jour ?

La mécanique de commercialisation du jeu était d'une efficacité redoutable. Une fois la base du jeu terminée avec les figurines initiales, on ne pouvait que souhaiter continuer l'aventure et donc acheter les extensions... Ou alors c'est juste moi qui aime bien les LEGO.



L'intérieur du Toy Pad est assez simple de conception : trois PCB interconnectés, étant à la fois des supports pour les LED RGB et des antennes pour les zones RFID/NFC.

3.3 Minifigures : la crypto s'en mêle !

Les véhicules/gadgets sont relativement faciles à gérer une fois les arguments de la commande et le format de la réponse connus. Les minifigures, en revanche, sont un peu plus délicates puisque les données obtenues via une commande `0xd4` sont chiffrées, tout comme les arguments (ou *payload*) de la commande elle-même. Là, un certain flou entoure toujours ces données, mais il est parfaitement possible d'arriver à identifier l'un de ces *tags*.

L'algorithme de chiffrement utilisé est TEA, pour *Tiny Encryption Algorithm* **[8]**. Il s'agit d'un algorithme de chiffrement par bloc simple et surtout s'implémentant avec une quantité de code relativement réduite. Ceci en fait un choix évident pour les applications embarquées et/ou à base de microcontrôleur, comme le Toy Pad. Une implémentation possible et facile à obtenir de TEA est celle tout simplement fournie sur la page Wikipédia prenant ici la forme des fonctions `teaencrypt()` et `teadecrypt()` et ayant deux arguments, un pointeur vers les données et un autre vers la clé à utiliser. À ce propos justement, je choisis ici de ne pas l'intégrer dans l'article puisque, même si le jeu est aujourd'hui obsolète, c'est une donnée qu'on peut considérer comme « sensible » (expliquer des algorithmes est une chose, diffuser des secrets en est une autre). Le lecteur curieux aura cependant vite fait de la piocher dans l'un des nombreux dépôts GitHub proposant des expérimentations autour du matériel LEGO Dimensions. Nous nous en tiendrons donc à la désigner par le nom de la variable la contenant : `unsigned char teakey[]`.

La clé utilisée étant initialement constituée de 16 `uint8_t` (`unsigned char`) alors que l'implémentation attend des `uint32_t`, nous créons deux fonctions intermédiaires spécialement dédiées au chiffrement/déchiffrement des données des minifigures :

```
void decryptPayload(unsigned char *buf) {
    uint32_t data[2];
    uint32_t keya[4];

    keya[0] = readUInt32LE(teakey,0);
    keya[1] = readUInt32LE(teakey,4);
    keya[2] = readUInt32LE(teakey,8);
    keya[3] = readUInt32LE(teakey,12);

    data[0] = readUInt32LE(buf,0);
    data[1] = readUInt32LE(buf,4);

    teadecrypt(data, keya);

    writeUInt32LE(data[0], buf, 0);
    writeUInt32LE(data[1], buf, 4);
}

void encryptPayload(unsigned char *buf) {
    uint32_t data[2];
    uint32_t keya[4];

    keya[0] = readUInt32LE(teakey,0);
    keya[1] = readUInt32LE(teakey,4);
    keya[2] = readUInt32LE(teakey,8);
    keya[3] = readUInt32LE(teakey,12);

    data[0] = readUInt32LE(buf,0);
    data[1] = readUInt32LE(buf,4);

    teaencrypt(data, keya);

    writeUInt32LE(data[0], buf, 0);
    writeUInt32LE(data[1], buf, 4);
}
```

Les fonctions `readUInt32LE()` et `writeUInt32LE()` nous permettent de simplement convertir 4 octets en un entier non signé sur 32 bits et, inversement, d'inscrire 8 octets dans un tableau à partir d'un `uint32_t`, le tout en respectant l'*endianness*. Ce sont là des fonctions utilitaires que je « trimbale » depuis fort longtemps (`readUInt32BE()`, `readUInt32LE()`, `writeUInt32BE()`, `writeUInt32LE()`, `readUInt16BE()`, `readUInt16LE()`, `writeUInt16BE()`, `writeUInt16LE()`) et généralement fort utiles dès lors qu'on joue avec des données brutes. Elles n'ont rien de spécial et sont parfaitement génériques.

Nous voici maintenant en mesure de chiffrer et déchiffrer un *payload* de 8 octets (2 pages de 4 octets d'un *tag*) à l'aide de la clé secrète et nous pouvons alors nous pencher sur la fameuse commande `0xd4` chargée d'interroger un *tag* de minifigure. Sa syntaxe est la suivante :

```
octet 0 : valeur magique (0x55)
octet 1 : taille du payload
octet 2 : commande 0xd4
octet 3 : compteur de messages
octet 4 à 11 : payload chiffré TEA
octet 12 : somme de contrôle
octet 13 à 31 : bourrage (0x00) pour arriver à 32 octets
```

Le *payload* avant chiffrement n'est que partiellement connu. Le premier des 8 octets correspond à l'index du *tag* sur le Pad et le reste est constitué des valeurs `4C 26 93 62 FF 06 84`. On pourrait penser que le 0x26 est un numéro de page dans le *tag*, mais en réalité, ce sont les pages 0x24 à 0x27 qui sont retournées (cf. partie sur le contenu des *tags*), déchiffrées depuis le « support » et rechiffrées avant transmission via USB HID. On comprendra aisément pourquoi ces mécanismes sont en place alors qu'il n'existe rien de tel pour les véhicules/gadgets, étant donné que ces minifigures permettent l'accès à des zones de jeux sous la forme de contenus téléchargeables et non simplement à de nouveaux personnages jouables.

Nos fonctions de chiffrement/déchiffrement agissant directement sur les données du *buffer* passé en argument, il nous suffira de composer le *payload*, ajuster l'index du *tag* à interroger, chiffrer et compléter avec le reste du message (et du *padding*) avant d'utiliser `send_message()` à destination du Toy Pad. La réponse pourra alors être traitée comme pour les véhicules/gadgets, mais en ajoutant un appel à `decryptPayload()` :

```
memcpy(cryptopayload, d4resp->teadata, 8);
decryptPayload(cryptopayload);
printf("Reply to D4 cmd (message %02X)  ", D4query_number);
tagCRefIdx = readUInt32LE(cryptopayload,0);
printf("character ID = 0x%08X = %u       ", readUInt32LE(cryptopayload,0),
tagCRefIdx);
if (tagCRefIdx < tagsCNamesLen) {
  printf("This is : " BOLDGREEN "%s" RESET "\n", legoCharacterStr[tagCRefIdx]);
} else {
  printf("CHARACTER NOT IN MY LIST !\n");
}
<file>
```

Là encore, nous « castons » les données sur une structure dédiée pour nous simplifier la vie :

```
<file>
typedef struct {
    u_char type;        // 0x55
    u_char size;        // taille payload
    u_char counter;     // compteur message
```

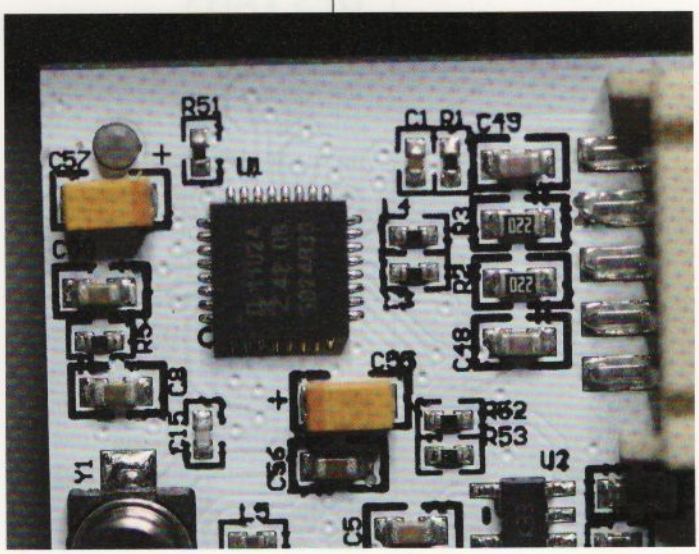
```
    u_char index;       // index
    u_char teadata[8];  // payload chiffré TEA
    u_char checksum;    // somme de contrôle
    u_char padding[19]; // bourrage
} D4response;
```

Et comme précédemment, nous utilisons les 4 premiers octets du *payload* déchiffré, convertis en entier non signé 32 bits *little endian* pour rechercher la chaîne de caractères correspondant au nom du personnage dans un tableau :

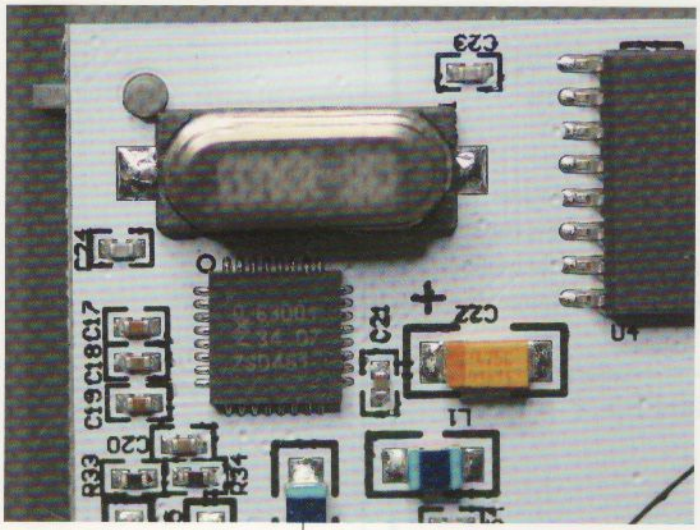
```
static const char *legoCharacterStr[] = {
    ">null<",
    "Batman",
    "Gandalf",
    "Wyldstyle",
    "Aquaman",
    "Bad Cop",
    "Bane",
    "Bart",
    "Benny",
    "Chell",
[...]
```

Ceci clôt la partie concernant le Toy Pad et la communication USB HID. Avec ces éléments, nous sommes en mesure de piloter le périphérique et de faire, plus ou moins, ce que le jeu fait dans son fonctionnement normal. Il nous est donc possible d'animer les LED et de lire, mais surtout d'identifier, aussi bien les véhicules/gadgets que les personnages. Nous avons également découvert que des *tags* NFC génériques peuvent être utilisés et sont susceptibles d'être lus (avec `0xd2`) et peut-être même intégralement. Ceci demande davantage d'expérimentations, mais ouvre des perspectives intéressantes.

Enfin, une commande particulière n'a pas été testée ici, c'est `0xd3` permettant d'écrire sur un *tag*. Au vu des risques de corruption de données pouvant découler de tels essais, dont la pire issue serait de rendre un ou plusieurs *tags* totalement inutilisables (pour le jeu comme pour une utilisation alternative), je préfère remettre cela à une autre fois (ou à d'autres courageux développeurs). Pour explorer sereinement



Le microcontrôleur utilisé est, sans surprise, un modèle de chez NXP : le LPC11U24. C'est un ARM Cortex-M0 avec 32 Kio de flash, 10 Kio de SRAM et 4 Kio d'EEPROM. C'est assez modeste, mais son principal avantage est de disposer d'un contrôleur USB 2.0 full-speed « périphérique », évitant ainsi l'ajout d'électronique supplémentaire.



La partie NFC du Toy Pad est gérée par ce MFRC630 de chez NXP, vraisemblablement connecté au microcontrôleur en i²c. Ce contrôleur est capable de prendre en charge les tags MIFARE Classic (1k et 4k), MIFARE Ultralight, MIFARE Ultralight C, MIFARE Plus et MIFARE DESFire. Ainsi que les NTAG21x, bien sûr.

une telle activité, il faudra tout d'abord être en mesure de procéder à une « sauvegarde » des informations des *tags* et, pour ce faire, en comprendre le contenu. Chose que nous allons voir sans attendre...

4. CONTENU DES TAGS

Contrairement au Toy Pad sur lequel je m'étais moi-même cassé les dents par le passé, avec un succès très relatif, les *tags* eux-mêmes seraient restés un mystère sans le fantastique travail d'Eric Betts (et d'autres de la communauté Proxmark3 version Iceman, comme Alina Shumann alias AlinaNova21/ags131) parfaitement synthétisé sous la forme d'une application macOS écrite en Swift, nommée *LDIO* et disponible sur GitHub **[9]**. Mais comme tout le monde n'a pas forcément d'argent à dépenser dans du matériel se périmant à vitesse grand V parce qu'un nouveau modèle vient de matérialiser la dernière révolution technologique (vous savez comme l'ajout d'un port HDMI ou le retour du MagSafe), explorons tout cela en C.

Avant toute chose, notez que nous utiliserons, naturellement, la LibNFC **[10]** mais aussi et surtout l'excellente LibFreeFare **[11]**. Celle-ci est une surcouche à la LibNFC, facilitant l'utilisation de *tags* FeliCa Lite, MIFARE Classic, MIFARE DESFire, MIFARE Mini, MIFARE Plus, MIFARE Ultralight, MIFARE Ultralight C et NTAG21x. Cependant, le support des NTAG21x n'est présent que dans la version de développement, post-version 0.4.0 généralement incluse dans les distributions GNU/Linux et autres systèmes *open source*. Cette dernière version stable date de 2015 et la 0.5.0 se fait attendre depuis bien longtemps. Vous n'aurez donc d'autre choix que de recompiler vous-même une version de développement. Les utilisateurs Debian (ou Raspbian / Raspberry Pi OS) seront ravis d'apprendre qu'il est possible de produire un paquet « propre » à partir des sources avec un minimum d'ajustements (du contenu de `debian/`). Vous pourrez également trouver un port pour FreeBSD directement sur mon GitLab **[12]**.

Ceci étant précisé, attaquons le sujet. Les NTAG213 sont des *tags* NFC compatibles Forum Type 2 produits par NXP et sont très courants, car (relativement) simples d'utilisation, peu chers et compatibles avec de très nombreux matériels (smartphones, en particulier). Le NTAG213 est l'un des membres de la famille NTAG21x incluant également le NTAG215 et NTAG216 (les NTAG210 et NTAG212 semblent peu courants). Rien d'étonnant donc qu'ils forment la base des *tags* LEGO Dimensions.

– Jouons aux LEGO... avec des tags NFC –

En dehors du stockage de données par page de 4 octets comme nous l'avons évoqué, et l'existence d'un UID programmé en usine sur 7 octets, les NTAG21x possèdent un certain nombre de caractéristiques intéressantes :

- limitation de l'accès en lecture et/ou écriture par mot de passe de 32 bits (cf. plus loin) ;
- possibilité de verrouiller en écriture tout ou partie des pages ;
- possibilité de verrouiller définitivement la configuration du verrouillage (et donc le contenu des pages concernées) ;
- possibilité de verrouiller définitivement la configuration du *tag* elle-même.

La granularité de la configuration et de la limitation d'accès n'est pas aussi fine qu'avec d'autres *tags* (comme les MIFARE Classic), mais c'est généralement suffisant pour un usage courant. Un excellent exemple concerne la configuration de l'authentification pour l'accès aux pages, puisque nous ne pouvons que préciser à partir de quel numéro de page celle-ci devient nécessaire. On ne peut donc pas dire « *je veux que l'accès aux pages 4, 5 et 6 nécessite une authentification, mais pas les autres comme 7, 8 ou 9* ». Autre point **très** important : le mécanisme de verrouillage, assez alambiqué, permet de rendre un *tag* complètement inutilisable en écriture, et ce, de façon définitive. Couplé à l'authentification qui peut être configurée pour la lecture/écriture et non simplement l'écriture, rendant ainsi illisible la configuration des pages sans le mot de passe, le risque d'erreur de manipulation conduisant un *tag* directement vers la poubelle n'est pas négligeable.

Fort heureusement, notre objectif ici n'est que de lire ce qui existe et non d'écrire dans les *tags* originaux. Mais ce n'est pas tout, en plus des sécurités fournies par NXP pour ce composant, nous devons également gérer celles propres au jeu et au Toy Pad.

Mais avant cela, voyons ce qui se passe lorsque nous tentons naïvement de lire un de ces *tags* LEGO avec un smartphone et l'une des applications Android citées précédemment. Les *tags*, qu'il s'agisse de véhicules/gadgets ou de personnages sont structurés de la même manière. Tout d'abord, leur configuration est verrouillée de façon définitive, il n'est donc plus possible de modifier quelles pages nécessitent une authentification et quelles sont celles qui sont en lecture seule. Le contenu « utilisateur » se divise en deux parties :

- De la page 4 à la page 29 (0x1d) le contenu est lisible librement et est constitué d'un enregistrement NDEF de type texte contenant ce qui semble être une identification (« 965354753715 » pour la minifigure *Doctor Who*, par exemple). Cette identification est répétée en pages 26, 27 et 28 sous la forme d'une chaîne ASCII brute. Il semblerait que, du point de vue du Toy Pad et du jeu, ces deux occurrences de l'identifiant ne soient pas utilisées.
- À partir de la page 30 (0x1e), une authentification est nécessaire, à la fois pour l'écriture et la lecture, rendant inaccessibles, sans le mot de passe, les données que nous obtenons avec le Toy Pad.

À cela s'ajoutent différents verrouillages en lecture seule interdisant pour l'un et l'autre type de *tag* la modification des données NDEF, de la chaîne répétée en pages 26 à 28 et de la configuration. Seules différences, les *tags* minifigures ont leurs pages 36 et 37 en lecture seule alors que les véhicules/gadgets permettent la réécriture des données (ce qui est logique). La page 38, contenant le `00 01 00 00` d'un

véhicule/gadget, est également verrouillée sur les deux types de *tags*, rendant donc impossible de permuter un personnage en véhicule et inversement. Tout ce qu'il est donc possible de faire, c'est lire les données et c'est d'ailleurs notre unique objectif ici.

4.1 S'authentifier auprès du tag

Pour s'enquérir du contenu d'un *tag* et identifier l'élément de jeu qu'il référence, il nous est nécessaire de disposer du mot de passe configuré dans le *tag*. Plus exactement, le mot de passe configuré pour **chaque** *tag*, car celui-ci n'est pas unique. Le mot de passe est en réalité calculé à partir de l'UID du *tag* et, bien entendu, la communauté de développeurs s'étant penchée sur le sujet a trouvé l'algorithme utilisé. En C, ceci se traduit ainsi :

```
void magicHash(uint8_t *data, size_t len, uint8_t *hash)
{
    uint32_t v2 = 0;
    for (int i = 0; i < len/4; i++) {
        int v4 = rotr32(25, v2);
        int v5 = rotr32(10, v2);
        uint32_t b = readUInt32LE(data, i*4);
        v2 = b + v4 + v5 - v2;
    }
    writeUInt32LE(v2, hash, 0);
}

void pwdgen(char *tag_uid, uint8_t *key)
{
    uint8_t base[] = {
        0x04, 0x4b, 0xd4, 0x0a, 0x9a, 0x40, 0x81, 0x28,
        0x63, 0x29, 0x20, 0x43, 0x6f, 0x70, 0x79, 0x72,
        0x69, 0x67, 0x68, 0x74, 0x20, 0x4c, 0x45, 0x47,
        0x4f, 0x20, 0x32, 0x30, 0x31, 0x34, 0xaa, 0xaa
    };
    uint8_t uid[7] = { 0 };
    uint8_t hash[4] = { 0 };

    hex2array(tag_uid, uid, 7);

    for (int i = 0; i < 7; i++) {
        base[i] = uid[i];
    }

    magicHash(base, 32, hash);
    memcpy(key, hash, 4);
}
```

Le tableau `base[]` de 32 octets est composé des 7 octets d'UID du *tag*, de la chaîne de caractères « © *Copyright LEGO 2014* » et d'un bourrage constitué de deux occurrences de `0xaa`. La fonction `magicHash()` procède à quelques opérations de décalage et d'addition et produit les 32 bits qu'on pourra directement utiliser avec la fonction `ntag21x_authenticate()` de la LibFreeFare, après avoir créé une clé à partir des 4 octets avec `ntag21x_key_new()`. Notez que cette fonction prend en argument deux éléments : un pointeur vers la clé (ou mot de passe), mais également un autre vers deux octets de *PACK*, pour *Password ACKnowledge*. Le mécanisme d'authentification pour les NTAG21x repose sur une logique peu courante. Pour valider l'accès, on fournit non seulement le mot de passe de 32 bits (*PWD* dans la documentation), mais également une valeur à obtenir en cas de réussite, le PACK. C'est la correspondance de ces deux éléments qui valide l'authentification et l'un comme l'autre est laissé au choix du développeur lors de la configuration du *tag*. L'authentification repose donc sur une valeur de 48 bits en réalité (PWD + PACK) et non juste les 32 bits de PWD. Dans le cas des *tags* LEGO Dimensions, le PACK est `0xaa55` et ne change pas d'un *tag* à l'autre.

Autre remarque intéressante concernant ce morceau de code, notez l'utilisation de la fonction `hex2array()` pour obtenir un tableau de `uint8_t` à partir de la chaîne retournée par `freefare_get_tag_uid()`. C'est étrange, mais il ne semble pas y avoir de fonction de la LibFreeFare permettant de remonter cette information sous une autre forme, alors que les manipulations sur les UID sont relativement courantes dans le domaine.

À ce stade, et en version simplifiée, notre code ressemble à quelque chose comme :

```
// initialisation LibNFC
nfc_init(&context);

// connexion au lecteur
pnd = nfc_open(context, NULL);

// obtention des tags présents
tags = freefare_get_tags(pnd);

// récupération de l'UID du premier tag
tag_uid = freefare_get_tag_uid(tags[0]));

// génération de PWD
pwdgen(tag_uid, legokey);

// création d'une clé
ntagkey = ntag21x_key_new(legokey, pack);

// connexion au tag
ntag21x_connect(tags[0]);

// authentification
ntag21x_authenticate(tags[0], ntagkey);
```

Toutes ces fonctions LibFreeFare doivent retourner `NFC_SUCCESS` et ceci doit, bien entendu, être vérifié, même si ceci n'est pas présenté ici pour limiter la taille de l'article.

Une fois la clé dérivée et utilisée, avec le PACK, pour s'authentifier auprès du *tag*, il est enfin possible de lire son contenu :

```
for (uint8_t p = 0; p <= ntag21x_get_last_page(tags[0]); p++) {
  if(ntag21x_read4(tags[0], p, data) == NFC_SUCCESS) {
    for (int i=0; i < 4; i++) {
      tagdata[(p*4)+i] = data[i];
    }
  } else {
    warnx("Unable to read page. Auth failed ?");
    for (int i=0; i < 4; i++) {
      tagdata[(p*4)+i] = 0xff;
    }
  }
}
```

Une fois l'ensemble des données obtenues, la technique pour identifier le *tag* est strictement identique à celle utilisée avec le Toy Pad. Mais ceci ne fonctionne qu'avec les véhicules/gadgets, repérables via le `00 01 00 00` en page 38. Pour les minifigures, les choses sont une nouvelle fois un peu différentes...

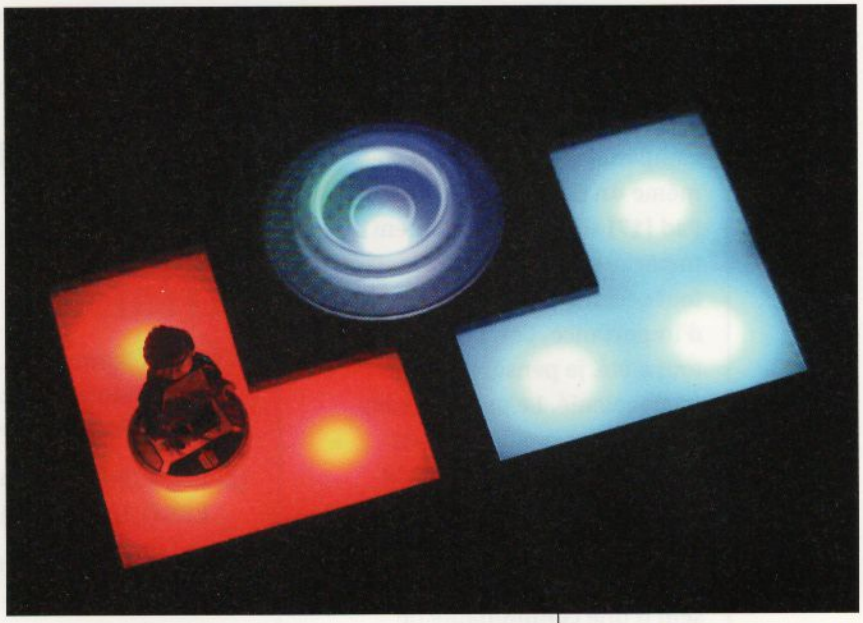
4.2 Chiffrement des données des minifigures

De la même manière que l'accès aux données des minifigures via le Toy Pad repose sur du chiffrement, les pages 36 et 37 ne contiennent pas de données directement exploitables. Sans surprise, le même algorithme de chiffrement, TEA, est utilisé, mais la clé n'est pas la même que celle du Toy Pad. Une nouvelle fois, la clé est calculée en fonction de l'UID du *tag* et sera donc unique à chaque *tag*.

```
void keygen(char *tag_uid)
{
    uint8_t minifigHashConstant[16] = {
       0xb7, 0xd5, 0xd7, 0xe6, 0xe7, 0xba, 0x3c, 0xa8,
       0xd8, 0x75, 0x47, 0x68, 0xcf, 0x23, 0xe9, 0xfe
    };

    uint8_t uid[7] = { 0 };
    uint8_t tmp[24];
    uint8_t tmphash[4];

    hex2array(tag_uid, uid, 7);

    for (int i = 0; i < 4; i++) {
        memcpy(tmp, uid, 7);
        memcpy(tmp+7, minifigHashConstant, (i+1)*4);
        tmp[7+((i+1)*4)] = 0xaa;
        magicHash(tmp, 7+((i+1)*4)+1, tmphash);
        memcpy(teakey+(i*4), tmphash, 4);
    }
}
```

Une fois encore, ceci n'est ici que l'implémentation C du code Swift d'Eric Betts. Nous obtenons ainsi la clé de chiffrement/déchiffrement TEA dans la variable globale `teakey[]` et pouvons l'utiliser avec exactement la même implémentation que pour le code USB HID. Il suffira donc d'appeler `decryptPayload()` sur les 8 octets des pages 36 et 37 et nous obtiendrons l'identifiant du personnage. Nous pouvons dès lors vérifier clairement quel *tag* correspond à quel élément du jeu sans avoir recours au Toy Pad.

Comme précédemment, je n'ai pas exploré la partie écriture ou création de *tags* à partir de NTAG213 vierges/neufs. En premier lieu parce que ceci ne m'intéresse pas réellement, si ce n'est pour créer des « sauvegardes » des minifigures en ma possession, mais également parce que, pour faire les choses correctement, il faudrait verrouiller les *tags* de la même manière que ceux d'origine, et donc les rendre inutilisables pour autre chose. Il est possible que ce verrouillage ne soit pas vérifié par le Toy Pad et/ou par le jeu (comme le laisse supposer le code de *phogar* sur GitHub **[13]**), mais ceci ne peut être confirmé que par une démarche que je n'ai pas envie de suivre (et ma PS3 prend la poussière dans un placard).



Une fois le contrôle du Toy Pad obtenu au moyen de notre code « maison », on peut l'utiliser comme le ferait le jeu officiel, en contrôlant les LED et en lisant le contenu des tags qui y sont déposés.

CONCLUSION

Il n'est pas courant, dans le magazine, de traiter d'un matériel sans en faire un usage concret. Et pourtant, nous voici à la fin de l'article, sans une réelle mise en œuvre du Toy Pad LEGO Dimensions ou de ses *tags* (si ce n'est un code relativement fonctionnel disponible dans mon GitLab **[14]**). Pour autant, nous avons exploré une technologie fort intéressante et susceptible d'être réutilisée, au-delà du défi technique, pour des usages pratiques. Les bases ayant été jetées, la suite dépendra de votre imagination et les idées ne manquent pas : créer un conteur d'histoire comme celui étudié dans le numéro 45 **[15]**, inventer un jeu de toutes pièces comme une sorte de Simon **[16]** en NFC, piloter la lecture de sa collection de MP3 en intégrant des *tags* dans les pochettes CD, utiliser le Pad comme système de signalisation lumineuse (sans les *tags* donc), etc.

Mais LEGO Dimensions est également un rappel et une illustration intéressante concernant la sécurité et la vie d'un produit.

Aujourd'hui, il existe des outils et même des applications Android et iOS permettant de créer illégalement des *tags* compatibles avec le jeu. Il y a même un émulateur de Toy Pad **[17]** simulant totalement le matériel et les figurines, mais tout ceci n'a plus réellement d'importance. Car, voyez-vous, tout a été, je pense, parfaitement calculé. Le premier *commit* pour cet émulateur, par exemple, date d'août 2021 et même si une version a parfaitement pu circuler des années avant, cela n'aura été qu'au sein d'une communauté de passionnés qui n'ont que faire du jeu lui-même ou de porter préjudice commercialement à son éditeur.

Aucune sécurité n'est vraiment totalement impénétrable, c'est une simple question de connaissances, de moyens et surtout de temps. La vie de LEGO Dimensions s'est étalée sur une période suffisamment réduite pour que les expérimentations ne finissent pas en usine de contrefaçons. Et maintenant que ce serait aisément possible pour un mauvais acteur, il n'y a plus d'argent à se faire puisque l'effet de mode est passé. Personne n'a plus intérêt à vendre de fausses figurines sur AliExpress. C'est un parfait timing, la sécurité a duré le temps nécessaire et les ingénieurs à l'origine du produit l'ont certainement conçue dans ce sens. Ce n'est que lorsque les concepteurs, ou plus justement ceux aux commandes de la stratégie commerciale s'imaginent faire reposer leur produit sur une sécurité « absolue et éternelle » que les ennuis commencent... **DB**

RÉFÉRENCES

[1] **https://gamergen.com/actualites/lego-dimensions-deux-ans-plus-tard-fini-extensions-jeu-toys-to-life-286762-1**

[2] **https://lego.fandom.com/fr/wiki/LEGO_Dimensions**

[3] **https://github.com/RfidResearchGroup/proxmark3**

[4] **https://github.com/libusb/hidapi**

[5] **https://usb.org/document-library/hid-descriptor-tool**

[6] **https://elecCelerator.com/usbdescreqparser/**

[7] **https://github.com/woodenphone/lego_dimensions_protocol**

[8] **https://fr.wikipedia.org/wiki/Tiny_Encryption_Algorithm**

[9] **https://github.com/bettse/LDIO**

[10] **https://github.com/nfc-tools/libnfc**

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

[12] **https://gitlab.com/0xDRRB/drrbFreeBSDports**

[13] **https://github.com/phogar/ldnfctags**

[14] **https://gitlab.com/0xDRRB**

[15] **https://connect.ed-diamond.com/hackable/hk-045/reutilisation-d-un-lecteur-audio-de-figurines**

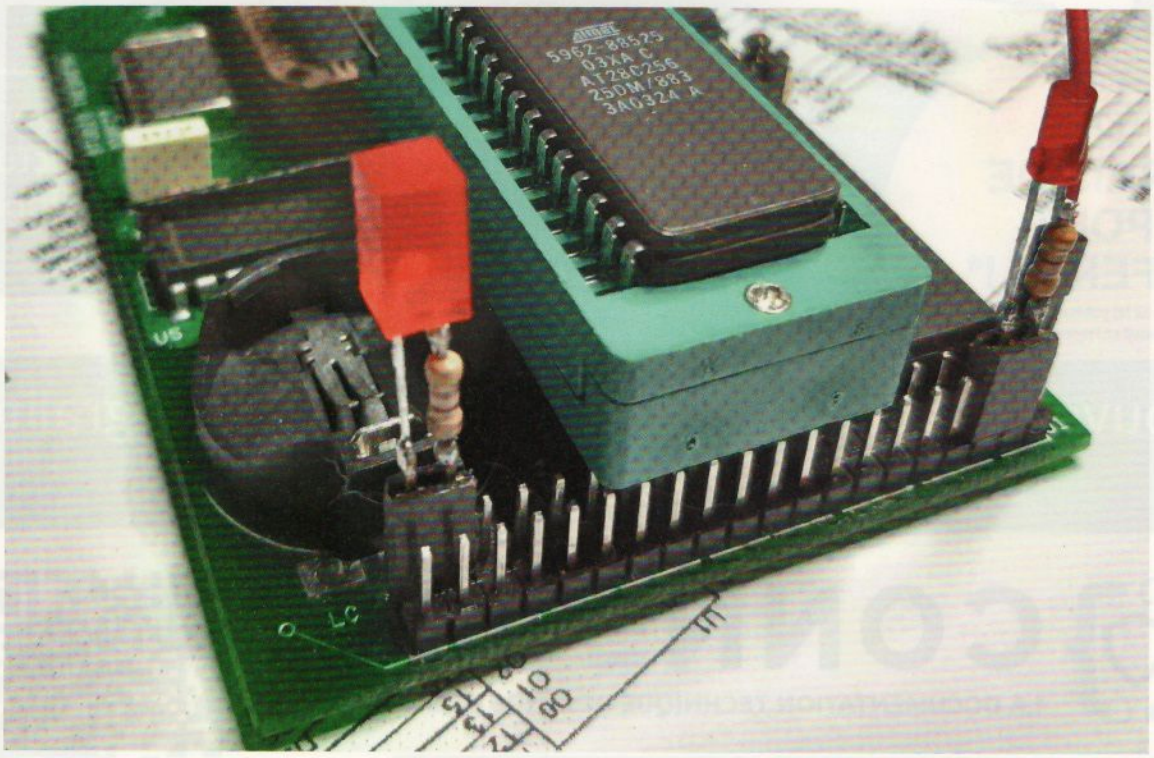
[16] **https://fr.wikipedia.org/wiki/Simon_(jeu)**

[17] **https://github.com/Berny23/LD-ToyPad-Emulator**

CARTE Z180 : LE MYSTÈRE DE LA RAM

Denis Bodor

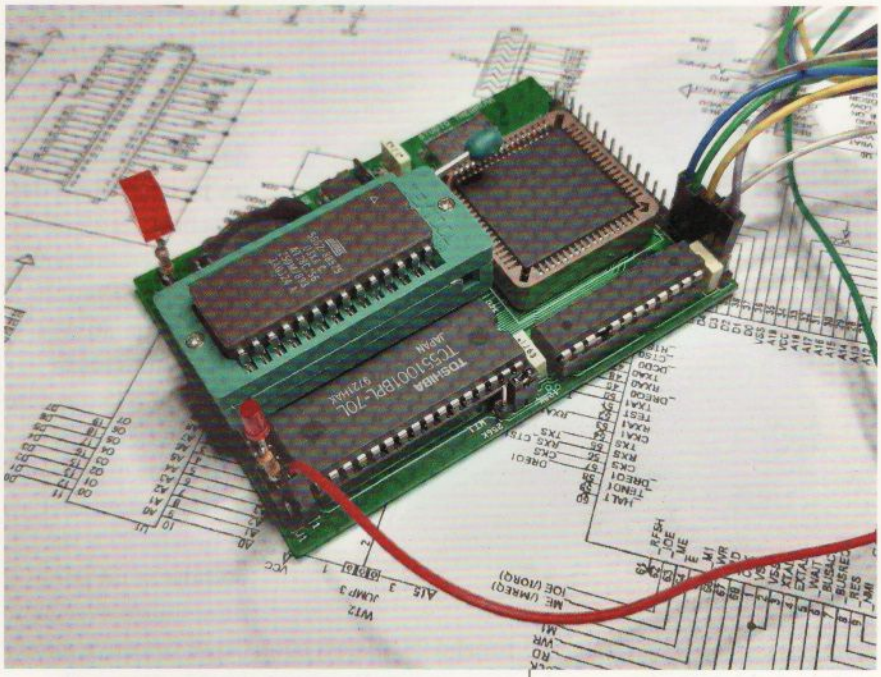
Dans l'épisode précédent, nous avons fait connaissance avec une carte industrielle à base de Z180 et nous sommes fixé comme objectif d'en faire notre « ordinateur 8 bits ». Nous avons chargé et exécuté notre programme en EEPROM et l'avons fait communiquer via une liaison série. Pour aller plus loin, nous devons cependant régler un problème qui consiste à comprendre comment est gérée la RAM et arriver à l'utiliser pour, au minimum, disposer d'une pile (stack) nous permettant d'appeler des sous-routines. Attaquons-nous au problème sans attendre...



– Carte Z180 : le mystère de la RAM –

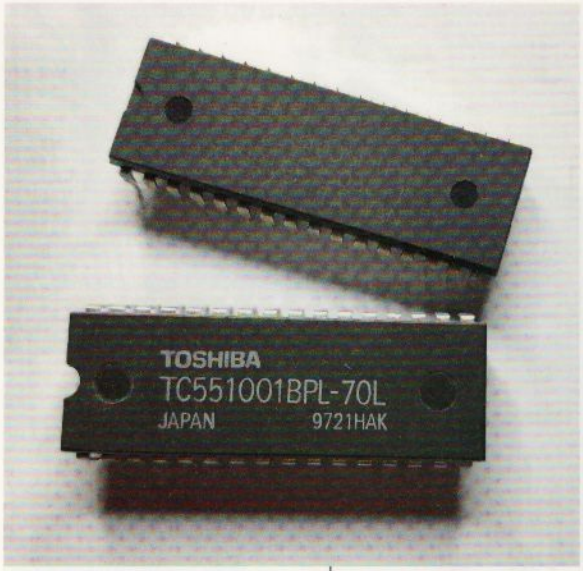
Avant de nous pencher sur la suite, revenons un instant sur l'article précédent, avec un potentiel problème d'une part et une amélioration de l'autre. Le problème concerne la communication série qui, après réception d'autres exemplaires de la carte et les expérimentations qui s'en sont suivies, se manifeste de la pire façon possible : parfois, l'envoi de données cesse et reprend de manière aléatoire. Pire encore, le fait de tripoter la carte influe sur ce comportement. Quelque chose ne va pas et ceci ressemble à une broche flottante qui devrait clairement être à la masse ou tirée à Vcc. Après avoir épuisé les potentielles solutions (filtrage de l'alimentation, vérification des condensateurs polyester, essai d'autres adaptateurs USB/série, etc.), j'en suis arrivé à utiliser une, ancienne et souvent oubliée, règle de base : « *Lorsque vous avez absolument tout essayé et que ça ne fonctionne toujours pas, en dernier recours... lisez la doc* ».

Le problème rencontré vient du fait que l'ASCI0 utilise un contrôle de flux matériel (RTS/CTS) par défaut qui, d'après ma compréhension de la *datasheet* du Z180, ne peut être désactivé, contrairement à ASCI1 (bit CTS1E de STAT1 impactant le rôle de la broche RXS (*Clocked Serial Receive Data* qui devient /CTS1)). Le fait de laisser la broche 18 de J2 non connectée fait que l'UART ne sait pas réellement s'il est possible d'envoyer les données ou non. Connecter cette broche /CTS0 au RTS de l'adaptateur USB/série (marqué « DTR » sur la sérigraphie) règle immédiatement le problème, et on connectera bien entendu aussi le CTS de l'adaptateur sur /RTS0, broche 19, de J2. Et c'est ainsi que les comportements aléatoires ont disparu, ou presque.



La carte de base a sensiblement évolué et s'est étoffée de quelques ajouts comme le support ZIF version 2, une SRAM plus modeste, une LED pour l'alimentation et une autre pour le signal d'horloge.

« Presque », car il arrive tantôt que la communication cesse... jusqu'à ce qu'une gentille petite tape soit administrée à l'EEPROM, ou la carte légèrement bousculée. Il y a un faux contact quelque part, ou une mauvaise soudure, et ceci nous amène à une amélioration notable : plutôt que de fabriquer un adaptateur 27C256/28C256 à l'aspect et à la robustesse plus que douteux, pourquoi ne pas simplement modifier la carte ?



La carte est initialement livrée avec une SRAM de 512 Kio (en haut) qui sera certainement mieux (et plus facilement) utilisée pour un autre projet. Certainement à base de Motorola MC68008, un processeur 8/16/32 bits de la famille m68k.

La modification est relativement simple, même si elle nécessite un minimum de dextérité. On coupe les pistes partant des broches 1 et 27 de l'emplacement pour l'EPROM, on établit une liaison du A14 de la RAM vers la broche 1 à l'aide d'un bout de fil (*bodge wire* dans le jargon d'usage) et on met la broche 27 à Vcc en faisant un pont de soudure avec la broche 28, juste à côté. On se retrouve alors avec un emplacement capable d'accueillir directement une EEPROM 28C256. Mais attention, la carte n'est ensuite plus compatible 27C256/27C512, mais ceci facilite grandement les choses, y compris pour ajouter un support ZIF par-dessus l'existant. Je n'ai pour l'instant modifié qu'une paire de cartes sur les 6 maintenant en ma possession et j'avoue hésiter à propager la modification, étant partagé entre l'aspect pratique et le côté « vintage » d'utiliser, à terme, une belle UVPROM. On remarquera cependant que le support ZIF ainsi placé bloque une partie des broches sur J1 et en particulier les signaux /WR, /IORQ (« _IOE » sur le schéma) et RES pour le *reset*, ainsi qu'une partie du bus de données et d'adresses exposé à cet endroit. Empiler les *sockets* DIP28 est une option, mais pas forcément une bonne idée pour l'intégrité des signaux.

Après ce petit interlude sur ce qui a déjà été commis comme exactions concernant cette carte, il est temps de s'attacher aux évolutions futures et à l'épineux problème de la RAM.

Si nous avions un Z80 dans une configuration similaire, avec une ROM de 32 Kio, nous utiliserions simplement une RAM de taille identique, séparant l'espace d'adressage en son milieu. Ceci serait très facile à faire, à l'aide d'un simple circuit logique, comme un ensemble de portes *OU* inversées. En effet, dans cette situation, la ligne A15 du bus d'adresses serait active uniquement quand la RAM est utilisée, de `0x8000` à `0xffff` et inactive pour la ROM de `0x0000` à `0x7fff`. Autrement dit, il suffit d'activer /CS (*Chip Select* ou *Chip Enable*) de l'un ou l'autre composant quand /MREQ (*Memory REQuest*) est à l'état bas et A15 haut pour la RAM, et bas pour la ROM.

Avec un Z180, les choses sont très différentes. En effet, ce processeur dispose de 20 lignes d'adresses, permettant donc d'adresser 2 puissance 20 octets, soit 1048576 ou 1 Mio. Pour autant, les registres, et *HL* en particulier (souvent utilisé pour pointer des données en mémoire), ne font que 16 bits et il n'est donc pas possible d'adresser plus de 65536 emplacements de mémoire. Comment donc le Z180 fait-il alors pour accéder à des adresses au-delà de 65535 (`0xffff`) ? La réponse tient en trois

lettres : MMU, pour *Memory Management Unit* ou unité de gestion mémoire, en bon français.

Le travail de la MMU (ici et de l'époque des Z80/Z180) est de gérer un espace d'adressage « virtuel » visible par le code et, en coulisse, de traduire cela en adresse « réelle » sur le bus. On parle alors d'adresses logiques pour ce qui est visible par le code et d'adresses physiques pour ce qui concerne matériellement les mémoires. L'astuce consiste à définir différentes zones, en mémoire physique et de les faire apparaître dans l'espace d'adressage logique comme étant une unique mémoire, ici de 64 Kio. La tâche de la MMU est de s'occuper non seulement de la traduction des adresses à la volée, mais également de permettre de changer ce « mapping » à souhait.

Imaginons, hypothétiquement, que notre division par deux soit configurée. Nous pourrions avoir la ROM de `0x0000` à `0x7fff` et **un morceau de RAM** de `0x8000` à `0xffff`, qui correspondrait à la plage physique de `0x20000` à `0x27fff`. Notre code pourrait alors placer 32 Kio de données à cet endroit, puis modifier la configuration de la MMU pour que cette traduction d'adresses change. Ce faisant, la même zone logique `0x8000` à `0xffff` correspondrait alors à, par exemple, la plage physique de `0x30000` à `0x37fff`, laissant les données de `0x20000` à `0x27fff` parfaitement intactes et utilisables par la suite, après une nouvelle configuration de la MMU.

Ce mécanisme, appelé *banking*, consistant à diviser la mémoire physique en zones, ou banques (*banks*), de tailles fixes et permutables au cours de la vie d'un programme, est celui utilisé par la MMU du Z180. Notez qu'il est parfaitement possible de faire exactement la même chose avec un Z80, en utilisant un périphérique externe pour basculer d'une zone à une autre ou d'un composant RAM à un autre. Avec le Z180, tout est intégré et nécessite une configuration, car il n'est pas possible de désactiver cette fonctionnalité.

« Pourquoi nos deux précédents codes fonctionnent-ils ? », me demanderez-vous. Tout simplement parce que la configuration par défaut, appliquée au *reset* du Z180, ainsi que certaines contraintes fixées par le constructeur le permettent. Pour le comprendre, nous devons savoir comment marche la MMU du Z180 : retroussage de manches et lecture de la *datasheet* !

1. PLUS EN PROFONDEUR DANS LA MMU DU Z180

Faisons l'impasse ici sur l'histoire de cette MMU directement héritée de l'Hitachi 64180, si ce n'est en mentionnant qu'elle est relativement simple, par rapport à d'autres, existant en composants distincts accompagnant, par exemple, le Motorola 68000. Elle fonctionne exactement comme je viens de le décrire dans mon exemple hypothétique, à quelques nuances près.

Nous avons donc deux espaces d'adresses liés entre eux, un logique et un physique (en plus des ports E/S), et ils se divisent en trois parties ou sections, qui apparaissent **toujours** dans cet ordre **dans l'espace d'adresses logiques** :

- « *Common 0* » débute toujours à `0x0000` et est toujours traduite (ou « mappée ») au début de la mémoire physique (`0x00000`). Sa taille est fixée par une valeur de 4 bits appelée *BA*, correspondant à la moitié basse du registre *CBAR*

(port E/S `0x3a`), multiplié par 4 Kio (ou `0x1000`). Au *reset*, cette valeur est `0x0` et cette section fera donc, par défaut `0x0` fois `0x1000`, soit 0 octet.

- « *Bank* » (ou *Banked*, *Bank area* ou *Banked area*) débute à *BA* * `0x1000` et s'étend jusqu'à *CA* * `0x1000`. *CA* est l'autre moitié du registre *CBAR*, les 4 bits de poids fort, et la valeur par défaut au *reset* est `0xf` (*CBAR* = `0xf0`). Ceci signifie donc que, par défaut, la taille de cette section est de (*CA* - *BA*) * `0x1000`, et donc `0xf000`.
- « *Common 1* » (parfois appelé juste *Common*) débute à *CA* * `0x1000` et s'étend jusqu'à l'adresse logique la plus haute, soit `0xffff`. Au *reset*, comme *CA* vaut `0xf`, nous avons donc (`0x10` - `0xf`) * `0x1000`, soit `0x1000`.

Nous pouvons choisir n'importe quelle valeur à mettre dans le registre *CBAR* à condition que *CA* soit toujours supérieur à *BA*, de manière à ce que *Bank* apparaisse **avant** *Common 1* (autrement dit, *Common 1* doit **toujours** débuter **après** *Bank*).

Voilà pour ce qui est de l'espace d'adressage logique et de la taille des trois sections. Celles-ci seront, bien entendu, identiques en espace d'adressage physique, mais nous pouvons placer *Bank* et *Common 1* où cela nous chante. « *Common 0* », en revanche, et comme dit précédemment, débute **toujours** à `0x00000` en adressage physique.

Pour placer les deux sections, deux autres registres sont à notre disposition, *BBR* (*Bank Base Register*, `0x39`) et *CBR* (*Common Base Register*, `0x38`). Notez au passage que j'utilise le mot « registre » au sens large du terme, il s'agit de ports E/S accessibles via `out0` et `in0`, comme les registres des ASCI vus la fois précédente. Ces deux registres reposent sur le même mode de calcul que celui que nous venons de voir, en multiple de `0x1000` (4 Kio) :

- *BBR* contient la valeur à ajouter à *BA* avant d'être multiplié par `0x1000`. Le calcul donne l'adresse physique à laquelle débute la section *Bank*. Au *reset*, *BBR* (comme *CBR*) est à `0x00`. De ce fait, l'adresse physique par défaut du début de *Bank* est (`0x00` + `0x0`) * `0x1000` = `0x00000`. Ici, on retombe donc sur nos pieds puisque l'EEPROM est précisément à cet endroit, physiquement.
- *CBR* est la valeur à ajouter à *CA* avant de multiplier par `0x1000` pour obtenir l'adresse de début de *Common 1* en mémoire physique. Par défaut, nous avons donc (`0x00` + `0xf`) * `0x1000` = `0x0f000`.

Au final, notre code précédent fonctionne, car les adresses logiques de `0x0000` à `0xffff` se retrouvent forcément aux adresses physiques `0x00000` à `0x0ffff`, qui matériellement correspondent à l'EEPROM (si elle était de 64 Kio).

Une autre manière de voir les choses, et de traduire une adresse logique en adresse physique, est :

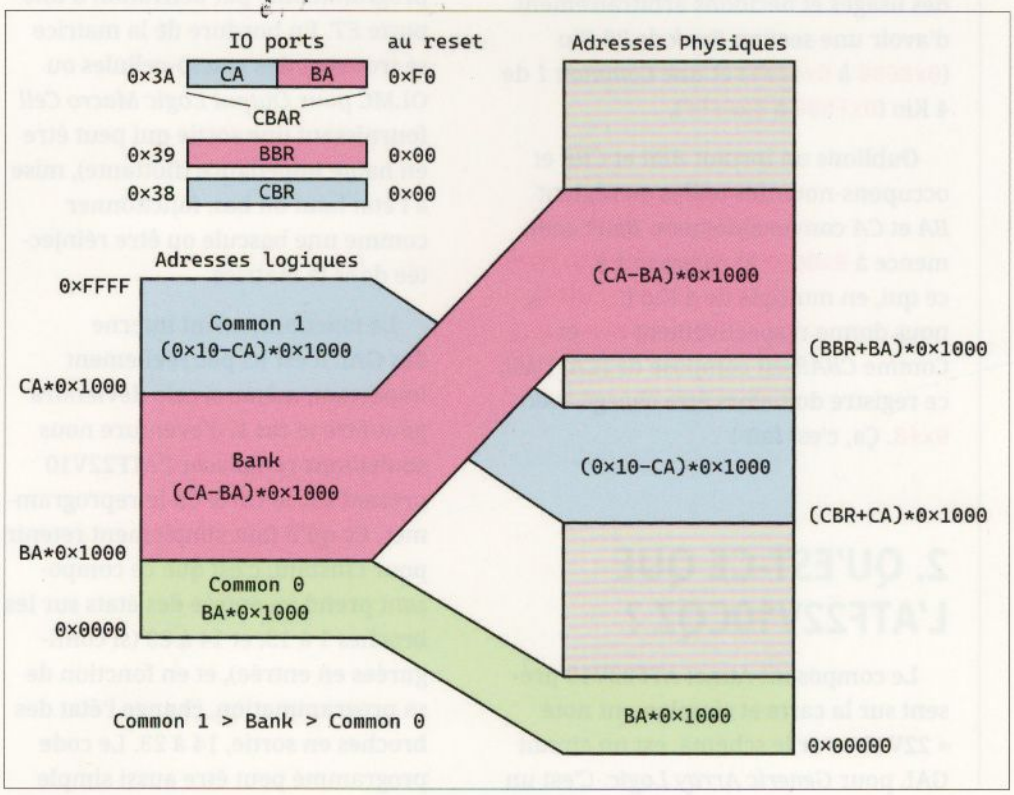
- si l'adresse logique est supérieure à *BA* * `0x1000` et inférieure à *CA* * `0x1000`, alors l'adresse physique correspondante est l'adresse logique plus *BBR* multiplié par `0x1000` (4 Kio) ;
- si l'adresse logique est supérieure à *CA* * `0x1000`, alors l'adresse physique sera l'adresse logique plus *CBR* fois `0x1000`.

Attention : les documentations sont parfois trompeuses (erronées) et les quelques articles du Web comparant l'Hitachi 64180 au Z180 ajoutent à la confusion, laissant entendre que les valeurs après *reset* pourraient être différentes entre les deux processeurs. Le manuel du Z180 **[1]**, page 60, indique bien une valeur au *reset* pour *CBAR* de `11110000`, soit *CA* à `0xf` et *BA* à `0x0`. Mais le PDF

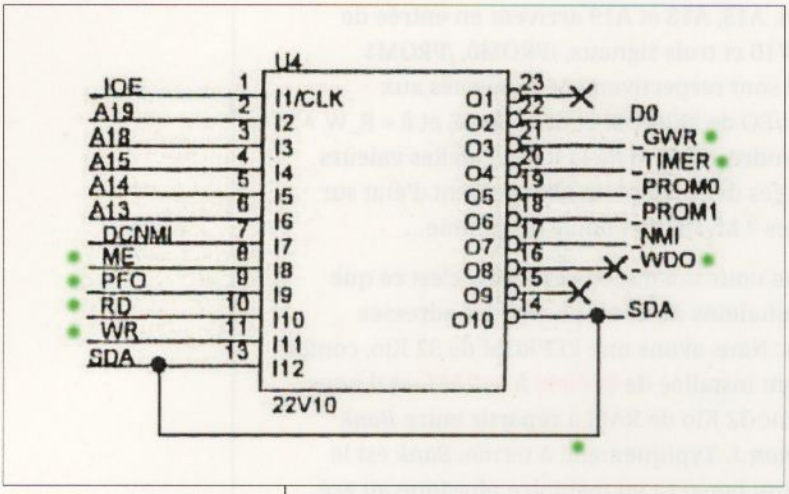
« *80180 Microprocessor Unit Product Specification* » **[2]**, page 72, indique « *All bits of CA are set to 1 during RESET* », mais surtout « *All bits of BA are set to 1 during RESET* », **ce qui est totalement faux** !

En pratique, qu'est-ce que cela donne dans notre cas ? Ne sachant pas encore exactement comment est configurée la RAM et sont traitées matériellement les valeurs sur le bus d'adresses, il est difficile de répondre concernant la mémoire physique. À ce stade, tout ce que nous savons c'est que les lignes A13, A14, A15, A18 et A19 arrivent en entrée de l'ATF22V10 et trois signaux, /PROM0, /PROM1 et /GWR sont respectivement connectés aux broches /EO de l'EPROM et de la RAM, et à « R_W » (comprendre « /WE ») de la RAM. Quelles valeurs des entrées définissent un changement d'état sur les sorties ? Mystère et boule de gomme...

Ce que nous savons, en revanche, c'est ce que nous souhaitons avoir au niveau des adresses logiques. Nous avons une EEPROM de 32 Kio, confortablement installée de `0x0000` à `0x7fff`, et il nous reste donc 32 Kio de RAM à répartir entre *Bank* et *Common 1*. Typiquement, à terme, *Bank* est la section qui bougera en mémoire physique au gré des besoins. C'est d'ailleurs pour cette raison que *Common 1* porte ce nom et non *Bank 1* (*Bank* étant alors *Bank 0*). *Common 0* est utilisé pour le code initial et les routines courantes, *Bank* pour le code susceptible de changer (charger/décharger) et *Common 1* pour les « données communes ». Il est, semble-t-il,



Résumé graphique du fonctionnement de la MMU du Zilog Z180 et des registres associés.



Voici le schéma corrigé (points verts) du brochage effectif du composant GAL ATF22V10CQZ tel qu'il est connecté sur le circuit imprimé. Cette erreur étrange est à l'origine d'une perte de temps importante, mais aussi d'une bonne leçon : ne pas croire aveuglément les schémas et toujours vérifier.

généralement conseillé de garder *Bank* à une taille minimum, pour avoir davantage d'espace pour les données. Mais nous sommes loin d'être à ce niveau de décision et souhaitons, avant tout, que cela fonctionne. Nous faisons donc fi des usages et décidons arbitrairement d'avoir une section *Bank* de 28 Kio (`0x8000` à `0xefff`) et une *Common 1* de 4 Kio (`0xf000` à `0xffff`).

Oublions un instant *BBR* et *CBR* et occupons-nous des tailles en réglant *BA* et *CA* convenablement. *Bank* commence à `0x8000` et *common 1* à `0xf000` ce qui, en multiple de 4 Kio (`0x1000`), nous donne respectivement `0x8` et `0xf`. Comme *CBAR* est composé de [*CA*] [*BA*], ce registre doit alors être chargé avec `0xf8`. Ça, c'est fait !

2. QU'EST-CE QUE L'ATF22V10CQZ ?

Le composant Atmel ATF22V10 présent sur la carte et simplement noté « 22V10 » sur le schéma, est un circuit GAL pour *Generic Array Logic*. C'est un composant à logique programmable successeur des PAL (*Programmable Array Logic*) et prédécesseur des FPGA. Son principal intérêt est de permettre de construire un circuit logique dans le composant lui-même et d'ainsi remplacer des portes *ET*, *OU*, *NON-ET*, *XOR*, etc., prenant physiquement la forme de circuits intégrés de type 74* ou de la série 4000.

Les GAL sont reprogrammables et on utilise habituellement un langage de description pour détailler ce qu'on souhaite obtenir. Cette « source » est ensuite assemblée pour produire un fichier JEDEC que l'on peut utiliser pour programmer la mémoire du composant. En interne, les GAL contiennent une matrice où chaque intersection entre une ligne verticale et horizontale est programmable par activation d'une porte *ET*. En bordure de la matrice se trouvent des macro-cellules ou OLMC pour *Output Logic Macro Cell* fournissant une sortie qui peut être en haute impédance (flottante), mise à l'état haut ou bas, fonctionner comme une bascule ou être réinjectée dans la matrice.

Le fonctionnement interne des GAL n'est ici pas réellement important, même si cela deviendra peut-être le cas si d'aventure nous souhaitons remplacer l'ATF22V10 présent sur la carte ou le reprogrammer. Ce qu'il faut simplement retenir pour l'instant, c'est que ce composant prend en entrée des états sur les broches 1 à 13, et 14 à 23 (si configurées en entrée), et en fonction de sa programmation, change l'état des broches en sortie, 14 à 23. Le code programmé peut être aussi simple

qu'un « *sortie = (entrée1 OU entrée2) ET (entrée3 ET NON-entrée4)* » ce qui, littéralement, en code donnerait quelque chose comme : `OUT = (IN1 + IN2) * (IN3 * /IN4)` après assignation des noms à différentes broches du composant (si le sujet vous intéresse, jeter un œil à GALasm **[3]** et/ou Galette **[4]**).

Si vous disposez d'un programmeur d'EEPROM TL866CS, par exemple, la bonne nouvelle c'est que vous pouvez parfaitement l'utiliser pour programmer des GAL et PAL, ou lire leur contenu. La mauvaise nouvelle, c'est que les GAL disposent d'un bit de sécurité empêchant leur lecture et que ce bit ne peut être réinitialisé qu'en effaçant le composant. Dans notre cas, bien entendu, le bit est activé et nous ne pouvons donc pas lire la configuration pour, éventuellement, en déduire la logique interne qui y est programmée.

Mais le schéma nous permet de déduire plusieurs choses :

- La broche 1 n'est certainement pas utilisée comme un signal d'horloge permettant d'utiliser l'ATF22V10 comme un *latch* (l'un des modes de fonctionnement des GAL), puisque celle-ci est connectée à /IORQ sur le Z180, qui indique un accès E/S par opposition à /MREQ (accès mémoire).
- Les broches 18, 19 et 20, respectivement /PROM0, /PROM1 et /GWR (*General WRite* ?), sont forcément des sorties, puisque connectées à des entrées sur la ROM et la RAM.
- Les broches 2 à 13 étant forcément des entrées, les états de A13, A14, A15, A18 et A19 sont surveillés et utilisés pour définir une action en sortie. Il en va de même pour /RD, /WR, /MREQ (« /ME »), etc.
- Une partie des entrées est reliée aux sorties du MAX691 (/PFO et /WDO) et la broche 17 est connectée à /NMI (*Non-Maskable Interrupt*), une entrée du Z180 provoquant une interruption non masquable. 17 est donc une sortie, très certainement utilisée pour signifier au processeur une rupture d'alimentation ou un déclenchement du chien de garde.

On peut supposer, en toute logique, que les lignes d'adresses en entrée, accompagnées des signaux /RD, /WR, /MREQ et /IORQ servent à définir l'état de /PROM0, /PROM1 et /GWR, et donc de sélectionner la ROM ou la RAM en fonction de l'adresse utilisée. Notez d'ailleurs un point intéressant du schéma qui est assez surprenant : les broches /CE (*Chip Enable*) de la ROM et de la RAM sont directement connectées à la masse, activant en permanence les deux mémoires. Généralement, ces lignes sont liées à /MREQ via des portes logiques puisque les composants sont accédés par une requête mémoire et c'est /RD ou /WR qui impacte directement /OE (*output Enable*) et /WE (*Write Enable*) pour la lecture ou l'écriture. Ici, l'ATF22V10 semble combiner les entrées et uniquement impacter /OE et /WE (alias /WR), ce n'est pas ce qu'il y a de plus courant...

3. TESTONS LA LOGIQUE DE L'ATF22V10

Pour tester la logique interne du composant sans avoir accès à sa mémoire, nous pouvons le mettre à l'épreuve. En d'autres termes, nous pouvons nous amuser à présenter différentes combinaisons d'état sur les broches qui nous intéressent et observer l'état en sortie sur /PROM0, /PROM1 et /GWR. Naïvement, j'ai commencé à faire cela en plaçant l'ATF22V10 sur une

platine à essais et en y connectant quelques entrées à Vcc ou à la masse via des résistances de 10 kΩ. La tâche est fastidieuse et les résultats incohérents, les trois signaux en sortie étaient toujours à l'état haut. Tout ceci n'avait aucun sens.

Mais comme le dirait l'agent Smith, « *n'envoyez jamais un humain faire le travail d'une machine* », qui ici se traduit en « *voilà un boulot pour une carte Arduino* », car 9 entrées pouvant avoir deux états nous donnent littéralement 2^9 tests, soit 512. L'agent Smith a donc raison.

Et voici donc ce que la « machine » doit faire :

```
#define GAL_RD    3   // orange
#define GAL_RW    2   // jaune
#define GAL_ME    4   // brun
#define GAL_IOE   10  // rouge

#define GAL_A13   5   // gris
#define GAL_A14   6   // violet
#define GAL_A15   7   // vert
#define GAL_A18   8   // blanc
#define GAL_A19   9   // bleu

#define GAL_GWR   A0  // vert
#define GAL_PROM0 A1  // jaune
#define GAL_PROM1 A2  // bleu

void setup() {
  pinMode(GAL_RD, OUTPUT);
  pinMode(GAL_RW, OUTPUT);
  pinMode(GAL_ME, OUTPUT);
  pinMode(GAL_A13, OUTPUT);
  pinMode(GAL_A14, OUTPUT);
  pinMode(GAL_A15, OUTPUT);
  pinMode(GAL_A18, OUTPUT);
  pinMode(GAL_A19, OUTPUT);
  pinMode(GAL_IOE, OUTPUT);

  pinMode(GAL_GWR, INPUT);
  pinMode(GAL_PROM0, INPUT);
  pinMode(GAL_PROM1, INPUT);

  digitalWrite(GAL_A13, LOW);
  digitalWrite(GAL_A14, LOW);
  digitalWrite(GAL_A15, LOW);
  digitalWrite(GAL_A18, LOW);
  digitalWrite(GAL_A19, LOW);

  digitalWrite(GAL_IOE, HIGH);
  digitalWrite(GAL_RW, HIGH);
  digitalWrite(GAL_RD, HIGH);

  Serial.begin(115200);
}

void test() {
  if (digitalRead(GAL_PROM0))
    Serial.print(" OE_ROM  ");
  else
    Serial.print("/OE_ROM  ");
  if (digitalRead(GAL_PROM1))
    Serial.print(" OE_RAM  ");
  else
    Serial.print("/OE_RAM  ");
  if (digitalRead(GAL_GWR))
    Serial.println(" WE_RAM");
  else
    Serial.println("/WE_RAM");
}

//  xx xx xx A19 A18 A15 A14 A13
// 128 64 32  16   8   4   2   1
void pattern(int val) {
  if ( val & 16) {
    digitalWrite(GAL_A19, HIGH);
    Serial.print(" 1  ");
  } else {
    digitalWrite(GAL_A19, LOW);
    Serial.print(" 0  ");
  }
  if ( val & 8)  {
    digitalWrite(GAL_A18, HIGH);
    Serial.print(" 1  ");
  } else {
    digitalWrite(GAL_A18, LOW);
    Serial.print(" 0  ");
  }
  if ( val & 4)  {
    digitalWrite(GAL_A15, HIGH);
```

```
    Serial.print(" 1  ");
  } else {
    digitalWrite(GAL_A15, LOW);
    Serial.print(" 0  ");
  }
  if ( val & 2)  {
    digitalWrite(GAL_A14, HIGH);
    Serial.print(" 1  ");
  } else {
    digitalWrite(GAL_A14, LOW);
    Serial.print(" 0  ");
  }
  if ( val & 1)  {
    digitalWrite(GAL_A13, HIGH);
    Serial.print(" 1  ");
  } else {
    digitalWrite(GAL_A13, LOW);
    Serial.print(" 0  ");
  }
  Serial.print("   ");
}

void dopattern() {
  Serial.println("A19 A18 A15 A14 A13");
  for (int i=0; i<=31; i++) {
    pattern(i);
    test();
    delay(10);
  }
}

void loop() {
  digitalWrite(GAL_IOE, HIGH);
  digitalWrite(GAL_ME, LOW);
  digitalWrite(GAL_RW, HIGH);
  digitalWrite(GAL_RD, HIGH);
  dopattern();
  Serial.println("-----------");
  delay(1000);
}
```

L'exécution de ce code avec l'Arduino dûment connecté à l'ATF22V10 nous donne... absolument n'importe quoi ! /WE_RAM (alias /GWR) s'active lors d'une lecture, /OE_ROM est actif lors d'une écriture et la plupart du temps, toutes les sorties sont à l'état haut, exactement comme avec le test manuel. Quelque chose ne va vraiment pas. Aurais-je grillé le composant ? Non, la carte fonctionne et exécute le code lorsqu'il est en place. Comment est-ce possible avec un /OE_ROM qui n'est jamais à la masse ?

La réponse est simple : le schéma est faux. Vraiment, vraiment faux, car :

- les entrées /MREQ (9) et /PFO (8) sont inversées ;
- /RD (11) et /WR (10) sont inversées ;
- et en sortie, /GWR (20) et /TIMER (21) sont inversées.

Ceci s'est révélé alors que je vérifiais, matériellement au testeur, où allait effectivement /GWR. Et après cette correction, c'est /RD qui activait /GWS, etc., ainsi de suite. La réelle nomenclature des broches de l'ATF22V10 est, semble-t-il (certains signaux ne concernent ni le CPU ni la ROM ou la RAM) :

```
/IORQ   1       24  Vcc
   A19  2       23  -
   A18  3       22  D0
   A15  4       21  /GWR
   A14  5       20  /TIMER
   A13  6       19  /PROM0
/DCNMI  7       18  /PROM1
/MREQ   8       17  /NMI
  /PFO  9       16  /WDO (patte coupée)
   /RD  10      15  -
   /WR  11      14  SDA
Masse   12      13  SDA
```

Une fois ceci remis dans le bon ordre, le test prend tout son sens, que ce soit en lecture :

```
A19 A18 A15 A14 A13
0   0   0   0   0     /OE_ROM   OE_RAM   WE_RAM
0   0   0   0   1     /OE_ROM   OE_RAM   WE_RAM
0   0   0   1   0     /OE_ROM   OE_RAM   WE_RAM
0   0   0   1   1     /OE_ROM   OE_RAM   WE_RAM
0   0   1   0   0     /OE_ROM   OE_RAM   WE_RAM
0   0   1   0   1     /OE_ROM   OE_RAM   WE_RAM
0   0   1   1   0     /OE_ROM   OE_RAM   WE_RAM
0   0   1   1   1     /OE_ROM   OE_RAM   WE_RAM
0   1   0   0   0      OE_ROM  /OE_RAM   WE_RAM
0   1   0   0   1      OE_ROM  /OE_RAM   WE_RAM
0   1   0   1   0      OE_ROM  /OE_RAM   WE_RAM
0   1   0   1   1      OE_ROM  /OE_RAM   WE_RAM
0   1   1   0   0      OE_ROM  /OE_RAM   WE_RAM
0   1   1   0   1      OE_ROM  /OE_RAM   WE_RAM
0   1   1   1   0      OE_ROM  /OE_RAM   WE_RAM
0   1   1   1   1      OE_ROM  /OE_RAM   WE_RAM
1   0   0   0   0      OE_ROM  /OE_RAM   WE_RAM
1   0   0   0   1      OE_ROM  /OE_RAM   WE_RAM
1   0   0   1   0      OE_ROM  /OE_RAM   WE_RAM
1   0   0   1   1      OE_ROM  /OE_RAM   WE_RAM
1   0   1   0   0      OE_ROM  /OE_RAM   WE_RAM
```

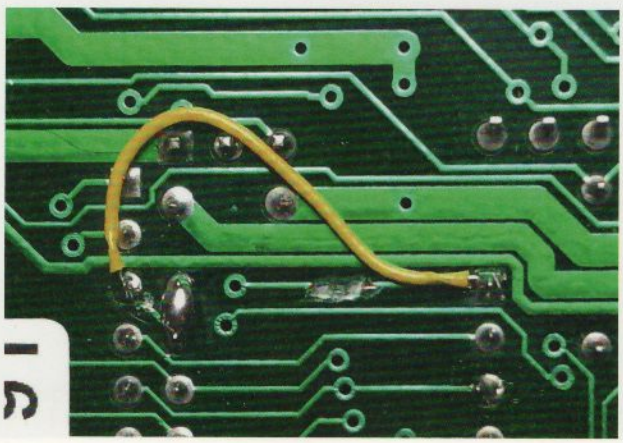
```
1   0   1   0   1       OE_ROM   /OE_RAM   WE_RAM
1   0   1   1   0       OE_ROM   /OE_RAM   WE_RAM
1   0   1   1   1       OE_ROM   /OE_RAM   WE_RAM
1   1   0   0   0      /OE_ROM    OE_RAM   WE_RAM
1   1   0   0   1      /OE_ROM    OE_RAM   WE_RAM
1   1   0   1   0      /OE_ROM    OE_RAM   WE_RAM
1   1   0   1   1      /OE_ROM    OE_RAM   WE_RAM
1   1   1   0   0      /OE_ROM    OE_RAM   WE_RAM
1   1   1   0   1      /OE_ROM    OE_RAM   WE_RAM
1   1   1   1   0      /OE_ROM    OE_RAM   WE_RAM
1   1   1   1   1      /OE_ROM    OE_RAM   WE_RAM
```

Ou en écriture :

```
A19 A18 A15 A14 A13
0   0   0   0   0       OE_ROM   OE_RAM   WE_RAM
0   0   0   0   1       OE_ROM   OE_RAM   WE_RAM
0   0   0   1   0       OE_ROM   OE_RAM   WE_RAM
0   0   0   1   1       OE_ROM   OE_RAM   WE_RAM
0   0   1   0   0       OE_ROM   OE_RAM   WE_RAM
0   0   1   0   1       OE_ROM   OE_RAM   WE_RAM
0   0   1   1   0       OE_ROM   OE_RAM   WE_RAM
0   0   1   1   1       OE_ROM   OE_RAM   WE_RAM
0   1   0   0   0       OE_ROM   OE_RAM  /WE_RAM
0   1   0   0   1       OE_ROM   OE_RAM  /WE_RAM
0   1   0   1   0       OE_ROM   OE_RAM  /WE_RAM
0   1   0   1   1       OE_ROM   OE_RAM  /WE_RAM
0   1   1   0   0       OE_ROM   OE_RAM  /WE_RAM
0   1   1   0   1       OE_ROM   OE_RAM  /WE_RAM
0   1   1   1   0       OE_ROM   OE_RAM  /WE_RAM
0   1   1   1   1       OE_ROM   OE_RAM  /WE_RAM
1   0   0   0   0       OE_ROM   OE_RAM  /WE_RAM
1   0   0   0   1       OE_ROM   OE_RAM  /WE_RAM
1   0   0   1   0       OE_ROM   OE_RAM  /WE_RAM
1   0   0   1   1       OE_ROM   OE_RAM  /WE_RAM
1   0   1   0   0       OE_ROM   OE_RAM  /WE_RAM
1   0   1   0   1       OE_ROM   OE_RAM  /WE_RAM
1   0   1   1   0       OE_ROM   OE_RAM  /WE_RAM
1   0   1   1   1       OE_ROM   OE_RAM  /WE_RAM
1   1   0   0   0       OE_ROM   OE_RAM   WE_RAM
1   1   0   0   1       OE_ROM   OE_RAM   WE_RAM
1   1   0   1   0       OE_ROM   OE_RAM   WE_RAM
1   1   0   1   1       OE_ROM   OE_RAM   WE_RAM
1   1   1   0   0       OE_ROM   OE_RAM   WE_RAM
1   1   1   0   1       OE_ROM   OE_RAM   WE_RAM
1   1   1   1   0       OE_ROM   OE_RAM   WE_RAM
1   1   1   1   1       OE_ROM   OE_RAM   WE_RAM
```

Dans les deux cas, la RAM est accédée en lecture ou en écriture lorsque A18 **ou** A19 est à l'état haut et donc que le bit d'adresse 18 **ou** 19 est à 1. Mais pas 18 **et** 19. Ceci est précisément l'information dont nous avions besoin et cela signifie que notre RAM s'active à partir de l'adresse `01000000000000000000` en binaire et donc `0x40000` en hexadécimal, soit à 256 Kio du début de l'espace d'adressage physique. Le cavalier WT1 sur la carte met la broche A17 de la RAM à la tension d'alimentation (VRAM et non Vcc, car gérée par le MAX691) ou la connecte sur A17 du bus (et donc du processeur). Ceci, car une RAM Toshiba TC551001CP de 128 Kio (131072 mots de 8 bits), comme l'indique le schéma, dispose en réalité d'une broche /CE2 à cet endroit, qui correspond effectivement à A17 sur une RAM de taille plus importante. Ce cavalier sert donc effectivement à activer une RAM de 256 Kio (ou plus), malgré le fait que c'est une BSI BS62LV4006 de 512 Kio qui est présente sur la carte.

Comme A18 et A19, ensemble à l'état haut, n'activent pas le /OE de la RAM, ceci signifie que l'adresse `10111111111111111111`, ou `0xbffff` est la plus haute possible pour activer cette mémoire. Ceci se trouve à `0x7ffff` du début de l'espace d'adresses, soit 512 Kio. Il semble donc parfaitement possible d'utiliser les 512 Kio du composant, mais avec une organisation un peu particulière. En effet, utiliser une adresse entre `0x40000` et `0x7ffff` nous donne accès à la seconde moitié de cette quantité de mémoire, et faire de même avec une adresse entre `0x80000` et `0xbffff` accède en réalité aux premiers 256 Kio du composant. Voilà donc pourquoi une BS62LV4006 est insérée sur la carte alors que le cavalier est libellé « 128K 256K » et que le schéma précise une TC551001. Quoi qu'il en soit, 128 Kio sont amplement suffisants et je vais donc m'en tenir à mes TC551001BPL en stock, les BS62LV4006 de 512 Kio pouvant servir à un autre projet (par exemple à base de Motorola 68008, qui est un CPU d'un tout autre calibre).



Faire en sorte que la carte accepte une EEPROM 28C256 à la place d'une 27C256 se résume à couper deux pistes et à ajouter une connexion. Tout cela pour simplement inverser deux broches de l'emplacement.

L'utilisation des lignes A13 à A15, cependant, reste mystérieuse et leurs états n'ont aucun impact sur les trois sorties, que ce soit avec /MREQ ou /IORQ. Il en va de même pour D0 qui peut être une entrée comme une sortie et dont l'usage nous est, à ce stade, inconnu.

Mais ce n'est pas grave, car nous avons maintenant les derniers éléments nécessaires pour la configuration de la MMU. Nous savons où commence la RAM, à `0x40000`, et qu'en restant conservateur pour commencer, nous pouvons partir du principe qu'elle s'étend jusqu'à `0x5ffff` (128 Kio). Comme nous avons décidé de définir *CBAR* à `0xf8` pour 28 Kio de *Bank* et 4 Kio de *Common 1*, nous avons les valeurs *BA* et *CA* qui, combinées respectivement à

BBR et *CBR* nous permettront de positionner ces espaces en mémoire physique. Pour être un peu originaux, nous allons placer *Common 1* avant *Bank*, puisque cela est possible en mémoire physique. Nous voulons donc que *Common 1* débute en `0x40000` et *Bank* à `0x41000`. Le fait que cette section se trouve précisément à 4 Kio après le début de *Common 1* n'est absolument pas une nécessité, nous pourrions parfaitement utiliser `0x42000`, `0x42000`, `0x50000`, etc., du moment que les 28 Kio ne « dépassent » pas de la mémoire effectivement disponible.

Comme (*BBR* + *BA*) * `0x1000` nous donne l'adresse, et que *BA* vaut `0x8`, nous devons mettre `0x39` dans *BBR*. De manière similaire, nous avons (*CBR* + *CA*) * `0x1000` pour l'adresse physique de début de *Common 1* et donc un *CBR* à `0x31`. Youpi, nous avons notre configuration !

Voilà pour la théorie, maintenant vérifions que mademoiselle la pratique valide bien l'approche (*Pratique* qui, de ce qu'on m'a dit, est la belle-sœur de *Prudence*, elle-même mère de *Sûreté*).

4. TESTONS L'ACCÈS À LA RAM

Il est temps de faire un peu d'assembleur, et cela risque d'être toujours très spartiate. En effet, ne sachant pas si la RAM fonctionne et si notre compréhension de la MMU est correcte, nous allons devoir continuer à exécuter notre code entièrement en ROM. Ceci signifie « pas de pile » et donc toujours pas de sous-routine (`call`/`ret`).

La démarche cependant sera relativement simple : une fois la MMU configurée, nous allons écrire une valeur 8 bits arbitraire quelque part en RAM (donc après `0x8000`), puis lire à nouveau cet emplacement. Si nous obtenons la même valeur, c'est que la RAM est présente et inscriptible.

Notre code débute comme le précédent :

```
        .module uarttest
        .z180

; ASCI Registers port 0 and 1
CNTLA0  .equ    0x00    ; ASCI Channel Control Register A 0
CNTLA1  .equ    0x01    ; ASCI Channel Control Register A 1
CNTLB0  .equ    0x02    ; ASCI Control Register B 0
CNTLB1  .equ    0x03    ; ASCI Control Register B 1
STAT0   .equ    0x04    ; ASCI Status Register 0
STAT1   .equ    0x05    ; ASCI Status Register 1
TDR0    .equ    0x06    ; ASCI Transmit Data Register 0
TDR1    .equ    0x07    ; ASCI Transmit Data Register 1
RDR0    .equ    0x08    ; ASCI Receive Data FIFO 0
RDR1    .equ    0x09    ; ASCI Receive Data FIFO 1
CBR     .equ    0x38    ; MMU Common Base Register
BBR     .equ    0x39    ; MMU Bank Base Register
CBAR    .equ    0x3a    ; MMU Common/Bank Area Register
ICR     .equ    0x3f    ; I/O Control Register
```

```
        .area   _HEADER (ABS)
        ;; Reset vector
        .org    0x0
        jp      init

        .org    0x100
init:
        ; Pour le linker
        .area   _HOME
        .area   _CODE
        .area   _INITIALIZER
        .area   _GSINIT
        .area   _GSFINAL

        .area   _DATA
        .area   _INITIALIZED
        .area   _BSEG
        .area   _BSS
        .area   _HEAP

        .area   _CODE
```

Nous avons cependant ajouté les symboles pour les registres des ports CBAR, BBR et CBR. Sans attendre, nous configurons la MMU avec les valeurs précédemment calculées :

```
mmuinit:
        ld      a,#0xf8
        out0    (CBAR),a

        ld      a,#0x31
        out0    (CBR),a

        ld      a,#0x39
        out0    (BBR),a
```

Puis nous configurons l'ASCI0 (sortie série), exactement comme la dernière fois :

```
asci0init:
        ld      a,#0x67 ; 8O2
        out0    (CNTLA0),a

        ld      a,#0x21
        out0    (CNTLB0),a
```

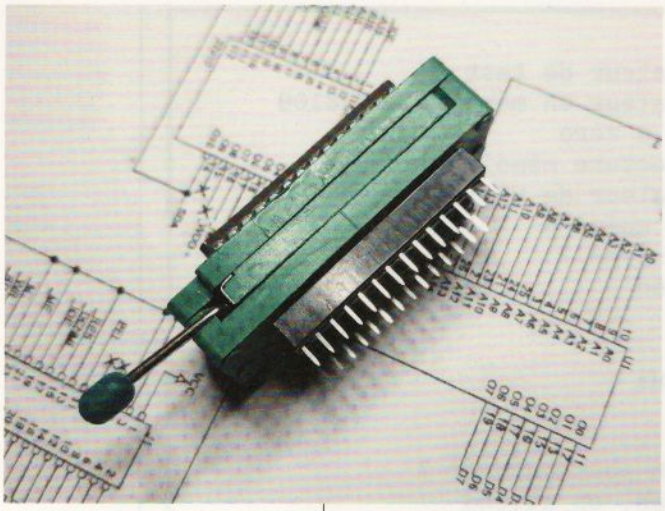
Ceci nous amène enfin au test à proprement parler. Nous utilisons l'adresse `0x8100` comme point de vérification. Si notre configuration est correcte, cette adresse logique de *Bank* est traduite en *0x42000* en mémoire physique, et donc en RAM.

```
main:
        ld      a,#0x42         ; valeur de test
        ld      (0x8100),a      ; valeur en mémoire à 0x8100
        ld      a,#0x00         ; A à zéro
        ld      a,(0x8100)      ; lecture mémoire à 0x8100
        ld      b,#0x42         ; valeur de vérification
        sub     b               ; A moins B
        jp      z,ok            ; A-B==0 ?
        ld      hl,#textpasok   ; non
        jp      asci0putc
ok:
        ld      hl,#textok      ; oui
asci0putc:
        in0     a,(STAT0)
        bit     1,a             ; On peut envoyer ?
        jr      z,asci0putc     ; non, on attend
        ld      a,(hl)          ; oui
        inc     a
        dec     a
        jp      z,main          ; c'était \0
        out0    (TDR0),a        ; A!=0 donc envoi
        inc     hl              ; caractère suivant
```

Nous jonglons ici avec deux chaînes de caractères, mais le principe est strictement identique à ce que nous avons développé pour l'article précédent. Nous envoyons, via l'ASCI0, l'une ou l'autre chaîne en fonction de l'égalité entre la valeur lue à `0x8100` (dans *A*) et notre valeur test placée dans *B*. Le reste du code est constitué de la boucle d'attente déjà connue et des chaînes à utiliser :

```
delay:
        ld      de,#0x2000      ; init compteur
Inner:
        dec     de              ; compteur--
        ld      a,d
        or      e
        jp      nz,Inner        ; zéro ?

        jp      asci0putc       ; loop

textok:
        .ascii  "RAM works.\r\n"
        .db     0x00
textpasok:
        .ascii  "RAM doesn't works !!!\r\n"
        .db     0x00
```

Le `Makefile` est strictement identique au précédent et une fois le code assemblé, « linké » et programmé en EEPROM, la carte Z180 nous dit « RAM works. » à répétition. Nous pouvons réitérer le test avec différentes adresses, en *Bank* ou en *Commun 1* et obtenons toujours ce même fantastique et plaisant résultat. Rhaaaa... dopamine...



Modifier la carte pour accepter des 28C256 (ou 28C512) nous permet de simplifier notre « adaptateur » ZIF. Il suffit maintenant de le surélever légèrement en intercalant un emplacement à souder DIP-28, et le tour est joué.

5. ET SI ON FAISAIT UN PEU DE C ? PAS TOUT DE SUITE

Oui, nous pourrions, puisque nous avons maintenant tout le nécessaire pour le faire. Mais ne nous précipitons pas, voulez-vous. Ce que je vous propose, en revanche, est de garder cela pour plus tard et plutôt revoir notre code, toujours en assembleur, de façon à changer la valeur de *BBR* pour faire « glisser » *Bank* au travers de la mémoire physique et donc faire quelque chose qui se rapproche du *banking* tel qu'il pourrait être utilisé dans un vrai code utilisable.

Comme nous savons que la RAM fonctionne, nous allons utiliser *Common 1*, qui se trouve tout en haut de l'espace d'adresses logiques, pour **enfin** avoir une pile (*stack*) et pouvoir faire appel aux instructions `call` et `ret` pour utiliser des sous-routines. De ce fait, juste après la configuration initiale de la MMU, qui reste ici identique, nous ajoutons une simple ligne :

```
ld      sp,#0xffff
```

Nous chargeons dans le registre *SP* (*Stack Pointer*) la valeur littérale `0xffff`, correspondant à l'adresse la plus élevée de l'espace d'adressage logique. Rappelons que la pile croit vers le bas et qu'un empilement (avec `push` ou un `call`) va décrémenter le contenu de ce registre, et inversement, un dépilement (avec `pop` ou `ret`) va l'incrémenter.

Nous gardons l'initialisation de l'ASCI0 dans le flux standard du code, mais nous transformons en sous-routine quelques fonctions, à commencer par la boucle de temporisation, que nous améliorons au passage :

```
delay:
        ld      bc,#0x0040
outer:
        ld      de,#0x1000
inner:
        dec     de
        ld      a,d
        or      e
        jp      nz,inner
        dec     bc
        ld      a,b
        or      c
        jp      nz,outer
        ret
```

Nous ne passons toujours pas de valeur en paramètre, mais nous utilisons à présent deux boucles imbriquées et donc deux compteurs. Vous comprenez maintenant pourquoi nous avions une étiquette `inner` précédemment. Il s'agissait d'une version réduite du présent code.

De la même façon, l'envoi d'un caractère, précédemment référencé par le label `asci0putc`, devient la routine `putc` que voici :

```
putc:
        in0     a,(STAT0)
        bit     1,a
        jr      z,putc
        out0    (TDR0),c
        ret
```

Le mécanisme est le même, j'ai simplement changé l'un des registres utilisés. Ceci nous permet de développer une autre routine reposant sur celle-ci. C'est `putstr` qui sera chargé d'envoyer toute la chaîne se terminant par un caractère `NUL` :

```
putstr:
        ld      c,(hl)
        inc     c
        dec     c
        jp      z, putstrbye
        call    putc
        inc     hl
        jp      putstr
putstrbye:
        ret
```

C'est peu ou prou le même code que précédemment, mais nous ajustons simplement l'issue si la condition de fin (le caractère est `NUL`) est vérifiée. Ces deux routines obtiennent leurs arguments, le caractère à envoyer pour `putc` et l'adresse de début de la chaîne pour `putstr`, via des registres. Respectivement *C* et *HL*. Ce n'est non seulement pas la seule façon de procéder, puisqu'il est également possible d'utiliser la pile comme le fait le *C*, mais également un point sensible, car il faut prendre en considération qu'une routine puisse changer la valeur contenue dans un registre, et donc impacter le déroulement de la suite du code. Un bel exemple est la routine suivante, chargée d'envoyer deux caractères ASCII représentant, en hexadécimal, une valeur passée en argument :

```
printhex:
        push    af
        push    bc
        push    af
        call    num1
        ld      c,a
        call    putc
        pop     af
        call    num2
        ld      c,a
        call    putc
        pop     bc
        pop     af
        ret
num1:
        rra
        rra
        rra
        rra
num2:
        or      #0xF0
        daa
        add     a,#0xa0
        adc     a,#0x40
        ret
```

Cette routine manipule les registres *A*, *B* et *C* pour faire son travail et, dans le corps même de celle-ci, le problème se pose déjà. La valeur à traiter est attendue dans le registre *A* lors de l'appel, mais après que `num1` soit utilisé, *A* n'est plus la valeur initiale. De la même manière, sans prendre de mesures particulières, un code comme celui-ci résultera sur un contenu de *A* corrompu et donc inutilisable pour la suite :

```
        ld      a,#0x74
        call    printhex
        inc     a
```

Ici, *A* ne serait pas égal à `0x75`. Pour éviter le problème, nous pouvons utiliser la pile, c'est pourquoi dès le début de la routine, nous y plaçons *AF* et *BC*, afin de sauvegarder les valeurs des registres *A*, *F*, *B* et *C*. Nous refaisons un `push af` afin de pouvoir restaurer *A* avant l'appel à `num2` pour traiter le second chiffre de la valeur initiale, sans oublier de restaurer également les 4 registres juste avant le `ret`. Ceci est important pour deux raisons : suite à un appel à `printhex`, *A* a toujours sa valeur de départ et, là c'est capital, l'adresse de retour, pour revenir au flux normal du code, est stockée sur la pile et dépilée au moment du `ret`. Si nous « pourrissons » les deux valeurs sur le dessus de la pile, le code reprendra à une adresse différente et incohérente. Nous devons donc dépiler de manière à ce que les prochaines valeurs soient bien l'adresse de retour, et rien d'autre.

Avant d'attaquer la partie principale du code, nous devons parler de la manière de faire évoluer la valeur dans *BBR* au cours de l'exécution. Nous jouons avec les registres un peu partout, appelons des routines et utilisons déjà la pile de façon intéressante. La question est donc : comment faire pour maintenir un compteur sans monopoliser bêtement un registre ? La réponse est simple : nous avons de la RAM et elle est faite pour cela. Mais attention, pas n'importe quelle RAM puisque *Bank* va « glisser » en mémoire physique et nous risquerions de ne pas retrouver nos petits. Nous devons choisir un emplacement en *Common 1* et c'est précisément l'objet de la définition d'un nouveau symbole au début de notre code assembleur :

```
SAVEDBBR        .equ    0xf000
```

`0xf000` est l'adresse (logique) du tout début de *Common 1* et est suffisamment éloignée de base de la pile (`0xffff`) pour éviter les problèmes. Ici, nous stockerons la valeur de départ de *BBR* (`0x39`) et rafraîchirons après chaque tour dans la boucle, jusqu'à la valeur maximum pour 512 Kio de RAM, soit `0xb1` et donc une adresse physique de `0xb9000` (à `0x7000` de la fin de la RAM à `0xbffff`). Notez au passage que ceci n'est absolument pas une façon de vérifier que chaque emplacement mémoire fonctionne, car même avec une RAM de 128 Kio, nous ne tombons jamais « nulle part », mais accédons aux mêmes emplacements via plusieurs adresses différentes. En d'autres termes, ce n'est pas un *memtest86*, mais uniquement une façon de s'assurer que nous avons compris le fonctionnement de la MMU, des adresses physiques/logiques et des différentes sections de mémoire.

Sans plus attendre, voici le corps du programme :

```
main:
        call    delay
        ld      hl,#bonjour
        call    putstr
        ld      a,#0x39
        ld      (SAVEDBBR),a

mainloop:
        ld      hl,#justbbr
        call    putstr
        ld      a,(SAVEDBBR)
        call    printhex
        out0    (BBR),a

        ld      a,#0x42
        ld      (0x8100),a
        ld      a,#0x00
        ld      a,(0x8100)
        ld      b,#0x42
        sub     b
        ld      hl,#textok
        jp      z,ok
        ld      hl,#textpasok
```

```
ok:
        call    putstr
        ld      a,(SAVEDBBR)
        inc     a
        ld      (SAVEDBBR),a
        ld      b,#0xb2
        sub     b
        jp      z,main
        jp      mainloop
```

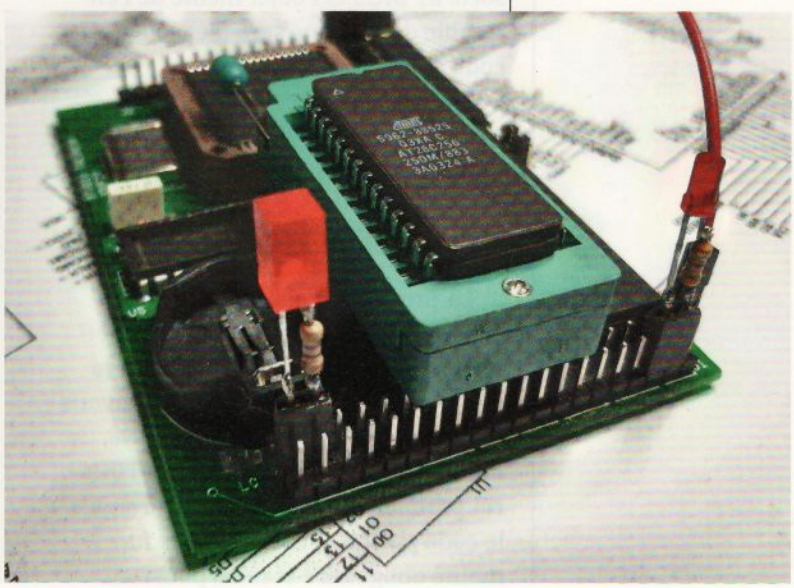
Rien de bien particulier ici, il s'agit d'une simple implémentation de ce que nous avons décrit précédemment, en utilisant des techniques et des éléments déjà vus par ailleurs. Cette source se conclura, bien entendu, par les habituelles chaînes de caractères :

```
justbbr:
        .ascii  "BBR=0x"
        .db     0x00
textok:
        .ascii  "  RAM works.\r\n"
        .db     0x00
textpasok:
        .ascii  "  RAM doesn't works !!!\r\n"
        .db     0x00
bonjour:
        .ascii  "\r\n\r\nLet's go!\r\n"
        .db     0x00
```

Et, non, `justbbr` n'est pas un mauvais jeu de mots sur « Justin Bieber » (je viens de m'en rendre compte), mais « *juste BBR* » comme dans « *on affiche juste 'BBR'* ». Une fois l'EEPROM programmée et placée sur la carte, la mise sous tension nous montre la progression du test :

```
[...]
BBR=0x8C  RAM works.
BBR=0x8D  RAM works.
BBR=0x8E  RAM works.
BBR=0x8F  RAM works.
[...]
```

Nous avons changé l'endroit où la pause est faite et à présent, les lignes défilent dans leur intégralité rapidement jusqu'à ce qu'on retourne au label `main` pour une nouvelle exécution. Nous constatons que jouer avec *BBR* n'est pas un problème et l'ensemble fonctionne correctement.



Cette nouvelle solution permettant de facilement retirer l'EEPROM pour la reprogrammer n'a pas que des avantages : le support ZIF dépasse sur une partie des connecteurs avec les signaux du processeur, le bus de données et une partie du bus d'adresses.

Ce style de SRAM se trouve généralement sur d'anciennes cartes mères (cache) ou carte graphique. Il était, fut un temps, très facile de s'en procurer gratuitement, mais avec la disparition progressive des machines « vintage » (486, en particulier) et le regain d'intérêt actuel pour le retrocomputing, les prix ne cessent de grimper...

Bien entendu, nous avons là quelque chose de relativement simple et peu représentatif de ce qu'il est réellement possible de faire avec le *banking*. Deux approches générales peuvent être envisagées : avoir le code en EEPROM assisté de *Common 1* et *banker* les données, ou alors, minimiser *Common 0*, augmenter sensiblement *Common 1* pour les données et *banker* du code. Cette dernière option permettra, par exemple, d'avoir un minimum de routines génériques en ROM et de charger des programmes en RAM depuis une source (stockage, communication série, etc.). Le code en ROM est alors une sorte de BIOS et l'objet même de l'ensemble du système sera d'exécuter un code chargé séparément.

6. CE SERA TOUT POUR AUJOURD'HUI

La prochaine fois, nous nous attacherons à faire fonctionner un code en C sur cette plateforme. Ce ne sera pas très différent de ce que nous avons vu par ailleurs avec le Z80 puisque, après tout, c'est historiquement l'intérêt premier du C dont il s'agit : rendre le code portable. Ceci sera une formalité obligatoire et nous pourrons ensuite regarder de plus près comment fonctionnent les autres périphériques du Z180, qui sont habituellement des « pièces rapportées » avec un ordinateur à base de Z80 ou de 6502. Ensuite, mais rien n'est moins sûr, nous verrons comment il est possible d'utiliser la MMU et le *banking* en C. SDCC ne supporte pas cela nativement, mais il existe des solutions, parmi lesquelles nous avons deux débuts de piste : d'une part, la version adaptée de SDCC par le légendaire Alan Cox (oui, oui, cet Alan Cox là !) pour son système FUZIX **[5]**, et d'autre part, ce que décrit la documentation du RTOS Contiki concernant le *code banking* avec le microcontrôleur Intel 8051.

Bref, encore beaucoup de choses à explorer, en espérant que je résiste suffisamment longtemps aux yeux doux que me fait ce Motorola 68008 qui, lui, utilise une architecture 8/32 bits ne nécessitant pas de *banking* pour accéder à l'ensemble de la mémoire adressable de 1 Mio... Dieu, qu'il serait mignon sur une platine à essais ! **DB**

RÉFÉRENCES

[1] **http://www.zilog.com/docs/z180/um0050.pdf**

[2] **https://www.zilog.com/docs/z180/ps0140.pdf**

[3] **https://github.com/daveho/GALasm**

[4] **https://github.com/simon-frankau/galette**

[5] **https://github.com/EtchedPixels/FUZIX**

CSAW'23 EUROPE

CYBERSECURITY COMPETITION

The greatest academic cybersecurity event in the world!
4 European competitions run in Valence on November 9-10, 2023

PUF - enabled Security Challenge

European students in cybersecurity (per team)

Competition dedicated to the safety of embedded and cyber-physical systems

Submission deadline: August 31, 2023

Applied Research Competition

European PhD students in cybersecurity

Election for the best cybersecurity article published in a high-ranking journal

Submission deadline: September 10, 2023

Red Team Competition

French High school students (per team)

Cybersecurity challenges for high school students (in partnership with Root-Me Pro)

Qualifying Round: sept 14 - oct 16, 2023

Capture the flag

European students in cybersecurity (per team of 4)

36-hour competition to solve cybersecurity challenges qualifications

Qualifying Round: September 8-10, 2023

Meet with the greatest talents in cybersecurity and win the European challenge !

More information and registration
csaw-europe@esisar.grenoble-inp.com

GRENOBLE INP UGA Esisar VALENCE

IN PARTNERSHIP WITH HACKABLE

MINISTÈRE
DES ARMÉES
Liberté
Égalité
Fraternité

#cyberexpert @DGA

Rejoignez-nous à Rennes, dans les métiers de :
Lutte Informatique Défensive et Lutte Informatique Offensive

du 16 nov
au 30 nov 2023
www.dghack.fr

DGA Maîtrise de l'information

POSTULEZ en cybersécurité !

CONTACT Envoyez votre candidature

dga-mi-bruz.recrutement.fct@intradef.gouv.fr

DIRECTION
GÉNÉRALE
DE L'ARMEMENT
DGA
Maîtrise
de l'information