



ÉLECTRONIQUE | EMBARQUÉ | RADIO | IOT

HACKABLE

L'EMBARQUÉ À SA SOURCE

N° 49

JUILLET / AOÛT 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 - 49 - F: 14,90 € - RD



CPPAP: K92470

SBC / PHOTOBOOTH

Surprenez vos invités grâce à votre **cabine photo** automatisée à base de **Raspberry Pi** et **PHP** p.66

RETROBREW / Z80

Le Z80 sur platine à essais : cette fois, on fait ça comme les grands avec **EEPROM** et **RAM** ! p.108

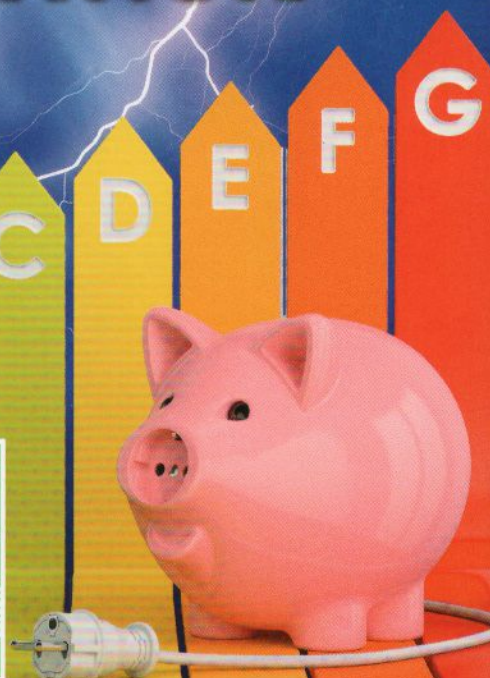
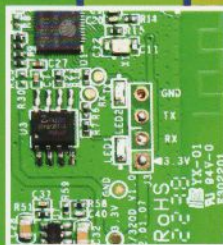
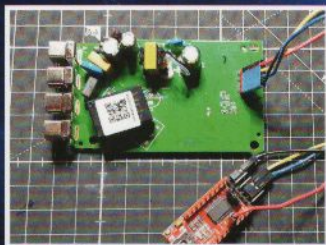
Domotique / économies / Home Assistant

Terrifié par vos dernières factures d'électricité ?

DÉTECTEZ VOS SOURCES DE SURCONSOMMATION ÉLECTRIQUE !

p.38

- Déploiement des capteurs
- Configuration de Home Assistant
- Mesures et calcul des coûts



MICROCONTRÔLEUR / RP2040

FUZIX : Transformez une simple **Raspberry Pi Pico** en un ordinateur exécutant un vrai **Unix** p.18

HACK / REVERSE

Rétro-ingénierie et exploration d'une **carte industrielle** à microprocesseur Zilog 8 bits p.90

ATMEL / SIGNAUX

FFT sur **Atmega32U4** : du domaine temporel au domaine spectral dans seulement **2,5 Kio de RAM** ! p.04



ÉDITO



C'est la première fois !

En presque 50 numéros du magazine, c'est la première fois qu'un projet en couverture impacte à ce point mon quotidien. Tout fier d'utiliser la même machine depuis plus de 13 ans, sans ralentissement notable ou mise à niveau autre qu'une carte graphique il y a une paire d'années, je pensais réellement que c'était une belle économie. Après tout, ceci veut dire « ne pas dépenser d'argent pour une nouvelle machine », ça semble logique. Et ça l'est... presque. La machine en ques-

tion est construite autour d'une carte mère ASUS Z8NA-D6 avec deux Intel Xeon E5520 à 2,3 GHz et 24 Gio de RAM, et pour l'usage que j'en ai (pas de jeu ou de 3D, mais beaucoup de compilations), elle affiche des performances encore vraiment très agréables. Le pari, en 2009, d'acheter un « monstre », mais qui dure longtemps était donc judicieux...

Mais cela fait maintenant une semaine que j'ai solidement finalisé l'installation dont je vous parle dans ce numéro, et les faits sont là. Quelque 1400 kWh par an sont surconsommés par ce PC, soit près de 300 € d'électricité en plus (au tarif actuel) par rapport à une configuration récente, comme un mini PC équipé d'un processeur sensiblement plus rapide. PC pouvant coûter, grand maximum pour mon budget, quelque chose comme 700 € (Ryzen 5 + 32 Go de RAM).

En d'autres termes, là, sous mes yeux et grâce à Home Assistant, ESPHome et le matériel Sonoff, je vois mon erreur dans toute sa splendeur. Même avec un tarif au kWh de 2022 ou 2021, garder une machine « qui tient encore la route », mais est un gouffre à énergie est un mauvais calcul, basé sur des impressions plus que sur des informations tangibles.

Bien sûr, ceci est amplifié par le coût actuel de l'énergie, mais la leçon n'en est pas moins mémorable. « Beast » (c'est son *hostname*) va donc prendre sa retraite sous peu et j'espère que cette petite histoire vous aura donné envie, à vous aussi, de savoir exactement où va toute cette énergie qui vous est facturée...

Denis Bodor

Hackable Magazine

est édité par Les Éditions Diamond



12, place du Capitaine Dreyfus - 68000 Colmar - 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
12 Place du Capitaine Dreyfus, 68000 Colmar, 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 €

La rédaction n'est pas responsable des textes, illustrations et photos qui lui sont communiqués par leurs auteurs. La reproduction totale ou partielle des articles publiés dans Hackable Magazine est interdite sans accord écrit de la société Les Éditions Diamond. Sauf accord particulier, les manuscrits, photos et dessins adressés à Hackable Magazine, publiés ou non, ne sont ni rendus, ni renvoyés. Les indications de prix et d'adresses figurant dans les pages rédactionnelles sont données à titre d'information, sans aucun but publicitaire. Toutes les marques citées dans ce numéro sont déposées par leur propriétaire respectif. Tous les logos représentés dans le magazine sont la propriété de leur ayant droit respectif.

Suivez-nous sur Twitter

 @hackablemag



SOMMAIRE

MICROCONTRÔLEURS & ARDUINO

04 Du domaine temporel au domaine spectral dans 2,5 kB de mémoire : transformée de Fourier rapide sur Atmega32U4 et quelques subtilités du C

18 FUZIX : un « vrai » Unix pour Raspberry Pi Pico !

DOMOTIQUE & CAPTEURS

38 Surveillez votre consommation électrique et faites des économies

SBC & RASPBERRY PI

58 Créer son photobooth custom

HACK & UPCYCLING

90 Exploration d'une carte industrielle à base de Z180

RÉTRO

110 Zilog Z80 : le retour du fils de la vengeance !

ABONNEMENT

75 Abonnement

À PROPOS DE HACKABLE...

HACKS, HACKERS & HACKABLE

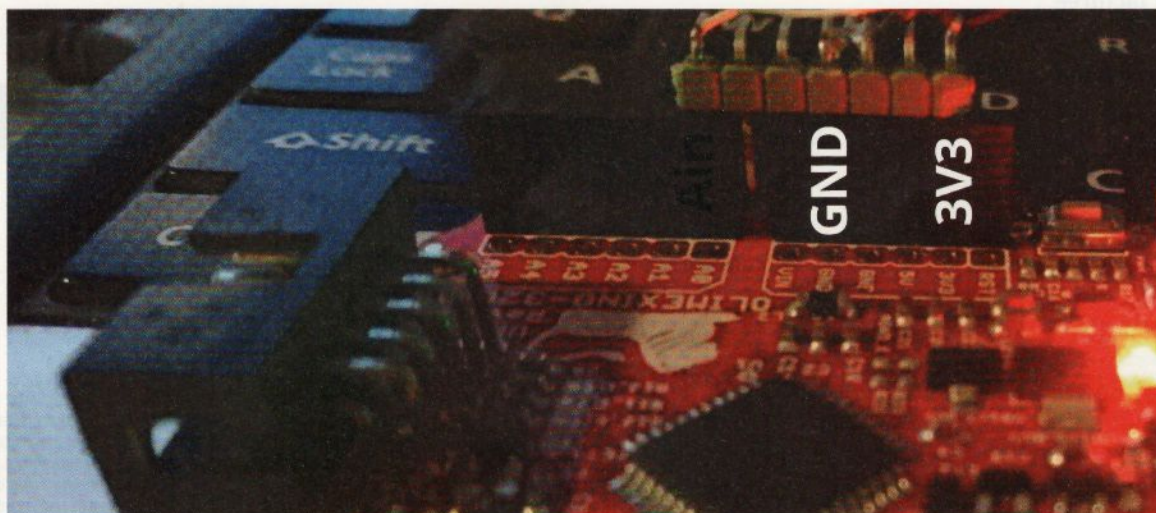
Ce magazine ne traite pas de piratage. Un **hack** est une solution rapide et bricolée pour régler un problème, tantôt élégante, tantôt brouillonne, mais systématiquement créative. Les personnes utilisant ce type de techniques sont appelées **hackers**, quel que soit le domaine technologique. C'est un 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.

DU DOMAINE TEMPOREL AU DOMAINE SPECTRAL DANS 2,5 KB DE MÉMOIRE : TRANSFORMÉE DE FOURIER RAPIDE SUR ATMEGA32U4 ET QUELQUES SUBTILITÉS DU C

J.-M FRIEDT

[Enseignant-chercheur à l'université de Franche-Comté à Besançon]

Nous avons exploré diverses implémentations libres de transformées de Fourier discrètes rapides (FFT), mais leur occupation en mémoire reste de la dizaine de kilooctets. Que peut-on faire avec 2,5 kB de mémoire ? La vénérable note d'application 3722 de Maxim IC nous enseigne comment implémenter efficacement une FFT sur microcontrôleur 8-bits et l'arithmétique en virgule fixe, et la notation en complément à deux au passage.



– Du domaine temporel au domaine spectral dans 2,5 kB de mémoire : transformée de Fourier rapide... –

La note d'application 3722 de Maxim [1] devrait être une lecture obligatoire pour tout développeur désireux d'appréhender toute la puissance des petits microcontrôleurs 8 bits (taille des données manipulées par leur unité arithmétique et logique) que nous pourrions choisir pour leur faible consommation par exemple, mais elle est malheureusement devenue introuvable sur le Web. Heureusement, nous avons archivé ce document avant que Maxim ne soit racheté par Analog Devices et le mettons à disposition, dans <https://github.com/jmfriedt/l3ep>, dans le répertoire FFT avec son archive associée de codes sources. Cette note d'application contient toutes les informations nécessaires pour développer efficacement de l'arithmétique sur des entiers codés en virgule fixe, en définissant le mode d'encodage des informations en format Qm.n pour attribuer m bits à la partie entière du nombre et n bits à fractionnaire, en produisant automatiquement le code effectuant la transformée de Fourier rapide (*Fast Fourier Transform* ou FFT), et une méthode efficace de calcul du module s'affranchissant de multiplications et de la racine carrée en passant par une table d'allocation (*Look Up Table*) précalculée et donc stockée en mémoire non volatile, car préfixée dans sa définition du terme **const**. Nous nous proposons de reprendre ce texte fondateur sur un microcontrôleur plus actuel qu'est l'Atmega32U4 avec ses 2,5 kB de RAM et 32 kB de mémoire non volatile et voir comment y porter le code pour effectuer le passage du domaine temporel au domaine spectral de mesures acquises sur le convertisseur analogique numérique.

1. PRISE EN MAIN DE LA NOTE D'APPLICATION

La note d'application AN3722 venait avec une archive contenant le logiciel de démonstration pour compilateur propriétaire IAR visant une architecture un peu exotique nommée MAXQ2000 (<https://www.analog.com/en/products/maxq2000.html>) spécifiquement conçue pour commander un écran LCD. Une fois débarrassé des fichiers de l'IDE `fft_r2*` pour se concentrer sur les codes sources, il ne reste que l'implémentation de la FFT `maxqfft.c` et les prototypes de fonctions associées `maxqfft.h`, le fichier `iomaxq200x.h` étant spécifique au microcontrôleur qui sert de prétexte à cette note d'application. Ce code en C devrait être portable, sauf :

```
/*
 * Hardware Multiplier Macros
 *
 * These macros are used to access the hardware multiplier. For
 * the MAXQ, registers MA and MB are the hardware multiplier
 * operands while MC1:MC0 store the hardware multiplier result.
 *
 * (1) MUL_1(A,B,C) : C=A*B (result converted to Q8.7 notation)
 * (2) MUL_2(A,C)   : C=A*MB (result converted to Q8.7 notation)
 * (3) MUL_INIT(B)  : MB=B
 * (4) MUL_NC(A,C)  : C=A*MB (result not converted to Q8.7 notation)
 */
#define MUL_1(A,B,C) MA=A;MB=B; NOP; tmp_16.LSW=MC0; tmp_16.MSW=MC1; C=tmp_32>>7
#define MUL_2(A,C)   MA=A;      NOP; tmp_16.LSW=MC0; tmp_16.MSW=MC1; C=tmp_32>>7
#define MUL_INIT(B)  MB=B
#define MUL_NC(A,C)  MA=A;      NOP; C=MC0
```


qui prétend forcer les multiplieurs matériels du microcontrôleur. Ces registres n'ayant aucun sens pour l'Atmega32U4, nous nous contentons de les déclarer comme variables globales du programme pour permettre la compilation. Le sens de ce code est limpide puisque toute la FFT est implémentée sur des entiers codés sur 16 bits, dont les 8 bits de poids fort sont la partie entière de la représentation en virgule fixe et les 8 bits de poids faible sont la partie fractionnaire, avec la perte de précision lors de la multiplication associée à la perte des bits de poids faible que nous observons avec les décalages en fin d'opération. Ainsi, cette fonction se traduit trivialement par :

```
short MB;
#define MUL_1(A,B,C) MB=B;C=(A*B)>>7
#define MUL_2(A,C)   C=(A*MB)>>7
#define MUL_INIT(B)  MB=B
#define MUL_NC(A,C)  C=A*MB
```

et le tour est joué pour compiler le code sur PC (nous verrons plus loin que ce n'est pas si simple !).

2. PREMIERS ESSAIS SUR PC : LES STUBS

Développer sur microcontrôleur c'est bien, mais c'est long, fastidieux et peu d'outils de débogage (**gdb**) sont disponibles. Développer sur PC, c'est mieux : comment prendre le code destiné à un microcontrôleur et le tester sur PC ? En prenant soin de séparer la partie algorithmique du code, portable, des aspects d'initialisation du matériel et de communication, par exemple : au moyen de *stubs* qui émuleront le fonctionnement du matériel sur le microcontrôleur lors de l'exécution sur PC. Ainsi, alors que la communication se fera de façon asynchrone par port série compatible RS232 ou USB sur le microcontrôleur, la fonction d'affichage implémentée sur PC peut se contenter d'un **printf()** sur la console. Le choix du *stub* se fait lors de la compilation, en liant l'exécutable avec la version appropriée du code. Cette approche de développement du code permet de bénéficier d'outils puissants sur PC tels que **valgrind** ou **lint** pour profiler le code afin d'identifier de potentielles sources de dysfonctionnement avant de l'exécuter sur microcontrôleur, dont l'absence de gestionnaire de mémoire (MMU) rend les dépassements de mémoire indétectables autrement que par un comportement aléatoire et inattendu du code.

```
void uart_init(void)
{
    #if defined (__AVR_ATmega32U4__)
    #define USART_BAUDRATE (9600)
    unsigned short baud;
    UCSR1A = 0; // importantly U2X1 = 0
    UCSR1B = 0;
    UCSR1B = (1 << RXEN1)|(1 << TXEN1); // enable receiver and transmitter
    UCSR1C = _BV(UCSZ11) | _BV(UCSZ10); // 8N1
```


– Du domaine temporel au domaine spectral dans 2,5 kB de mémoire : transformée de Fourier rapide... –

```

    baud    = ((( F_CPU / ( USART_BAUDRATE * 16UL))) - 1));
    UBRR1H = (unsigned char)(baud>>8);
    UBRR1L = (unsigned char)baud;
#endif
}

void uart_write(short c)
{
    #if defined (__AVR_ATmega32U4__)
        char s[7]; // -32768:32767\0
        int k=0;
        sprintf(s, "%hd", c);
        do {uart_transmit(s[k]);k++;} while (s[k]!=0);
        uart_transmit(' ');
    #else
        printf("%hx ",c); // %hd en decimal
    #endif
}

```

Ainsi, les deux fonctions dépendantes du matériel que sont l'initialisation du port de communication et la fonction d'affichage des valeurs elle-même sont remplacées dans l'implémentation sur PC par une fonction vide pour l'initialisation et `printf("%hx", val)`; pour la seconde, toutes les variables manipulées étant codées sur 16 bits. Plutôt que de s'attaquer tout de suite à des données expérimentales, la fonction d'acquisition est remplacée par la génération d'un signal de forme connue, par exemple sinusoïde ou créneau, dont la transformée de Fourier est connue, permettant donc de valider le bon fonctionnement de la bibliothèque.

```

#include <math.h>
#include "fft.h"

void getSamplesFromADC(short *x_n_re)
{int i;
 for (i=0;i<N;i++)
    {//if ((i%20)<10) x_n_re[i]=127; else x_n_re[i]=-127;
      //if (x_n_re[i]&0x0080) x_n_re[i]=0xFF00;
      x_n_re[i]=(short)(127.*sin((float)(i)/4.));
    }
}

```

La compilation conditionnelle exploite le drapeau `__AVR_ATmega32U4__` identifié dans `/lib/avr/include/avr/io.h` du paquet `avr-libc` de Debian/GNU Linux et défini par `avr-gcc` lorsque l'option de compilation `-mmcu=atmega32u4` est activée. En l'absence de ce drapeau, nous supposons que la compilation cible le processeur compatible Intel du PC.

Lors de l'exécution de ce code, nous constatons que tous les $256/2=128$ coefficients de Fourier aux fréquences positives sont nulles lorsque les données représentent une sinusoïde, sauf le 1er coefficient. Ce résultat est en accord avec les attentes : le signal analysé est défini comme

$127 \times \sin(i/4.)$ avec i l'indice du point. L'argument de la fonction trigonométrique s'exprime comme une fréquence f qui fait tourner la phase en fonction du temps t comme $2\pi f t$ qui vaut ici $1/4$ donc $f = 1/(2\pi \times 4) \approx 0,04$ et comme la transformée de Fourier couvre la gamme des fréquences allant de 0 à la fréquence d'échantillonnage sur les $N = 256$ points considérés, l'abscisse de la composante spectrale est $0,04 \times 256 = 10,2$. Le premier point étant d'indice 0 pour la composante continue du signal, le 11e point représente bien la composante spectrale $1/4$ utilisée dans la définition du signal.

De la même façon, si nous décommentons la définition des échantillons en créneaux de rapport cyclique 50 % et de période 20 échantillons – représentation en complément à deux qui propage donc le bit de signe de l'octet de poids faible si celui-ci est à 1 pour indiquer un nombre négatif – nous obtenons des coefficients non nuls aux abscisses (en commençant à compter avec l'indice 0) 13, 38, 64, 90, et 115 pour des valeurs de 143, 32, 22, 16 et 16, respectivement. La transformée de Fourier discrète d'un créneau est caractérisée par une série de raies spectrales aux harmoniques impaires

du mode fondamental dont l'amplitude décroît comme l'indice de l'harmonique. Les indices sont bien aux abscisses prévues, tandis que l'axe des ordonnées présente la bonne tendance sans suivre exactement la loi attendue, compte tenu de la fuite d'énergie dans les composantes spectrales adjacentes (143 est entouré de 35 et 16, 32 est suivi de 16).

3. VALIDATION SUR ATMEGA32U4 : LA TAILLE D'UN INT

Le code fonctionnel sur PC est *cross-compilé* pour Atmega32U4 en remplaçant gcc par `avr-gcc -mmcu=atmega32u4` dans le `Makefile` et... l'exécution échoue lamentablement avec le renvoi exclusivement de 0 s après la FFT d'un signal qui devrait présenter une raie spectrale. La promesse de portabilité du code en C entre plateformes n'est donc pas respectée.

Dans la série des choses qui ne sont pas définies dans le langage C :

- coder un entier sur 8 bits en signé ou non signé n'est pas défini par défaut si nous omettons le préfixe `signed` ou `unsigned`. En effet, <https://gcc.gnu.org/onlinedocs/gcc/C-Dialect-Options.html> affirme dans la description de l'option `-funsigned-char` que « *Each kind of machine has a default for what char should be. It is either like unsigned char by default or like signed char by default. Ideally, a portable program should always use signed char or unsigned char when it depends on the signedness of an object. But many programs have been written to use plain char and expect it to be signed, or expect it to be unsigned, depending on the machines they were written for.* » tandis que l'`int` est toujours signé par défaut tel que l'affirme l'option `-fno-unsigned-bitfields` de cette même page avec « *the basic integer types such as int are signed types* », mais sa taille dépend de l'architecture.
- la taille de l'`int` utilisé tout le temps dans un code C dépend de l'architecture. En effet, `int` est défini comme la taille « native » de l'entier sur l'architecture ciblée, mais capable de représenter au moins 32767 donc sur 2 octets au minimum. Le seul type strictement défini en C est le `short` avec ses deux octets, puisque nous constatons dans `/usr/include/stdint.h` d'une machine sous GNU/Linux que :

– Du domaine temporel au domaine spectral dans 2,5 kB de mémoire : transformée de Fourier rapide... –

```
/* Signed. */
typedef signed char      int_fast8_t;
#if __WORDSIZE == 64
typedef long int         int_fast16_t;
typedef long int         int_fast32_t;
typedef long int         int_fast64_t;
#else
typedef int              int_fast16_t;
typedef int              int_fast32_t;
__extension__
typedef long long int    int_fast64_t;
#endif
```

et que donc sur un PC basé sur un processeur compatible Intel 32 bits le **long** est codé sur 4 octets et le **long long** sur 8, tandis que sur une machine 64 bits les deux **long** et **long long** sont tous deux codés sur 8 octets.

- l'ordre d'analyse des arguments d'une fonction dépend du compilateur : LLVM (compilateur **clang**) et GCC n'interprètent pas dans le même ordre les arguments de **printf("%d %d %d\n", i, f(&i), i);** pour une fonction **f(int *i) {(*i)++; return(*i);}** qui modifie le contenu de la variable. En effet, lors de la compilation du code :

```
#include <stdio.h>
int f(int* i) {(*i)++; return(*i);}
int main()
{int i=1; printf("%d %d %d\n", i, f(&i), i);}
```

nous obtenons un résultat différent selon le compilateur utilisé :

```
$ clang -Wall -o arg arg.c && ./arg
1 2 2
$ gcc -Wall -o arg arg.c && ./arg
2 2 1
```

Le problème que nous venons de rencontrer est du deuxième type, à savoir que le calcul sur le PC est passé par un **int** codé sur 4 octets qui s'est réduit à deux octets sur Atmega32U4, résultant dans l'échec de la multiplication de deux nombres en virgule fixe codés sur 16 bits qui doivent absolument passer par une variable intermédiaire sur 32 bits, avant de tronquer les bits de poids faible. Le problème se résout en définissant :

```
#define MUL_1(A,B,C) MB=B; C=((long)A*(long)B)>>7;
```

afin d'imposer des opérations arithmétiques sur 32 bits et non sur les 16 bits qu'occupent **A** et **B**, et cette fois le résultat sur Atmega32U4 est strictement identique à l'exécution sur PC.

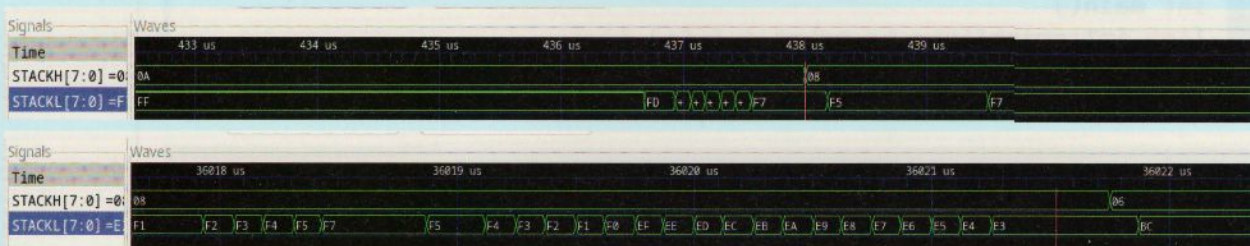
ANALYSE DE L'OCCUPATION DE LA PILE DANS LE SIMULATEUR SIMAVR

Nous avons déjà expliqué dans [2] la puissance du simulateur **simavr** et sa capacité à tracer l'état interne des registres pour en mémoriser les changements d'état dans un fichier VCD (*Value Change Dump*) lisible par **gtkwave**. Ici, nous avons un doute sur la capacité de l'Atmega32U4 à stocker toutes les informations utiles à une transformée de Fourier rapide sur 256 points et désirons tester l'état de la pile en cours d'exécution. L'Atmega32U4 a cette particularité que le pointeur de pile – la zone brouillon où le microcontrôleur stocke les variables temporaires en partant de l'adresse la plus élevée de la RAM – n'est pas un registre proche de l'unité arithmétique et logique, mais un registre adressable aux emplacements **0x5D** et **0x5E**. Nous indiquons à **simavr** de mémoriser l'état de ces deux registres par un entête qui ne sera pas lié au binaire flashé dans le microcontrôleur, mais uniquement comme consigne à l'émulateur :

```
#if defined (__AVR_ATmega32U4__)
#define F_CPU 16000000UL
#include "avr_mcu_section.h"
AVR_MCU(F_CPU, "atmega32u4");
AVR_MCU_VCD_FILE("atmega32u4.vcd", 1000);

const struct avr_mmcu_vcd_trace_t _mytrace[] _MMCU_ = {
    {AVR_MCU_VCD_SYMBOL("STACKL"), .what=( void *) (0x5D), },
    {AVR_MCU_VCD_SYMBOL("STACKH"), .what=( void *) (0x5E), },
};
#endif
```

dont l'exécution par **simavr --freq 16000000 --mcu atmega32u4 fft** se traduit par une trace dont l'affichage est découpé dans les zones les plus intéressantes :



4. AFFICHAGE EN WATERFALL EN ASCII

Maintenant que nous avons validé le bon fonctionnement de la bibliothèque sur PC et sur Atmega32U4, il reste à analyser un vrai signal physique et pas une simple simulation. Pour ce faire, le convertisseur analogique numérique de l'Atmega32U4 est connecté à la carte son d'un PC faisant office de générateur de fonction. La carte son génère un signal centré sur 0, tantôt positif et tantôt négatif, quand le convertisseur analogique numérique ne peut que mesurer une tension entre la masse et sa tension de référence. Il faut donc introduire une tension de biais constante entre la sortie de la carte son et l'entrée du convertisseur (Fig. 1) : afin de s'affranchir de tout composant actif (p. ex. amplificateur opérationnel), la solution la plus simple est le pont de résistances pour définir la tension de biais, et un condensateur pour protéger la sortie de la carte son du potentiel constant. Ce montage nommé T de polarisation suppose de respecter deux hypothèses :

- le pont diviseur ne produit en son point milieu un potentiel pondéré par les valeurs des résis-

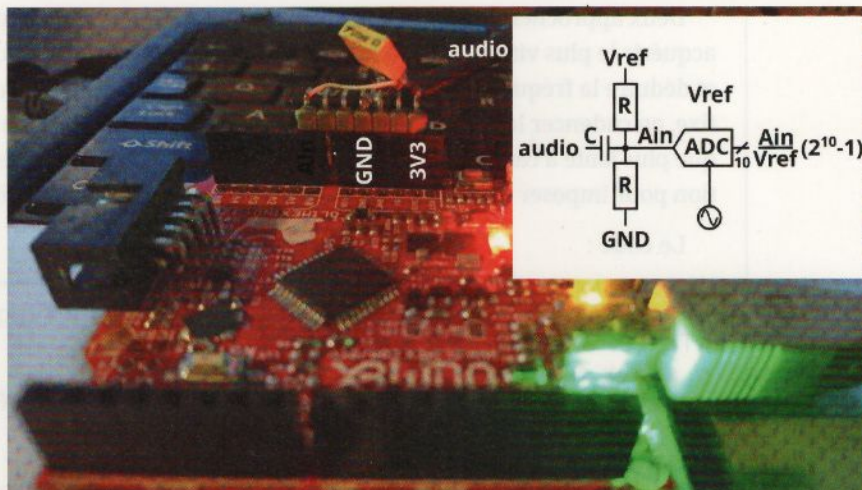


Figure 1 : Montage exploitant la carte son d'un ordinateur comme générateur de fonction pour fournir un signal au convertisseur analogique numérique (ADC), ici codé sur 10 bits, après translation du signal par un biais sélectionné comme la moitié de la tension de référence Vref. La communication se fait par USB sur un port série virtuel (droite).

tances que si le courant qui circule dans les résistances est grand devant le courant consommé par le montage connecté au point milieu. Or, le convertisseur analogique numérique présente [3, p.306] une impédance caractéristique d'une dizaine de k Ω . Le choix de résistances de quelques k Ω tout au plus permettra de vérifier la condition de fonctionnement du pont diviseur de tension ;

- le condensateur se comporte toujours comme un circuit ouvert pour la composante continue introduite par le pont de résistances, mais doit laisser passer le signal audiofréquence comme le ferait un fil. Le module de l'impédance $|Z|$ du condensateur C traversé par un signal de fréquence f est $|Z| = 1/(C2\pi f)$ qui sera d'autant plus faible que f est élevé. Le problème se pose donc aux basses fréquences : notre choix d'un condensateur non polarisé de valeur arbitrairement sélectionnée à 100 nF implique une impédance de 10 k Ω ou plus lorsque la fréquence passe en dessous de 150 Hz. Lors de nos essais, nous avons par conséquent toujours mesuré des signaux audiofréquences au-dessus de 500 Hz pour limiter l'impact du condensateur sur l'amplitude du signal vu par le convertisseur analogique numérique.

Deux approches s'offrent à nous pour cadencer le convertisseur analogique numérique : soit acquérir le plus vite possible dans une boucle de N éléments en vue d'effectuer la FFT de ce tableau et déduire la fréquence d'échantillonnage, inconnue *a priori*, d'une mesure d'un signal à fréquence fixe, ou cadencer le convertisseur sur un *timer* dont l'interruption déclenche la conversion. Bien que plus lente à cause de la surcharge de gestion des interruptions, nous utilisons la seconde solution pour imposer la fréquence d'échantillonnage f_s et donc l'axe des abscisses de la FFT.

Le code :

```
// http://www.avrfreaks.net/forum/atmega32u4-starting-adc-conversion-timer4
EMPTY_INTERRUPT(TIMER4_OVF_vect);
volatile short x_n_re[N], flag;
ISR(ADC_vect) {if (flag<N) {x_n_re[flag] = ADC-512; flag++;}}

void adc_init()
{ADMUX = (1<<REFS0)+7; // tension de référence & canal
 ADCSRB = (1<<ADTS3); // source de déclenchement de la mesure
 ADCSRA = (1<<ADEN) | (1<<ADSC) | (1<<ADATE) | (1<<ADIE) | (1<<ADPS2) |
 (1<<ADPS1) | (1<<ADPS0);
}

void timer_init()
{TCR4B = (0<<CS43) | (1<<CS42) | (0<<CS41) | (1<<CS40); // CK/16
 TIMSK4 = (1<<TOIE4);
 TC4H = 0; OCR4C = 200; // 16 MHz/16/100=5 kHz
}
```

initialise le *timer* 4 (codé sur 10 bits) pour se déclencher tous les 200 coups d'horloge après une décimation d'un facteur 16 de l'oscillateur à 16 MHz qui cadence le microcontrôleur, résultant en une fréquence d'échantillonnage de 5 kHz puisque `adc_init()` déclenche la conversion en configurant `ADCSRB` sur le *timer*. La voie mesurée est définie dans `ADMUX` comme étant le convertisseur numéro 7... noté A0 dans la sérigraphie compatible Arduino de la carte Olimexino32U4. Chaque fois que la conversion est achevée, l'interruption liée à la fin de conversion analogique numérique `ADC_vect` est appelée et se charge de stocker dans le tableau des valeurs à convertir par FFT la nouvelle mesure. Lorsque le tableau est plein, la conversion est effectuée et son résultat affiché à l'écran :

```
adc_init();
timer_init();
sei();
flag=0;
while (1)
{if (flag==N)
{fft(x_n_re);
 for (k=2;k<N_DIV_2_PLUS_1;k++) {putc(' '+x_n_re[k],&USBSerialStream);}
 putc('\r',&USBSerialStream);
 putc('\n',&USBSerialStream);
 flag = 0;
}
```


– Du domaine temporel au domaine spectral dans 2,5 kB de mémoire : transformée de Fourier rapide... –

en s'appuyant sur la bibliothèque LUFA et sa version allégée réduite à la communication s'apparentant au lien asynchrone compatible RS232 fournie à http://jmfriedt.free.fr/LUFA_light_EEA.tar.gz pour communiquer par USB.

La FFT étant bijective, les N points acquis dans le temps deviennent N points dans le domaine spectral s'étendant de $-f_s/2$ à $+f_s/2$ donc un pas de fréquence de f_s/N . Puisque le module de la FFT d'un signal réel est une fonction paire, nous nous contentons de n'afficher que les N/2 points compris entre 0 et $f_s/2$: avec N=256, ces 128 points tiennent dans un terminal un peu élargi par rapport aux 80 colonnes standard, et afficher successivement les FFT en codant l'amplitude de chaque raie spectrale comme un symbole différent permet d'afficher en ASCII un *waterfall* dans lequel l'axe des abscisses est la fréquence et l'ordonnée le temps, permettant d'observer les évolutions des composantes spectrales caractérisant un signal en fonction du temps. La Fig. 2 illustre le résultat en codant (gauche et milieu) l'intensité spectrale par la lettre « A » à laquelle s'ajoute un indice proportionnel à l'amplitude du pic, permettant de voir toute l'étendue du spectre, et à droite la solution quand le symbole de référence en l'absence d'énergie est l'espace, améliorant le contraste, mais ne permettant pas d'identifier la fréquence maximale sur l'écran. Le signal a été produit par *sox* comme un *chirp*, donc une sinusoïde de fréquence variable à gauche entre 500 et 2000 Hz, et au milieu et à droite entre 500 et 2500 Hz, par les commandes :

```
while true; do play -n synth 4 sin 500-2000;done
```

et :

```
while true; do play -n synth 4 sin 500-2500;done
```

respectivement, avec pour premier argument la durée du *chirp* (4 secondes) suivi de la nature du signal (une sinusoïde) et la gamme de fréquences.

Figure 2 : Affichage en waterfall en ASCII dans un terminal : la lettre « A » (gauche et milieu) ou l'espace (droite) indiquent l'absence de puissance dans une raie spectrale, et l'intensité est codée par un ajout à cette valeur de référence proportionnel à la puissance de chaque raie spectrale. L'axe des abscisses est imposé par la fréquence d'échantillonnage, et la vitesse de rafraîchissement en ordonnée est déterminée par le temps d'acquisition et de calcul. Chaque chirp dure 4 secondes et se répète dans une boucle infinie.

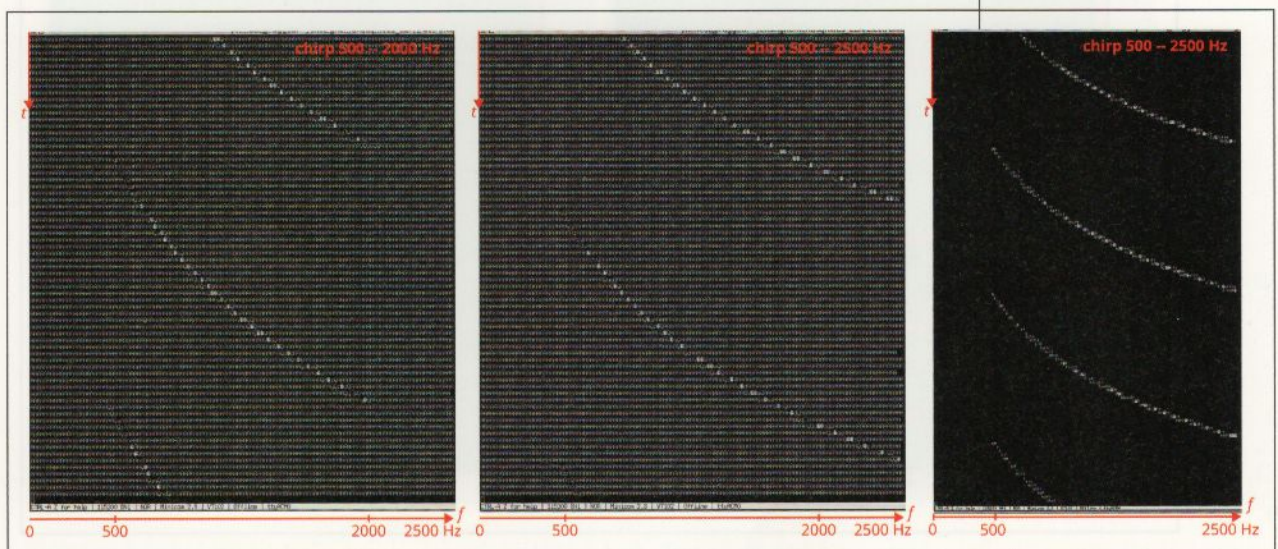
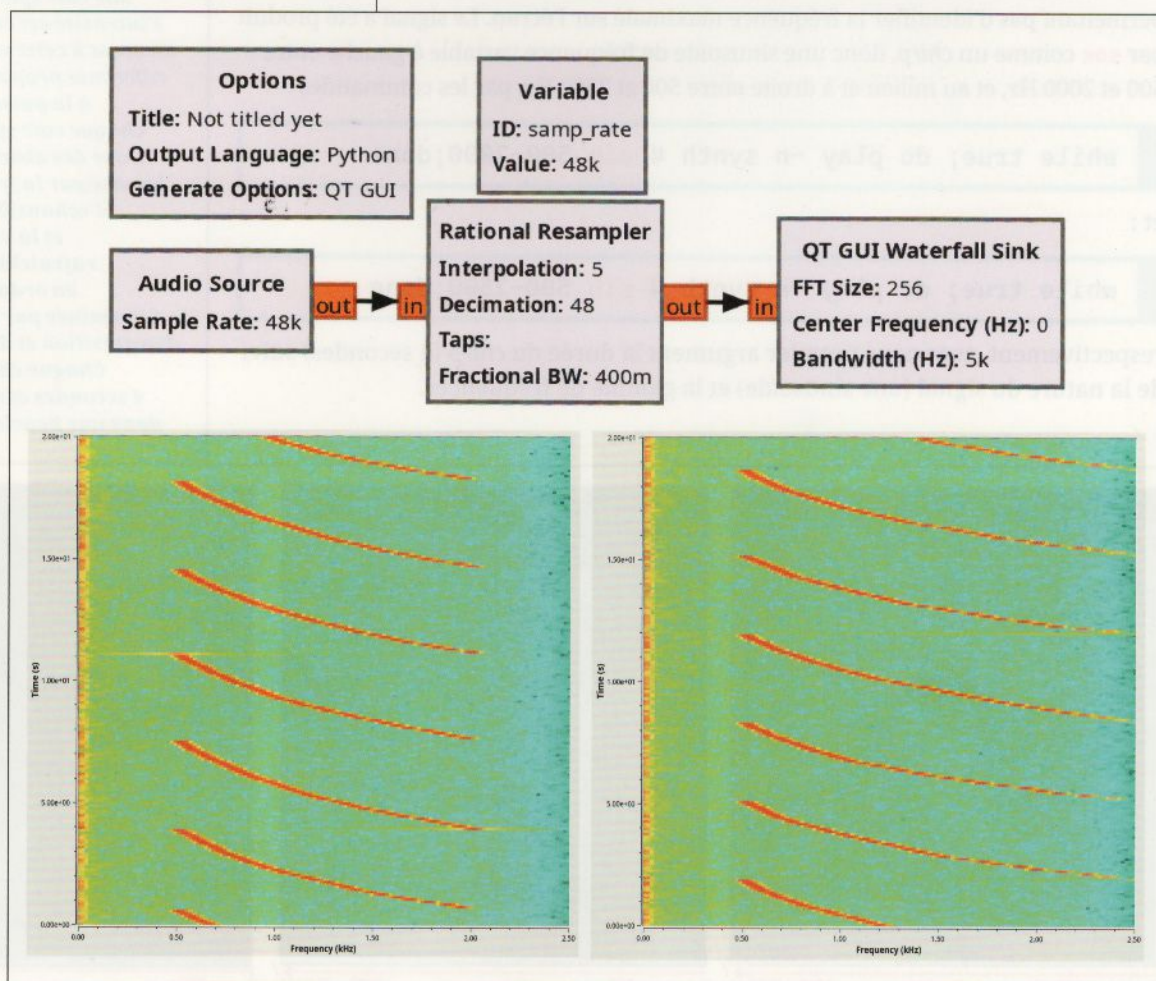


Figure 3 : En haut : chaîne de traitement GNU Radio faisant l'acquisition du signal sur l'entrée microphone de la carte son, ré-échantillonnant le signal de 48 à 5 kHz, et affichant l'évolution temporelle de la transformée de Fourier court terme calculée sur 256 points dans le waterfall. En bas à gauche : le signal émis balaie 500 à 2000 Hz en 4 s. En bas à droite : le signal émis balaie 500 à 2500 Hz, la fréquence de Nyquist, en 4 s.

N'ayant aucune raison d'avoir confiance dans notre résultat, nous désirons le comparer à une approche indépendante du même problème, par exemple par l'implémentation du *waterfall* dans GNU Radio. Cette comparaison est proposée en Fig. 3 avec une chaîne d'acquisition et de traitement réduite à sa plus simple expression, avec la seule subtilité de passer de la fréquence d'acquisition imposée par la carte son du PC – 48 kS/s – à une fréquence comparable à celle utilisée sur microcontrôleur – 5 kS/s. Pour ce faire, le **Rational Resampler** se charge de décimer et d'interpoler pour atteindre la fréquence d'échantillonnage requise : nous conservons les paramètres par défaut du bloc et enregistrons le flux audiofréquence sur l'entrée microphone alors que **sox** joue au moyen de **play** les *chirps* sur la sortie casque, l'entrée et la sortie étant reliées par un câble audio jack mâle/jack mâle.



– Du domaine temporel au domaine spectral dans 2,5 kB de mémoire : transformée de Fourier rapide... –

Nous pourrions être surpris par la forme du *chirp* dans le *waterfall* : pourquoi l'évolution de la fréquence n'est-elle pas linéaire dans le temps ? La page de manuel ou les sources disponibles à <https://sourceforge.net/projects/sox/> nous indiquent en lignes 81 et 262 de `synth.c` que :

```
typedef enum {Linear, Square, Exp, Exp_cycle} sweep_t;
static const char sweeps[] = ":+/-";
```

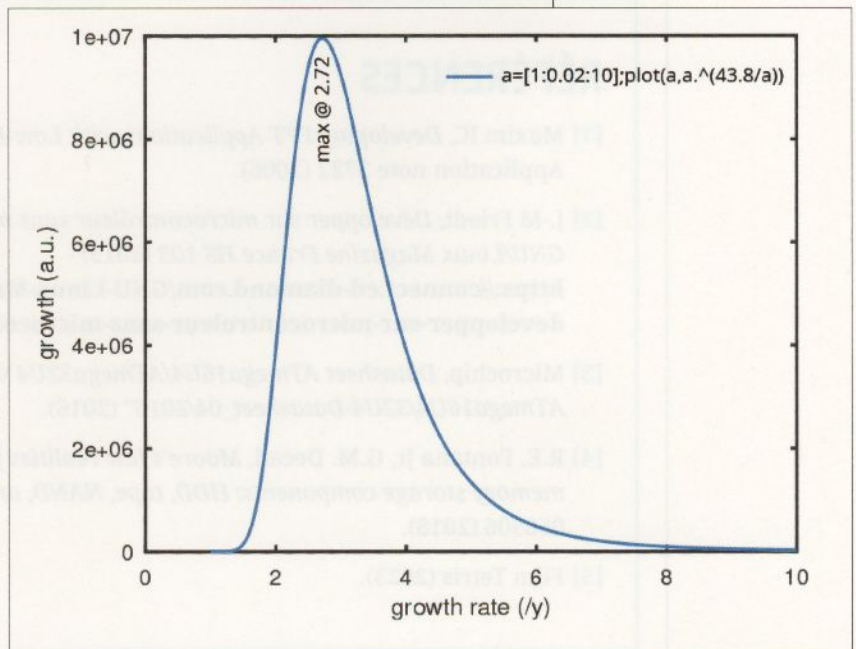
signifiant que le symbole utilisé entre les deux fréquences balayées définit la nature du balayage : l'utilisation de « - » comme séparateur se traduit par « *the tone will change by a fixed number of semitones per second.* », donc une évolution exponentielle de la fréquence telle que nous l'observons.

CONCLUSION

Porter un code sur une architecture aux ressources contraintes est une opportunité de s'assurer de comprendre en détail le fonctionnement des outils utilisés, ici le langage C et quelques-unes de ses subtilités, afin d'utiliser aussi efficacement que possible les ressources disponibles. Ce faisant, nous avons atteint notre objectif d'implémenter une transformée de Fourier rapide – FFT – sur un microcontrôleur 8 bits ne proposant que 2,5 kB de mémoire volatile.

La disparition de Gordon Moore, fondateur d'Intel et auteur de la loi empirique qui porte son nom annonçant que le nombre de transistors double tous les deux ans, ne doit pas nous faire oublier que la consommation électrique d'un processeur croît comme le nombre de commutations des portes logiques, donc croît avec la fréquence de cadencement et le nombre de processeurs. Cependant, la vitesse n'est qu'un paramètre d'un système de calcul qui ne met en jeu que notre patience à attendre la fin du calcul. Plus dramatique est la taille de stockage : alors que nous avons commencé à programmer sur des disquettes ou

Figure 4 : Gain d'un facteur n toutes les n années pour $n \in [1:10]$ sur une durée permettant d'atteindre un gain de 10^7 .



cassettes d'une capacité de 360 kB, il est maintenant devenu courant d'acquérir un support de stockage qualifié en téraoctets, soit 7 ordres de grandeur en 40 ans, avec une conséquence bien plus dramatique sur notre capacité à appréhender une telle masse d'informations.

Une croissance régulière d'une propriété, par exemple d'un facteur r tous les y ans, s'exprime classiquement par une loi exponentielle de la forme $f(t) = f(0) \times r^{(t/y)}$ avec f la grandeur analysée en fonction du temps t de valeur initiale $f(0)$. Afin de rendre la loi plus facile à retenir, il est classique d'annoncer qu'une grandeur « multiplie sa capacité d'un facteur n tous les n ans », imposant donc que r et y soient égales : la loi devient donc $n^{(t/n)}$ et nous pouvons nous demander quel n induit la croissance la plus rapide, par exemple une multiplication modeste, mais rapide des capacités (n petit) ou une multiplication importante mais lente (n grand) : la figure 4 indique le facteur multiplicatif après 43,8 ans ajusté pour atteindre le facteur 10^7 de croissance de la capacité de stockage, qui est atteint pour une croissance d'un facteur 2,72 tous les 2,72 ans. La loi de Gordon Moore doublant la capacité tous les deux ans empiriquement observée pour le nombre de transistors dans les processeurs, donc $n = 2$, permet dans la même durée de multiplier le nombre d'unités de calcul d'un facteur $2 \cdot 10^6$!

Il peut avoir du bon de se rappeler ce qu'on peut faire avec 2560 octets, ou pour se laisser un peu de marge avec 8 kB pour l'incroyable Tetris de la console portable Gameboy originale [5] ! Les codes décrits dans ce document sont disponibles à <https://github.com/jmfriedt/13ep> dans le sous-répertoire FFT. **JMF**

RÉFÉRENCES

- [1] Maxim IC, *Developing FFT Applications with Low-Power Microcontrollers*, Application note 3722 (2006).
- [2] J.-M Friedt, *Développer sur microcontrôleur sans microcontrôleur : les émulateurs*, GNU/Linux Magazine France HS 103 (2019) - <https://connect.ed-diamond.com/GNU-Linux-Magazine/glmfhs-103/developper-sur-microcontroleur-sans-microcontroleur-les-emulateurs>
- [3] Microchip, *Datasheet ATmega16U4/ATmega32U4 version "Atmel-7766J-USB-ATmega16U4/32U4-Datasheet_04/2016"* (2016).
- [4] R.E. Fontana Jr, G.M. Decad, *Moore's law realities for recording systems and memory storage components: HDD, tape, NAND, and optical*, AIP Advances 8(5) 056506 (2018).
- [5] Film Tetris (2023).

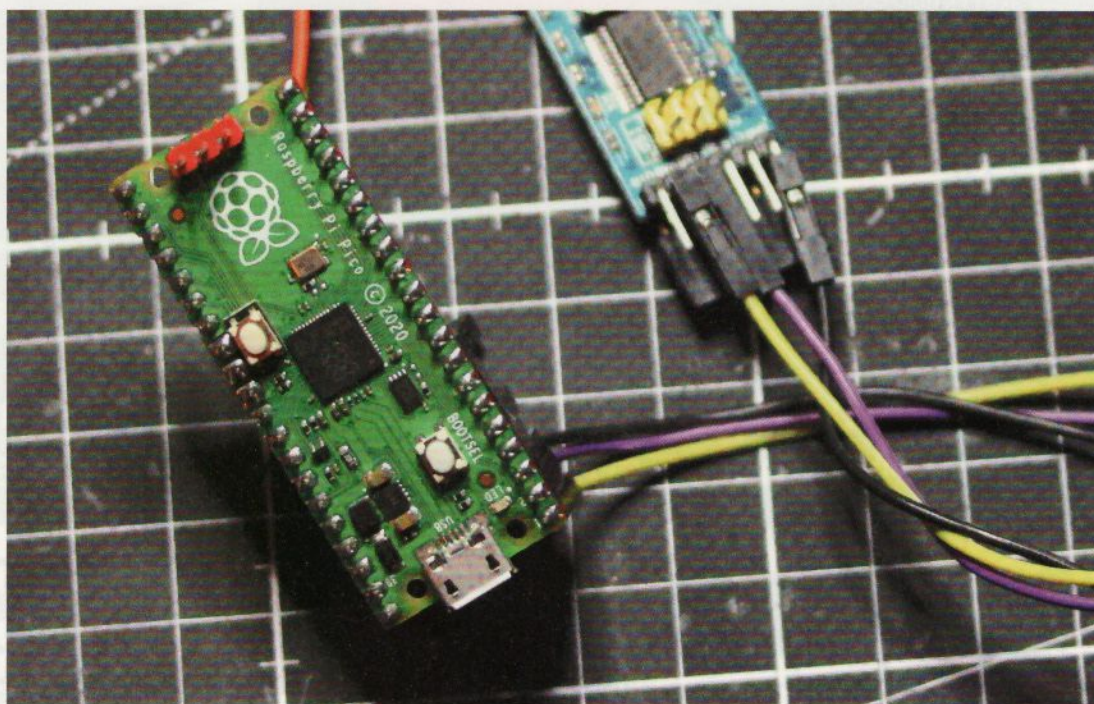
FUZIX : UN « VRAI » UNIX POUR RASPBERRY PI PICO !

Denis BODOR

À l'arrivée de la carte Raspberry Pi Pico, la principale difficulté était de faire comprendre au plus grand nombre qu'il ne s'agissait absolument pas d'un SBC (*Single Board Computer*), comme tout le reste de la famille de Raspberry Pi qui l'a précédé, mais de quelque chose de totalement différent : une carte microcontrôleur plus proche d'un Arduino, d'un ESP8266 ou d'une Bluepill (STM32).

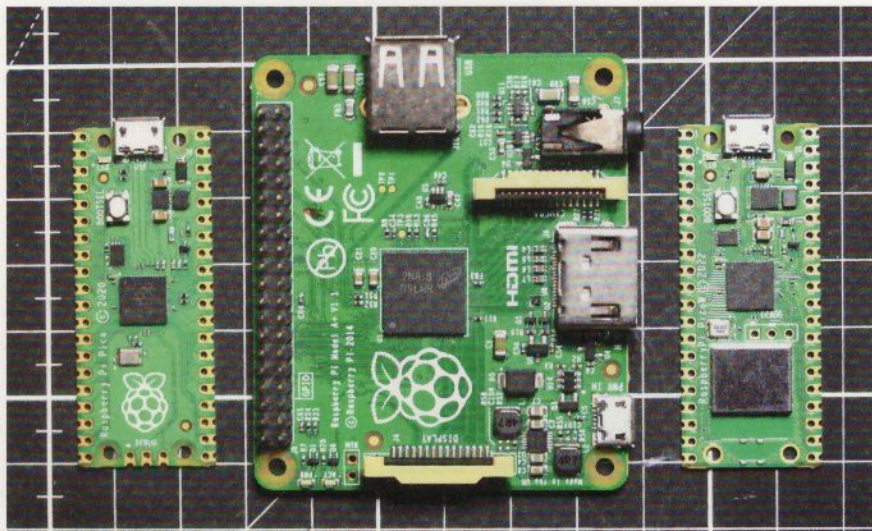
Pas question donc de faire fonctionner une distribution comme Raspbian (maintenant Raspberry Pi OS) ou un système de type Unix comme GNU/Linux... Ah oui ? Vraiment ? Pas tout à fait...

Floutons un peu les lignes, voulez-vous ?



Dans un précédent article du numéro 46 [1], nous avons expérimenté autour d'ESPPDP, un émulateur/simulateur SIMH de DEC PDP-11, porté sur Espressif ESP32, permettant de faire fonctionner 2.11BSD, l'une des premières versions d'UNIX issues du CSRG (Computer Systems Research Group) de Berkeley (voir la page Wikipédia [2] de BSD, le vrai). Il n'était donc pas réellement question de disposer d'un système Unix pour ESP32 puisque tout cela reste totalement simulé.

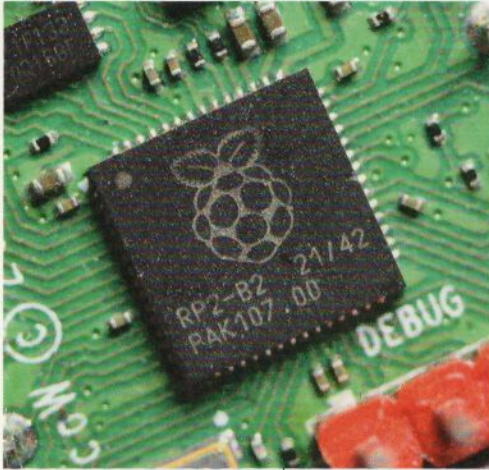
Mais aujourd'hui, nous allons voir quelque chose de différent : un système d'exploitation Unix destiné à fonctionner sur une architecture 8 bits, pour de vrai. Il s'agit de FUZIX, un système s'inspirant des anciens UNIX (V7, System V, etc.), construit/structuré de la même manière et bien entendu, écrit en C. FUZIX est toujours activement développé et est un projet initié en 2014 par Alan Cox [3] et annoncé, non sans humour, par cette magnifique série de questions : « Gavé par systemd ? Kdbus était la goutte d'eau ? La communauté Linux est trop grande et bruyante ? Vous rêvez du bon vieux



temps où vous connaissiez les contributeurs par leurs noms et où les sources pouvaient tenir une unique disquette ? » (« Fed up of systemd? Kdbus the final straw? Linux community too large and noisy? Yearn for the good old days when you knew every contributor by name and the source code fitted on a single floppy disc? »). Amusant de constater qu'il y a 10 ans, on ne se doutait pas que la situation pourrait encore bien empirer (Docker, Ansible, K8S, JS partout, et j'en passe).

L'idée derrière FUZIX n'est pas simplement ludique, il s'agit d'avoir un système Unix à une taille compréhensible par un humain en un temps raisonnable, mais ayant, tout de même, des fonctionnalités dignes d'intérêt et une architecture « traditionnelle » qu'on a maintenant bien du mal à réellement discerner avec un GNU/Linux et même, dans une certaine mesure avec un [Free | Open | Net]BSD (« pataper ! », ces systèmes sont par définition plus complexes qu'un 2.11BSD). Précisons cependant de suite que FUZIX est un développement en cours et qu'un certain nombre de fonctionnalités sont encore absentes, dont le support complet de TCP/IP, ce qui n'est pas un problème dans le cas des Raspberry Pi Pico « standard » (non, « W ») ou encore le `ptrace`, `select()`/`poll()` et des systèmes de fichiers de plus de 32 Mio.

Ça ressemble à des framboises, ça a la couleur des framboises, c'est plein de GPIO comme des framboises, mais ce ne sont pas les mêmes framboises et ça ne fait pas tout à fait la même confiture...



Le RP2040 est un microcontrôleur ARM Cortex-M0+ bicœur très performant intégrant RAM, flash et divers périphériques en une seule et unique puce.

Mais l'objet même de FUZIX est d'être utilisable et compréhensible sur des architectures qui le sont tout autant. Le système est ainsi disponible pour DEC PDP-11, MOS 6502, Motorola 68000, Zilog Z80/Z180, Intel 8080/8085/8086, Ti MSP430, esp8266 et, bien sûr ARM32 (Cortex-M0 et M4). C'est donc un système très intéressant pour le *retrobrew*, puisqu'il est ini-

tialement développé dans cette optique et pour des plateformes modestes des années 80 et 90. Mais acheter ou construire un ordinateur 8 bits n'est pas à la portée de tous (financière ou temporelle) et notre « victime » du jour sera donc une carte microcontrôleur Raspberry Pi Pico.

Une fois FUZIX installé, vous vous retrouverez donc avec un système interactif, prenant en charge le matériel et capable d'exécuter des programmes, un shell, des services, de monter

des systèmes de fichiers, etc., que vous pourrez entièrement personnaliser. Au point d'aller fouiller dans les sources du noyau pour y faire des horreurs, puisque c'est un projet en logiciel libre sous licence GNU GPLv2.

1. INSTALLATION

Le site officiel de FUZIX [5] met à disposition un certain nombre d'images de systèmes prêts à l'usage pour différentes plateformes génériques et/ou ordinateurs 8 bits (Amstrad NC200, Tandy TRS80, MTX512, Sinclair Timex, etc.) ainsi que des systèmes de fichiers racines (*rootfs*) pour des architectures de base (6809, Z80, 68000, etc.). Comme nous avons un *vrai* système

BARE METAL, RTOS ET OS

Qu'il s'agisse d'une plateforme basée sur un microcontrôleur (MCU) ou sur un processeur intégrant un certain nombre de périphériques (SoC ou APU), il faut distinguer plusieurs types de programmation dont le choix impactera les performances, la compacité du code, les ressources utilisées et la facilité de maintenance, d'évolution, de portabilité et de développement.

La programmation dite « *bare metal* » (métal brut ou nu) est généralement utilisée sur microcontrôleurs, avec des ressources limitées. Là, l'ensemble du code que le développeur produit a pour tâche de gérer entièrement le matériel. Il n'y a pas de système d'exploitation et le code machine fonctionne directement sur le matériel sans aucune abstraction, si ce n'est via d'éventuelles bibliothèques mises à disposition par le fabricant. La notion de tâches concurrentes n'existe pas en tant que telle, bien qu'il soit possible pour un code d'utiliser plusieurs cœurs, voir de paralléliser les opérations (avec des *coroutines*, par exemple).

Un cran au-dessus en termes d'abstraction, nous avons le RTOS (*Real Time Operating System*) qui, comme son nom ne l'indique pas vraiment, ne se limite plus vraiment à la notion de temps réel, par un étrange effet d'abus de langage. Des systèmes comme FreeRTOS (voir article de Jean-Michel Friedt

d'exploitation, celui-ci se divise en deux parties : le noyau (*kernel*) et l'espace utilisateur, contenant les programmes que vous pouvez exécuter. Ces systèmes de fichiers racines vous évitent de devoir tout recompiler et de vous concentrer sur le noyau, afin de l'adapter aux spécificités de votre matériel et donc porter FUZIX.

Dans notre cas, nous n'avons pas de développement à faire puisque, grâce au travail de David Given [6], qui est maintenant intégré aux sources officielles, le portage pour Raspberry Pi Pico est déjà prêt. Nous serons, en revanche, obligés de recompiler l'ensemble du système puisqu'il n'existe pas d'image toute faite. Ce

n'est pas un problème, bien au contraire, puisque c'est plus ou moins le « style de la maison », mais aussi du fait que les configurations actuelles rendent l'opération excessivement rapide (littéralement 16 secondes sur un bi-Xeon E5520 de plus de 10 ans avec 24 Go de RAM).

Pour construire FUZIX, vous aurez besoin d'une machine sous GNU/Linux (le *cross*-compilateur ARM de FreeBSD 13.2 ne semble pas d'humeur), disposant d'un compilateur ARM (*arm-none-eabi*) et de l'environnement de développement (SDK) Raspberry Pi Pico correctement installé et prêt à l'emploi. Normalement, les dépendances induites par le SDK sont suffisantes pour rendre la construction de FUZIX possible, il n'y a rien d'autre à installer.

Concernant la carte Raspberry Pi, vous aurez également besoin, mais ce n'est pas strictement nécessaire, d'un support de carte SD pour pouvoir monter des systèmes de fichiers et étendre les capacités de stockage, ainsi qu'un adaptateur USB/série. Une option permet d'utiliser le port USB de la Pico comme interface utilisateur (pour accéder au shell), mais le fait de plutôt utiliser un adaptateur USB distinct nous permet de voir les messages de démarrage et les éventuelles erreurs. En l'absence de cet ajout, il reste possible d'utiliser FUZIX, mais de façon non optimale puisqu'il faut attendre le démarrage du noyau pour voir apparaître l'interface série CDC...

dans le précédent numéro [4]), Contiki, RIOT ou encore Zephyr vont bien au-delà du pur RTOS. En laissant de côté l'aspect temps réel, la tâche d'un RTOS est de fournir au programme (et non à un utilisateur) une interface unique lui permettant d'interagir avec le matériel. Le RTOS gère également la concurrence des tâches et la protection de la mémoire, mais ne doit pas être vu pour autant comme un système tel que GNU/Linux ou macOS, proposant une ligne de commande ou une interface permettant de lancer et stopper des programmes. La plupart du temps, le code du développeur est compilé avec le RTOS pour former un unique binaire, sans chargement/déchargement ou système de fichiers.

Et enfin, nous avons l'OS ou système d'exploitation « généraliste », comme vous pouvez en utiliser sur votre PC/Mac, vos SBC de type Raspberry Pi ou encore sur un smartphone. En plus de gérer l'accès au matériel, la protection de la mémoire, l'accès aux réseaux et la concurrence des tâches, il permet à l'utilisateur de choisir quels programmes sont démarrés et arrêtés, via une interface (en ligne de commandes ou graphique). Ce à quoi peut éventuellement s'ajouter également une notion temps réel si nécessaire pour les tâches nécessitant un contrôle précis de l'exécution, avec des contraintes temporelles dans l'exécution des traitements.

Le microcontrôleur RP2040 dispose de 2 Mio de flash NAND et c'est là que se trouvera, par défaut, le système de fichiers racine (en lecture/écriture), ainsi que de 269 Kio de RAM, ce qui est largement suffisant pour satisfaire FUZIX. Les deux cœurs Cortex M0+ ne sont pas utilisés puisque le système ne le gère pas (encore), mais la fréquence par défaut de 130 MHz est plus qu'il n'en faut pour rendre le système très réactif, même avec un seul cœur actif.

Il est important de noter que la documentation de FUZIX n'est pas « optimale » et que l'intégration des différents supports pour les plateformes n'est pas nécessairement reflétée correctement dans cette dernière. Mais commençons par le commencement en téléchargeant les sources du système depuis GitHub :

```
$ cd kkpарт
$ git clone https://github.com/EtchedPixels/FUZIX.git
Clonage dans 'FUZIX'...
remote: Enumerating objects: 65681, done.
remote: Counting objects: 100% (6907/6907), done.
remote: Compressing objects: 100% (2185/2185), done.
remote: Total 65681 (delta 4730), reused 6828 (delta 4696)
Réception d'objets: 100% (65681/65681), 18.72 MiB | 19.74 MiB/s, fait.
Résolution des deltas: 100% (47194/47194), fait.

$ cd FUZIX
```

L'arborescence des sources se divise principalement en quelques répertoires importants :

- **Kernel/** qui, comme son nom l'indique, contient les sources du noyau (assembleur et C) ainsi que des sous-répertoires spécifiques à chaque plateforme supportée, et donc **platform-rpipico** (et **cpu-armm0**) pour celle que nous utilisons ici.
- **Applications/** contient les sources des outils de l'espace utilisateur (*userland*) comme le shell, les outils, les jeux, etc.
- **Library/** qui regroupe toutes les bibliothèques système, en version source, dont une libc permettant aux programmes standard de fonctionner, ainsi que les fichiers d'en-tête (**includes/**).
- **Standalone/** : contient différents outils « à part » ainsi que les fichiers de configuration du système, qui se retrouveront dans **/etc** sur la cible.

Théoriquement, avec le SDK Pico installé, tout ce que vous avez à faire est d'éditer le fichier **Makefile** et de changer la ligne :

```
TARGET=rc2014
```

en :

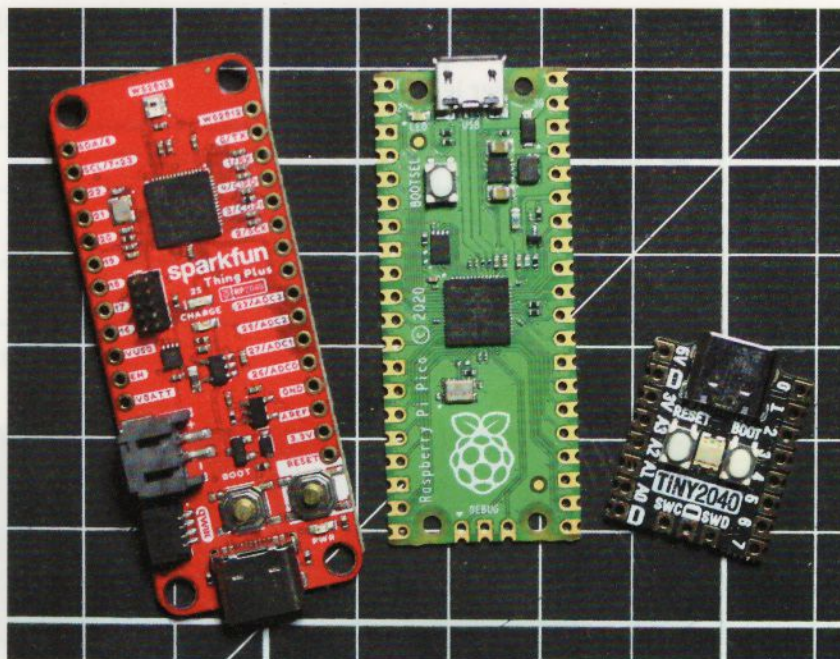
```
TARGET=rpipico
```


– FUZIX : un « vrai » Unix pour Raspberry Pi Pico ! –

Puis de lancer la construction avec un simple `make` (ou plutôt un `make -j 'nproc'` pour utiliser tous les cœurs/*threads* disponibles). Mais ça, c'est la théorie, car même si vous obtenez bien les deux fichiers UF2 (`fuzix.uf2` et `filesystem.uf2`) dans `Kernel/platform-rpipico` (et non `Images/rpipico` comme on devrait le supprimer), vous avez également un message d'erreur : « `chmod: can't stat file 'issue'` » et une partie des commandes/outils qu'on s'attend à trouver dans un système Unix seront absents.

Nous devons donc procéder à quelques ajustements et ceci s'étend également au support matériel et à la carte SD en particulier. Le premier problème, le plus évident, concerne l'absence du fichier `issue`, qui est référencé dans `Kernel/platform-rpipico/update-flash.sh`. Placez-vous donc dans ce répertoire (car nous pouvons lancer la construction depuis cet endroit) et éditez le script en question. Le fichier manquant est attendu dans `Standalone/filesystem-src/etc-files/issue` alors qu'il se trouve, en réalité, dans `Standalone/filesystem-src/templates/issue`.

`update-flash.sh` est un script shell, mais celui-ci fait usage de `ucp` (`Standalone/ucp`) compilé en même temps que



le système. Cet outil permet de composer et de manipuler un système de fichiers FUZIX et nous trouvons donc, dans le script, une série de commandes proches de celles d'un shell, créant des répertoires, copiant des fichiers et changeant les permissions. À noter que ce même outil, `ucp`, nous permettra également de manipuler le système de fichiers que nous créerons sur la carte SD plus tard et donc de facilement transférer des fichiers entre la machine de développement et la carte Pico.

Tant que nous y sommes et puisque nous éditons `update-flash.sh`, nous pouvons ajouter quelques outils qui manquent à l'appel, comme `uname` par exemple. Le système de construction a compilé l'ensemble des outils utilisateur pouvant aller dans le système de fichier racine, mais `update-flash.sh` n'en copie qu'une partie. Si vous jetez un œil dans `Applications/util` (`../../../../Applications/util` même), vous y trouverez une bonne centaine de binaires déclinés en une version *debug* et une version

Depuis quelque temps déjà, le microcontrôleur RP2040 est disponible seul et un certain nombre de fabricants comme SparkFun (à gauche) ou Pimoroni (à droite) proposent leur propre carte de développement avec des caractéristiques sensiblement différentes de la Raspberry Pi Pico.

stripée (sans extension et sans symbole de débogage). Pour ajouter celui ou ceux qui vous intéressent, vous n'avez que quelques petites lignes à ajouter au script. Par exemple, pour **uname**, nous devons copier le fichier :

```
bget ../../Applications/util/uname
```

Et ne pas oublier de le rendre exécutable :

```
chmod 0755 uname
```

Depuis le répertoire **Kernel/platform-rpipico**, nous pouvons alors reconstruire une image avec :

```
$ make -j `nproc` image
[...]
./update-flash.sh
Making 512 byte/block filesystem with normal byte order
on device filesystem.img with fsize = 2547 and isize = 32.
../../Standalone/mkftl -s 1952 -e 0x1000 -g 10
filesystem.img -o filesystem.ftl
Number of physical erase blocks: 488
Maximum logical size: 1277kB (2555 sectors)
Used space: 1273kB (2547 sectors)
arm-none-eabi-objcopy -I binary -O elf32-littlearm
--change-section-vma .data=0x10018000
filesystem.ftl filesystem.elf
./uf2conv filesystem.ftl filesystem.uf2 0x10018000
Wrote 7808 blocks to filesystem.uf2
```

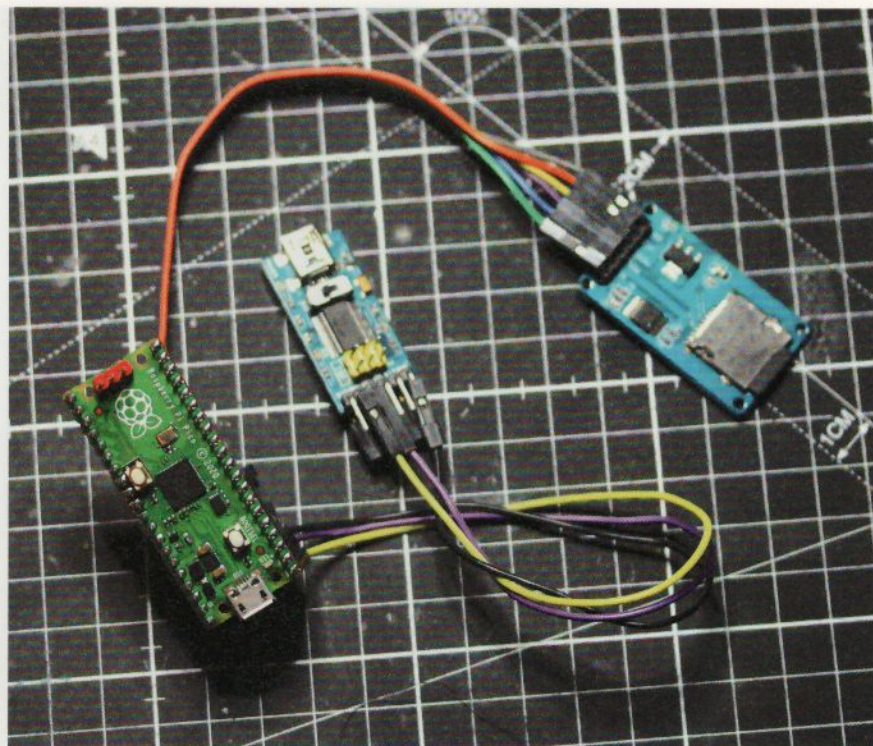
Nous pouvons alors utiliser les deux fichiers UF2 en les copiant simplement sur le support de stockage qui apparaît en connectant la Pico tout en maintenant le bouton BOOTSEL enfoncé. Personnellement, je n'apprécie pas particulièrement cette technique et préfère grandement utiliser **picotool** fourni avec le SDK Pico. Une fois la carte passée en mode BOOTSEL, on pourra, par exemple, utiliser les commandes suivantes :

```
$ picotool load -v ./build/fuzix.bin
Loading into Flash: [=====] 100%
Verifying Flash:   [=====] 100%
OK

$ picotool load -v filesystem.ftl -t bin -o 0x10018000
Loading into Flash: [=====] 100%
Verifying Flash:   [=====] 100%
OK
```


– FUZIX : un « vrai » Unix pour Raspberry Pi Pico ! –

Il est également possible d'utiliser les fichiers UF2 avec **picotool** et non simplement les binaires bruts. Si tel est votre choix, il ne sera alors pas nécessaire de spécifier une adresse de chargement avec l'option **-o** pour le système de fichiers racine (**filesystem.ftl**), car celle-ci est directement intégrée dans le format. Remarquez d'ailleurs la structure utilisée, avec un noyau chargé par défaut à l'adresse **0x10000000** et le système de fichiers, dans la même flash NAND, plus loin.



2. DÉMARRAGE, CONFIGURATION, SD ET CORRECTIONS

Comme nous n'avons pas utilisé l'option **-x** pour réinitialiser la carte après programmation, nous avons l'opportunité d'y connecter l'adaptateur USB/série (broches GP0, GP1 et GND) et de lancer un émulateur de terminal comme GNU Screen ou Minicom (115200 8N1). Dès le *reset* manuel (en mettant la broche RUN à la masse ou en déconnectant/reconnectant l'USB), vous devriez voir ceci défiler devant vos yeux écarquillés :

Le montage idéal pour tester FUZIX dans de bonnes conditions se compose d'une carte Pico, d'un adaptateur USB/série (3,3V) et d'un support pour carte SD ou microSD.

```
FUZIX version 0.4pre1
Copyright (c) 1988-2002 by H.F.Bower, D.Braun, S.Nitschke, H.Peraza
Copyright (c) 1997-2001 by Arcady Schekochikhin, Adriano C. R. da Cunha
Copyright (c) 2013-2015 Will Sowerbutts <will@sowerbutts.com>
Copyright (c) 2014-2023 Alan Cox <alan@etchedpixels.co.uk>
Devboot
264kB total RAM, 160kB available to processes (15 processes max)
Enabling interrupts ... ok.
NAND flash, 1952kB physical 1296kB logical at 0x13018000: hda:
SD drive 0: no card found
```



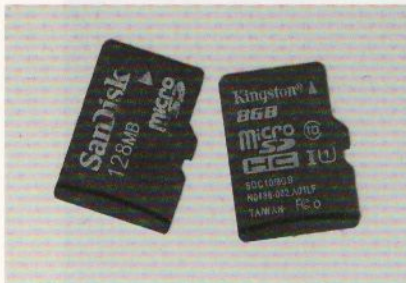
```
Mounting root fs (root_dev=0, ro): warning: mounting dirty file system,
forcing r/o.
OK
Starting /init
init version 0.9.1
Cannot open file
Current date is Fri 2023-06-02
Enter new date:
Current time is 15:18:39
Enter new time:

^ ^
n n   Fuzix XXX
>@<
      Welcome to Fuzix
m m

login: root

Welcome to FUZIX.
# uname -a
Fuzix 0.4pre1 rpipico
# df
Filesystem      Blocks   Used  Free  %Used Mounted on
/dev/hda         2515   1921   594   76% /
# mount
/dev/hda mounted on / read-write
#
```

FUZIX n'étant en mesure de gérer que des systèmes de fichiers d'une taille maximum de 32 Mio, et ce, avec un nombre de partitions maximum de 16, n'importe quelle SD fera l'affaire. En particulier, celles certainement inutilisables pour d'autres projets, à base de SBC, comme celle-ci (en haut), de seulement 128 Mio.



Le microcontrôleur RP2040 ne dispose pas de RTC et FUZIX est donc incapable de connaître, et de conserver, la date et heure entre deux démarrages. Il vous proposera donc de valider les informations provenant de la construction du système. Suite à quoi, vous arriverez, tout naturellement, sur l'invite de `login`. Le seul compte présent est **root** et il ne dispose pas de mot de passe. Notez que notre **uname**, ajouté manuellement, fonctionne à merveille et qu'il nous reste environ 25 % d'espace de stockage, soit 594 blocs de 512 octets soit presque 300 Kio. Le système de fichiers racine est monté en lecture/écriture, ce qui signifie que vous pouvez parfaitement utiliser le système normalement et que vos informations perdureront entre deux démarrages.

Il y a juste un petit problème. Si vous redémarrez la carte, vous verrez apparaître le message suivant : « *Mounting root fs (root_dev=0, ro): warning: mounting dirty file system, forcing r/o* ». La racine est ensuite remontée en lecture/écriture, mais ce n'est pas la bonne façon de procéder, à moins qu'on aime les données corrompues. En réalité, le problème vient, encore une fois, de **update-flash.sh**, qui oublie des choses.

– FUZIX : un « vrai » Unix pour Raspberry Pi Pico ! –

Mais avant de corriger cela, et comme le problème est lié aux systèmes de fichiers, nous pouvons nous pencher sur la carte SD. Le fichier `README.md` dans `Kernel/platform-rpipico` précise :

If you have an SD card reader, connect the SD card to the following pins:

Pico pin	RP2040 pin	SD card pin
16	12	MISO
17	13	CS
19	14	SCK
20	15	MOSI

Si vous le faites, et insérez une carte dans le support, vous aurez le même sympathique message que celui apparaissant déjà : « *SD drive 0: no card found* ». Et pour cause, le brochage détaillé dans le `README.md` ne correspond pas au code qui, lui, dit :

```
#ifndef CONFIG_RC2040

/* RC2040 board */
/* Pico SPI GPIO connected to SD SPI1 */
#define Pico_SD_SCK 14
#define Pico_SD_TX 15
#define Pico_SD_RX 12
#define Pico_SD_CS 13

//Pico spi0 or spi1 must match GPIO pins used above.
#define Pico_SD_SPI_MOD spi1

#else

/* Pico SPI GPIO connected to SD SPI0
- David Given's Arrangement */
#define Pico_SD_SCK 2
#define Pico_SD_TX 3
#define Pico_SD_RX 4
#define Pico_SD_CS 5

//Pico spi0 or spi1 must match GPIO pins used above.
#define Pico_SD_SPI_MOD spi0

#endif
```

Le RC2040 [7] est un projet intéressant de carte basée sur la Pico, émulant un système Z80 et faisant fonctionner un système CP/M (celui-là même qui a servi d'inspiration à MS-DOS). Il semble donc logique que ce soit possible de faire fonctionner ce FUZIX sur cette carte. Mais c'est plutôt l'autre partie du `ifndef` qui est intéressante, car le brochage est totalement différent de celui du `README`. Si nous connectons le support SD de cette façon (SCK=2, MOSI=3, MISO=4 et CS=5), subitement, la carte est parfaitement reconnue :


```
NAND flash, 1952kB physical
1296kB logical at 0x13018000: hda:
SD drive 0: hdb: hdb1 hdb2 hdb3 hdb4
```

Les partitions utilisées sont des partitions PC standard (DOS), mais le système de fichiers FUZIX ne supporte pas plus de 32 Mio. Pour éviter de gâcher du matériel, on s'attachera à trouver la plus petite carte SD de sa collection, ici une microSD de 128 Mio (oui, « Mio », pas « Gio ») qui sera partitionnée en autant de morceaux de 32 Mio que nécessaire. Cela vous semble peu ? Dites-vous que tout le système, kernel inclus, tient actuellement dans 2 Mio de NAND avec encore 300 Kio libres. Je pense que même une seule partition de 32 Mio sera déjà bien suffisante. Après tout, il n'y a ni LibreOffice ni Firefox ni systemd sous FUZIX...

Ce nouvel agencement des connexions rendra notre SD opérationnelle, ce qui nous ramène au problème précédent concernant le fait d'arrêter proprement le système en gérant correctement le système de fichiers. C'est quelque chose que vous pouvez faire directement puisque tous les outils sont présents. Ainsi, une fois le système démarré, vous pouvez créer un système de fichiers sur la première partition de la SD :

```
# mkfs -f /dev/hdb1 32 65535
Making filesystem on device /dev/hdb1
with isize 32 fsize 65535. Confirm? y
Clearing blocks .....
Building free list...
Writing initial inode and superblock...
Done.
```

Soyez patient, la seconde partie du « formatage » prend un certain temps. Nous pouvons ensuite monter ce système de fichiers sans le moindre problème :

```
# mkdir /mnt/sd1
# mount /dev/hdb1 /mnt/sd1
# df
```

Filesystem	Blocks	Used	Free	%Used	Mounted on
/dev/hda	2515	1912	603	76%	/
/dev/hdb1	65503	1	65502	0%	/mnt/sd1

Mais celui-ci, comme /, doit normalement être démonté avant de pouvoir couper l'alimentation de la carte ou, au minimum, être remonté en lecture seule. Inversement, un système de fichiers non démonté avant redémarrage risque d'être corrompu et devra être vérifié. Exemple :

```
# mount /dev/hdb1 /mnt/sd1
warning: mounting dirty file system, forcing r/o.

# umount /dev/hdb1

# fsck /dev/hdb1
Device 0 has fsize = 65535 and isize = 32. Continue? y
```



```

Pass 1: Checking inodes...
Pass 2: Rebuilding free list...
Rebuild free list? y
Pass 3: Checking block allocation...
Pass 4: Checking directory entries...
Pass 5: Checking link counts...

```

Tout le nécessaire est présent dans `Standalone/filesystem-src/etc-files`, mais, pour dire les choses sans vergogne, c'est un peu le bordel. Pourquoi ne pas laisser ce répertoire tranquille, ainsi que `update-flash.sh` et construire quelque chose de réellement sur mesure ? Personnellement, et bien que la flash NAND du RP2040 soit plus rapide qu'une SD, je préfère avoir un noyau en flash, mais un système de fichiers racine sur la première partition de la SD. Ce sera plus lent, mais nous aurons davantage d'espace pour nous faire plaisir et ce sera plus pratique pour transférer des fichiers.

3. PERSONNALISATION

Il y a beaucoup à faire, mais nous avons de quoi nous inspirer avec le contenu de `Standalone/filesystem-src/etc-files`. Comme nous ne voulons pas (trop) modifier les éléments originaux, nous allons commencer par créer un répertoire `my-etc-filesSD` dans `Kernel/platform-rpipico` pour y copier et créer les fichiers qui trouveront place, ensuite, dans `/etc/`.

`inittab`, `termcap`, `motd` et `issue` ne changent pas et peuvent être directement repris. Le reste, en revanche, va subir un petit lifting, à commencer par `rc`, le script de démarrage :

```

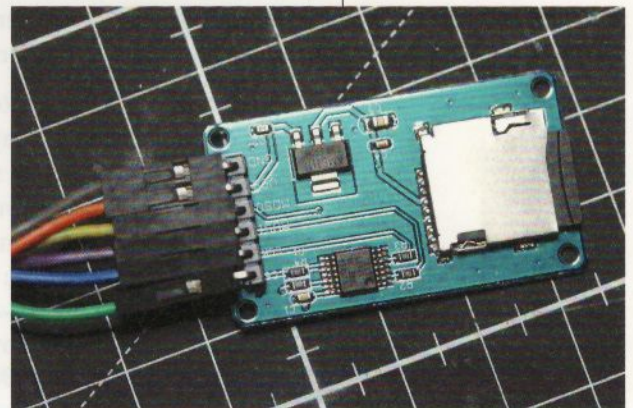
umask 022
fsck -a
setdate
remount -n / rw
prtrout >/etc/mtab
mount -a
>/var/run/utmp

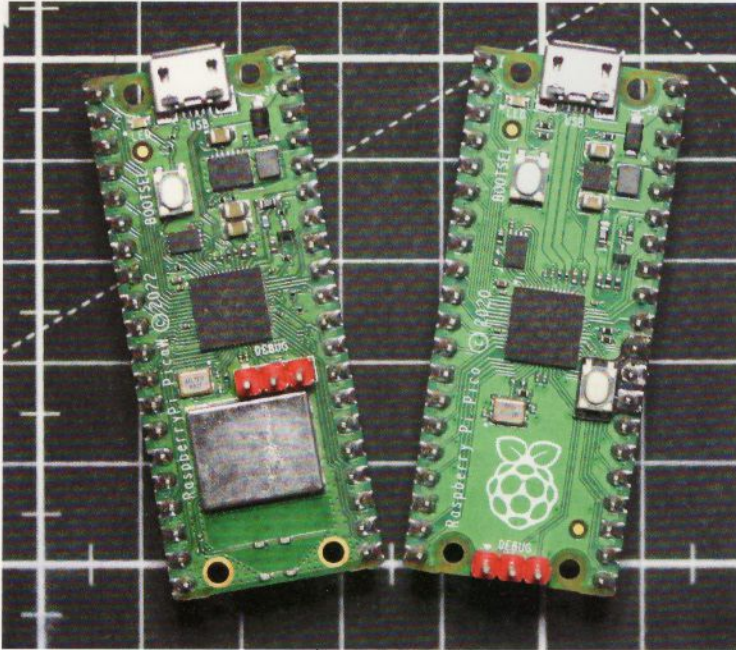
```

Ceci reprend le `rc` existant, mais part du principe qu'il y aura d'autres systèmes de fichiers présents à l'avenir, d'où le `fsck -a` pour vérifier tous les systèmes de fichiers présents dans `/etc/fstab` et `mount -a` pour les monter (si ce n'est déjà fait). Notre `fstab` ressemblera pour l'instant à ceci :

```
/dev/hdb1 / fuzix defaults 0 0
```

Les modules permettant de connecter une SD ou microSD à une carte microcontrôleur sont très faciles à trouver en ligne pour un budget très réduit. Attention cependant, la qualité est proportionnelle au prix et il faudra parfois se montrer tolérant. Ici, par exemple, même si ce n'est pas excessivement visible sur la photo, nous avons les broches GND, VCC, SCK, CS, MOSI et... MOSO (?).





Une récente déclinaison de la Pico, intégrant un contrôleur Wi-Fi, est disponible, c'est la Pico W. Ceci ne présente cependant aucun intérêt pour FUZIX, dont le support réseau n'en est qu'à ses balbutiements.

Notre racine n'est plus `/dev/hda`, mais plutôt la première partition de la carte SD, et nous allons immédiatement nous charger de gérer l'extinction correctement. Je vous fais ici la version courte, car ceci a demandé quelques fouilles dans les sources (mais après tout, c'est l'idée avec FUZIX). La commande `reboot` demande en réalité à `init` de changer de `runlevel` et ceci à pour effet d'invoquer des scripts placés dans `/etc` et spécifiés dans `inittab`. De plus, en fonction du nom de la commande, il s'agira d'un redémarrage ou d'une extinction. Pour nous, cela ne change pas grand-chose, mais nous devons le prendre en compte. Les deux scripts à créer sont totalement similaires. Voici celui pour le redémarrage, `rc.reboot` :

```
echo "Kill all process"
killall
echo "syncing disks"
sync
echo "Remount / (ro)"
remount / ro
sleep 1
echo "Done. Ready for reset"
```

Oui, mes messages sont en anglais, parce que c'est la langue de l'informatique et parce que personne n'a l'intention de faire parler FUZIX dans la langue de Molière (qui soit dit en passant était différente du français moderne). Le script pour l'arrêt, `rc.halt`, est identique, si ce n'est à la dernière ligne (`echo`). Ce simple `remount` réglera le problème de vérification forcée au redémarrage et nous protégera d'une corruption de données.

Il ne nous reste ensuite que deux autres fichiers, qui concernent la gestion des utilisateurs. Par défaut, FUZIX ne dispose que du super-utilisateur `root` et comme tout le monde le sait (ou devrait le savoir), utiliser `root` pour des tâches courantes, c'est mal ! Nous allons donc en profiter pour créer un utilisateur standard, `denis`, avec un ID utilisateur de `1000` et un ID groupe identique. Voici le fichier `passwd` révisé :

```
root::0:0:./root:/bin/sh
denis::1000:1000:Demo User:/home/denis:/bin/sh
```


Et le **group** qui va bien :

```
root:x:0:
denis:x:1000:w
```

Nous aurons donc à disposition un utilisateur standard, capable de basculer en super-utilisateur si nécessaire, via la commande **su**, qui est disponible.

Ces fichiers prêts, nous pouvons nous tourner vers le script composant le système de fichiers racine. Là encore, nous allons partir d'une base existante en copiant **update-flash.sh** sous le nom **update-sd.sh**. Dès le début du script, nous avons un choix à faire ; nous pouvons générer une image qui devra ensuite être transférée sur la première partition de la SD :

```
IMG=filesystemSD.img

rm -f ${IMG}
../../Standalone/mkfs ${IMG} 32 65535
../../Standalone/ucp ${IMG} <<EOF
```

ou nous pouvons directement manipuler la SD placée dans un lecteur USB (à condition de lancer le script avec **sudo**) :

```
IMG=/dev/sdd1

../../Standalone/mkfs ${IMG} 32 65535
```

Ici, mon lecteur USB/SD est **/dev/sdd** et la première partition **sdd1** (je pourrai également utiliser **/dev/disk/by-id/usb-Generic_Mass-Storage-0\:0-part1** pour limiter les risques de confusion). Notez les arguments de **mkfs** qui sont différents avec toujours des *inodes* de 32 octets, mais un nombre de blocs de 65535 ($65535 * 512 = 32 \text{ Mio}$) et non plus 2547 comme pour la flash NAND du microcontrôleur RP2040.

Nous passons ensuite à la copie des fichiers de configuration en supprimant ce qui est actuellement référencé via **../../Standalone/filesystem-src/etc-files** et en remplaçant par :

```
cd /etc
bget my-etc-filesSD/fstab
bget my-etc-filesSD/group
bget my-etc-filesSD/inittab
bget my-etc-filesSD/issue
bget my-etc-filesSD/motd
bget my-etc-filesSD/passwd
bget my-etc-filesSD/rc
bget my-etc-filesSD/rc.reboot
bget my-etc-filesSD/rc.halt
bget my-etc-filesSD/termcap
```

Sans oublier d'ajuster les permissions :

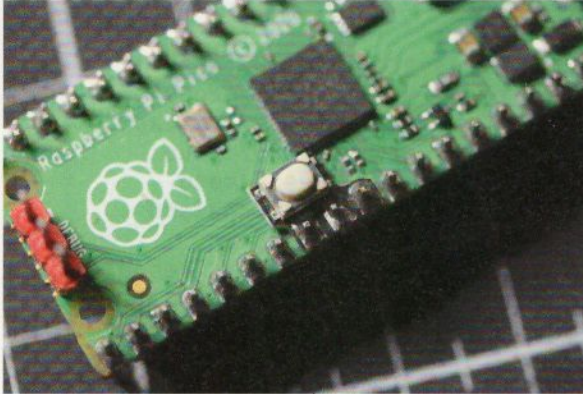
```
chmod 0644 issue
chmod 0644 motd
chmod 0644 passwd
chmod 0644 inittab
chmod 0644 termcap
chmod 0644 fstab
chmod 0644 group
chmod 0755 rc
chmod 0755 rc.reboot
chmod 0755 rc.halt
```

Notez que **rc**, **rc.reboot** et **rc.halt** sont des scripts et doivent donc être exécutables. Nous ajoutons également quelques créations de répertoires, pour plus tard (**/mnt/***), mais aussi pour notre utilisateur :

```
cd /
mkdir mnt/sd2
mkdir mnt/sd3
mkdir mnt/sd4
mkdir home
mkdir home/denis

cd /home
chmod 0755 denis
```

Et enfin, nous ajoutons les binaires qui manquent à l'appel dans la liste, mais qui sont effectivement disponibles après construction/compilation :



Fut un temps, Pimoroni commercialisait le « Captain Resetti », un petit bouton-poussoir à souder directement sur la Pico et permettant de procéder facilement à un reset du microcontrôleur. Ce produit est malheureusement épuisé et c'est tout à fait regrettable, car pour un projet comme celui-ci, l'avantage est évident.

```
cd /bin
bget ../../Applications/util/killall
ln reboot shutdown
chmod 0755 killall
```

Je pars ici du principe que vous avez déjà intégré **uname** précédemment, mais libre à vous d'ajouter ou non n'importe quoi se trouvant dans **Applications/** ou l'un de ses sous-répertoires (nous avons de la place maintenant).

Le script est maintenant prêt et nous pourrions parfaitement nous en satisfaire en l'appelant manuellement. Mais pourquoi se priver, puisque nous sommes déjà si bien engagés ? Autant intégrer cela directement dans le **Makefile** pour automatiser au maximum en ajoutant une cible, juste après la section concernant la cible **image** :

```
sd: world
    sudo ./update-sd.sh
```

Notez que le **sudo** n'est là qu'au cas où vous accéder directement à la SD depuis **update-sd.sh**. À présent, un simple **make sd**, à la place de **make image**, dans le répertoire **Kernel/platform-rpipico** générera directement notre SD, en construisant ce qui manque (dépendance **world**). Il ne nous reste alors plus qu'à expliquer au noyau que notre système de fichiers racine n'est plus la flash NAND (**hda**), mais se trouve sur la première partition de la SD. Il n'y a pas de **bootloader** avec FUZIX, mais une macro nous permet de spécifier le **boot device**. Celle-ci est déclarée dans **config.h** où nous trouvons actuellement :

```
#define BOOTDEVICE 0x0000 /* hda */
```

Cette valeur sur 16 bits est documentée dans **Kernel/start.c** (ligne 128) et encode la désignation du périphérique avec **hda=0x0000**, **hda1=0x0001**, **hda2=0x0002**, **hdb=0x0010**, etc. Notre **hdb1** sera donc :

```
#define BOOTDEVICE 0x0011 /* hdb2 */
```

C'est tout ! Il ne nous reste plus qu'à tout reconstruire avec un petit **make clean** suivi d'un :

```
$ make -j `nproc` sd
mkdir -p build
(cd build && cmake ..)
Using PICO_SDK_PATH from environment ('/home/denis/PIPICO/pico-sdk')
Using PICO_SDK_FETCH_FROM_GIT from environment ('yes')
```


- FUZIX : un « vrai » Unix pour Raspberry Pi Pico ! -

```
PICO_SDK_PATH is /home/denis/PIPICO/pico-sdk
Defaulting PICO_PLATFORM to rp2040 since not specified.
Defaulting PICO platform compiler to pico_arm_gcc since not specified.
-- Defaulting build type to 'Release' since not specified.
PICO compiler is pico_arm_gcc
-- The C compiler identification is GNU 10.3.1
[...]
make[1] : on quitte le répertoire
" /home/denis/HUBpub/tmp/FUZIX/Standalone "
./update-sd.sh
Making 512 byte/block filesystem with normal byte order on
device filesystemSD.img with fsize = 65535 and isize = 32.
```

Comme nous avons un nouveau noyau binaire, différent du précédent, il sera nécessaire de reflasher celui-ci sur la Pico avec `picotool load -v ./build/fuzix.bin` par exemple, ou en copiant le fichier `build/fuzix.uf2` sur le support USB émulé par la Pico en mode BOOTSEL. Le système de fichiers racine, en revanche, trouvera place sur la SD, et si vous avez préféré l'option « image » plutôt qu'un dangereux accès direct au support, il faudra le transférer avec `dd` :

```
$ sudo dd if=filesystemSD.img of=/dev/votre_lecteur_part1
31+1 enregistrements lus
31+1 enregistrements écrits
33553920 bytes (34 MB, 32 MiB) copied, 12,3293 s, 2,7 MB/s
```

Nous pouvons ensuite placer la SD dans le support, croiser les doigts et redémarrer la Pico :

```
FUZIX version 0.4pre1
Copyright (c) 1988-2002 by H.F.Bower, D.Braun, S.Nitschke, H.Peraza
Copyright (c) 1997-2001 by Arcady Schekochikhin, Adriano C. R. da Cunha
Copyright (c) 2013-2015 Will Sowerbutts <will@sowerbutts.com>
Copyright (c) 2014-2023 Alan Cox <alan@etchedpixels.co.uk>
Devboot
264kB total RAM, 160kB available to processes (15 processes max)
Enabling interrupts ... ok.
NAND flash, 1952kB physical 1296kB logical at 0x13018000: hda:
SD drive 0: hdb: hdb1 hdb2 hdb3 hdb4
Mounting root fs (root_dev=17, ro): OK
Starting /init
init version 0.9.1
/dev/hdb1:
Checking root file system.
Done.
Current date is Sat 2023-06-03
Enter new date:
Current time is 14:41:25
Enter new time:
```



```

^ ^
n n  Fuzix XXX
>@<
      Welcome to Fuzix
m m

login: root

Welcome to FUZIX.
# df
Filesystem            Blocks    Used   Free   %Used Mounted on
/dev/hdb1              65503    1958   63545    2% /

# /usr/games/cowsay Tadaaaa!
-----
< Tadaaaa! >
-----
      \      ^__^
         (oo)\_______
            (__)\       )\/\
                ||----w |
                ||     ||

#

```

32 Mio de stockage et 98 % libre pour notre plus grand plaisir. Notez le message au démarrage, `root_dev=17`, c'est notre `0x0011` en décimal. Mais nous n'en avons pas fini, car il reste un petit ajustement qu'il n'était pas possible de faire avec `ucp`, donner la propriété du répertoire `/home/denis` à son légitime utilisateur :

```

# cd /home
# chown denis denis
# chgrp denis denis
# ls -la
drwxr-xr-x  3 root  root   512 Jun 03 14:41 .
drwxr-xr-x 12 root  root   512 Jun 03 14:41 ..
drwxr-xr-x  2 denis denis  512 Jun 03 14:41 denis

```

Et nous en profitons pour ajouter un peu de sécurité (on sait jamais) :

```

# passwd
Changing password for root
New password:
Re-enter new password:
Password changed.

# ^D
^ ^
n n  Fuzix XXX
>@<
      Welcome to Fuzix
m m

```


– FUZIX : un « vrai » Unix pour Raspberry Pi Pico ! –

```
login: denis

Welcome to FUZIX.
$ id
uid=1000(denis) gid=1000(denis)
$ who
denis    tty01    Sat Jun 03 14:52:54 2023
$ pwd
/home/denis
$ touch toto
$ ls -la
drwxr-xr-x  2 denis    denis          512 Jun 03 14:53 .
drwxr-xr-x  3 root     root            512 Jun 03 14:41 ..
-rw-r--r--  1 denis    denis            0 Jun 03 14:53 toto
$ rm toto

$ passwd
Changing password for denis
New password:
Re-enter new password:
Password changed.

$
```

Nous pouvons effectivement nous connecter avec ce nouvel utilisateur, qui se retrouve dans son **\$HOME** comme prévu et il a toute liberté de faire ce qui lui chante dans ce répertoire. Magnifique !

Il ne nous reste plus qu'à passer **root** pour arrêter le système et espérer que nos scripts fonctionnent :

```
$ su
Password:
# shutdown
Kill all process
syncing disks
Remount / (ro)
Done. Ready for power off
```

Et un petit redémarrage plus loin, nous pouvons constater que nous n'avons aucun message d'erreur, le système de fichiers est propre, comme prévu.

4. POUR FINIR

Même si FUZIX est petit, nous n'avons couvert ici qu'une infime partie de ce qu'il est possible de faire. En réalité, avec ce système, nous avons réellement un microscopique ordinateur, sous Unix, que nous pouvons étudier, bricoler et personnaliser. Et ça, c'est, je trouve, tout simplement extraordinaire !

L'outil **ucp** est bien plus qu'un élément de construction du système de fichiers racine. C'est une interface à privilégier lorsqu'il s'agit d'accéder aux fichiers du système sur la Pico. En effet, en l'absence de connectivité réseau, il est relativement difficile de transférer des données dans un sens ou dans l'autre, sauf éventuellement via la liaison série et un encodage quelconque (comme Zmodem, à condition de porter l'outil).

C'est aussi pour cette raison que le fait d'opter pour un *rootfs* sur SD et non en flash NAND est une bonne idée. Il devient possible de simplement prendre le support, le glisser dans un emplacement ou un adaptateur USB sur PC et d'utiliser **ucp** en mode interactif. Pour cela, rien de plus simple, il suffit de l'invoquer via **sudo** en spécifiant le chemin du périphérique bloc contenant le système de fichiers FUZIX :

```
$ sudo Standalone/ucp /dev/sdd1
Fuzix UCP version 1.4ac. Type ? for help.
ucp: ls
drwxr-xr-x 12      1      512 Jun 03 123 15:13 ./
drwxr-xr-x 12      1      512 Jun 03 123 15:13 ../
drwxr-xr-x 2      51     4096 Jun 03 123 15:13 bin/
drwxr-xr-x 2      50     1536 Jun 03 123 15:13 dev/
drwxr-xr-x 2      49     512 Jun 03 123 15:14 etc/
drwxrwxrwx 5      48     512 Jun 03 123 15:13 mnt/
drwx--x--- 2      44     512 Jun 03 123 15:13 root/
drwxrwxrwx 2      43     512 Jun 03 123 15:13 tmp/
drwxr-xr-x 5      42     512 Jun 03 123 15:13 usr/
drwxr-xr-x 3      41     512 Jun 03 123 15:13 var/
drwxr-xr-x 3      40     512 Jun 03 123 15:13 home/
-rwxr-xr-x 1      2     11200 Jun 03 123 15:13 init
ucp: df
0: 1960 blks used, 63543 free; 189 inodes used, 51 free
ucp:
```

On peut alors, lister les fichiers (**ls** ou **dir**), créer des répertoires (**mkdir**), changer les permissions (**chmod**), supprimer des éléments (**rm/rmdir**), récupérer des fichiers (**bput**) ou en ajouter (**bget**), et même avoir une idée de l'occupation de l'espace (**df**). L'interface n'est pas idéale (un petit coup de *readline* serait bien agréable), mais cela fait le travail à merveille, tout en étant scriptable. Que du bonheur !

Mais le plus intéressant n'est pas ce qui apparaît à l'écran. Non, ce sont les sources du système, qu'il s'agisse du noyau, des bibliothèques ou des outils de l'espace utilisateur. Le noyau FUZIX tient dans quelque 36 000 lignes de C et d'assembleur, et l'espace utilisateur dans son ensemble, y compris ce qui n'a pas été installé/utilisé ici, ce n'est que quelque 500 fichiers source C (la plupart des programmes étant contenus en un unique fichier C). Que vous ayez envie de savoir ce que fait un système Unix, comment est organisé le système de fichiers ou que vous vouliez ajouter vos propres *syscall*, juste pour vous amuser, c'est possible ! Tout est possible !

– FUZIX : un « vrai » Unix... –

Et vous pouvez, bien sûr, apporter votre contribution au projet, qui est toujours très actif. Peut-être en ajoutant les commandes **chown** et **chgrp** dans **ucp** puisque celles-ci font défaut. Ou peut-être en ajoutant un peu d'ordre de-ci de-là, car, oui, soyons honnête, un petit brin de ménage ne ferait pas de mal (essayez simplement un **git status** après un **make clean** et vous comprendrez).

Personnellement, je trouve ce système absolument captivant et magique. Le travail réalisé par Alan Cox et les contributeurs pour les différentes plateformes supportées est tout simplement phénoménal. C'est une base de travail idéal pour mieux comprendre le fonctionnement d'un système d'exploitation (mieux que Minix, en réalité). Mon seul regret est que FUZIX ne se satisfait pas de 64 Kio d'espace d'adressage et de ce fait, n'est pas utilisable en l'état avec un ordinateur *retrobrew* simpliste à base de Z80 comme celui que vous pourrez découvrir dans le présent numéro. Mais qui sait, peut être que sur un Z180 avec 256 Kio de RAM... **DB**

RÉFÉRENCES

- [1] <https://connect.ed-diamond.com/hackable/hk-046/revivez-la-legende-unix-2.11bsd-pour-pdp-11-sur-un-esp32>
- [2] https://fr.wikipedia.org/wiki/Berkeley_Software_Distribution
- [3] <https://twitter.com/EtchedPixels>
- [4] <https://connect.ed-diamond.com/hackable/hk-048/freertos-dans-25-kb-de-ram-sur-atmega32u4>
- [5] <https://www.fuzix.org/>
- [6] <http://cowlark.com/2021-02-16-fuzix-pi-pico/>
- [7] <https://github.com/ExtremeElectronics/RC2040>

Chez votre marchand de journaux !

Et sur : ed-diamond.com

N°138 ADMINISTRATION SYSTÈME & RÉSEAU
LINUX PRATIQUE
 GNU
 ORCHESTRATION
 Prenez en main Nomad, un projet plus souple que Kubernetes pour l'orchestration de conteneurs p.20
 RÉSEAU / WEB
 Découvrez ce que sont les URI /well-known/ et leur réelle utilité p.67
 MULTIPLATEFORME / GO
 À la recherche d'une alternative à Apache légère, sécurisée et sans dépendances ?
TESTEZ CADDY, LE SERVEUR WEB ULTIME ! p.44
 • Présentation et avantages
 • Installation
 • Configuration
 E-COMMERCE
 Gros plan sur Sylius, une solution e-commerce headless bâtie sur Symfony p.74
 SYSTÈME / BSD
 « La grande migration », épisode 2 : mails et IPv6 p.08
 KUBERNETES / SNAPSHOT
 Protégez vos données et applications d'environnements conteneurisés à l'aide de TrilioVault p.31
 FRAIS DE PORT OFFERTS !*
 *Offre valable sur les publications en kiosque pour toute livraison en France Métropolitaine.

NOUVEAU !

LINUX PRATIQUE N°138



Également disponible en version lecture numérique Flipbook HTML5**



Retrouvez ce nouveau numéro, ainsi que l'intégralité de Linux Pratique sur notre base documentaire :



CONNECT
 LA DOCUMENTATION TECHNIQUE DES PROS DE L'IT

connect.ed-diamond.com

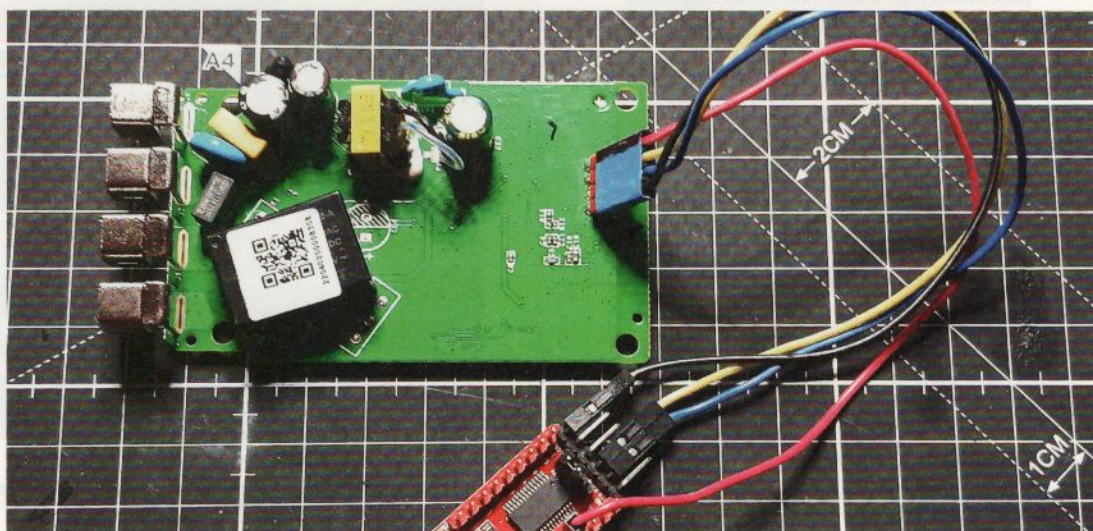


**L'offre Flipbook HTML5 est réservée aux clients particuliers.

SURVEILLEZ VOTRE CONSOMMATION ÉLECTRIQUE ET FAITES DES ÉCONOMIES

Denis BODOR

Comme pour tout le monde, les dernières factures de mon fournisseur d'énergie n'ont pas été une bonne nouvelle, bien au contraire. L'augmentation du coût de l'électricité a un impact notable sur nos finances et souvent, de manière générale, dès lors qu'il est question de technologie, faire des économies nécessite un certain investissement. L'idée est simple et consiste à engager un certain budget qui, dans un temps déterminé, sera remboursé par l'économie réalisée, puis deviendra un bénéfice net à long terme. Mais pour décider de ce qu'il faut changer, encore faut-il savoir où l'on peut effectivement faire des économies.



« **C**e n'est pas Versailles, ici ! » : vous vous souvenez sans doute de cette publicité mettant en avant des « services » pour réduire sa consommation et donc faire des économies. Bien que relativement amusante, cette campagne est cependant trompeuse dans le sens où l'expression semble être une référence aux somptueux luminaires du château de Versailles et aux dépenses exorbitantes investies par Louis XIV et son entourage dans le nombre de bougies consommées, qui se chiffrait en plusieurs dizaines de milliers d'exemplaires par bal organisé. Seulement voilà, aujourd'hui l'éclairage n'est pas le poste de consommation le plus important d'une habitation, loin de là, en particulier depuis l'abandon des ampoules incandescentes.

Lorsqu'on parle d'économie d'électricité dans une habitation, il faut généralement regarder du côté du chauffage, du ballon d'eau chaude, du four, du réfrigérateur, des plaques de cuisson, etc. Mais, là encore, ce sont des considérations théoriques qui demandent à être vérifiées, quantifiées et éventuellement révisées. Votre équipement informatique, par exemple, est-il plus ou moins consommateur que votre congélateur ? Quid de votre four ? Lever sérieusement le pied sur les pizzas et les poissons panés aura-t-il un effet visible sur votre facture (comme sur votre ligne) ? Ou plus massivement, est-ce que le cumul de tout un tas de petits changements d'habitudes, avec les difficultés que cela suppose (se reprogrammer n'est jamais facile), aura-t-il vraiment des conséquences positives notables ?

Le zélateur écologiste, roulant dans une vieille guimbarde brûlant autant d'huile que d'essence, mais affublé de l'indispensable autocollant « Nucléaire ? Non merci ! » vous répondra sans hésitation : « Oui ! Bien sûr ! Sauvons la planète ! ». Et il aura très

LINKY OR NOT LINKY... LA QUESTION NE SE POSE PAS.

La question ne se pose pas, non parce qu'il y aurait un quelconque problème avec ce matériel ou qu'il serait nécessaire de l'entourer de cristaux (programmés une nuit sans lune avec de l'eau provenant d'un haut lieu vibratoire cosmotellurique) pour se protéger contre les ondes OGM de la 5G du vaccin qui vous manipulent pour vous faire croire que la terre est ronde (alors qu'à l'évidence, c'est un disque creux triangulaire à cinq faces), mais tout simplement parce que je ne dispose que d'un bon vieux compteur électromécanique (vous savez, le modèle « vintage », noir, avec un disque qui tourne).

Nous n'avons pas pour habitude, dans ce magazine, de parler de choses qui n'ont pas été mises en pratique réellement, mais, bien qu'il soit possible d'utiliser la prise téléinformatique de ce type d'équipement pour suivre sa consommation, l'intérêt est assez limité. Je m'explique : ceci peut sembler être une affirmation étonnante, mais lorsqu'il s'agit de vouloir faire des économies, une valeur globale, pour toute la maison ou l'appartement, apporte peu de données et est assez peu pratique.

Comment allez-vous déterminer que votre four doit être utilisé avec parcimonie ? Arrêter de l'utiliser pendant une semaine et observer les effets ? Mais avez-vous vraiment utilisé les autres équipements (lave-linge, aspirateur, bouilloire, PC, etc.) exactement de la même manière que la semaine de référence ? Quid du congélateur ou du réfrigérateur ? Allez-vous également l'arrêter pour une semaine, histoire de voir s'il faut le remplacer par un modèle plus économe ?

Une unique mesure comme la consommation globale d'un habitat apporte une information générale, mais ne vous permet pas réellement d'identifier les problèmes ou les postes sur lesquels travailler. De plus, vous n'avez pas réellement besoin de téléinformation numérique, ou TIC (Télé-Information Client), pour cela. Il suffit de relever le compteur de façon hebdomadaire, faire un joli petit tableau dans LibreOffice et le tour est joué, vous avez votre information.



Principal avantage du vieux compteur électromécanique : il n'est pas peint dans le vert le plus horrible de l'Univers.

Donc non, je ne parlerai pas de Linky ou de TIC et ceci n'est absolument pas grave, car nous allons utiliser des équipements et des solutions qui nous permettent d'avoir une vraie granularité dans nos mesures. De vraies informations pour juger de la situation.

vaguement raison (sans qu'il sache lui-même techniquement pourquoi) dans l'absolu, car chaque kilowatt-heure compte, en principe. Mais une réponse de principe, manichéenne au possible, n'est pas suffisante. Il faut quantifier pour décider et changer, et ce, aussi bien pour des raisons financières qu'idéologiques. Car voyez-vous, une vieille ampoule à filament allumée 3 fois l'an dans le fond d'un grenier aura clairement moins d'impact « sur la planète » si elle est laissée tranquille

les 10 prochaines années, plutôt que recyclée et remplacée par un modèle à LED, acheté en ligne ou dans un magasin de bricolage à 15 min en voiture de chez soi.

1. LA MÉTHODE DU CONSTRUCTEUR : SONOFF, EWELINK, IFTTT

Je passerai rapidement sur cette partie, car comme vous le savez peut-être, je suis atteint d'une sévère allergie à la dissémination irresponsable de données à caractère privé (les miennes, en particulier) et, en dehors de quelques éléments (santé, préférences sexuelles, croyances, etc.), il n'y a pas plus privé que ce qui se passe dans son lieu de vie. Mouvement, température, activation de tel ou tel éclairage, état du chauffage ou encore consommation électrique détaillée n'ont strictement rien à faire sur l'ordinateur de quelqu'un d'autre, actuellement appelé « le cloud ».

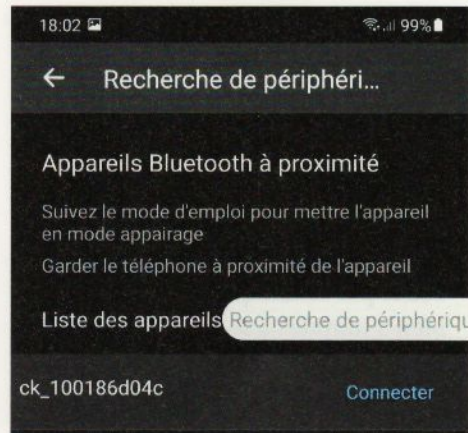
Quoi qu'il en soit, chaque fabricant met généralement à disposition, ou rend son produit compatible avec un ou plusieurs services en ligne permettant de simplifier la gestion de sa domotique. Si nous prenons le cas de Sonoff, ceci se passe avec eWeLink, un service et une application pour smartphone édités par CoolKit Technologies basé à Shenzhen (vous vous souvenez que j'ai parlé de « dissémination irresponsable » ?). Cette application, disponible via Google Play [1] pourra être installée sur votre smartphone très rapidement et, après avoir activé le Bluetooth, pourra être lancée pour configurer n'importe quel appareil Sonoff compatible, comme le POWR316 que nous étudierons en détail plus loin dans l'article.

Le principe de fonctionnement est simple : non configuré, le POWR316 ne se connecte pas au Wi-Fi, mais attend un appairage Bluetooth pour valider cette étape. Ensuite, tout se passera via le réseau local. Dès le démarrage, l'application vous

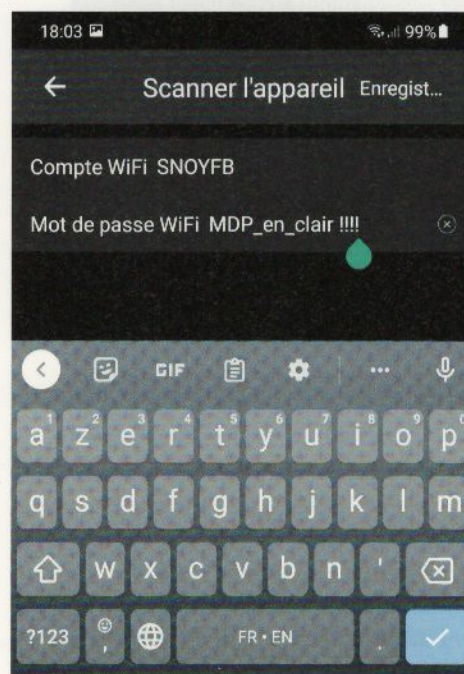
demandera, bien entendu, de créer un compte puis de confirmer votre adresse mail par la saisie d'un code reçu par ce biais. Ce n'est qu'ensuite qu'il sera possible d'appairer le capteur en choisissant d'ajouter un appareil :



Plusieurs façons de configurer un appareil sont disponibles, du Bluetooth au QRcode en passant par la configuration manuelle. L'association Bluetooth est assez classique, mais mérite d'être détaillée pour vous faire une idée de ce qui vous attend d'un point de vue qualitatif. Le scan des appareils Bluetooth fera normalement apparaître le POWR316 en attente de configuration :



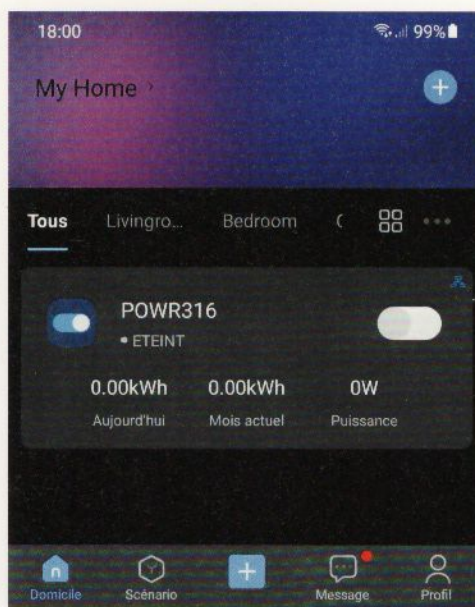
Notez, non seulement, le nom de l'appareil, « *ck_100186d04c* » qui est absolument identifiable (sic), mais également l'interface qui semble avoir quelques difficultés avec la résolution d'écran (Samsung Galaxy A40) et/ou les traductions des messages et textes. Passons. Une petite tape sur **Connecter** entamera la configuration en vous demandant de choisir un réseau Wi-Fi et de préciser la phrase de passe associée pour se connecter au point d'accès :



Remarquez qu'il ne s'agit pas d'une connexion du smartphone, mais bien d'un passage de configuration à l'appareil Sonoff et surtout que le mot de passe saisi apparaît en clair. Là encore, la qualité de l'interface est plus que douteuse, en particulier par le fait qu'on passera bien 5 minutes à se demander quelle est l'étape suivante, avant de se rendre compte qu'il y a un bouton **Enregist...** en haut à droite de l'écran. Cette découverte faite et le bouton utilisé, l'appareil sera configuré, reconnecté en Wi-Fi et ajouté :



Là, vous pourrez choisir la pièce associée et finalement terminer la configuration. Vous vous retrouverez ensuite dans l'interface principale listant votre flotte de capteurs :



Une sélection de l'un d'entre eux vous permettra d'afficher les données en temps réel ainsi que l'historique des mesures :



Il faut avouer que, si l'on est que peu attentif aux questions de confidentialité, il s'agit d'une méthode qui peut paraître intéressante pour très rapidement connaître la consommation d'un équipement ou d'un ensemble d'équipements. On notera cependant que, si on se limite à ce genre de choses, des produits moins bavards (car non connectés) existent et se présentent sous la forme d'adaptateurs avec un afficheur LCD se plaçant, tout simplement, sur la prise. Si vous ajoutez à cela un petit carnet et un crayon, vous avez votre « *monitoring* » maison pour trois francs six sous, parfaitement étanche aux pirates et gouvernements étrangers bien trop curieux.

Mais si vous voulez votre vie totalement ouverte et souhaitez exprimer votre exhibitionnisme numérique, il est possible d'ajouter une couche avec un autre service en ligne, américain cette fois (pour l'équité CPC/PLA/NSA/CIA). En effet, eWeLink permet de centraliser les informations des capteurs, mais est très limité en ce qui concerne les automatisations. Fort heureusement (ou pas), il y a IFTTT, littéralement « *IF This Then That* » (« Si ceci alors cela ») qui, comme l'acronyme l'indique, permet de déclencher des actions en fonction de conditions.

IFTTT est un service en ligne (et une application iOS/Android [2]) capable de prendre en compte quantité de services, allant d'Amazon Alexa à l'ensemble des produits Google, en passant par Blogger, Twitch, Twitter ou encore eWeLink. IFTTT fonctionne avec la notion d'Applets (limités à 2 avec la version gratuite) qui sont de simples scénarios composés de déclencheurs, de filtres et d'actions.

Une page du constructeur Sonoff [3] détaille la manière de lier les comptes des deux services/applications et ainsi pouvoir associer déclencheurs et actions propres aux périphériques Sonoff. Un essai rapide montre qu'ici, pour la mesure de consommation, les fonctionnalités sont très limitées, voir inexistantes

puisque vous ne pouvez utiliser que ce qui est supporté par IFTTT (ce n'est pas vous qui êtes aux commandes). Pour un Sonoff POWR316, ceci se limite à la fonction d'interrupteur Wi-Fi et rien de plus.

On voit ici les problèmes soulevés par ces solutions :

- vos données fuient chez des prestataires que vous ne connaissez pas et en qui vous ne pouvez pas avoir confiance ;
- vous être limité par les fonctionnalités mises à disposition ;
- vous êtes tributaire de l'existence du service et d'éventuels changements, à la fois dans les API et dans la politique commerciale ;
- vous n'avez strictement aucune maîtrise de la solution et aucun contrôle ;
- si vous n'avez plus d'Internet, vous n'avez plus de domotique.

Donc, en ce qui me concerne, ce dernier mail d'IFTTT « *Sorry to hear that you are deleting your account* » ne suscite pas une once de regret, pas plus que la suppression du compte (et de l'application) eWeLink :



Oh oui, je suis sûr, aucun doute !

Je voulais savoir, je voulais vous montrer. C'est fait. Passons à autre chose.

2. ON ARRÊTE DE RIGOLER ET ON FAIT ÇA PROPREMENT

C'est bien gentil tout ça, mais le moment est venu de faire les choses de manière sérieuse et responsable, sans fuiter tout ce qui se passe dans notre demeure en direction de l'administration de monsieur Xi Jinping ou des sous-fifres du général Nakasone. Le mot d'ordre est clair : ce qui se passe dans la maison reste dans la maison. Ce qui veut dire pas de *cloud*, pas de service à inscription obligatoire et pas de connexion permanente à Internet.

Nous avons abordé dans le numéro 46 toute la procédure de mise en marche d'une installation domotique en utilisant deux composants principaux : Home Assistant pour la centralisation des données, les scripts et les automatisations, et ESPHome, un SDK complet, très simple d'utilisation pour remplacer le *firmware* de très nombreux capteurs et périphériques, voire de montages « faits maison » à base d'ESP8266, ESP32 et Raspberry Pi Pico W. Dans les numéros 47 et 48, nous avons approfondi le sujet et complété l'installation de différentes manières.

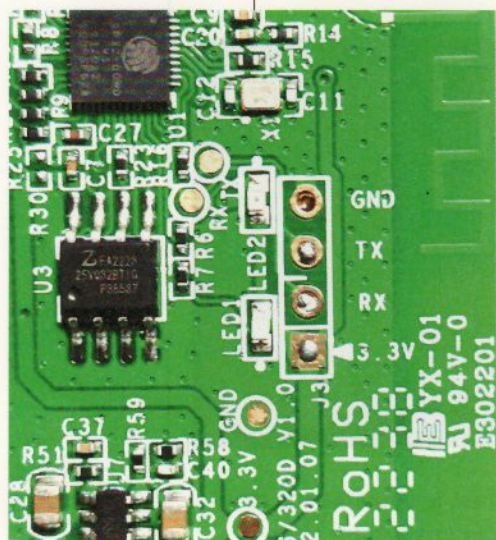
Je ne me relancerai pas ici dans des explications exhaustives sur le sujet, mais pour ceux d'entre vous qui prennent le train en marche, nous allons commencer par un rappel rapide.

présentant les informations structurées via une interface web et/ou mobile et permettant de mettre en place des automatisations, des scènes et des scénarios.

L'installation de Home Assistant, ou HA pour les intimes, peut se faire de différentes façons (installation totale, conteneur, service local, etc.), sur différentes plateformes (PC, Mac, SBC ARM, etc.) et différents systèmes (Windows, GNU/Linux, FreeBSD, macOS, etc.). La solution la plus simple pour très rapidement disposer d'un serveur HA est de lui dédier une machine, que ce soit un PC ou un SBC ARM comme une Raspberry Pi (personnellement, j'utilise une Pi 3). Pour cela, vous pouvez utiliser HAOS pour *Home Assistant Operating System*, un GNU/Linux déjà configuré, faisant fonctionner HA dans un conteneur Docker.

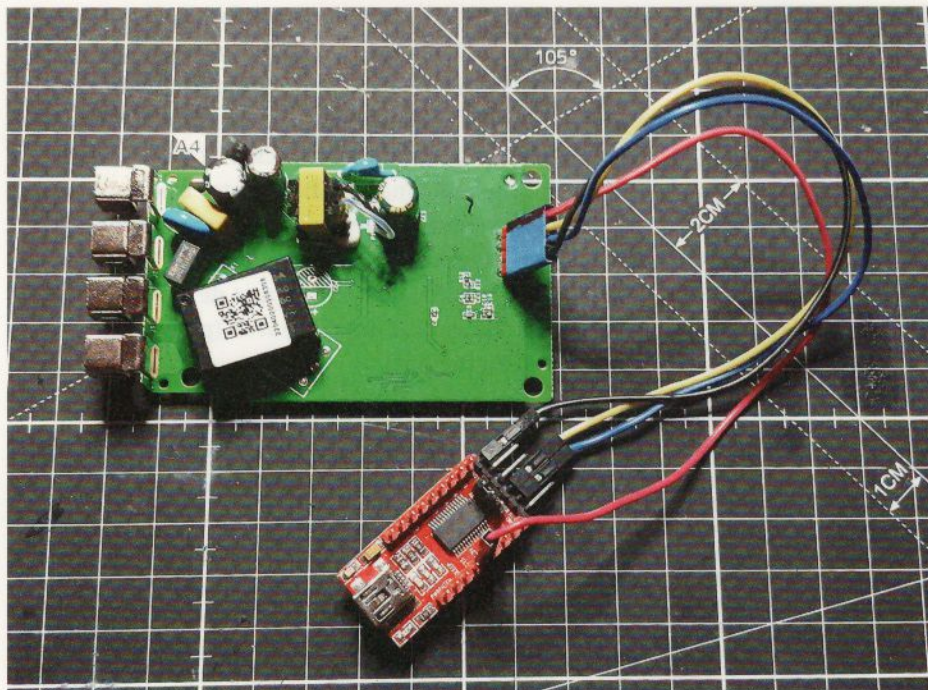
Si votre cible est une Pi, les choses sont encore plus simples, puisque HAOS est listé dans les systèmes alternatifs de *Raspberry Pi Imager* [5], l'outil officiel de création de cartes SD pour les Pi, disponible pour GNU/Linux, Windows et macOS. Téléchargez l'outil, choisissez le système, enregistrez le support, placez la SD dans la Pi, alimentez le SBC et l'affaire est dans le sac.

Comme sur bien des produits Sonoff, le circuit imprimé dans les POWR316 et POWR320D dispose d'un emplacement pour souder des broches et reprogrammer le microcontrôleur Espressif intégré.



2.1 Home Assistant, HAOS et ESPHome : cours de rattrapage accéléré

Home Assistant [4] est un projet *open source* visant à créer un serveur domotique centralisant les informations des capteurs rattachés (température, mouvement, caméra, etc.), actionnant des périphériques pilotés (interrupteurs, relais, etc.),



Flasher un nouveau firmware est un jeu d'enfant, il vous suffit d'utiliser un adaptateur USB/série et un outil logiciel proposé, en open source, par les projets ESPHome ou Tasmota.

Il en va presque de même pour une installation sur PC (comme un mini PC type Intel NUC ou un vieux portable). Pointez un navigateur sur la page GitHub de HAOS [6], allez dans « Releases » et piochez la dernière image ([haos_generic-x86-64-10.2.img.xz](https://github.com/HAOS/HAOS/releases/download/v1.0.2/haos_generic-x86-64-10.2.img.xz) à ce jour). Décompressez, inscrivez l'image sur le support de démarrage (disque dur de la cible branché temporairement en USB via un adaptateur ou clé pour un démarrage sur USB) avec **dd** en ligne de commandes ou *Etcher* [7] et démarrer la machine (cf. le site HAOS [8] pour plus de détails).

Que ce soit sur Pi ou PC, une fois HAOS démarré et grâce à la magie de mDNS, vous n'aurez qu'à pointer votre navigateur sur <http://homeassistant.local:8123> pour découvrir l'interface de Home

Assistant et vous laisser guider pour la configuration de base. Tout est parfaitement ergonomique et non technique lorsqu'on s'en tient aux fonctionnalités basiques, ce n'est qu'une question de se familiariser avec l'interface. Pour le reste, j'ai couvert cela plus largement dans le Hackable 46 (puis le 47 et 48 pour des éléments spécifiques).

L'autre pièce maîtresse est le *firmware* qui prendra place dans les périphériques en remplaçant ce que le constructeur y aura programmé pour son architecture. La quasi-totalité des équipements utilisables et *hackables*, comme les Sonoff et les Shelly, utilisent un microcontrôleur ESP8266 ou ESP32 d'un modèle ou d'un autre. Pour avoir un nouveau *firmware*, *open source* et maîtrisable, le code qui le compose doit être en mesure de supporter les composants intégrés (afficheurs, LED, GPIO, entrées, capteurs divers, etc.).

Fort heureusement, deux projets sont là pour nous éviter de devoir tout coder nous-mêmes : Tasmota [9] et ESPHome [10]. Les deux ont leurs avantages et leurs inconvénients, les deux fonctionnent parfaitement avec Home Assistant et les deux font parfaitement le



L'afficheur LCD du Sonoff POWR320D est du plus bel effet, certes, mais l'intérêt pratique est en réalité assez limité, même pour seulement quelques euros de plus par rapport au modèle POWR316.

travail. Tasmota se présente sous la forme d'une interface graphique (*Tasmota Web Installer* dans Chrome, *Tasmotizer* et *ESP-Flasher* pour Windows, GNU/Linux ou macOS) et ESPHome s'utilise en ligne de commandes en reposant sur un conteneur Docker et des fichiers de description au format YAML. À noter que les projets Home Assistant et ESPHome sont très proches et collaborent intensivement, et qu'il est également possible d'intégrer ESPHome dans HA et de mettre à jour vos périphériques via l'interface web (pour ceux qui aiment, moi j'aime pas).

ESPHome est pour moi un choix évident, car c'est simple d'utilisation (la simplicité n'est pas une GUI bling-bling) et les descriptions en YAML sont faciles à gérer, composer, modifier et versionner (avec Git). Les images de conteneurs sont disponibles pour AMD64, ARM et AARCH64 et s'installeront sans problème sur n'importe quelle machine GNU/Linux supportant Docker (Pi incluse), via :

```
$ docker pull ghcr.io/esphome/esphome
```

Vous vous retrouverez avec une image de conteneur **esphome** que vous pouvez instancier au besoin (lancer un conteneur, puis le détruire). J'ai ma petite architecture par défaut pour tous mes fichiers YAML et en voici un exemple le plus simple (et court) possible :

```
substitutions:
  devicename: sonoff666
esphome:
  name: $devicename
esp8266:
  board: sonoff_basic
logger:
  <<: !include inc/wifi.yaml
  <<: !include inc/auth.yaml
binary_sensor:
  - platform: gpio
    pin:
      number: GPIO00
      mode:
        input: true
        pullup: true
        inverted: true
    name: $devicename Basic Bouton
```


– Surveillez votre consommation électrique et faites des économies –

```

on_click:
  - switch.toggle: relay
switch:
  - platform: gpio
    name: $devicename Basic Relais
    pin: GPIO12
    id: relay
status_led:
  pin:
    number: GPIO13
    inverted: yes

```

Ceci sera stocké dans un fichier, `sonoffbasic_simple.yaml` par exemple, dans un répertoire dédié à ESPHome. Ce fichier contient tout ce que ESPHome doit savoir sur le matériel que vous utilisez (ici, un interrupteur Sonoff Basic) : le type de microcontrôleur, le nom du périphérique que vous créez, où est branché le bouton, quelle broche pilote la LED et quelle fonction celle-ci doit avoir, ce qui se passe si on appuie sur le bouton...

Un répertoire `inc/` est également utilisé pour y placer des éléments qui seront communs à tous vos périphériques ESPHome, ce qui vous évitera de les copier/coller dans chaque fichier YAML de chaque périphérique. Nous avons ainsi les informations concernant l'installation HA (`auth.yaml`) :

```

# Enable Home Assistant API
api:
  encryption:
    key: !secret api_key
ota:
  password: !secret ota_password

```

et celles pour votre point d'accès Wi-Fi (`wifi.yaml`) :

```

wifi:
  ssid: !secret wifi_ssid
  password: !secret wifi_password
  # fallback hotspot
  ap:
    ssid: $devicename Fallback AP
    password: !secret ap_password

```

Les éléments « sensibles » ne sont pas dans ces fichiers, mais stockés dans `secrets.yaml` se trouvant également dans `inc/` :

```

wifi_ssid: "MonAPwifi"
wifi_password: "MaPhraseDePasse"
ap_password: "IEg1S4k1ZkRk"
ota_password: "123456"
api_key: "ODcyOWE0MWJjYmEzNDdjNDgxNzQ5NzNmMGIXNTEyOTI="

```


Nous avons, dans l'ordre, le nom du point d'accès, son mot de passe, le mot de passe de l'AP de secours (pour se connecter directement au périphérique en cas de problème), le mot de passe de mise à jour OTA et une clé secrète prépartagée, constituée de 32 octets encodés en base64. Lorsqu'un nouveau périphérique flashé avec cette configuration sera détecté par votre serveur HA, celui-ci vous demandera s'il faut l'intégrer et il vous faudra alors spécifier cette clé pour que les deux intervenants (périphérique et serveur) s'authentifient. Vous pouvez obtenir une clé aléatoire en allant sur <https://esphome.io/components/api.html>. Celle qui apparaît là change à chaque rechargement de la page, ou alors vous pouvez en générer une vous-même avec la commande `echo -n 'echo -n "mon_secret" | md5sum | awk '{print $1}' | base64`.

Une fois un joli script YAML dûment créé, générer un *firmware* correspondant, le compiler et le programmer dans un périphérique compatible se fait en une ligne de commande invoquant Docker et le conteneur **esphome** :

```
$ docker run --network=host --rm \
-v "${PWD}":/config --device=/dev/ttyUSB1 \
-it esphome/esphome -s devicename sonoff666 \
compile sonoffbasic_simple.yaml
```

`-s devicename sonoff666` est une option nous permettant de changer le nom du périphérique directement à la construction, ce qui est fort pratique si vous avez une série d'appareils identiques à reprogrammer et ne voulez pas éditer le YAML à chaque fois. `--device=/dev/ttyUSB1` désigne le port série sur lequel est connecté le périphérique en mode programmation (GPIO0 à la masse sur ESP8266/32) et `sonoffbasic_simple.yaml` est notre script YAML.

Notez qu'un périphérique ESPHome en fonction et connecté au réseau Wi-Fi local pourra être reprogrammé directement sans connexion série, via OTA. Vous n'aurez plus besoin de spécifier le port à utiliser et le nom (`devicename`) sera le nom d'hôte pour contacter le périphérique (`sonoffbasic_simple.local` ici).

Arrêtons-là le cours de rattrapage et acceptez mes excuses si vous avez déjà lu tout cela dans un numéro précédent. J'ai condensé au maximum le B.A.BA de HA et d'ESPHome, tout en essayant de rendre cela intelligible pour que le plus grand nombre profite du contenu de cet article.

Pour surveiller la consommation d'un appareil ou un groupe d'appareils en particulier, la solution consiste à utiliser ces équipements comme des adaptateurs s'intercalant entre la prise murale et votre matériel.



2.2 Sonoff POWR316 alias Sonoff POW Origin 16A

Le Sonoff POWR316 se trouve facilement sur des sites comme AliExpress ou Amazon pour quelques euros. Construit autour d'un ESP32 et d'un capteur de courant ChipSea CSE7766 connecté en interne via une liaison série de 4800 bps, celui-ci dispose d'un bornier permettant de connecter phase et neutre en entrée et en sortie. Couplé à des prises mâles et femelles, ou placé dans un tableau électrique, il agit comme une prise télécommandée pouvant se placer entre une source AC 230 V et un ou plusieurs appareils. Ce modèle, comme son nom l'indique, est limité à 16 ampères, qui est le courant généralement associé aux circuits de prises électriques (par opposition aux luminaires généralement à 10 A et aux circuits spécifiques à 20 A ou 32 A). Il n'est bien entendu pas question de dépasser cette limite.

Reprogrammer un POWR316, comme les autres modèles de la même famille, est relativement facile. Il suffit d'ouvrir le boîtier (4 simples vis) et on découvre un sympathique emplacement, avec les noms des signaux sérigraphiés sur le circuit, afin de souder des broches pour y connecter un adaptateur USB/série 3,3 V. Il suffira ensuite de presser l'unique bouton présent tout en branchant l'adaptateur. Cette mise sous tension spécifique démarrera l'ESP32 avec GPIO0 à la masse et donc en mode « bootloader » (programmation).

ATTENTION DANGER !

Nous parlons ici d'équipements connectés au courant domestique qui, je le rappelle, est **potentiellement mortel**. Prenez toutes les dispositions nécessaires pour assurer votre sécurité et celle de votre entourage (chats inclus).

Ne travaillez **JAMAIS** sur le circuit d'un équipement connecté au courant domestique. Je pense en particulier à la programmation du *firmware*. Respecter les usages. Quand le boîtier est ouvert, il n'est pas branché au ~230 V. Point ! Ne connectez **jamais** l'adaptateur USB/série, ou tout autre équipement de mesure, lorsque le boîtier est connecté au courant domestique. Point ! Laissez cela à des personnes immortelles ou qui savent parfaitement ce qu'elles font.

Et enfin, la connexion à la terre est importante. Ces boîtiers, Sonoff, Shelly ou autre n'utilisent généralement que des connexions pour la phase (« L » pour « live », brun) et le neutre (« N », bleu), et n'ont pas de liaison à la terre. Mais contrairement aux Sonoff Basic, par exemple, prévus pour des luminaires, nous parlons ici de contrôler des équipements différents (réfrigérateur, PC, four, lave-linge, etc.) qui **DOIVENT** être reliés à la terre. Il est donc **crucial pour votre sécurité** de relier la terre de la fiche mâle (vert/jaune) à la terre de la fiche femelle, en passant donc à côté du Sonoff/Shelly. Je le répète : un équipement **DOIT** être relié à la terre, même piloté par un IoT domotique. Et ceci ne souffre d'aucune discussion. Point !

Dès lors, on pourra utiliser le conteneur ESPHome pour construire et flasher un nouveau *firmware*. Ce dernier sera produit par le script YAML suivant, qui débute par les mentions classiques (nom, plateforme, carte, informations de connexion) :

```
substitutions:
  devicename: sonoffpowr316-1

esphome:
  name: $devicename

esp32:
  board: esp32dev

logger:

<<: !include inc/wifi.yaml
<<: !include inc/auth.yaml
```


Nous avons ensuite besoin de déclarer le port série pour le capteur d'énergie, ainsi que spécifier l'utilisation du composant **time** de HA pour que le capteur synchronise son horloge et puisse nous fournir une quantité d'énergie quotidienne et non simplement en temps réel ou cumulé :

```
uart:
  rx_pin: GPIO16
  baud_rate: 4800

time:
  - platform: homeassistant
    id: homeassistant_time
```

Puis viennent les déclarations des capteurs, à commencer par le CSE7766 :

```
sensor:
  - platform: cse7766
    update_interval: 10s
    current:
      name: $devicename Current
      id: a_sensor
    voltage:
      name: $devicename Voltage
      id: v_sensor
    power:
      name: $devicename Power
      id: w_sensor
    energy:
      name: $devicename Energy
      id: wh_sensor
  - platform: total_daily_energy
    name: $devicename Total Daily Energy
    power_id: w_sensor
```

Ces informations, courant, tension, puissance, énergie sont directement celles fournies par le composant ChipSea, mais nous ajoutons également le volume d'énergie « pour la journée », qui se base sur **w_sensor**. Il s'agit, dans les grandes lignes, d'une intégration avec le cumul de puissance pour les 24 dernières heures, ramené à zéro à minuit.

Nous configurons ensuite l'usage du bouton :

```
binary_sensor:
  - platform: gpio
    pin:
      number: GPIO0
      mode:
        input: true
        pullup: true
        inverted: true
    name: $devicename Bouton
    on_press:
      - switch.toggle: relay
      - light.toggle: redled
```

Nous pouvons faire ce qui nous chante, mais je pense que basculer le relais et piloter la LED rouge intégrée est une bonne idée. Notez que la relation entre le bouton (GPIO0) et le relais ne passe pas par HA, ce qui est fort pratique si le Sonoff en question est celui qui mesure et contrôle la multiprise sur laquelle est branchée la Pi avec le serveur HA et que vous avez la bonne idée de couper l'alimentation « pour voir si ça marche » (vécu, et pas drôle).

À propos des GPIO justement, nous avons ensuite les LED :

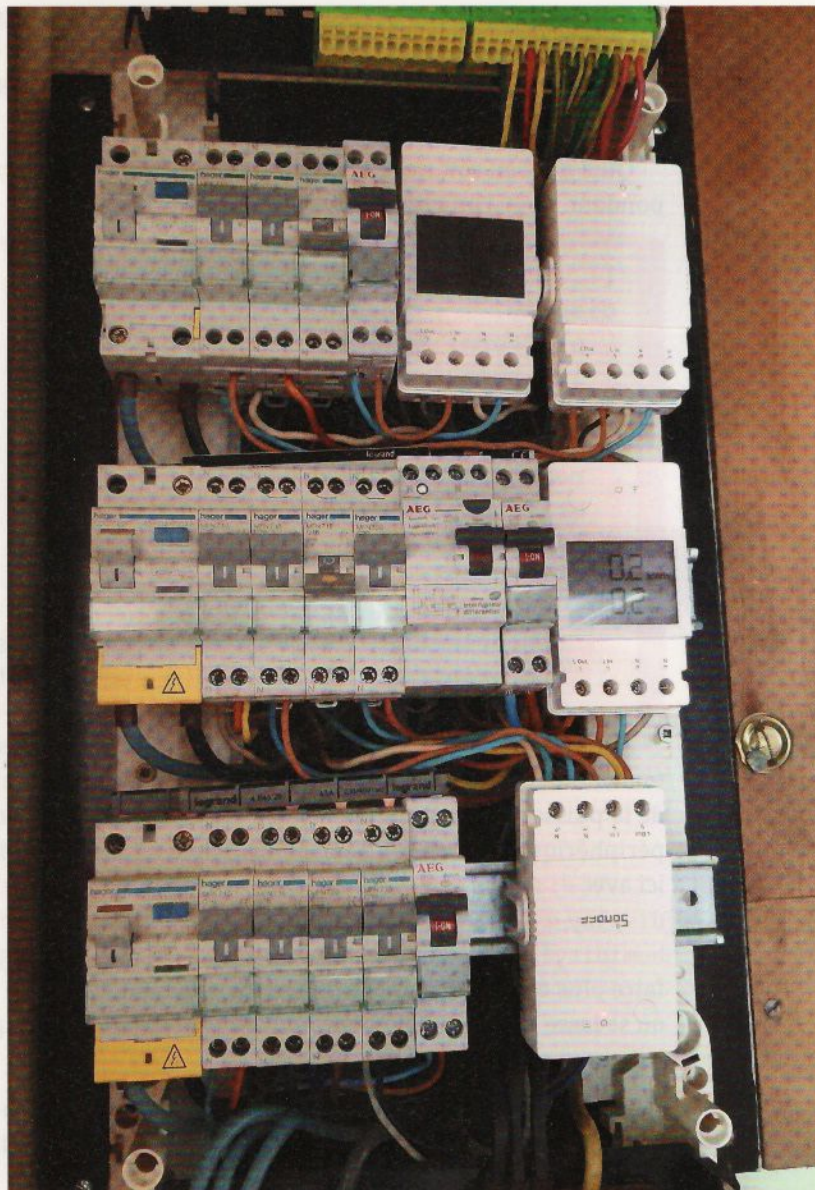
```
light:
  - platform: status_led
    name: $devicename blueled
    id: blueled
    pin:
      number: GPIO05
      inverted: True
  - platform: status_led
    name: $devicename redled
    id: redled
    restore_mode: RESTORE_DEFAULT_ON
    pin:
      number: GPIO18
      inverted: True
```

Et nous terminons par le relais sur GPIO13 :

```
switch:
  - platform: gpio
    name: $devicename Relais
    pin: GPIO13
    id: relay
    restore_mode: RESTORE_DEFAULT_ON
```


Notez le **RESTORE_DEFAULT_ON** permettant, en cas de coupure ou de redémarrage, de spécifier un état précis. Ici, il s'agit de revenir au précédent état connu ou, s'il est indéfini, d'activer le composant (relais). Ceci nous permet donc d'avoir une solution en cas de coupure de délestage ou autre, et ne pas avoir à faire le tour des Sonoff pour tout rallumer. Une autre option valide aurait été **ALWAYS_ON** pour ignorer l'état précédent et toujours être actif.

Une fois tout ceci enregistré dans un fichier **sonoff_powr316.yaml**, il ne restera plus qu'à construire le *firmware* et reflasher le ou les périphériques avec une simple ligne de commande, comme détaillé dans le petit cours de rattrapage express. Notez que, contrairement aux Sonoff Basic par exemple, il ne semble pas possible d'alimenter ces modèles depuis les broches de programmation. Le *firmware* démarre, mais ne cesse de faire *rebooter* l'ESP32. Pour mettre le périphérique en fonction et l'intégrer dans Home Assistant, il faudra donc l'installer réellement en production, et donc l'alimenter avec du courant domestique (boîtier fermé, donc).



2.3 Un mot sur les Sonoff POWR320D

Le POWR320D est une déclinaison 20 A (parfait pour mon four) du POWR316. Le circuit imprimé est identique, tout comme la plupart des composants qui s'y trouvent. Seul deux sont différents, le relais qui est bistable (deux GPIO de contrôle plutôt qu'un qui bascule) et un contrôleur d'affichage Titan Micro TM1621 assorti d'un écran LCD qui n'existe pas dans la version PWR3xx (sans le « D » pour *display*).

Deux POWR316 et deux POWR320D, directement sur le tableau électrique : de quoi couvrir la surveillance des prises de deux étages de la maison, du four et de l'ensemble micro-onde + réfrigérateur. Mais je commence à être un peu à l'étroit...

Honnêtement, pour quelques euros de plus, je pensais que l'écran LCD serait du plus bel effet et très utile, mais ce n'est pas du tout le cas. C'est amusant de voir le nombre de watts, la tension, le courant ou l'énergie consommée en temps réel, mais on s'en lasse rapidement, d'autant que tout est affiché, en mieux, dans HA.

Quoi qu'il en soit, ces petites différences doivent être prises en compte dans le YAML correspondant. Pour l'affichage, nous avons :

```
[...]
display:
  platform: tm1621
  id: tm1621_display
  cs_pin: GPIO25
  data_pin: GPIO14
  read_pin: GPIO26
  write_pin: GPIO27
  lambda: |-
    it.display_voltage(false);
    it.display_kwh(true);
    it.printf(0, "%.1f", id(wh_sensor).state);
    it.printf(1, "%.1f", id(w_sensor).state);
```

Nous spécifions la connexion aux GPIO et nous utilisons un *template* permettant de directement modifier les informations à l'écran en fonction des valeurs d'énergie consommée et de la puissance en temps réel. Notez que l'afficheur et le contrôleur sont utilisés sur plusieurs périphériques Sonoff et que l'unité affichée est une sorte de paire d'icônes, que nous activons ici avec `display_kwh()`. Il en existe trois autres : `display_voltage()` (tension et courant), `display_celsius()` (température), `display_fahrenheit()` (température) et `display_humidity()` (hygrométrie). Eh oui, il n'y a que « kWh » et pas de « Wh ». Il est possible de faire plus étoffé, comme détaillé sur la page dédiée du site ESPHome [11], mais à quoi bon, on se lasse après quelques coups d'œil post-installation et personne ne va jamais appuyer sur le bouton pour basculer l'affichage.

Le relais, pour sa part, est géré différemment puisque nous avons GPIO02 pour l'activer et GPIO04 pour le désactiver. Il est donc nécessaire de déclarer une entité (*template*) pouvant basculer et agissant sur les deux GPIO alternativement :

```
switch:
  - platform: gpio
    restore_mode: ALWAYS_OFF
    internal: true
    id: relay_off
    pin: GPIO04
    on_turn_on:
      - delay: 500ms
      - switch.turn_off: relay_off
```


– Surveillez votre consommation électrique et faites des économies –

```
- platform: gpio
  restore_mode: ALWAYS_OFF
  internal: true
  id: relay_on
  pin: GPIO02
  on_turn_on:
    - delay: 500ms
    - switch.turn_off: relay_on
- platform: template
  name: $devicename Relais
  optimistic: true
  id: relay
  restore_mode: ALWAYS_ON
  turn_off_action:
    - switch.turn_on: relay_off
    - light.turn_off: redled
  turn_on_action:
    - switch.turn_on: relay_on
    - light.turn_on: redled
```

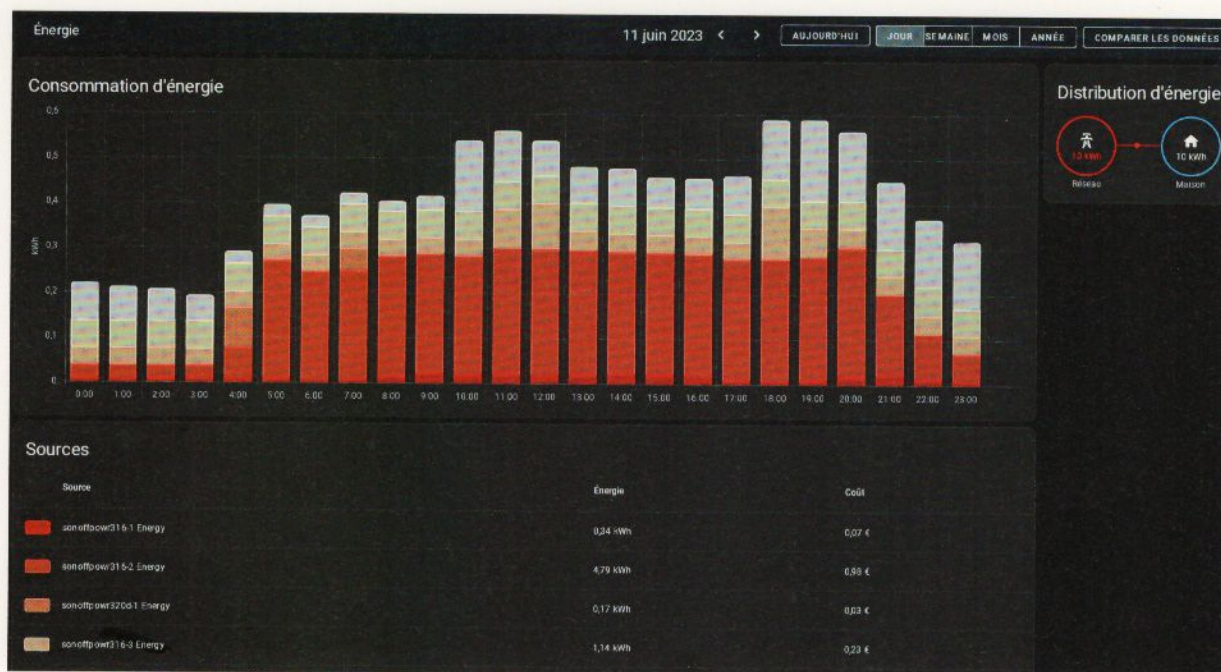
C'est tout. Compilez et flashez, et voici votre POWR320D prêt à servir.

2.4 L'énergie dans Home Assistant

Une fois le ou les Sonoff POWR alimentés et connectés au Wi-Fi de la maison, ceux-ci seront détectés par le serveur HA qui vous invitera à les intégrer en spécifiant la pièce dans laquelle ils se trouvent et la clé d'authentification. Avec les configurations spécifiées ici, chaque Sonoff fournit alors 9 entités. Il vous sera possible de faire apparaître chacune d'elles dans l'interface (tableau de bord ou *dashboard*) pour surveiller tension, courant, puissance, etc.

Mais HA va plus loin dans ce domaine, en proposant un tableau de bord spécialisé nommé « Énergie ». Un simple clic dans la barre latérale vous permettra d'accéder à ce dernier et de débiter la configuration. Là, vous pourrez alors spécifier chaque entité remontant des valeurs en joules (GJ ou MJ) ou en watt-heure (kWh, MWh ou Wh). Remarquez que HA est conçu pour des installations bien plus avancées que ma modeste demeure, en permettant, en plus de l'énergie provenant du réseau électrique, de prendre en charge la production de panneaux solaires pouvant être réinjectée dans le réseau ou stockée localement (batteries). La consommation de gaz et d'eau peut également être ajoutée, mais ceci demandera des installations spécifiques.

Une fois toutes les entités ajoutées, le tableau de bord « Énergie » pourra faire son travail. Comme le précisent les différents messages de l'interface, laissez un peu de temps à HA pour synthétiser tout cela et remonter l'ensemble des informations. Mon expérience montre que ce n'est certes pas immédiat, après une modification de la configuration, mais nous sommes tout de même loin des 2 heures annoncées. Mais une fois que cela fonctionne, c'est un véritable bonheur.



Le tableau de bord « Énergie » centralise toutes les informations utiles et permet de savoir précisément tout ce qui se passe en un clin d'œil, y compris le coût de votre consommation. En jouant avec les thèmes HA, il est même possible, comme ici, d'obtenir quelque chose de fort plaisant pour les yeux (à défaut de l'être pour le port-monnaie).

On voit l'ensemble de la consommation surveillée apparaître sous la forme d'un magnifique graphique décliné en heures, jours, semaines, mois et années, donnant une vision globale de ce qui se passe et, par définition, des gains, si vous comptez faire des efforts pour ajuster votre consommation (une fonction de comparaison sur différentes plages de temps est disponible).

Tout n'est cependant pas parfait et deux problèmes notables sont omniprésents dans les forums du projet. Le premier concerne les coûts. En effet, il est possible d'associer un prix au watt-heure consommé et donc d'obtenir une vision financière de la consommation. Malheureusement, le tarif en question est spécifié, au moment de l'ajout d'une entité, en ne pouvant préciser que deux chiffres après la virgule, ce qui pose un gros problème, car avec le kilowatt-heure à 0,1708 € hors taxe (tarif de mon fournisseur local), ceci n'est pas possible. L'astuce consiste à ne pas choisir *Utiliser un prix statique* mais *Utiliser une entité avec le prix actuel*.

Ceci ne peut être fait qu'après avoir effectivement créé l'entité en question. Pour cela, rendez-vous dans **Paramètres, Appareils et services**, onglet **Entrées**, puis choisissez **Créer une entrée** (en bas à droite). Dans la liste, choisissez **Nombre** et dans l'écran suivant, spécifiez un nom (« tarifelec », par exemple), précisez **Champ de saisie** pour le **Mode d'affichage** et « EUR/MWh » (ou « EUR/kWh ») comme unité de mesure. Validez et vous devrez alors voir votre « entrée » dans la liste. Cliquez dessus et spécifiez le tarif dans l'unité choisie (TTC de préférence). Attention, le coût est utilisé et enregistré en temps réel et n'est pas rétroactif. Donc si vous le faites après avoir commencé à collecter les mesures, celles du jour même seront partiellement fausses et celles des jours précédents à zéro. Cependant, dès cette manipulation faite, les prix commencent à apparaître en compagnie des consommations en kWh.

Watts			Dailly		
⚡	Bench	21,8 W	⚡	Bench	145,581 Wh
⚡	PC++	262,8 W	⚡	PC++	1 884,823 Wh
⚡	Frigo + micro-onde	124,8 W	⚡	Frigo + microonde	435,713 Wh
⚡	Chambre + Jaune (baie)	57,7 W	⚡	Chambre + Jaune (baie)	328,715 Wh
⚡	Four + Plan	0,2 W	⚡	Four + Plan	157,707 Wh
⚡	Haut Emerick	1,6 W	⚡	Haut Emerick	75,584 Wh

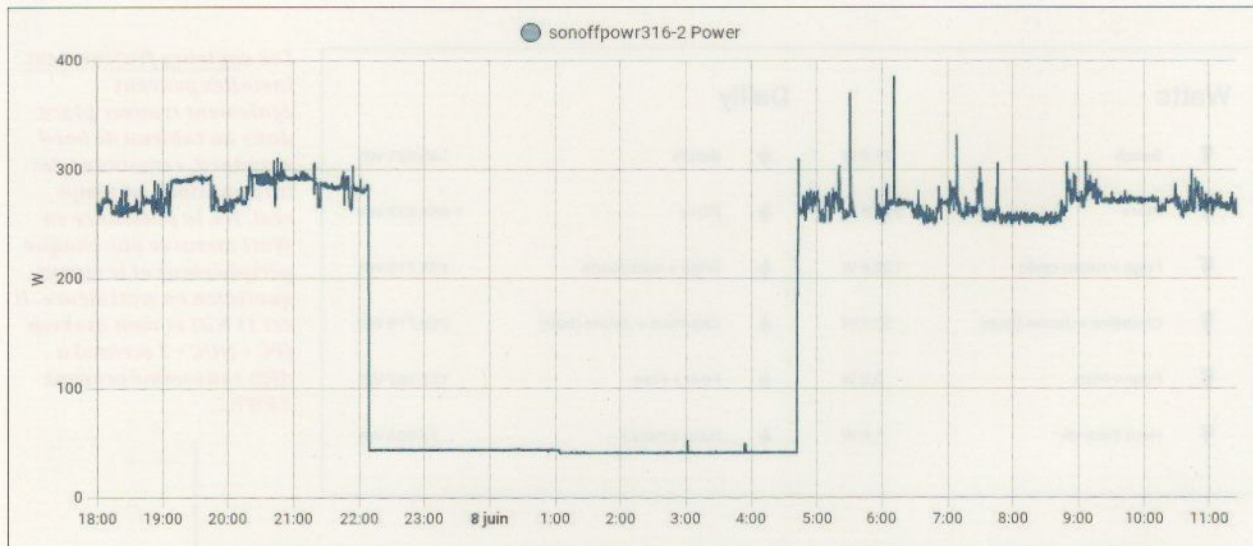
Les capteurs fraîchement installés peuvent également trouver place dans un tableau de bord standard, remontant des informations en temps réel. Ici, la puissance en Watt mesurée par chaque périphérique et le cumul quotidien en watt-heure. Il est 11 h 31 et mon desktop (PC + NUC + 2 écrans) a déjà consommé presque 2 kWh...

Le second problème est davantage cosmétique, mais est assez énervant à long terme. Il tient dans le fait que la représentation graphique de la consommation sur le *dashboard* utilise les identifiants des capteurs (ID) et non le nom de l'entité (*friendly name*). Il est donc difficile de voir rapidement quel capteur « consomme » quoi et ce à quoi il est connecté (four, PC, box, bouilloire, etc.). Deux solutions existent, mais sont assez pénibles à mettre en œuvre : soit on renomme les entités dans HA, soit on crée un *template* afin de disposer de nouvelles entités, avec des noms différents, ne faisant que reprendre les valeurs des capteurs. Cette solution, temporaire, est celle proposée par *elvo* dans les forums HA [12], mais elle suppose de tirer un trait sur les données déjà collectées, pour partir sur cette nouvelle base. Vous savez comment ça marche, quand on commence à avoir des mesures, on n'a pas envie de les perdre et personnellement, comme il ne s'agit en principe que d'une question d'interface, je vais laisser tout cela en l'état et miser sur une éventuelle future évolution et mise à jour du *dashboard* « Énergie ».

CONCLUSION

Au moment de la clôture de la rédaction du présent article, cela ne fait que quelques jours que j'ai totalement intégré la gestion de la consommation électrique à mon installation Home Assistant. Je vous ferai peut-être un retour au prochain numéro, mais les constations intéressantes sont déjà là. Le four est un gouffre à énergie et je me pose la question d'éventuellement acquérir un mini four de table dédié (~90 €), moins gourmand (1200 W), car plus compact, pour les pizzas de mon fils (moi, je ne mange pas ça).

Mais le plus choquant est sans doute les quelque 5 kWh de mon PC de plus de 10 ans d'âge. Il va falloir sérieusement réfléchir à une solution. Idem pour la bouilloire, utilisée pour les cafés (solubles) du matin, qui mérite de l'attention. Mais le réfrigérateur, lui, un Samsung RT34MASW1 de presque 15 ans n'est étonnamment pas très gourmand. Les plaques vitrocéramiques De Dietrich sont un problème, mais d'un tout autre acabit puisque le courant nécessaire (32 A) dépasse les capacités des Sonoff/Shelly (16 A ou 20 A). Il va falloir que je trouve une solution, car c'est certainement un gros poste de consommation...



Ceci est un graphique très intéressant quand on sait qu'il s'agit là de la consommation de deux machines. Un bi-Xeon E5520 de 2009 (Z8NA-D6) sous GNU/Linux avec ses 2 écrans d'une part, éteints la nuit, et un mini PC Asus PN51 Ryzen 5 5500U headless sous FreeBSD 13.1 acheté en juillet 2022 de l'autre, constamment en fonction. « Étonnant, non ? » comme le dirait ce vraiment très regretté Pierre Desproges...

Au final, j'ai acquis 5 Sonoff POWR316 [13] (dont un que j'ai suicidé en l'alimentant en 5 V) et 2 POWR320D [14]. Cet investissement de quelque 125 € devra, en premier lieu, se rentabiliser durant les 12 prochains mois. Ceci revient à économiser quelques ~10 €/mois (soit ~50 kWh) et me semble tout à fait faisable en changeant quelques habitudes. Nous verrons, mais une chose est claire : maintenant, au moins, je sais ! **DB**

RÉFÉRENCES

- [1] <https://play.google.com/store/apps/details?id=com.coolkit>
- [2] <https://play.google.com/store/apps/details?id=com.ifttt.ifttt>
- [3] <https://sonoff.tech/product-review/tutorial/works-with-ifttt-instruction-guide/>
- [4] <https://www.home-assistant.io/>
- [5] <https://www.raspberrypi.com/software/>
- [6] <https://github.com/home-assistant/operating-system>
- [7] <https://etcher.balena.io/>
- [8] <https://www.home-assistant.io/installation/generic-x86-64>
- [9] <https://tasmota.github.io/docs/>
- [10] <https://esphome.io/>
- [11] <https://devices.esphome.io/devices/Sonoff-POW-Elite-20a>
- [12] <https://community.home-assistant.io/t/energy-custom-names-for-individual-devices/372943/14>
- [13] <https://fr.aliexpress.com/item/1005004975263034.html>
- [14] <https://fr.aliexpress.com/item/1005004663114139.html>

CRÉER SON PHOTOBOOTH CUSTOM

Michaël BERTOCCHI

[Ingénieur développement - EEX Paris, auteur de dupot.org]

Vous avez peut-être déjà vu pendant un mariage ou autre événement festif un espace réservé où des personnes se prenaient automatiquement en photo via un drôle d'appareil. Cette installation de prise de photo automatique s'appelle un photobooth, et nous allons voir dans cet article comment vous pouvez facilement créer le vôtre.



Pour les 40 ans de ma femme, j'ai eu l'ambitieuse idée de vouloir construire un objet particulier pour cette soirée, en lien avec son personnage favori. J'avais, au cours d'un mariage puis d'une soirée d'entreprise, découvert cet appareil original qui permettait d'avoir un stand autonome pour laisser les invités libres de se photographier en toute autonomie. En regardant rapidement l'objet, je m'étais dit que ça ne devrait pas être très compliqué à concevoir.

Nous allons ici voir les étapes de conception tant matérielle que logicielle qui m'ont permis de réaliser cet automate bien utile pour vos prochaines soirées (anniversaire, mariage, etc.).

1. TAILLEZ VOTRE CRAYON, ON COMMENCE SUR PAPIER

Pour ce genre de projet, je préconise avant toute chose de se poser au calme, de prendre un papier et un crayon et d'essayer de conceptualiser la cible.

Dessiner le projet final permet d'avoir une vue d'ensemble : ici, on veut un objet à une certaine hauteur avec un écran pour la prévisualisation et un appareil pour prendre la photo.

Pour compliquer la chose, j'avais en tête de mettre un fond vert afin d'incruster après coup une image particulière : pour cela, il nous faut donc une surface verte assez importante à positionner derrière.

Pour l'écran, j'ai souhaité ne pas partir sur une version tactile, de peur que les enfants ou même certains adultes, à force d'appuyer sur celui-ci, déplacent la structure, modifiant la mise en place initiale ou le fasse tomber : je suis donc parti sur l'idée d'une navigation via une manette USB.

Enfin, pour la prise de vue, j'ai pensé qu'il serait plus simple de partir sur une webcam orientable plutôt qu'un appareil photo qu'il faudrait installer de manière fixe et donc trop peu réglable sur place.

En posant tout ceci, on peut déterminer les prérequis :

- une « boîte » qui contiendra l'ensemble ;
- un petit écran ;
- un mini-ordinateur qui ne chauffe pas (un Raspberry Pi, par exemple) ;
- un pied (pour tenir cette boîte assez haute) ;
- une webcam orientable et fixable ;
- une manette de jeu USB ;
- une structure montable ainsi qu'un grand tissu vert.

2. LE CŒUR DE LA BÊTE : LA PARTIE LOGICIELLE

Pour être honnête, j'ai dû abandonner la première version que j'avais développée en **React.js**, après avoir validé sur mon ordinateur que tout fonctionnait pour le mieux, je me suis rendu compte en l'installant sur le Raspberry Pi que la solution était trop gourmande en ressources.

Je suis donc reparti de zéro : une *stack* beaucoup plus légère PHP/HTML5/Vanilla JS.

2.1 Création du projet et prise en main

Pour le projet PHP, je souhaitais partir du strict minimum permettant de gérer de simples changements de pages avec au minimum des **views** et un **layout** pour simplifier la partie graphique.

Je me suis donc orienté vers le *microframework* PHP `dupot/static-management-framework`.

Ce *microframework* a été à l'origine conçu pour créer des sites d'administration de sites statiques conçus avec le *framework* `dupot/static-generation-framework`.

On initie avec une simple commande `composer` la création du projet :

```
composer create-project dupot/static-management-skeleton myPhotobooth
```

Après cette commande, on se retrouve avec une arborescence de départ ainsi qu'une configuration `composer` déjà paramétrée. Vous voyez en effet un fichier `docker-compose.yaml` ainsi qu'un fichier `vhost/mysite.local.conf`.

Si vous ne possédez pas déjà `composer` sur votre ordinateur, vous pouvez l'installer via la commande suivante :

```
sudo apt install docker-compose
sudo usermod -aG docker $USER
newgrp docker
```

On ajoute le paquet `docker-compose` qui, via la gestion de dépendances, installera les autres paquets `docker` nécessaires.

Ensuite, on ajoute notre utilisateur courant au groupe `docker` pour éviter que l'on ait besoin de passer en administrateur pour utiliser Docker.

Une fois installé, vous pouvez lancer le projet via la commande suivante :

```
docker-compose up
```

Le terminal va vous afficher une longue liste de commandes et lorsqu'elle s'arrêtera de défiler, vous pourrez ouvrir votre navigateur sur l'adresse `http://localhost/` pour voir s'afficher une simple page de démarrage.

Regardons rapidement certains fichiers de notre projet :

- `public/index.php` : c'est le fichier de départ de votre site web, qui charge votre fichier de configuration et démarre votre application ;
- `src/conf/routing.json` : le paramétrage de vos `routes` et les classes PHP à appeler en fonction.

```
[
  {
    "pattern": "#/myOtherPage.html#",
    "class": "My\\Infrastructure\\Pages\\OtherPage",
    "method": "index"
  },
```



```
{
    "pattern": "#/#",
    "class": "My\\Infrastructure\\Pages\\HomePage",
    "method": "index"
}
```

Le *framework* est conçu en respectant le principe de la *clean architecture*, ce qui explique notamment que les classes de pages ne sont pas directement dans **src/**, mais dans **src/Infrastructure/** pour bien différencier la partie métier (**domain**) de la partie infrastructure.

Pour ce simple projet, nous n'utiliserons pas de partie domaine, le projet étant trop petit pour envisager une méthode TDD (*Test Driven Development*) ou DDD (*Domain Driven Development*). Mais il est bon à savoir que ce modeste *microframework* le permet.

On peut facilement comprendre ici que pour chaque **url** qui correspondrait à un bloc JSON, le *framework*instanciera la classe renseignée et appellera sa méthode indiquée : simple/efficace.

- **src/Infrastructure/Pages** : qui contient vos différentes classes de pages.

Ce répertoire contient également un sous-répertoire **Layouts** pour héberger les *templates* de votre site et **View** pour les fichiers d'affichage de chaque page.

- Fichier **src/Infrastructure/Pages/HomePage.php** :

```
<?php
namespace My\Infrastructure\Pages;
use Dupa\StaticManagementFramework\Page\PageAbstract;
use Dupa\StaticManagementFramework\Render\Layout;
use Dupa\StaticManagementFramework\Render\View;

class HomePage extends PageAbstract
{
    protected $layout = null;
    public function __construct()
    {
        $this->layout = new Layout(__DIR__ . '/Layouts/default.php');
    }
    public function index()
    {
        $view = new View(
            __DIR__ . '/View/home.php',
            []
        );
```



```

        $this->layout->appendContext('contentList', $view);
        return $this->render();
    }
    public function render()
    {
        echo $this->layout->render();
    }
}

```

On a dans ce projet de départ `public/index.php` qui va lire les différentes routes, analyser et choisir la première expression régulière qui correspond, instancier la classe indiquée et appeler sa méthode. Dans notre cas, il va instancier la classe `HomePage` et appeler sa méthode `index()`.

Dans la logique, on appelle en premier le constructeur qui instancie un objet `Layout` à partir de notre fichier `Layouts/default.php`, puis la méthode `index()` qui instancie un objet `View` à partir de notre fichier d'affichage `View/home.php` qu'elle ajoute à notre `Layout` dans sa variable de contexte `contentList`, avant de retourner le rendu de celui-ci.

En jetant un petit coup d'œil sur le fichier de `Layout` :

- Fichier `src/Infrastructure/Pages/Layout/default.php` :

```

<!doctype html>
<html lang="en">

<head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>My Website</title>
    <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/
css/bootstrap.min.css" rel="stylesheet" integrity="sha384-KK94CHFLLe+nY2dmCW
GMq91rCGa5gtU4mk92HdvYe+M/SXH301p5ILy+dN9+nJ0Z" crossorigin="anonymous">
    <style>
    </style>
</head>

<body>
    <div class="container">
        <?php foreach ($this->contextList['contentList'] as $contentLoop) : ?>
            <?php echo $contentLoop->render(); ?>
        <?php endforeach; ?>
    </div>
    <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/
js/bootstrap.bundle.min.js" integrity="sha384-ENjd04Dr2bkBIFxQpeoTz1Hicje39W
m4jDKdf19U8gI4ddQ3GYNS7NTKfAdVQSZ" crossorigin="anonymous"></script>
</body>
</html>

```

On peut observer ici une boucle sur la variable de contexte `contentList` pour appeler la méthode `render()` de chaque objet du tableau pour les afficher : rien de bien sorcier, donc.

2.2 Thème et première page de menu

Pour rester transparent, quand lors du dernier jour de préparation, je me suis aperçu que ma solution qui était déjà développée devait être abandonnée et réécrite, je me suis dit : vu le peu de temps qu'il me reste, je ne vais pas refaire dans le même ordre : validation technique du concept, suivie de la partie graphique après coup.

En effet, cette fois, je trouvais plus pertinent/efficient d'avancer pas à pas en validant/verrouillant chaque page, sans besoin de revenir par la suite après coup. Pour ce genre de projet, dont le rendu final graphique a une grande importance, je vous invite à faire de même : dès le départ, prendre la taille de l'écran cible (forcément différente de votre ordinateur), et valider étape par étape les écrans sur la cible, ainsi une fois la dernière page terminée, vous pouvez passer à la suite.

Pour le thème graphique ici, j'ai voulu faire simple : Bootstrap 5 pour le design CSS et un simple fond uni avec une image de Snoopy pour être dans le thème de la soirée.

Pour cela, avec ce *framework*, rien de plus simple, il suffit d'éditer le fichier :

- Fichier `src/Infrastructure/Pages/Layouts/default.php` :

```
<!doctype html>
<html lang="en">

<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title><?php echo $this->contextList['title'] ?></title>
  <link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-KK94CHFLLe+nY2dmCWGMq91rCGa5gtU4mk92HdvYe+M/SXH301p5ILy+dN9+nJ0Z" crossorigin="anonymous">
  <style>
    body {
      background: #0d68abff url('/css/images/layoutBackground.png') top left no-repeat;
    }
  </style>
</head>

<body>
  <div class="container">
    <?php foreach ($this->contextList['contentList'] as $contentLoop) : ?>
      <?php echo $contentLoop->render(); ?>
    <?php endforeach; ?>
  </div>
  <script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.0-alpha3/dist/js/bootstrap.bundle.min.js" integrity="sha384-ENjdO4Dr2bkBIFxQpeoTz1HIcje39Wm4jDKdf19U8gI4ddQ3GYNS7NTKfAdVQSZe" crossorigin="anonymous"></script>
</body>
</html>
```


Ici, deux choses importantes :

- Premièrement : modifiez la balise style pour mettre une couleur de fond en utilisant les couleurs de l'image de fond stockée dans `public/css/images/`.
- Deuxièmement, remplacez la balise HTML de titre par l'utilisation d'une variable de contexte `title`, cela nous permettra de la changer à chacune de nos pages.

Repassons à notre page d'accueil qui nous permettra d'afficher les actions disponibles sur notre appareil :

- Fichier `src/Infrastructure/Pages/HomePage.php` :

```
<?php
namespace My\Infrastructure\Pages;
...
class HomePage extends PageAbstract
{
    protected $layout = null;
    public function __construct()
    {
        $this->layout = new Layout(__DIR__ . '/Layouts/default.php');
        $this->layout->assignContext('title', 'Bienvenue sur Photobooth');
    }
    ...
}
```

Une seule chose à changer ici : l'assignation du titre, le plus gros étant dans la vue utilisée :

- Fichier `src/Infrastructure/Pages/View/home.php` :

```
<div class=" d-flex align-items-center justify-content-center" style="height:
700px;">

    <a href="/takePicture.html" class="btn btn-primary btn-lg" role="button">

        <svg xmlns="http://www.w3.org/2000/svg" width="16" height="16"
fill="currentColor" class="bi bi-camera-fill" viewBox="0 0 16 16">
            <path d="M10.5 8.5a2.5 2.5 0 1 1-5 0 2.5 2.5 0 0 1 5 0z" />
            <path d="M2 4a2 2 0 0 0-2 2v6a2 2 0 0 0 2 2h12a2 2 0 0 0 2-2V6a2
2 0 0 0-2-2h-1.172a2 2 0 0 1-1.414-.586l-.828-.828A 2 2 0 0 0 9.172 2H6.828a2
2 0 0 0-1.414.586l-.828.828A 2 2 0 0 1 3.172 4H2zm9 2.5a3.5 3.5 0 1 1-7 0 3.5 3.5 0 0 1 7 0z" />
        </svg><br />
        Prendre une photo</a>
</div>
```

On affiche ici au centre un simple bouton avec une icône d'appareil photo qui nous mènera sur la page suivante `/takePicture.html`.

Vous avez ici deux possibilités : écrire le nom de vos pages en dur, ou utiliser des constantes dans les classes de pages appelées.

2.3 Nouvelle page et petite réorganisation

Nous avons, dans la page précédente, affiché un bouton menant à la page de prise de photo. Pour l'ajouter, c'est très simple, on crée d'abord la classe de cette page :

- Fichier `src/Infrastructure/Pages/TakePicturePage.php` :

```
<?php
namespace My\Infrastructure\Pages;

use Dupot\StaticManagementFramework\Page\PageAbstract;
use Dupot\StaticManagementFramework\Render\Layout;
use Dupot\StaticManagementFramework\Render\View;

class TakePicturePage extends PageAbstract
{
    protected $layout = null;
    public function __construct()
    {
        $this->layout = new Layout(__DIR__ . '/Layouts/default.php');
        $this->layout->assignContext('title', 'Souriez, dites whistiti');
    }

    public function index()
    {
        $view = new View(
            __DIR__ . '/View/take.php',
            []
        );
        $this->layout->appendContext('contentList', $view);
        return $this->render();
    }

    public function render()
    {
        echo $this->layout->render();
    }
}
```

Celle-ci ressemble beaucoup à la précédente, et l'on remarque déjà que l'on se répète beaucoup trop. Mais avec la magie des héritages, on pourrait simplifier.

Créons une classe de page de base :

- Fichier `src/Infrastructure/Pages/BasePage.php` :

```
<?php
namespace My\Infrastructure\Pages;

use Dupot\StaticManagementFramework\Page\PageAbstract;
use Dupot\StaticManagementFramework\Render\Layout;
use Dupot\StaticManagementFramework\Render\View;
```



```
class BasePage extends PageAbstract
{
    protected $layout = null;
    protected $title = null;

    public function __construct()
    {
        $this->layout = new Layout(__DIR__ . '/Layouts/default.php');
        $this->layout->assignContext('title', $this->title);
    }

    public function renderWithContentView(View $view)
    {
        $this->layout->appendContext('contentList', $view);
        return $this->render();
    }

    public function render()
    {
        echo $this->layout->render();
    }
}
```

Ici, on rassemble les méthodes communes, ce qui simplifiera énormément nos futures pages.

Nous avons désormais :

- Fichier **src/Infrastructure/Pages/HomePage.php** :

```
<?php
namespace My\Infrastructure\Pages;

use Dupot\StaticManagementFramework\Render\View;
class HomePage extends BasePage
{
    protected $title = 'Bienvenue sur Photobooth';
    public function index()
    {
        return $this->renderWithContentView(
            new View(
                __DIR__ . '/View/home.php',
                []
            )
        );
    }
}
```

On a simplement le titre de notre page comme propriété et l'on se contente d'appeler une méthode pour lui envoyer notre vue avec ses paramètres.

Idem pour la page de prise de vue :

- Fichier `src/Infrastructure/Pages/TakePicturePage.php` :

```
<?php
namespace My\Infrastructure\Pages;

use Dupot\StaticManagementFramework\Render\View;
class TakePicturePage extends BasePage
{
    protected $title = 'Souriez, dites Cheeese';
    public function index()
    {
        return $this->renderWithContentView(
            new View(
                __DIR__ . '/View/takeApicture.php',
                []
            )
        );
    }
}
```

Pour que la page soit prise en compte lors de la navigation, il suffit de l'ajouter dans le fichier de `routes` vu précédemment :

- Fichier `src/conf/routing.json` :

```
[
    {
        "pattern": "#/takePicture.html#",
        "class": "My\\Infrastructure\\Pages\\TakePicturePage",
        "method": "index"
    },
    {
        "pattern": "#/#",
        "class": "My\\Infrastructure\\Pages\\HomePage",
        "method": "index"
    }
]
```

On reprend le nom de page indiqué dans la page d'accueil et on lui indique d'instancier cette nouvelle classe et d'appeler sa méthode `index`.

2.4 HTML5/JS : souriez, vous êtes filmé

Passons à la vue qui va permettre d'afficher la webcam. Dans cette page, on souhaite quatre choses :

1. Afficher un aperçu via la webcam.
2. Permettre via un bouton de lancer un décompte avant la prise de photo.
3. Prendre la capture de la webcam et la convertir en image.
4. Envoyer cette image à une API.

Pour la première partie, on peut utiliser le code suivant :

- Fichier `src/Infrastructure/Pages/View/takePicture.php` :

```
<div class=" d-flex align-items-center justify-content-center">
  <video id="video" playsinline autoplay></video>
</div>

<script>
  const video = document.getElementById('video');
  const constraints = {
    audio: false,
    video: {
      width: 1366,
      height: 910
    }
  };
  async function init() {
    const stream = await navigator.mediaDevices.getUserMedia(constraints);
    handleSuccess(stream);
  }
  function handleSuccess(stream) {
    window.stream = stream;
    video.srcObject = stream;
  }
  init();
</script>
```

On lit aisément que l'on écrit un bloc HTML5 `video`, on récupère le flux avec nos paramètres (pas de son et la résolution choisie) et on l'assigne à notre objet.

En vous rendant à cet instant sur votre page, vous devriez avoir une notification pour autoriser le navigateur à utiliser la webcam, puis vous devriez vous voir à l'écran.

Ajoutons le bouton de décompte qui prendra la photo :

```
<style>
  #capture {
    position: absolute;
    top: 50%;
    left: 50%;
    z-index: 200;
  }
  #capture span {
    color: white;
    font-size: 200px;
  }
</style>

<div class=" d-flex align-items-center justify-content-center">
  <video id="video" playsinline autoplay></video>
</div>
```



```
<div id="capture">
  <button class="btn btn-primary btn-lg" onclick="return
delayTakePicture()">Prendre la photo ^_^</button>
</div>
```

On ajoute une zone **capture** qui contiendra à la fois le bouton et le décompte en cours. Le tout est situé au premier plan : devant la prévisualisation de la webcam.

Le bouton pour prendre la photo appelle une nouvelle fonction qui gèrera à la fois le décompte et une fois atteint, la fonction de prise de capture de l'image et de conversion comme suit :

```
<script>
...
var countdown = 0;
var maxCountdown = 5;

function delayTakePicture() {
  countdown += 1;
  let captureDiv = document.getElementById('capture');
  if (captureDiv) {
    captureDiv.innerHTML = '<span>' + (maxCountdown - countdown) +
    '</span>';
  }
  if (countdown >= maxCountdown) {
    return takePicture();
  }
  setTimeout(delayTakePicture, 1000);
}

function takePicture() {
  let canvas = document.createElement('canvas');
  canvas.width = 1200;
  canvas.height = 800;
  let ctx = canvas.getContext('2d');
  ctx.drawImage(video, 0, 0, canvas.width, canvas.height);
  let image = canvas.toDataURL('image/jpeg');
  callApiToConvertImage(image);
}

async function callApiToConvertImage(image) {
  const formData = new FormData();
  formData.append("image", image);
  const response = await fetch("apiConvert.html", {
    method: "POST",
    body: formData,
  });
  const jsonData = await response.json();
  redirectToPreviewPage(jsonData.imagePath);
}

function redirectToPreviewPage(imagePath) {
  document.location.href = 'previewPicture.html?imagePath=' + imagePath;
}
...
</script>
```


Ce qui est intéressant ici, c'est la manière dont on passe d'un flux vidéo issu de la webcam à une image JPEG exploitable. Pour cela, comme vous pouvez le lire, on passe par la création d'un objet **canvas** en HTML5, sur lequel on vient « dessiner » notre flux vidéo à cet instant T.

Enfin, pour pouvoir détourner notre image, nous devons l'envoyer à notre API, et pour cela, nous utilisons la méthode **toDataURL** pour la convertir dans un format compatible avec l'envoi HTTP.

Enfin, on envoie à notre API **apiConvert.html** (que nous allons écrire ensuite), et nous récupérons le chemin de la nouvelle image après traitement par l'API, que nous transmettons à la page suivante pour la prévisualisation.

2.5 Une API de conversion, s'il vous plaît

Pour supprimer le fond vert, il nous faut installer une dépendance au sein de notre environnement de développement, pour cela, on va passer par une image Docker personnalisée.

Créons un fichier **DockerFile** :

- Fichier **Dockerfile** :

```
FROM lavoweb/php-7.4
RUN apt update
RUN apt install -y imagemagick
EXPOSE 80
```

On part de la même image renseignée dans notre fichier **docker-compose** de départ, à laquelle on ajoute le paquet **imagemagick**.

Ensuite, on modifie en conséquence le fichier **docker-compose** pour utiliser cette image personnalisée.

- Fichier **docker-compose.yml** :

```
version: "2"

services:
  web:
    build: .
    ports:
      - "80:80"
    volumes:
      - ../var/www/html
      - ../vhosts:/etc/apache2/sites-enabled/
```

Une fois modifié, il vous suffit de relancer la commande précédente pour permettre à Docker de construire et d'utiliser cette nouvelle image avec la commande :

```
docker-compose up
```


Créons maintenant cette API : pour cela, il nous suffit de créer une nouvelle classe, cette fois dans un autre répertoire **Apis** dans le dossier **Infrastructure** pour bien séparer les pages des API :

- Fichier **src/Infrastructure/Apis/ConvertApi.php** :

```
<?php
namespace My\Infrastructure\Apis;

use Dupot\StaticManagementFramework\Page\PageAbstract;
class ConvertApi extends PageAbstract
{
    const BIN_CONVERT = '/usr/bin/convert';
    const DEST_PATH = 'pictures';
    protected $logList;
    public function log($text)
    {
        $this->logList[] = $text;
    }
    public function convert()
    {
        if (!$this->getRequest()->isMethodPost()) {
            json_encode('Only post expected !!!');
        }
        $rawImage = $_POST['image'];
        $rawImageList = explode(',', $rawImage);
        $imageContent = base64_decode($rawImageList[1]);
        $originFilename = '/tmp/maPhoto' . date('YmdHis') . '.jpg';
        $this->log('Try to write temp image into ' . $originFilename);
        file_put_contents($originFilename, $imageContent);
        $this->log('Try to remove background ' . $originFilename);
        $imagePath = $this->removeBackgroundForImage($originFilename);
        $response = (object)[
            'imagePath' => $imagePath,
            'logList' => $this->logList
        ];
        return $this->returnJson($response);
    }
    protected function returnJson($response)
    {
        header('Content-Type: application/json; charset=utf-8');
        echo json_encode($response);
    }
    protected function getHexa($value)
    {
        $valueHexa = dechex($value);
        if (strlen($valueHexa) < 2) {
            $valueHexa = '0' . $valueHexa;
        }
        return $valueHexa;
    }
    protected function fromRGB($red, $green, $blue)
    {

```



```

    $redHexa = $this->getHexa($red);
    $greenHexa = $this->getHexa($green);
    $blueHexa = $this->getHexa($blue);

    return '#' . $redHexa . $greenHexa . $blueHexa;
}

protected function removeBackgroundForImage($tmpOriginImagePath)
{
    $tmpFilenameWithoutBackground = $tmpOriginImagePath . ".transparent.png";

    $command = self::BIN_CONVERT . ' -limit thread 4';
    $command .= ' ';
    $command .= $tmpOriginImagePath;
    $command .= ' ';
    $command .= "[1x1+10+10] -format '[fx:int(255*r)],[fx:int(255*g)],
%[fx:int(255*b)]' info:";

    $lastOutput = $this->logExecuteAndReturnOutput($command);

    list($red, $green, $blue) = explode(' ', $lastOutput);

    $rgbHexa = $this->fromRGB($red, $green, $blue);

    $transparentCommand = self::BIN_CONVERT;
    $transparentCommand .= ' ';
    $transparentCommand .= $tmpOriginImagePath;
    $transparentCommand .= ' -fuzz 20%% -transparent "' . $rgbHexa . '"';
    $transparentCommand .= ' ';
    $transparentCommand .= $tmpFilenameWithoutBackground;

    $this->logAndExecute($transparentCommand);
    $moveCommand = 'mv ' . $tmpFilenameWithoutBackground . ' ' . __DIR__ .
'/.../.../public/' . self::DEST_PATH;
    $this->logAndExecute($moveCommand);

    return self::DEST_PATH . '/' . basename($tmpFilenameWithoutBackground);
}

protected function logAndExecute($command)
{
    $this->log($command);
    exec($command);
}

protected function logExecuteAndReturnOutput($command)
{
    $this->log($command);
    exec($command, $outputList);

    return $outputList[0];
}
}

```


On commence par identifier la couleur présente en haut à gauche de notre photo, puis on demande de remplacer cette couleur (après l'avoir convertie en hexadécimal) en couleur transparente et d'enregistrer le tout au format PNG (format qui gère la transparence, contrairement au JPEG).

On se retrouve ainsi avec une image dont le fond vert a été remplacé par des pixels transparents, on la copie dans le répertoire `public/pictures` de notre application et on renvoie son adresse.

Notez l'ajout d'une méthode de *log* bien pratique pour déboguer notre API, pour les voir, il vous suffit de cliquer sur la touche F12 de votre navigateur et de vous rendre dans l'onglet **network**.

Avant de l'exécuter, pensez à créer un répertoire `pictures` et à lui paramétrer les bonnes permissions :

```
mkdir public/pictures/
chmod -R 777 public/pictures/
```

2.6 Prévisualisation de la photo prise

Si l'on revient en arrière, partie *front*, à la réception de la réponse API (de suppression du fond vert), on crée une redirection avec l'adresse de l'image :

- Fichier `src/Infrastructure/Pages/View/takePicture.php` :

```
<script>
  (...)
  async function callApiToConvertImage(image) {

    const formData = new FormData();
    formData.append("image", image);

    const response = await fetch("apiConvert.html", {
      method: "POST",
      body: formData,
    });
    const jsonData = await response.json();
    redirectToPreviewPage(jsonData.imagePath);
  }

  function redirectToPreviewPage(imagePath) {
    document.location.href = 'previewPicture.html?imagePath=' + imagePath;
  }
  (...)
</script>
```


Créons donc cette page de prévisualisation en commençant par la classe **page** :

- Fichier **src/Infrastructure/Pages/PreviewPicturePage.php** :

```
<?php
namespace My\Infrastructure\Pages;

use Dupot\StaticManagementFramework\Render\View;

class PreviewPicturePage extends BasePage
{
    protected $title = 'Merci de la photo';
    public function index()
    {
        return $this->renderWithContentView(
            new View(
                __DIR__ . '/View/preview.php',
                []
            )
        );
    }
}
```

Et son fichier d'affichage :

- Fichier **src/Infrastructure/Pages/View/preview.php** :

```
<style>
    #capture {
        position: absolute;
        top: 50%;
        left: 50%;
        z-index: 200;
    }
    #controls {}
</style>

<div class=" d-flex align-items-center justify-content-center" style="height: 700px;">
    
</div>

<div id="capture">
    <a href="#" class="btn btn-secondary btn-lg" onclick="nextBackground()">Fond
    suivant</a>
    <a autofocus href="#" class="btn btn-primary btn-lg"
    onclick="validateAndGoToHome()">Valider et retour au Menu</a>
</div>

<script>
    var backgroundNumberSelected = 1;
    var backgroundNumberMax = 3;
    var backgroundPattern = 'fond_0__NUMBER__.png';
```



```
function nextBackground() {
    var a = document.getElementById('previewImage');
    if (a) {
        a.style.backgroundImage = 'url(\'../backgrounds/' + backgroundPattern.
replace('__NUMBER__', backgroundNumberSelected) + '\')';

        backgroundNumberSelected++;

        if (backgroundNumberSelected > backgroundNumberMax) {
            backgroundNumberSelected = 1;
        }
    }
    nextBackground();
}

function validateAndGoToHome() {
    let image = '<?php echo $_GET['imagePath'] ?>';
    let background = 'backgrounds/' + backgroundPattern.replace('__NUMBER__',
backgroundNumberSelected);
    callApiToMergeImage(image, background);
}

async function callApiToMergeImage(imagePath, backgroundPath) {

    const formData = new FormData();
    formData.append("imagePath", imagePath);
    formData.append("backgroundPath", backgroundPath);

    const response = await fetch("apiMerge.html", {
        method: "POST",
        body: formData,
    });

    const jsonData = await response.json();

    redirectToHomePage(jsonData.imagePath);
}

function redirectToHomePage(imagePath) {
    document.location.href = '/';
}
}
</script>
```

Ici, deux choses : un bouton permet de changer dynamiquement le fond de l'image, et un autre permet de valider le choix, d'appeler une API pour fusionner le tout et revenir au menu.

La partie dynamique alterne la propriété CSS **backgroundImage** de l'image précédemment générée.

Quant à l'appel API, tout comme le précédent, on se contente d'envoyer en POST les paramètres pour la fusion avec l'image de base ainsi que l'image de fond validée.

Intéressons-nous à cette classe API de fusion :

- Fichier `src/Infrastructure/Api/MergeApi.php` :

```
<?php
namespace My\Infrastructure\Api;

use Dupot\StaticManagementFramework\Page\PageAbstract;

class MergeApi extends PageAbstract
{
    const BIN_CONVERT = '/usr/bin/convert';
    const DEST_PATH = 'pictures';
    const DEST_FINALPATH = 'pictures/final/';

    protected $logList;
    public function log($text)
    {
        $this->logList[] = $text;
    }
    public function merge()
    {
        if (!$this->getRequest()->isMethodPost()) {
            json_encode('Only post expected !!!');
        }

        $imagePath = $_POST['imagePath'];
        $backgroundPath = $_POST['backgroundPath'];

        $this->log('Try to merge image ' . $imagePath . ' and background ' .
$backgroundPath);

        $imageMergedPath = $this->mergeImageAndBackground($imagePath,
$backgroundPath);

        $response = (object)[
            'imagePath' => $imageMergedPath,
            'logList' => $this->logList
        ];

        return $this->returnJson($response);
    }
    protected function returnJson($response)
    {
        header('Content-Type: application/json; charset=utf-8');
        echo json_encode($response);
    }
    protected function mergeImageAndBackground($imagePath, $backgroundPath)
    {
        $rootDirectory = __DIR__ . '/../.../public/';
```



```

$destDirectory = __DIR__ . '/../../../public/' . self::DEST_FINALPATH;
$mergedImagePath = basename($imagePath) . '_final.png';

$mergeCommand = self::BIN_CONVERT;
$mergeCommand .= ' -composite';
$mergeCommand .= ' ';
$mergeCommand .= $rootDirectory . '/' . $backgroundPath;
$mergeCommand .= ' ';
$mergeCommand .= $rootDirectory . '/' . $imagePath;
$mergeCommand .= ' ';
$mergeCommand .= $destDirectory . '/' . $mergedImagePath;

$this->logAndExecute($mergeCommand);

return self::DEST_FINALPATH . '/' . basename($mergedImagePath);
}

protected function logAndExecute($command)
{
    $this->log($command);
    exec($command);
}
}

```

Dans cette classe, on peut voir que l'on appelle le binaire de conversion d'**imageMagick** pour fusionner l'image au fond supprimé plus l'image de fond, pour générer l'image finale stockée dans le sous-répertoire **final** de **pictures**.

N'oubliez pas de créer et de donner les bons droits à ce nouveau sous-répertoire :

```

mkdir public/pictures/final
chmod -R 777 public/pictures/final

```

Enfin, comme jusqu'à présent, il faut renseigner les différentes routes pour relier les deux précédentes classes :

- Fichier **src/conf/routing.json** :

```

{
    "pattern": "#/previewPicture.html#",
    "class": "My\\Infrastructure\\Pages\\PreviewPicturePage",
    "method": "index"
},
{
    "pattern": "#/apiMerge.html#",
    "class": "My\\Infrastructure\\Apis\\MergeApi",
    "method": "merge"
},
...

```

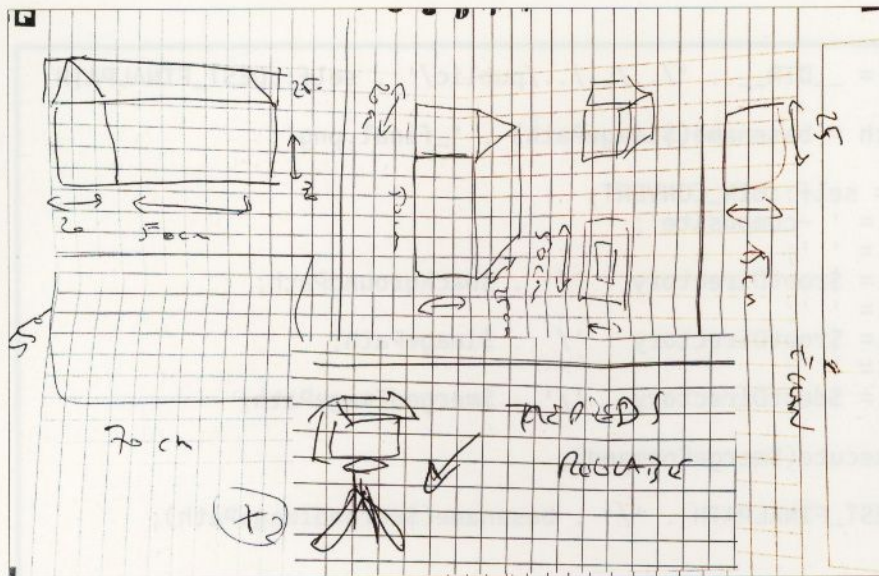



Fig. 1 : Une feuille et un crayon permettent facilement de mettre ses idées au clair (et de ne pas oublier de planches de bois).

Fig. 2 : Avec uniquement des planches, des tasseaux et des vis, on peut facilement monter toute la structure.



3. SI J'AVAIS UN MARTEAU... : LA PARTIE PHYSIQUE

3.1 Dessine-moi un mouton

Retour du crayon, cette fois pour le boîtier qui recevra tout cette électronique.

Pour construire un boîtier, il nous faut nous rendre dans des magasins de bricolage, et pour éviter de devoir perdre du temps en aller-retour pour des pièces oubliées, il est

important de bien poser sur papier le croquis de l'objet à construire, qu'importe si c'est ou non joliment dessiné, l'important c'est d'avoir sur le croquis l'ensemble des éléments, pour bien identifier les éléments à acheter et ceux à découper/percer pour construire l'ensemble.

Vous pouvez voir sur l'image 1 un des croquis en début de projet.

3.2 Sortez les outils, papa bricole

Rien de très compliqué, même si vous n'êtes pas très manuel : des planches de bois, des tasseaux, des vis et de quoi couper/percer/visser.

J'ai prévu que tout soit facilement démontable pour deux raisons :

1. Pouvoir transporter séparément le boîtier et la partie électronique plus fragile/délicate.
2. Pouvoir, pendant la soirée, intervenir pour déboguer/corriger l'installation.

Pour la partie *hardware*, un simple Raspberry Pi connecté à un petit écran installé pour être visible via une simple découpe dans une des planches de bois. Enfin, une webcam USB vissée au-dessus de l'écran.

Sur l'image 2, vous pouvez voir les principaux points « techniques » mis en place, que ce soit pour fixer les différentes planches sans devoir acheter un lot d'équerres, et également le fameux support permettant d'installer / enlever l'écran facilement.

3.3 Résultat final

Pas besoin ici d'un long discours pour expliquer comment installer un Raspberry Pi, j'ai simplement installé la dernière version de la distribution officielle Raspbian.

Vous pouvez apprécier sur l'image 3, page suivante, le rendu final durant la soirée : le pied de mon parasol faisant office de pied très stable, l'écran vert disposé derrière avec sa structure rétractable ainsi que les deux éclairages LED branchés à la fois sur la prise multiple (via un adaptateur secteur) ainsi que sur l'un des ports USB du Raspberry Pi.



Fig. 3 : Une fois peint, installé et allumé, le résultat final a eu beaucoup de succès.

enfants et même les adultes à force de toucher à celui-ci décalent l'installation de départ, faite pour avoir le fond vert cadré aux petits oignons.

L'idée était donc de permettre la navigation dans cette application web via une manette USB.

Première étape : installer les *drivers* pour la faire reconnaître par l'OS. Je me suis orienté sur une manette Nintendo Switch, je pensais qu'en choisissant une grande marque, je n'aurais pas de difficulté à trouver les *drivers* pour le Raspberry Pi.

En effet, il existe un projet GitHub à l'adresse <https://github.com/nicman23/dkms-hid-nintendo> qui permet facilement de faire reconnaître les manettes du constructeur sur nos OS préférés :

```
cd
mkdir drivers
cd drivers
git clone https://github.com/nicman23/dkms-hid-nintendo
cd dkms-hid-nintendo
sudo dkms add .
sudo dkms build nintendo -v 3.2
sudo dkms install nintendo -v 3.2
```

4. ONE MORE THING

4.1 Utilisation d'une manette

À cette étape, tout fonctionne au clic, ceci fonctionnerait très bien en l'état en achetant un simple écran tactile, mais comme dit précédemment, je craignais que les

Après un redémarrage, la manette est désormais reconnue par l'OS, il faut que le navigateur puisse la prendre en compte. Pour information, nativement aucun événement n'est émis à l'appui des touches de notre manette, il vous faut coder sa prise en charge grâce à l'API HTML5 **Gamepad**.

Comme nous avons besoin d'utiliser cette manette sur toutes les pages, nous ajoutons ce code dans notre **Layout**, il sera donc ainsi centralisé. Celui-ci permet d'émettre des événements lors de la détection d'appui sur certaines de nos touches :

- Fichier **src/Infrastructure/Pages/Layouts/default.php** :

```
...
<script>
window.addEventListener("gamepadconnected", function(e) {
  console.log("Manette connectée à l'indice %d : %s. %d boutons, %d axes.",
    e.gamepad.index, e.gamepad.id,
    e.gamepad.buttons.length, e.gamepad.axes.length);
  loopGamepad();
});

function loopGamepad() {

  let buttonA = false;
  let buttonB = false;
  let gamepadList = navigator.getGamepads()
  let myGamepad = gamepadList[0];

  if (myGamepad.buttons[2].pressed) {
    buttonA = true;
  }
  if (myGamepad.buttons[1].pressed) {
    buttonB = true;
  }
  if (buttonA) {
    console.log('button A');
    const event = new Event("buttonA");
    document.dispatchEvent(event);
  }
  if (buttonB) {
    console.log('button B');
    const event = new Event("buttonB");
    document.dispatchEvent(event);
  }
  setTimeout(loopGamepad, 500);
}
</script>
...
```

À la connexion de notre manette, nous appelons une fonction en boucle qui récupère la première (et unique) manette. Nous regardons si l'une des touches utilisées dans notre application est pressée et si tel est le cas, nous créons un **event** avec le nom de la touche pressée :

• Fichier `src/Infrastructure/Pages/View/home.php` :

```
<div class=" d-flex align-items-center justify-content-center" style="height:
700px;">

    <p>
        <a id="buttonGalery" href="/galery.html" class="btn btn-success btn-lg"
role="button">

            <svg xmlns="http://www.w3.org/2000/svg" width="16" height="16"
fill="currentColor" class="bi bi-camera-fill " viewBox="0 0 16 16">
                <path d="M10.5 8.5a2.5 2.5 0 1 1-5 0 2.5 2.5 0 0 1 5 0z" />
                <path d="M2 4a2 2 0 0 0-2 2v6a2 2 0 0 2 2h12a2 2 0 0 2-2V6a2 2 0
0 0-2-2h-1.172a2 2 0 0 1-1.414-.586l-.828-.828A2 2 0 0 0 9.172 2H6.828a2 2 0 0
0-1.414.586l-.828.828A2 2 0 0 1 3.172 4H2zm.5 2a.5.5 0 1 1 0-1 .5.5 0 0 1 0 1zm9
2.5a3.5 3.5 0 1 1-7 0 3.5 3.5 0 0 1 7 0z" />
            </svg><br />
            Galerie</a>
        </p>

        <p>
            <a id="buttonTakePicture" href="/takePicture.html" class="btn btn-primary
btn-lg" role="button">

                <svg xmlns="http://www.w3.org/2000/svg" width="16" height="16"
fill="currentColor" class="bi bi-camera-fill " viewBox="0 0 16 16">
                    <path d="M10.5 8.5a2.5 2.5 0 1 1-5 0 2.5 2.5 0 0 1 5 0z" />
                    <path d="M2 4a2 2 0 0 0-2 2v6a2 2 0 0 2 2h12a2 2 0 0 2-2V6a2 2
0 0 0-2-2h-1.172a2 2 0 0 1-1.414-.586l-.828-.828A2 2 0 0 0 9.172 2H6.828a2 2 0 0
0-1.414.586l-.828.828A2 2 0 0 1 3.172 4H2zm.5 2a.5.5 0 1 1 0-1 .5.5 0 0 1 0 1zm9
2.5a3.5 3.5 0 1 1-7 0 3.5 3.5 0 0 1 7 0z" />
                </svg><br />
                Prendre une photo</a>
            </p>
        </div>

<script>
    var buttonGalery = document.getElementById('buttonGalery');
    var buttonTakePicture = document.getElementById('buttonTakePicture');
    var buttonsList = [
        buttonGalery,
        buttonTakePicture
    ]
    var buttonSelected = 0;

    switchButton();

    function switchButton() {
        buttonsList[buttonSelected].style.height = '120px';
        buttonsList[buttonSelected].style.width = '350px';
        buttonsList[buttonSelected].style.opacity = '60%';
```



```

        if (buttonSelected === 0) {
            buttonSelected = 1;
        } else {
            buttonSelected = 0;
        }
        buttonsList[buttonSelected].style.height = '180px';
        buttonsList[buttonSelected].style.width = '360px';
        buttonsList[buttonSelected].style.opacity = '100%';
    }
    function gotoSelected() {
        buttonsList[buttonSelected].click();
    }
    document.addEventListener(
        "buttonA",
        (e) => {
            switchButton();
        },
        false
    );
    document.addEventListener(
        "buttonB",
        (e) => {
            gotoSelected();
        },
        false
    )
</script>
<div style="font-size:80px;color:white;position:absolute;bottom:0px;left:0px"
>A: choix, B: valider</div>

```

J'ai volontairement ajouté un nouveau bouton sur la page d'accueil permettant d'afficher une galerie des photos déjà prises, pour permettre de voir l'utilisation de deux boutons (pour rendre l'exemple plus intéressant).

On commence par enregistrer dans un tableau les deux boutons présents sur l'interface, puis on écoute les événements « touche » appuyés créés dans la partie **Layout** pour, au choix :

- soit passer le focus d'un bouton à l'autre ;
- soit valider le bouton sélectionné en appelant sa méthode **click()**.

4.2 Ajout d'une galerie des photos prises

Ajoutée précédemment pour rendre plus intéressante notre page d'accueil, nous pouvons facilement créer cette page de navigation :

- Création de la classe **src/Infrastructure/Pages/GalleryPage.php** :

```

<?php
namespace My\Infrastructure\Pages;

use Dupaot\StaticManagementFramework\Render\View;

```



```
class GaleryPage extends BasePage
{
    protected $title = 'Galerie des photos';
    public function index()
    {
        $pictureList = [];
        $fileList = scandir(__DIR__ . '/../../public/pictures/final');
        foreach ($fileList as $fileLoop) {
            if (substr($fileLoop, 0, 1) == '.') continue;
            $pictureList[] = basename($fileLoop);
        }
        return $this->renderWithContentView(
            new View(
                __DIR__ . '/View/galery.php',
                ['pictureList' => $pictureList]
            )
        );
    }
}
```

On récupère ici la liste des fichiers présents dans le répertoire utilisé pour générer les photos fusionnées et on les transmet à la vue :

• Fichier `src/Infrastructure/Pages/View/galery.php` :

```
<div id="pictureCarousel" class="carousel slide">
    <div class="carousel-indicators">
        <?php foreach (array_keys($this->contextList['pictureList']) as $i) : ?>
            <button type="button" data-bs-target="#carouselExampleIndicators" data-
            bs-slide-to="<?php echo $i ?>" <?php if ($i === 0) : ?>class="active"
            aria-current="true" <?php endif; ?> aria-label="Slide <?php echo $i ?>"></button>
        <?php endforeach; ?>
    </div>

    <div class="carousel-inner">
        <?php foreach ($this->contextList['pictureList'] as $i => $pictureLoop) : ?>
            <div class="carousel-item <?php if ($i === 0) : ?>active<?php endif; ?>">
                
            </div>
        <?php endforeach; ?>
    </div>

    <button class="carousel-control-prev" type="button" data-bs-
    target="#carouselExample" data-bs-slide="prev">
        <span class="carousel-control-prev-icon" aria-hidden="true"></span>
        <span class="visually-hidden">Previous</span>
    </button>
    <button class="carousel-control-next" type="button" data-bs-
    target="#carouselExample" data-bs-slide="next">
        <span class="carousel-control-next-icon" aria-hidden="true"></span>
```



```

        <span class="visually-hidden">Next</span>
      </button>
    </div>

    <script>
      const myCarouselElement = document.getElementById('pictureCarousel');
      var carousel;
      document.onreadystatechange = function() {
        if (document.readyState === "complete") {
          carousel = new bootstrap.Carousel(myCarouselElement, {
            interval: 5000,
            wrap: true,
            touch: true,
            keyboard: true,
            pause: "hover",
            prevNext: true,
            prevNextText: ""
          });
        }
      };

      document.addEventListener(
        "buttonA",
        (e) => {
          goToHome();
        },
        false
      );

      document.addEventListener(
        "buttonB",
        (e) => {
          carousel.next();
        },
        false
      );

      function goToHome() {
        document.location.href = '/';
      }
    </script>

```

Ici, on utilise le composant **carousel** de Bootstrap et on permet le passage à l'image suivante via le bouton B de la manette (en appelant la méthode **next()**) et le retour à la page d'accueil via le bouton A.

4.3 Attention à la mise en veille

Par défaut, le Raspberry Pi inclut un mécanisme de mise en veille de l'écran, pouvant provoquer quelques soucis lors d'une soirée, du genre « le photobooth s'est éteint... ».

Pour pallier ce souci, vous pouvez sur l'interface graphique du Raspberry Pi, dans le menu de configuration, désactiver la ligne **Screen blanking**.

Cette option est également disponible en ligne de commande via :

```
sudo raspi-config
```

Naviguez dans **Display option > Screen blanking**.

4.4 Plus de framboises ? Prenez une banane... ou pas...

Ayant suivi la pénurie de la fameuse carte, j'avais cherché une alternative peu onéreuse comme ordinateur pour ce photobooth, j'avais trouvé, commandé et reçu la « copie » assumée du Raspberry Pi, à savoir le Banana Pi.

J'aurai du présenter tout cet article en prenant pour base cette carte qui semblait la bonne alternative, sauf que je n'ai jamais réussi à la faire démarrer : uniquement voyant rouge de l'alimentation, pas de diode verte de lecture de la carte micro SD. J'ai eu beau essayer également de démarrer via une clef USB, sans succès.

J'hésite donc à vous conseiller cette alternative pourtant très bon marché : autour des 50 € sur le célèbre site chinois « Ali rapide », qui pour information nécessite un autre OS que la distribution dérivée de Debian : Raspbian.

Pour ceux que cela intéresse, il y a une page wiki bien expliquée à cette adresse : https://wiki.banana-pi.org/Banana_Pi_BPI-M2_Berry#Image_Release.

CONCLUSION

Dans cet article, vous avez pu découvrir que vous pouvez très facilement fabriquer votre propre photobooth et ceci très facilement. Une fois commencé, vous pouvez facilement agrémenter/améliorer le produit comme ajouter des fonctionnalités « post-prise de vue » pour y appliquer divers filtres.

Attention à bien éclairer le fond vert utilisé lors de l'installation, au risque d'avoir des difficultés à le supprimer automatiquement à cause des ombres. Au mieux, il vous faudrait un éclairage puissant derrière vos sujets éclairant uniquement votre fond ainsi qu'un éclairage plus léger éclairant juste les personnes au premier plan pour ne pas avoir à recourir à un flash.

Lors de l'installation, j'ai choisi de connecter le Raspberry Pi au Wi-Fi de l'ordinateur utilisé pendant la soirée tournant sous Kubuntu 22.04 (via un partage de connexion disponible dans KDE en 2 clics). Ceci m'a permis de mettre en **crontab** une commande **rsync** pour récupérer les photos du photobooth et les diffuser via un vidéo projecteur sous forme de diaporama.

Mais si vous le souhaitez, vous pouvez ajouter un bouton pour envoyer les photos par e-mail, voir proposer un bouton d'impression (même si vous prenez le risque de manger le papier trop rapidement).

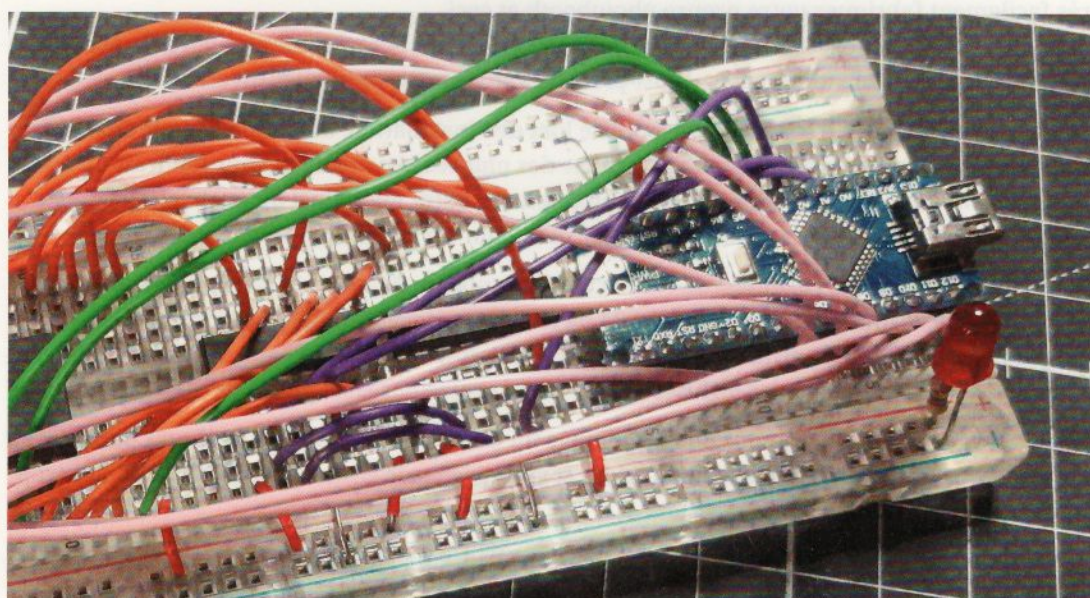
L'ensemble du projet se trouve sur mon GitHub à l'adresse : <https://github.com/imikado/articleLMphotoBooth>. **MB**



EXPLORATION D'UNE CARTE INDUSTRIELLE À BASE DE Z180

Denis BODOR

Dans mes pérégrinations visant à remettre en marche mon projet d'ordinateur 8 bits construit autour d'un modeste, mais classique Zilog Z80, j'ai croisé la route d'une carte très intéressante et surtout, très peu chère. Mais jeter son dévolu sur un produit clairement destiné à un autre usage qu'un projet didactique est une aventure, un défi et un pari plus qu'une approche raisonnée. La carte en question est ce qui semble être un contrôleur industriel créé par une défunte société italienne, vendu aux enchères sur eBay et livrée sans EPROM, mais avec un schéma. Enchère gagnée, livraison réceptionnée, que l'aventure commence !



Construire un ordinateur 8 bits est un projet passionnant, mais il faut avouer que la réalisation elle-même, pour arriver à quelque chose d'utilisable ou au minimum, de transportable (ce qui n'est pas le cas d'une salade de câbles entre plusieurs platines à essais), demande des efforts supplémentaires. La phase expérimentale débute avec la platine à essais, puis la plaque pastillée, pour arriver finalement, davantage par dépit ou obligation qu'autre chose, au vu des problèmes de stabilité, à la conception d'un vrai circuit imprimé avec un outil comme KiCad et une production de petite quantité par un prestataire comme PCBWay, NextPCB, EuroCircuit ou autre. En ce qui me concerne, voir la réalisation « faire quelque chose » est ce que je convoite en premier lieu. Devoir « faire propre », une fois que cela fonctionne, est beaucoup moins amusant. Ça ne l'était déjà pas à l'époque où il était question d'utiliser du *perchlo* et ça ne l'est pas davantage, aujourd'hui, en faisant fabriquer le circuit par un tiers.

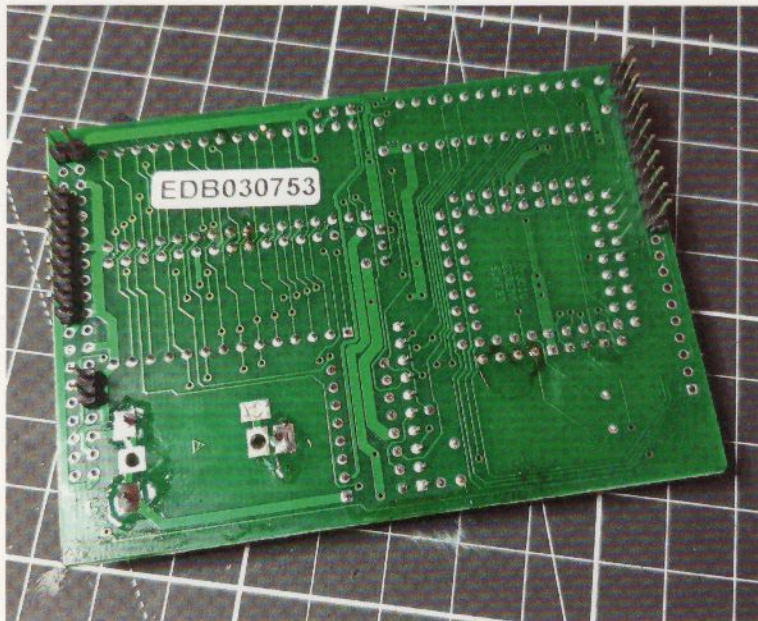
La tentation est grande alors de sauter des étapes et, plutôt que de se lancer



dans la conception d'un circuit, de chercher quelque chose de fini s'approchant suffisamment de ses propres préférences. Les projets d'ordinateurs « maison » sur base Z80 ou MOS 6502 ne manquent pas, mais en trouver un qui correspond exactement à ses besoins (ou envies), tout en restant à un coût raisonnable, est une tâche en elle-même. Beaucoup de projets visent un objectif très complet incluant généralement une sortie vidéo et audio pour reproduire, ni plus ni moins, ce que nos aïeux ont fait pour concevoir les ordinateurs familiaux 8 bits d'antan (ZX Spectrum, C64, Apple II, TRS 80, etc.).

C'est en cherchant une alternative fiable et économique, peut-être sous la forme d'un simple circuit imprimé à peupler de composants et éventuellement à adapter sauvagement à mes besoins que je suis tombé sur l'annonce eBay d'un vendeur italien, intitulée « Scheda core a microprocessore Zilog Z180 con schema - SENZA PREZZO DI RISERVA » (« Carte cœur à microprocesseur Zilog Z180 avec schéma - PAS DE PRIX DE RÉSERVE »). Mise aux enchères à seulement 4 € (mais avec 19 € de port), j'ai alors tenté ma chance et gagné l'enchère sans

Voici la carte telle que réceptionnée. C'est un ensemble relativement compact et simple se résumant à peu de composants. Notez que l'EPROM n'est pas présente et que cet emplacement est initialement prévu pour une UVPRM 27C256 ou 27C512.



L'arrière de la carte ne présente aucun composant et est équipé d'une série de connecteurs soudés. L'une des premières étapes, après nettoyage à l'alcool isopropylique (ou « isopropanol ») sera de dessolder ces connecteurs et d'en souder de nouveaux, de l'autre côté de la carte, et ce, sur l'ensemble des emplacements.

trop de difficulté, puisque j'étais le seul enchérisseur. Les 19 € de transport se sont expliqués d'eux-mêmes à la réception du colis, par UPS svp ! Je n'ai personnellement jamais vu ou reçu un paquet emballé avec autant de soin et de protections. Je pense que la camionnette UPS aurait pu rouler dessus que la carte n'aurait pas subi le moindre dommage...

Le vendeur, un certain Stefano (« raeltubes » [1] sur eBay), m'a confirmé avoir en sa possession « several of these boards » et cette première transaction, suivie des expérimentations que vous allez découvrir ont découlé sur l'acquisition de 5 autres exemplaires via une transaction en direct. Nul doute que bien d'autres exemplaires sont disponibles et n'attendent que votre bon vouloir pour reprendre vie. Si vous aussi souhaitez expérimenter, n'hésitez pas à contacter Stefano sur eBay, les premiers arrivés seront les premiers servis.

Quoi qu'il en soit, l'objet de l'article n'est pas tant de vous détailler l'utilisation de la carte que de partager les péripéties auxquelles on doit s'attendre lorsqu'on opte pour ce type d'approche. C'est aussi une belle occasion de faire un peu de rétro-ingénierie et de découvrir quelques techniques qui peuvent être utiles dans ce domaine, ceci sans pour autant me prétendre être un spécialiste sur le sujet.

1. LA CARTE

La bête se présente sous la forme d'un circuit imprimé de 65 x 90 mm, double-face, relativement compacte. Celle-ci dispose de deux séries de connecteurs avec J1 de 2x20 broches et J2 d'une simple ligne de 21 broches. La première étape de la prise en main a été d'immédiatement dessolder les quelques broches présentes, pour en ressouder de nouvelles dans la totalité des emplacements, mais surtout orientées côté composants pour faciliter l'accès. La carte était également équipée d'une pile bouton CR2430 soudée au PCB, qui a été retirée pour être remplacée par un support pour CR2032.

Faisons le tour du propriétaire :

- Un processeur Zilog Z180 en boîtier PLCC-68, grand frère du Z80, intégrant directement des compteurs/timers 16 bits, un gestionnaire d'interruption, deux ports série, un contrôleur DMA et une MMU (Memory Management Unit) assez basique. Le Z180 est très fortement inspiré du Hitachi 64180 (alias « Super Z80 »), une

version intégrée du Z80, restant compatible avec le Z80 standard, mais plus « capable » puisqu'en mesure de gérer un bus d'adresse de 20 bits et donc un espace mémoire de 1 Mio en lieu et place des quelque 64 Kio d'un Z80. Hitachi souhaitant rester compatible en raison de la quantité phénoménale de code préexistant à l'époque (nous sommes vers la fin des années 80), le 64180 reste un processeur 8 bits avec des registres de 16 bits qui, en principe, est donc bien incapable d'adresser plus de 2^{16} octets de mémoire. C'est là qu'intervient la MMU qui est strictement identique dans le Z180 qui fut produit par Zilog sous licence Hitachi (amusant retournement de situation) sous la forme du Z64180 puis du Z180. Notre processeur est équipé d'un quartz de 18,432 MHz, placé juste à côté.

- Une RAM statique BSI BS62LV4006 de 512 Kio, accompagnée d'un cavalier (« WT1 ») permettant de choisir une taille entre « 128K » et « 256K ». Les tailles sont différentes et le schéma livré avec le produit parle d'une RAM Toshiba TC551001BPL de 128 Kio. Le composant présent n'est-il pas celui d'origine ? Une évolution du circuit non documentée ? Une mauvaise documentation ? Mystère. Quoi qu'il en soit, et dans le doute, j'ai équipé la carte d'une RAM statique TC551001BPL en stock dans mes tiroirs. Nul besoin de dessouder quoi que ce soit, presque tous les composants sont montés sur *sockets* et peuvent être remplacés facilement.
- Un emplacement pour EPROM qui, d'après le schéma fourni par le vendeur, est normalement une UVPROM 27C256. Là encore, un cavalier (« WT2 ») à proximité de l'emplacement (vide pour le coup) permet de choisir « 32K » ou « 64K », ce qui laisse entendre qu'on peut également utiliser une 27C512.
- Un circuit de supervision pour processeur Maxim MAX691A, le seul circuit intégré soudé à la carte. Après un coup d'œil rapide à la *datasheet*, ce composant s'avère être en mesure de non seulement gérer le *reset* à la mise sous tension, mais également de surveiller la tension d'alimentation, d'alimenter temporairement la RAM et de signaler un problème au processeur via une ligne d'interruption. Un *watchdog* est également présent, mais ne semble pas utilisé ici (le signal WDO ne va nulle part, si ce n'est à une broche coupée du composant suivant).
- Un PLD (*Programmable Logic Device*), ancêtre des FPGA modernes, Atmel ATF22V10CQZ. Ce composant est programmable et est destiné à remplacer un jeu de composants logiques (74*). Son travail est d'être à l'écoute de différents signaux de contrôle en provenance du processeur (ou autre), qui provoqueront des changements d'état sur ses sorties en fonction du code avec lequel sa mémoire aura été programmée. Ceci est à la fois une aubaine, car les PLD de ce type (PAG/GAL) reposent sur une technologie très intéressante, et un problème, car il va être difficile de savoir ce que fait exactement ce composant.
- Tout ceci se complète d'un réseau de résistances de 10 kΩ pour tirer quelques signaux à Vcc, une résistance de 100 kΩ pour que le MAX691A mesure la tension et de quatre condensateurs de filtrage de 100 nF.

Et c'est tout ! La carte est, semble-t-il, très simple et ne devrait pas poser de problème... N'est-ce pas ? Grossière erreur que d'estimer une conception par son nombre de composants ! Le Z180 est bien plus complexe que le Z80, l'utilisation d'UVPROM est très pénible lorsqu'on tâtonne et le PLD est une boîte mystère. Mais nous avons un schéma, c'est déjà ça... ou pas (cf. le prochain épisode).

2. MISE EN MARCHÉ SIMPLE

Le Z180 est compatible, par son jeu d'instructions et par son fonctionnement, avec le Z80. Le premier objectif sera donc de lui faire faire quelque chose, comme une simple boucle tout en surveillant si les adresses apparaissant sur le bus sont les bonnes. Ceci fait, et puisque le Z180 intègre le nécessaire pour communiquer avec le monde extérieur, nous pourrions nous pencher sur les deux *ASCI (Asynchronous Serial Communication Interface)* ou en d'autres termes, les deux ports série.

Mais avant cela, nous devons trouver un moyen de faire exécuter un code au processeur et donc de faire en sorte qu'il lise le contenu de l'EPROM, qui n'a pas été livré avec le matériel. Trouver une ou plusieurs UVROM 27C256 n'est ni difficile, ni très coûteux. J'en possède d'ailleurs une large collection provenant en partie de récupérations, à une époque lointaine, sur des cartes mères et modems jugés alors obsolètes et inutiles (comment aurais-je pu me douter, à l'époque, que ces « vieux 80486 poussifs » sont maintenant une denrée rare et recherchée).

Le problème des UVROM n'est pas là, mais dans leur programmation. Pour y enregistrer des données, il est tout d'abord nécessaire de les effacer en exposant la puce, visible via une petite fenêtre en quartz par le dessus du composant, aux UV pendant 10 à 15 minutes. Ce n'est qu'ensuite qu'on peut, avec un programmeur capable de générer les 12,75 volts nécessaires, programmer

un nouveau contenu en mémoire. Autant vous dire que, lorsqu'on tâtonne pour faire fonctionner un code, ces opérations sont relativement pénibles et consommatrices de temps. Même en ayant un stock respectable, il arrive un moment où l'on est à court de composants vierges et l'on passe alors de longues heures à les exposer aux UV.

Une bien meilleure option est de s'orienter vers un composant similaire mais effaçable électriquement : les 28C256. Majoritairement compatibles, 27C256 et 28C256 ne diffèrent au niveau du brochage que par l'inversion de deux pattes : la broche 1-VPP (tension de programmation) d'une 27C256 devient A14 (ligne d'adresse 14) sur une 28C256, et la broche 27 (A14) devient /WE (« Write Enable », activation écriture). Pour utiliser une 28C256 en lieu et place d'une 27C256, et donc bien plus facilement mettre au point ses codes, il faut trouver un moyen de croiser ces broches.

Autant en profiter pour faire d'une pierre deux coups et utiliser un support ZIF (*Zero Insertion Force*) permettant de sécuriser le composant en place à l'aide d'un petit levier (comme un processeur moderne) plutôt

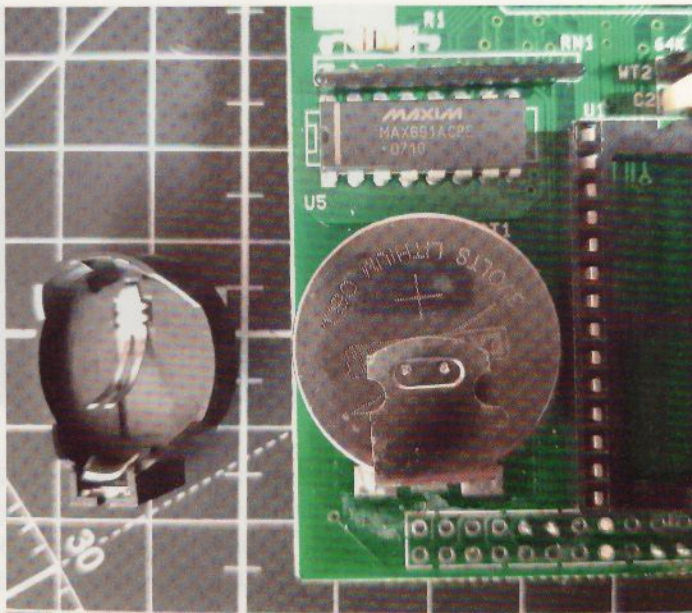
Au cœur de cette carte industrielle se trouve le Zilog Z180, grand frère du connu et très populaire Z80. Ce processeur intègre un certain nombre de périphériques, dont deux UART, un contrôleur DMA, une MMU et un gestionnaire d'interruptions. Il reste cependant parfaitement compatible au niveau du code et des binaires « dans une certaine mesure » avec le Z80.



que par friction dans le *socket*. L'idéal aurait été de pouvoir dessolder le support (*socket*) de l'EPROM pour souder le support ZIF à la place. Malheureusement, la proximité de la RAM, littéralement contre le support EPROM, ne laisse aucune option de ce style.

Ma solution est alors de simplement souder deux séries de connecteurs aux 2 fois 14 broches du support ZIF, à l'exception des broches 1 et 27, où sont ensuite soudés des fils de la broche 1 du ZIF à la 27 du connecteur, et de la broche 27 à la 1 du connecteur. Les deux « barrettes » de connecteurs pouvant être malencontreusement tordues lors de l'insertion dans le support présent sur le PCB, et surtout distendre les petites mâchoires métalliques qui le composent, un support DIP40 sacrificiel a été ajouté. Ceci apporte de la rigidité à l'ensemble et permet l'insertion sur le PCB sans avoir à forcer. Le résultat est visuellement horrible, mais cela reste très temporaire. Une fois un code digne de ce nom développé pour servir de moniteur, il n'y aura pas de problème pour le programmer dans une 27C256 qui sera, en plus, du plus bel effet.

Une fois la modification faite, il devient subitement possible de rapidement développer un code, l'inscrire en EEPROM et le tester. Bien entendu, un programmeur sera nécessaire, comme un TL866CS ou l'un des nouveaux T48 de XGecu (Windows uniquement). On peut également utiliser un montage maison, bien plus économique, avec un Arduino et deux 74HC595 (voir [2]). Ceci ne fonctionne qu'avec les 28C256, c'est lent, le code n'est pas ce qu'il y a de plus lisible et optimisé, mais c'est une solution viable qui fait le travail si l'on ne se mélange pas les pinceaux avec les connexions (ceci a été vérifié par votre dévoué serviteur).



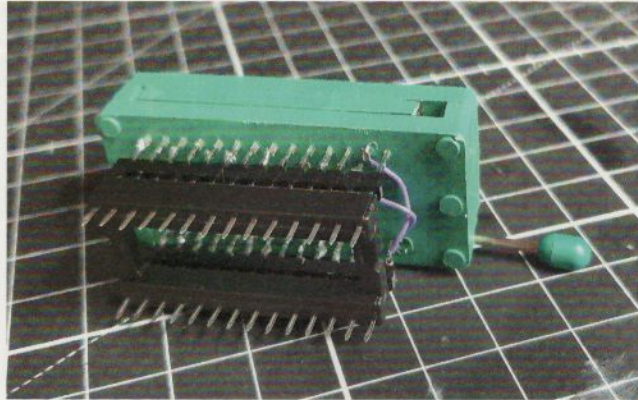
Une fois en mesure de faire s'exécuter un programme, il est temps de se pencher sur un premier essai d'une simplicité exemplaire. N'ayant pas d'information à ce stade sur l'utilisation de la RAM, nous nous contenterons d'une paire de sauts afin de vérifier que les instructions sont exécutées. Voici notre pathétique source en assembleur (oui, les **NOP** après **JP .INIT** ne servent à rien) :

```
org 0x0000

.INIT:  JP .PLOP
        NOP
        NOP
        NOP
.PLOP:  NOP
        NOP
        JP .INIT
        NOP
        NOP
        NOP
        NOP
        NOP
```

La carte est équipée d'une pile bouton au lithium 3 V CR2430 depuis longtemps épuisée. Contrairement à ce qu'on pourrait penser à première vue, le MAX691 juste au-dessus n'est pas une RTC, mais un gestionnaire d'alimentation capable de fournir un minimum de courant à la RAM en cas de problème, tout en signalant l'événement au processeur. Cette pile a été dessoudée et remplacée par un support pour CR2032 (à gauche).

Ce support ZIF a été sauvagement torturé de manière à permettre l'utilisation d'une EEPROM 28C256 en lieu et place de l'EPROM (UVPROM) 27C256 attendue par la carte. La seule différence physique entre ces deux mémoires se résume à l'inversion des pins 1 et 27.



Pour l'instant, aller à l'essentiel est suffisant et nous assemblerons cela avec *Z80asm* de Bas Wijnen [3], disponible pour toutes les distributions GNU/Linux et les différents *BSD. Obtenir un binaire se fera avec la commande `z80asm -o test.bin test.z80`, puis nous générerons une image de la bonne taille (32 Kio) pour l'EEPROM avec `dd if=test.bin of=test_ROM.bin bs=32k conv=sync`. Ceci pourra ensuite être inscrit dans la mémoire du composant et placé, via notre adaptateur, sur la carte. Le code assemblé est traduit en une série binaire de 16 valeurs : `c3 06 00 00 00 00 00 00 c3 00 00 00 00 00 00 00` et se trouvera donc dans les 16 premiers octets de l'espace d'adressage, sachant que les 5 **NOP** de fin ne seront jamais exécutés en raison du saut **JP** vers le label **.INIT**. Ceci signifie donc que surveiller les 4 bits de poids faible sur le bus d'adresse sera suffisant pour valider l'exécution.

Le connecteur 40 broches J1 nous permet de fournir l'alimentation +5 V sur 1 et la masse sur 2 (ou 3, 5, 8, 13, 21, 24, 25 et 37). Pour nous assurer que le code est exécuté, nous n'avons d'autre choix que d'utiliser un analyseur logique, comme un clone très peu cher (~5 €) de Salae Logic (puce Cypress FX2) et l'outil *open source* Sigrok/PulseView. Les différents signaux qui nous intéressent sont présents sur le connecteur J1 :

- les 8 premiers bits du bus d'adresse A0 à A8 (40, 38, 36, 34, 32, 30, 28, 26) ;
- /MREQ (23) noté **_M1** sur le schéma et passant à l'état bas lorsque le processeur souhaite accéder à la mémoire ;

- **_RD** (31) passant à l'état bas pour signifier une lecture en mémoire (par opposition à une écriture avec **_WR** sur 29) ;
- **_M1** (27) indiquant que le Z180 est dans le premier cycle machine correspondant à la récupération d'instruction (*opcode fetch*) ;
- et **CLK** (4) qui est le signal d'horloge sortant du Z180, généré en interne depuis le quartz qui est connecté au CPU (via XTAL et EXTAL).

Notez que les signaux dont le nom est précédé d'un trait de soulignement sur le schéma désignent l'utilisation d'une logique inversée. C'est quand la broche est à l'état bas (masse) que le signal est actif. Ceci est noté avec un « / » dans le manuel du Z180 et dans la plupart des autres documentations, même si l'on retrouve tantôt « _ » par ailleurs également. Dans la suite de l'article, nous utiliserons « / » indépendamment de la convention utilisée dans le schéma.

En branchant notre analyseur logique sur A0 à A3, /MREQ, /RD et CLK, nous pouvons alors observer les différentes adresses présentées successivement sur le bus par le processeur. Et

effectivement, nous voyons bien 0, 1, 2, 6, 7, 8, 9, A, 0, 1, 2, 6, 7, 8, 9, A, 0, 1, etc. Le code est bel et bien exécuté correctement et la carte fonctionne. C'est une excellente nouvelle, je n'ai pas perdu 23 €.

3. PARLER AU MONDE EXTÉRIEUR

À présent que nous savons que le Z180 est effectivement en mesure de lire et d'exécuter ce qui est stocké dans le 28C256, il est temps de nous attaquer à quelque chose de plus représentatif de cette exécution. Ce qui va suivre a été bien plus délicat et complexe que ce que je vais maintenant décrire. Je préfère cependant vous épargner les mésaventures et autres erreurs grossières pour aller directement à la solution fonctionnelle. Sachez cependant que se pencher sur un processeur et une carte inconnue implique des hauts et des bas. Si vous faites de même, ne vous attendez pas à ce que tout fonctionne du premier coup, bien au contraire. Mais c'est ainsi que l'on apprend...

Le Z180 intègre (presque) tout ce qu'il était nécessaire d'adjoindre sous la forme de composants

externes avec le Z80. Je pense, bien entendu, à l'utilisation d'une UART PC16550 ou autre, afin de pouvoir faire communiquer le système. Ici, cette fonctionnalité est déjà présente (en double) dans le processeur qui à peu de choses près (RAM et ROM) est, pour ainsi dire, un microcontrôleur. Et à l'instar d'UART « externes », comme une 16550, une Zilog Z8440, une Intel 8250 ou une Motorola MC6850, la configuration et l'utilisation se font via des ports d'entrée/sortie (IO Ports), permettant d'accéder aux registres du périphérique. En temps normal, c'est le concepteur du système qui détermine les adresses puisque la seule différence avec l'accès à la mémoire est l'utilisation du signal /IORQ (IO ReQuest), ou « _IOE » sur le schéma, en lieu et place de /MREQ (Memory REQuest). Dans le cas du Z180, les adresses sont prédéterminées au reset, car la fonctionnalité est physiquement « câblée » dans le silicium.

Ce TL866CS est un programmeur d'EPROM (entre autres) capable de supporter quelque 15 000 modèles de composants différents. Bien qu'il soit encore possible d'en trouver (tout comme son grand frère, le TL866-II), c'est un appareil en fin de vie, relativement rare et donc coûteux. Les nouveaux modèles du constructeur les T48 et T56 sont plus performants et moins chers, mais ils ne fonctionnent, pour l'instant, qu'avec le logiciel propriétaire du fabricant, sous Windows.



Pour le premier port série, ou plus exactement la première UART, « ASCIO » pour *Asynchronous Serial Communication Interface 0*, les registres sont les suivants :

- **CNTLA0** (port 0x00) : *CoNTroL Register A 0*, permet de configurer le type de communication (UART classique ou communication inter-Z180), l'activation de l'envoi et de la réception, ainsi que le format de données (taille des données, nombre de bits de stop, parité) ;
- **CNTLB0** (0x02) : *CoNTroL Register B 0*, configure le mode de communication inter-Z180 ainsi que la configuration des horloges et du diviseur pour paramétrer la vitesse de communication ;
- **STAT0** (0x04) : *STATus Register 0*, est le registre d'état permettant de connaître la disponibilité du *buffer* d'envoi et de réception, et l'existence d'erreur de communication ;
- **TDR0** (0x06) : *Transmit Data Register 0* est le *buffer* pour l'envoi de donnée ;
- **RDR0** (0x08) : *Receive Data Register 0*, idem pour la réception.

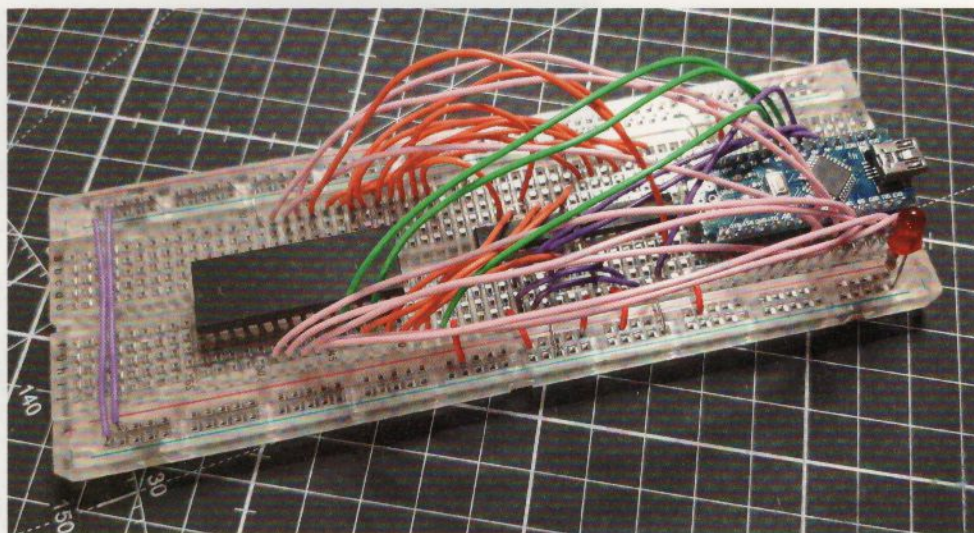
Notez qu'à la différence d'un 16550, nous avons ici deux registres/*buffers* pour la communication et non un seul qui est utilisé pour l'émission et la réception. Ceci signifie donc que pour envoyer des données, nous écrivons dans **TDR0** et pour obtenir ce qui arrive, nous lisons **RDR0**.

Mais avant de pouvoir faire cela, nous devons configurer l'UART pour la régler avec les paramètres souhaités pour la

communication : 9600 bps, 8 bits de données, deux bits de stop et un bit de parité impaire (9600 8O2). Pourquoi ne pas opter pour le classique 8N1 (8 bits, pas de parité et 1 bit de stop) ? Tout simplement parce que j'ai pris l'habitude, après de nombreux déboires dans mes expérimentations sur platine à essais, de mettre tous les atouts de mon côté et donc d'utiliser chaque mécanisme permettant de m'assurer de l'intégrité des données. Il s'avère que dans ce cas, ce n'était pas nécessaire, et que cela ne m'a pas empêché de faire une grossière erreur, par ailleurs.

Quoi qu'il en soit, votre bible en la matière doit être le « *Z8018x Family MPU User Manual* » (UM005004-0918) [4]. Absolument tout est dedans, des registres aux brochages en passant

Si vous ne comptez pas investir dans un programmeur d'EPROM, une alternative économique consiste à créer un montage utilisant seulement une carte Arduino (ici, une Nano pour la compacité) et deux registres à décalage 74HC595. Cela ne fonctionne qu'avec les 28C256 mais c'est suffisant.



par les *opcodes*, les caractéristiques électriques ou les différents modes de fonctionnement, même si les explications demandent parfois des compléments ou un brin d'expérimentation. C'est tout simplement la référence en quelque 300 pages seulement. Ceci pourra être complété du « *Z80180 Microprocessor Unit Product Specification* » (PS014005-0817) [5] pour une version résumée et plus compacte des caractéristiques et des registres.

Pour notre second code, et en préparation de l'utilisation du langage C par la suite, nous allons troquer *Z80asm* contre l'assembleur de SDCC [6], *sdasz80*. La syntaxe est légèrement différente de celle de *Z80asm* concernant les modes d'adressage et intègre quelques spécificités concernant les informations à destination de l'éditeur de liens, mais cela reste de l'assembleur tout ce qu'il y a de plus classique.

Notre objectif est simple : afficher à répétition une chaîne de caractères en sortie du port série ASCII. Pour cela, nous n'avons pas besoin de RAM ou d'une pile, tout se passera comme pour le code précédent, en ROM. Dans l'ordre, nous devons configurer le port, puis dans une boucle infinie :

1. Charger dans le registre HL l'adresse du début de notre chaîne de caractères.
2. Nous assurer que l'envoi est possible en lisant STAT et dans le cas contraire, attendre.
3. Accéder à la donnée et la charger dans le registre A.
4. Copier le caractère de A vers le port TDR0.
5. Incrémenter HL pour pointer sur le caractère suivant.
6. Revenir à 2 pour ce nouveau caractère.
7. Et ce, jusqu'à ce que l'octet chargé depuis la mémoire soit **0x00** et là, revenir à 1 pour boucler sans fin.

La partie délicate n'est pas le code assembleur, mais la configuration des registres **CNTLA0** et **CNTLB0**, et en particulier le réglage de la vitesse de communication. Voyons d'abord l'utilité de chaque bit des deux registres. Pour **CNTLA0** (page 125 du manuel), nous avons :

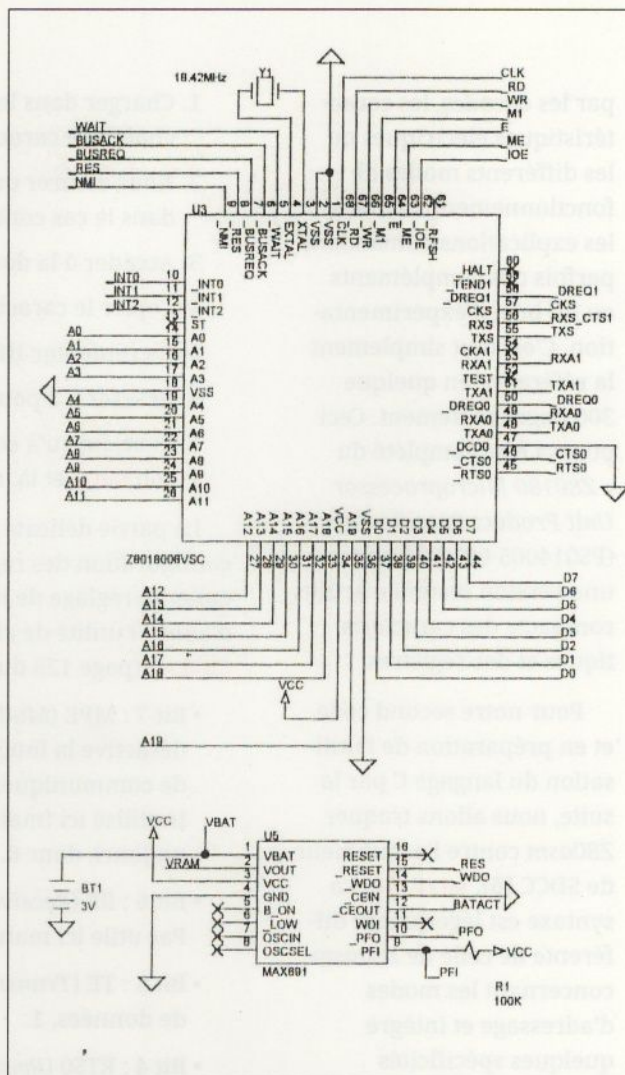
- Bit 7 : MPE (*Multi-processor Mode Enable*) active ou désactive la fonctionnalité permettant à plusieurs Z180 de communiquer entre eux sur un même bus série. Inutilisé ici (mais définitivement quelque chose à tester un jour), donc **0**.
- Bit 6 : RE (*Receiver Enable*) active ou non la réception. Pas utile ici mais activons tout même, **1**.
- Bit 5 : TE (*Transmitter Enable*) active ou non l'émission de données, **1**.
- Bit 4 : RTS0 (*Request to Send Channel 0*) est un contrôle de la ligne RTS pour le contrôle de flux matériel, inutile ici, **0**.
- Bit 3 : MPBR/EFR (*Multi-Processor Bit Receive/Error Flag Reset*) pour la gestion d'erreur en mode multiprocesseur, inutilisé ici, **0**.
- Bit 2 à 0 : MOD2-MOD0 (*ASCII Data Format Mode*) configure le format de la communication série avec MOD2 pour le nombre de bits de données (0=7, 1=8), MOD1 pour activer ou non la parité (0=non, 1=oui) et MOD0 pour le nombre de bits de stop (0=1, 1=2). Nous voulons 8O2, ce sera donc **111**. Notez que le manuel présente un tableau résumant tous les modes en page 131.

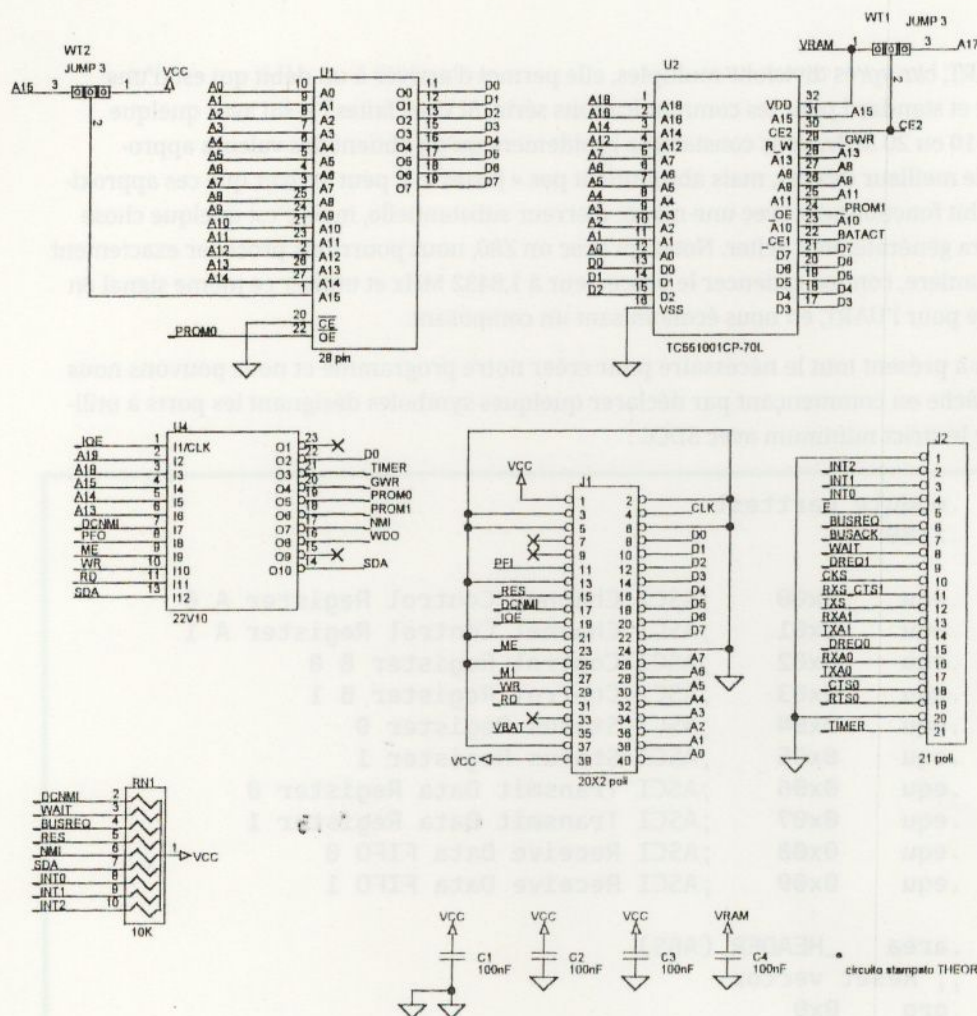
Nous avons ainsi notre octet à placer dans **CNTLA0**, c'est **01100111** en binaire, soit **0x67** en hexadécimal. Passons donc au second registre, **CNTLB0** (page 132 du manuel) :

- Bit 7 : MPBT (*Multi-Processor Bit Transmit*) configure l'état du bit supplémentaire en mode multiprocesseur. C'est inutile ici, donc 0.
- Bit 6 : MP (*Multiprocessor Mode*) configure le formatage de données pour la communication interprocesseur. Inutile ici, donc 0 (et, oui, le manuel utilise tantôt « *Multiprocessor* » et tantôt « *Multi-processor* »).
- Bit 5 : CTS/PS (*Clear to Send/ Prescale*) est le premier bit impactant la vitesse de communication et, en plus, il a deux usages. Lu, il donne l'état du signal CTS pour le contrôle de flux, mais écrit, il détermine le facteur de prédivison (*prescaler*). S'il est à 0, l'horloge système est divisée par 10, s'il est à 1, ce sera par 30.
- Bit 4 : PEO (*Parity Even Odd*) règle la parité si elle est activée dans **CNTLA0**. Si ce bit est à 0, la parité est paire, s'il est à 1, la parité est impaire. Nous avons donc ici 1.
- Bit 3 : DR (*Divide Ratio*) est le bit définissant le diviseur de fréquence permettant d'obtenir le débit en baud depuis l'horloge prédivisée (réglée par le *prescaler*). Si ce bit est à 0, une division par 16 est appliquée, s'il est à 1, la division se fait par 64.
- Bit 2 : SS2-SS0 (*Source/Speed Select*) permet de choisir la source d'horloge et la division finale de fréquence. Ces trois bits déterminent une division par 1 (000), 2 (001), 4 (010), 8 (011), 16 (100), 32 (101), 64 (110), ou l'utilisation d'un signal d'horloge externe différent de l'horloge système (111).

Pour arriver à obtenir nos 9600 bps, nous devons donc jouer sur le bit CTS/PS (diviser par 10 ou 30), le bit DR (diviser par 16 ou 64) et les bits SS2 à SS0. Ceci afin de partir de la fréquence de l'horloge système pour arriver à 9600 Hz. Mais quelle est la

fréquence système ? La pire réponse que vous puissiez donner et celle que j'ai utilisée initialement me faisant perdre un temps phénoménal est : « celle du quartz ». Or, la documentation précise en page 7, à propos de la broche CLOCK (68 en PLCC) : « *System Clock (Output, Active High). The output is used as a reference clock for the MPU and the external system. The frequency of this output is equal to one-half that*





Voici le schéma de la carte tel que fourni par le vendeur eBay (initialement en version papier, puis PDF par mail après demande). Attention, ne prenez pas ceci pour argent comptant (cf. le second volet de cette exploration, spécial « RAM », dans le futur N°50)...

of the crystal or input clock frequency. ». Autrement dit, l'horloge système est cadencée à la moitié de la fréquence du quartz.

Sur notre carte, nous avons un quartz de 18,432 MHz et l'horloge système du Z180 tourne donc à 9,216 MHz, soit 9216000 Hz. C'est donc cette valeur qui doit servir aux calculs et aux divisions successives. Notez au passage que cette

fréquence est divisible par un exposant de 2 (16) et ce n'est pas un hasard. Pour procéder aux calculs et trouver les bonnes valeurs pour obtenir finalement 9600 bps, le plus simple est de faire appel à une feuille de calcul dans un tableur. Là, on entre quelques formules et sur la base de la fréquence de base, on obtient instantanément toutes les vitesses possibles en fonction de toutes les

combinaisons de CTS/PS, DR et SS2-SS0. Immédiatement, on se rend compte qu'une seule est possible dans notre cas : $9216000 / 30 / 16 / 2 = 9600$. Il ne nous reste plus alors qu'à définir les bits, soit respectivement 1, 0, et 001, et nous obtenons l'octet final **00110001** pour **CNTLB0**, soit 0x31 en hexadécimal.

La fréquence 18,432 MHz est une valeur que l'on retrouve parfois pour les

quartz des UART, car après divisions multiples, elle permet d'arriver à un débit qui est d'une valeur précise et standard pour les communications série. Si vous faites l'essai avec quelque chose comme 10 ou 20 MHz, vous constaterez rapidement qu'on obtient des valeurs approchantes dans le meilleur des cas, mais absolument pas « justes ». Il peut arriver que ces approximations de débit fonctionnent avec une marge d'erreur substantielle, mais c'est quelque chose qu'on préférera généralement éviter. Notez qu'avec un Z80, nous pourrions procéder exactement de la même manière, comme cadencer le processeur à 1,8432 MHz et utiliser ce même signal en guise d'horloge pour l'UART, en nous économisant un composant.

Nous avons à présent tout le nécessaire pour créer notre programme et nous pouvons nous attacher à la tâche en commençant par déclarer quelques symboles désignant les ports à utiliser et intégrer le strict minimum avec SDCC :

```
.module uarttest
.z180

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

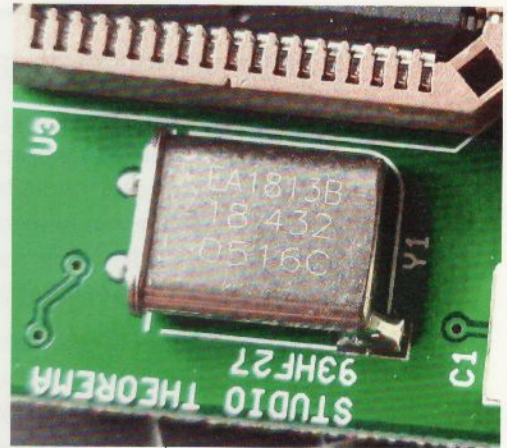
.area    _HEADER (ABS)
;; Reset vector
.org     0x0
jp      init
.org     0x100
init:
; Charge le Stack Pointer
; appel gsinit (Initialise global variables)
; appel main
; Ordre des segments pour le linker.
.area    _HOME
.area    _CODE
.area    _INITIALIZER
.area    _GSINIT
.area    _GSFINAL
.area    _DATA
.area    _INITIALIZED
.area    _BSEG
.area    _BSS
.area    _HEAP
.area    _CODE
```


Plusieurs éléments sont notables ici avec tout d'abord `.z180` nous permettant de signifier à l'assembleur que nous travaillons avec un Z180 et non un Z80. Ceci nous donne accès à quelques instructions supplémentaires n'existant que pour ce processeur (comme `out0`) qui n'existent pas avec le Z80. Autre point intéressant, le Z180, comme le Z80, possède une table de vecteurs de *reset*. Dans le code précédent, nous avons fait l'impasse sur ce point, mais avec SDCC, nous faisons les choses correctement. La première instruction utilisée sera donc un saut vers le début du code placé à l'adresse `0x0100` (`.org 0x100`). Là, théoriquement nous devrions initialiser le registre SP, le pointeur de la pile et appeler une routine permettant d'initialiser les variables globales. Ceci n'est un pré-requis que pour l'utilisation du langage C (SDCC est avant tout un compilateur C) et n'avons donc rien à faire ici, mais nous conservons tout de même une structure identique.

Autre point lié aux langages de haut niveau (oui, à cette époque et avec cette technologie, le C est un langage de haut niveau), SDCC a besoin de connaître l'organisation mémoire et plus précisément l'ordre des segments pour l'éditeur de liens. Encore une fois, en l'absence de C, ceci n'est pas utile, mais néanmoins important pour plus tard. Notez que nous avons deux fois `_CODE`. La première occurrence est là pour l'ordre des segments et la seconde pour signifier que ce qui va suivre doit aller dans la zone en question. Et ce qui suit est l'initialisation de l'ASCII :

```
ascii0init:
    ld    a, #0x67
    out0  (CNTLA0), a
    ld    a, #0x31
    out0  (CNTLB0), a
```

Ceci n'est rien de plus que ce dont nous venons de parler concernant ces deux registres. À noter que les *littéraux* (les éléments qui représentent directement une valeur, ici `0x67` est littéralement `0x67`) sont précédés d'un `#` et non de `$` comme c'est le cas avec l'assembleur GNU, par exemple. Autre point, nous utilisons `out0` (et plus tard `in0`) et non `out` qui existe également. Cette instruction a été spécifiquement ajoutée dans le Z180 (l'une des 12 spécifiques au Z180, voir page 173 du manuel) pour gérer plus facilement les périphériques intégrés, car les adresses des registres E/S sont en 16 bits. Pour accéder aux registres, les 8 bits de poids fort doivent être à 0 alors que l'instruction `out` place le contenu d'un registre (ici A) précisément à cet endroit. `out0` fait donc la même chose que `out`, mais force les 8 bits de poids fort à 0. Une autre technique, qu'on voit parfois pour le Z180, consiste à charger l'adresse du port en 16 bits dans BC et de faire `out (c), a`, mais comme nous avons basculé de `Z80asm` à `sdasz80` et spécifié `.z180`, nous n'avons pas besoin de cette astuce.



Ce quartz de 18,432 MHz, directement connecté au processeur, pourrait laisser entendre que l'horloge système est à cette fréquence. C'est une grossière erreur, car en réalité la fréquence est divisée par deux et oublier cet état de fait (parce qu'on ne prend pas le temps de lire la documentation) est source de nombreux problèmes lorsqu'on en vient à vouloir utiliser les communications séries.

PORTS INTERNES ET EXTERNES

La présence de différents périphériques dans le Z180 ne vous dispense pas d'en utiliser d'autres, externes. Le processeur ne dispose par exemple pas de GPIO et il pourra être souhaitable de lui adjoindre un Intel 8255. Les ports internes peuvent alors poser problème puisque, non seulement il s'agit de 64 adresses utilisées de 0x0000 à 0x003f (dans cet espace d'adressage), mais ces adresses seront également systématiquement présentées sur le bus, que le registre soit interne ou non.

Pour éviter des conflits et/ou devoir gérer cela avec une logique externe complexe, il est possible de déplacer la plage d'adresses par défaut, sur un multiple de 64, n'importe où entre 0x0000 et 0x00ff. Il suffit pour cela de jouer sur les bits 7 et 6 du registre ICR (*I/O Control Register*) pour choisir une autre plage : 0x0040 à 0x007f (01), 0x0080 à 0x00bf (10) ou 0x00c0 à 0x00ff (11).

Pour plus d'information, consultez la section « *Internal I/O Registers* » du manuel, page 10. Gardez à l'esprit que l'adresse du registre ICR change également en déplaçant la plage.

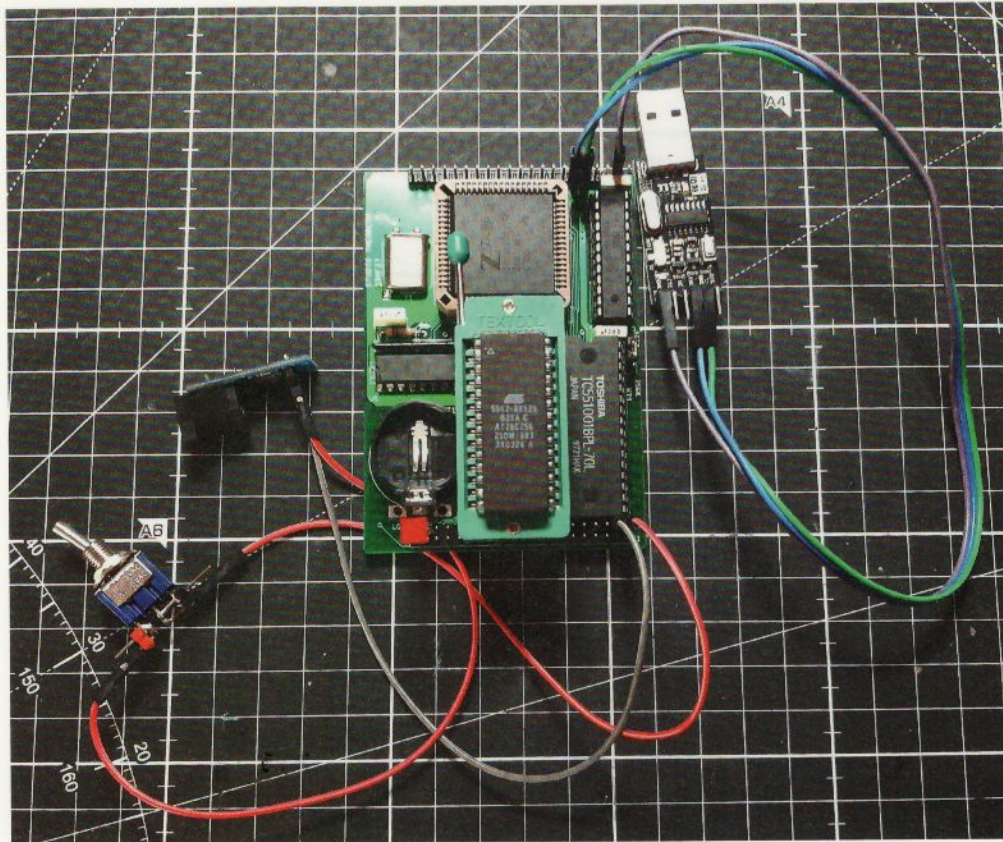
À ce stade, l'ASCII0 est prêt à être utilisé et nous pouvons nous lancer dans l'envoi des données :

```
main:
    ld      hl,#text      ; charger l'adresse du texte dans HL
ascii0putc:
    in0     a,(STAT0)     ; obtenir l'état
    bit     1,a           ; tester le bit TDRE de STAT0
    jr      z,ascii0putc  ; pas prêt
    ld      a,(hl)        ; charger le caractère
    inc     a             ; incrémenter A, puis...
    dec     a             ; décrémente A
    jp      z,main        ; c'était un \0 ?
    out0    (TDR0),a      ; non, envoyer caractère
    inc     hl            ; pointeur++
```

Habituellement, pour ce genre de test, on précise la taille des données à envoyer et on boucle sur cette quantité. Personnellement, je trouve plus simple d'ajouter un `\0` à la fin de la chaîne, comme en C, et tout simplement envoyer chaque caractère jusqu'à tomber sur ce marqueur de fin de chaîne. Ainsi, pour modifier le message affiché, il suffit de changer le texte dans le code source assembleur, sans avoir à ajuster une valeur en plus.

Notez l'astuce permettant de détecter ce marqueur. Les instructions `inc` et `dec` changent le drapeau « zéro » du registre d'état si la valeur obtenue est nulle après l'opération. Pour détecter un zéro, il suffit donc d'incrémenter puis de décrémente et de tester ce bit/drapeau.

Ce code se suffirait à lui-même en ajoutant un simple `jp ascii0putc` (une fois la chaîne spécifiée sous le label `text`), mais je n'aime pas particulièrement les avalanches rapides de caractères



La carte, après nombre d'expérimentations est sensiblement différente : support CR2032, connecteur sur le dessus, adaptateur pour l'EEPROM, alimentation +5 V avec interrupteur et liaison série via convertisseur USB. Ceci demande du travail, mais au final, on obtient une plateforme 8 bits furieusement intéressante et bien plus capable qu'un simple système à base de Z80.

dans mon terminal série. Ceci ne permet pas de voir ce qui se passe, car c'est trop rapide. Nous ajoutons donc quelques lignes supplémentaires pour induire un délai après l'envoi de chaque caractère :

```
delay:
    ld    de,#0x2000 ; initialise le compteur
Inner:
    dec   de          ; décrémente DE
    ld    a,d          ; copie D dans A
    or    e            ; A = D | E
    jp    nz,Inner     ; boucle si A != 0

    jp    asciiputc    ; boucle sur l'envoi
```

C'est une boucle de temporisation assez classique, avec une autre façon de tester si le contenu d'un registre est nul. Ici, nous avons une valeur sur 16 bits, mais ceci fonctionnerait également dans le cas de A (8 bits). Les opérations logiques, tout comme **inc** et **dec**, impactent le registre d'état et son bit/drapeau « zéro » et faire un OU logique entre les deux moitiés de DE nous donne le même résultat. Une fois les 0x2000 tours faits dans cette boucle, nous revenons à **asciiputc** : pour traiter le caractère suivant. Un caractère qui provient de la chaîne que nous n'avons plus qu'à spécifier pour conclure ce programme :


```
text:
    .ascii "Bonjour Monde !\r\n"
    .db    0x00
```

Pour produire notre binaire, nous allons créer un **Makefile**. Ceci simplifiera grandement les opérations d'assemblage, d'édition de liens, de génération du binaire pour l'EPROM et de programmation :

```
# Définition de quelques variables
ROMSIZE := 0x8000
TARGET  := uarttest
CC       := sdcc
AS       := sdasz80
CFLAGS   :=
LFLAGS   := -Wl-u
MFLAGS   := -mz180 --code-loc 0x0100 --data-loc 0x8000
ROMTYPE  := AT28C256

all: ${TARGET}.bin ${TARGET}_padded.bin

# Assemblage
${TARGET}.rel: ${TARGET}.z180
    $(AS) -plogffw ${TARGET}.rel ${TARGET}.z180

# Édition de liens
${TARGET}.ihx: ${TARGET}.rel
    $(CC) $(MFLAGS) $(LFLAGS) --no-std-crt0 \
    -o ${TARGET}.ihx ${TARGET}.rel

# Conversion de l'Intel Hex en binaire
${TARGET}.bin: ${TARGET}.ihx
    sdojcopy -Iihex -Obinary ${TARGET}.ihx ${TARGET}.bin

# Ajustement de la taille
${TARGET}_padded.bin: ${TARGET}.ihx
    sdojcopy -Iihex -Obinary --gap-fill 0x00 --pad-to \
    $(ROMSIZE) ${TARGET}.ihx ${TARGET}_padded.bin

# Programmation EPROM
prog: ${TARGET}_padded.bin
    minipro -p ${ROMTYPE} -w ${TARGET}_padded.bin

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


Une fois le binaire produit et programmé, nous pouvons constater, côté Z180, que tout fonctionne à merveille en connectant simplement un convertisseur USB/série aux broches 16 et 17 de J2. Cette mystérieuse carte industrielle est maintenant en mesure de communiquer avec nous. Victoire !

4. C'EST TOUT POUR LE MOMENT

Le fait de pouvoir obtenir une sortie série du Z180 ouvre des possibilités intéressantes, en plus d'être une preuve que nous commençons à avoir la main sur le fonctionnement de ce processeur qui, plus on creuse, plus s'avère à la fois proche et très différent du Z80. Non abordées ici, mes expérimentations sur le fait de déplacer la plage d'adresse des ports E/S sont un bon exemple, tout comme le fait de pouvoir désactiver le rafraîchissement de la DRAM en changeant un simple bit dans le registre RCR (0x36).

La communication série nous servira, la prochaine fois, à régler le problème d'accès à la RAM qui n'est largement pas aussi simple

qu'avec un Z80. Non seulement le Z180 intègre une MMU séparant l'espace d'adressage logique (sur 16 bits) de l'espace d'adressage physique (sur 20 bits) et celle-ci ne peut être désactivée, mais nous avons clairement un mécanisme obscur en place. En effet, la carte utilise un circuit logique programmable qui vient remplacer tout un lot de composants, et nous n'avons pas la moindre idée de ce que fait effectivement cet élément. Il sera donc nécessaire de tester son fonctionnement avec, pour objectif, de trouver quel signal active effectivement les broches /OE (*Output Enable*) de la RAM et de la ROM, et le /WE (*Write Enable*) de la RAM.

Mes expérimentations et notes prises en conséquence sont déjà bien avancées. Pour tout dire, le problème est déjà réglé et un code C fonctionnel ne saurait tarder à être testé. En attendant la conclusion de ce petit *cliffhanger*, et si vous vous intéressez à ces technologies 8 bits du passé, je ne saurais que trop vous conseiller d'acquérir l'une de ces cartes auprès du vendeur mentionné en début d'article. Je ne sais pas ce que *several* signifie dans l'esprit de Stefano (il n'a jamais mentionné de quantités précises en stock), mais il y a fort à parier que nombre de ces cartes sont toujours disponibles. Malgré les frais de port conséquents, il s'agit, je trouve, d'une alternative vraiment intéressante financièrement à l'achat des composants séparément et à la construction sur platine à essais. **DB**

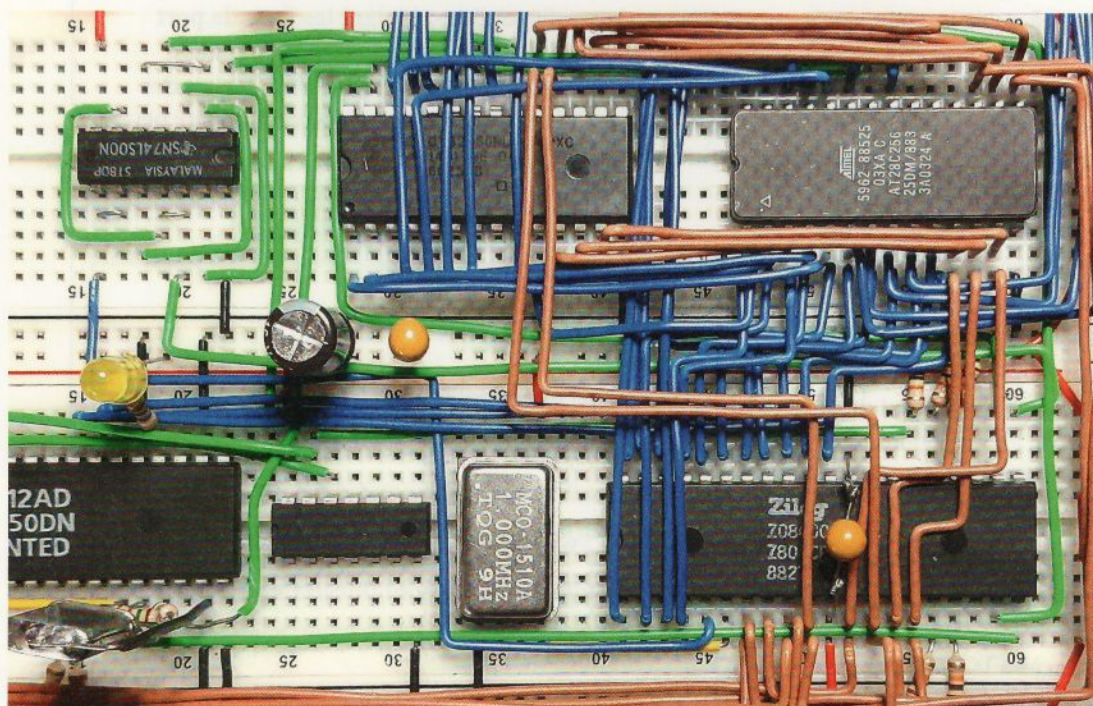
RÉFÉRENCES

- [1] <https://www.ebay.fr/usr/raeltubes>
- [2] <https://github.com/ppravatto/>
- [3] <https://www.nongnu.org/z80asm/>
- [4] <http://www.zilog.com/docs/z180/um0050.pdf>
- [5] <https://www.zilog.com/docs/z180/ps0140.pdf>
- [6] <https://sdcc.sourceforge.net/>

ZILOG Z80 : LE RETOUR DU FILS DE LA VENGEANCE !

Denis BODOR

Par le passé [1] [2] [3] [4] [5] [6] [7], et étalé du numéro 20 au 28, nous avons exploré la possibilité de construire son ordinateur 8 bits sur platine à essais avec l'assistance d'une carte Arduino pour émuler nombre de « périphériques » indispensables au système. Après plusieurs étapes décisives et très intéressantes cependant, le projet n'a mené nulle part pour diverses raisons. Il est temps aujourd'hui de nous remettre en selle pour prendre une direction différente et bien plus « traditionnelle ».

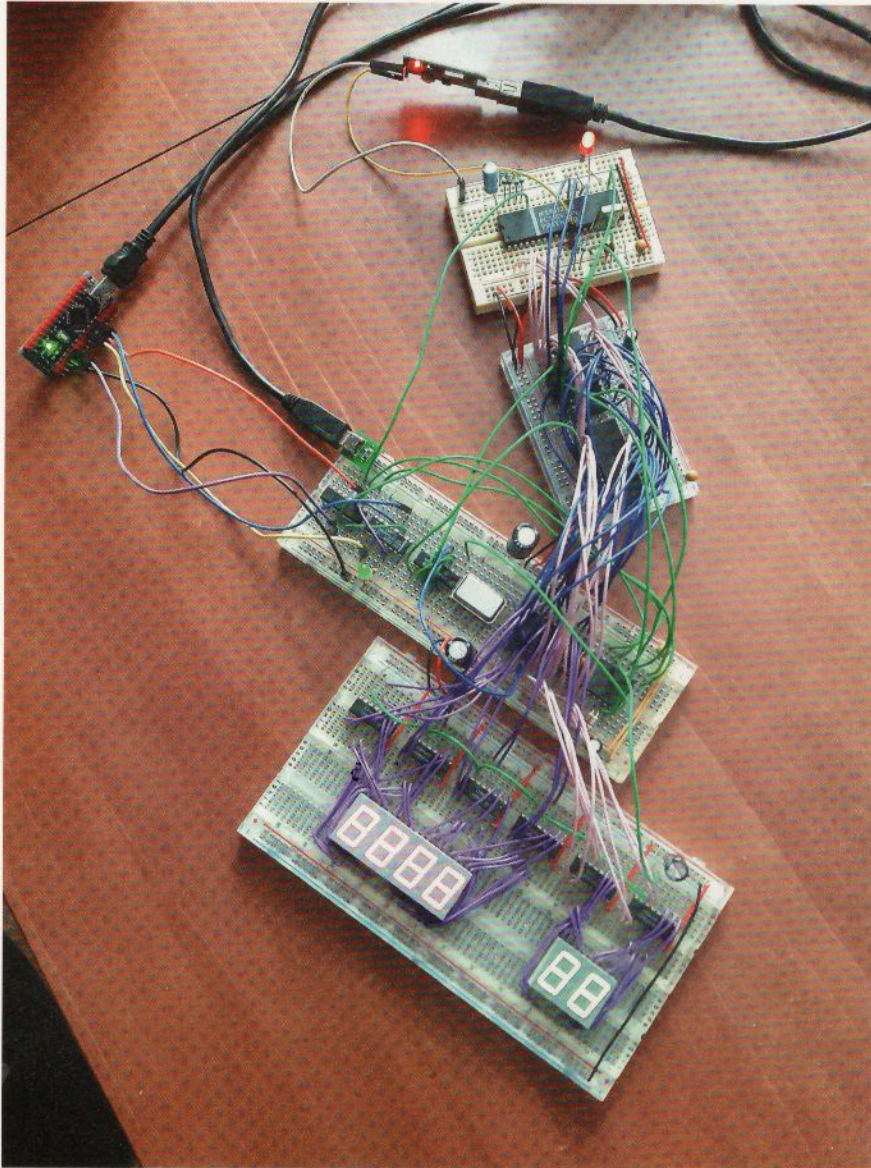


Le dernier article sur le sujet date du tout début 2019 et introduisait un circuit à base de compteur 74HCT193 et d'un sextuple inverseur 74LS04 (devenu entre temps un 74LS14, la même chose avec bascules à seuil (*triggers* de Schmitt) permettant de physiquement contrôler l'exécution pas-à-pas d'un code exécuté par le processeur, mais toujours à partir d'une carte Arduino. Je ne dirai pas que c'est là la raison de l'arrêt des articles, mais c'est une part importante de la réalisation que, quelque chose n'allait pas dans les fondations mêmes de ce projet. Oui, une carte à microcontrôleur est un avantage, mais on se retrouve finalement « entre deux mondes » avec des problématiques soulevées des deux côtés. Ce à quoi s'ajoute, forcément, une petite voix en arrière-plan qui ne cesse de dire « mais pourquoi diable vouloir contrôler un processeur avec un microcontrôleur qui en intègre déjà un ? ». C'est quelque chose qu'on retrouve par ailleurs [8] et j'avoue qu'instinctivement, je me dis immédiatement que « *ce n'est pas le vrai truc* ».

En réalité, il semblerait que la bonne approche consiste à d'abord construire le système, physiquement, traditionnellement et sans partie émulée, et seulement ensuite, d'éventuellement lui adjoindre un petit circuit de contrôle et une interface permettant à une carte microcontrôleur (Arduino, ESP32, Raspberry Pi Pico, Nucleo STM32 de STMicroelectronics, etc.) de simplement servir d'outil de mise au point. C'est ce que nombre de personnes avant nous ont fait et c'est ce que j'ai donc fait ces dernières semaines.

Notez que ce genre de réalisation ne cesse de gagner en popularité (toute relative) ces dernières années. À l'instar des ordinateurs *homebrew* (« faits maison ») des années 70/80 alors qu'il n'existait pas ou peu d'ordinateur susceptible d'être vendu aux passionnés et amateurs éclairés, nous avons aujourd'hui le *retrobrew*. Même principe, même matériel, mais nouveaux outils et moyens de se documenter et de partager son expérience. L'objectif n'est plus le même, car à l'époque, c'était la façon la plus économique (voire la seule durant une certaine période) d'avoir un ordinateur chez soi, alors qu'aujourd'hui, c'est un peu l'inverse : il y a des ordinateurs, voire des supercalculateurs partout, y compris dans nos poches, mais en maîtriser la conception au composant près est extrêmement difficile. À l'exception, peut-être, de petits microcontrôleurs, et encore, là aussi tout devient plus complexe, plus dense, plus nébuleux (soyons honnêtes, le temps des ATmega8 et des PIC16F84 est révolu).

Créer un système, quel que soit le processeur utilisé (Z80, Z180, 6502, 6809, 68008, 8088, etc.), vous coûtera **forcément** plus cher, et sera **forcément** plus difficile que de simplement acheter un clone Arduino, une *BluePill* (STM32) ou l'un des nombreux *devkits* ESP32/ESP8266. Cependant, l'idée n'est justement **pas** la facilité, mais d'avoir « son système à soi », réellement « fait maison », pour apprendre, comprendre et gagner en expérience à propos des fondations même de l'informatique. Voilà précisément la raison d'être du *retrobrew* et cet article doit être interprété dans ce sens : comme une source d'inspiration pour votre propre réalisation, totalement personnelle.



1. CAHIER DES CHARGES ET MATÉRIEL NÉCESSAIRE

Avant toute chose, il ne sera pas, en principe, nécessaire d'avoir lu les précédents articles pour explorer ce que nous allons voir ici. Je ne m'attarderai cependant pas excessivement sur les prémices du circuit, comme la classique utilisation de résistances et de LED pour simuler une mémoire remplie d'un unique *opcode* NOP (*No Operation*). Nous allons aller à l'essentiel, qui consiste à avoir un circuit relativement simple mais fonctionnel par lui-même.

Parmi les éléments toujours présents, nous avons le processeur, qui reste un Zilog Z80 et un UART (*Universal Asynchronous Receiver/Transmitter*) PC16550D. L'Arduino disparaît, tout comme le circuit d'exécution pas-à-pas, même s'il figure sur le schéma accompagnant le présent article. Il peut être

Heu... Alors, ce n'est pas vraiment complètement intuitif visuellement, mais ceci est un ordinateur 8 bits animé par un Zilog Z80, disposant de 32 Kio de ROM, 32 Kio de RAM et d'une interface série 9600 bps. À la charge du département marketing de faire croire au monde entier qu'il s'agit de la prochaine révolution IT !

ajouté si nécessaire, mais ne fait pas réellement partie du projet (pour l'instant). Pas plus que le sextuple afficheur 7 segments pilotés par des Motorola MC14495 (cf. photo), permettant de voir l'adresse et les données présentes à un instant *t* sur les deux bus. C'est très amusant au début, mais dès lors qu'il n'y a plus d'exécution pas-à-pas ou que le Z80 n'est plus cadencé à une vitesse d'escargot sous laudanum, l'intérêt autre que décoratif est drastiquement réduit.

Et justement, en l'absence de carte microcontrôleur et de *feedback* visuel, il nous faut une manière de dialoguer avec le système pour voir s'il fonctionne. L'UART est donc indispensable et nous fournira une liaison série avec un PC, via un adaptateur USB/série. Ici, c'est un PC16550D, mais n'importe quel composant similaire pourra faire l'affaire, à commencer par le Z8440 de Zilog (alias Z80-SIO), le W65C51 de WDC (alias ACIA), les deux étant toujours vendus chez Mouser, en version DIP40 et DIP28, ou encore le 8250 d'Intel ou le MC6850 de Motorola. La connexion sera très similaire entre ces différents composants, mais la manière de les configurer et de les utiliser sera différente et nécessitera donc quelques recherches de votre côté. L'objectif ici, et c'était sans doute une autre erreur de la série précédente, n'est pas de vous fournir une configuration clé en main, mais d'explorer le sujet, buter sur des problèmes et les régler. Si vous souhaitez quelque chose de plus « fini » pour vous intéresser principalement au code, un choix plus pertinent sera d'opter pour un kit tout fait, comme le bien connu RC2014 [9], le récent et très actif *Zeal 8-bit Computer* [10] ou n'importe lequel de ces

kits disponibles de-ci de-là sur le Net. Il est même possible de trouver des SBC industriels à base de Z80 ou de Z180 (le grand frère sur Z80 intégrant un certain nombre de périphériques) sur eBay pour une poignée d'euros, mais demandant généralement un certain travail d'investigation, voire de rétro-ingénierie.

Un ordinateur 8 bits, dans sa plus simple expression, se compose de peu d'éléments. À vrai dire, si on cherche quelque chose de drastiquement réduit, un processeur, une ROM (UVPRM, EPROM, EEPROM), un oscillateur à quartz et une poignée de composants passifs (condensateurs et résistances) sont tout ce qu'il faut. Mais pour pouvoir expérimenter et avoir davantage de liberté, l'UART et surtout une RAM s'avèrent rapidement nécessaires. Commençons donc par faire le tour de ce que vous devrez avoir sous la main pour créer un système identique, similaire ou proche du mien.

1.1 Les composants

La pièce maîtresse du système est, bien entendu, le processeur Zilog Z80. C'est un processeur 8 bits, toujours utilisé aujourd'hui (mais il se fait rare après presque 5 décennies de bons et loyaux services), dont les premières versions datent de 1976. Il est l'élément central de bien des ordinateurs des années 80, dont le TRS-80, les ZX80, ZX81 et ZX Spectrum, les Amstrad CPC (464, 6128, etc.) ou encore toute la série de machines reposant sur le standard MSX. Le Z80 est une extension et une évolution compatible logicielle de l'Intel 8080 et a été produit par Zilog, mais également, sous licence, par un grand nombre de fabricants (Nec, Sharp, Toshiba, Hitachi, SGS-Thomson (maintenant STMicroelectronics), etc.). C'est l'un des deux processeurs favoris des amateurs de construction d'ordinateurs 8 bits, l'autre étant bien entendu le MOS 6502 (Apple II, NES, Atari 2600, Commodore 64, etc.). En de rares occasions, on voit tantôt également de tels systèmes construits autour d'Intel 8088 ou de Motorola 6809 (parfois de 680×0). Trouver un Z80 sur eBay n'est pas difficile et les prix s'étalent de 4 € à 15 €, en fonction de la provenance. Notez que le Z80 est toujours produit, y compris en boîtier DIP40, le format dont nous avons besoin, et qu'une version à 6 ou 10 MHz (max) pourra être achetée, neuve, pour une dizaine d'euros chez des revendeurs comme Mouser, RS ou encore Digi-Key.

Pour cadencer le processeur, nous aurons besoin d'une source d'horloge qui, cette fois, ne sera donc pas une carte Arduino. La façon la plus simple de fournir ce signal au processeur est d'utiliser un oscillateur à quartz. Attention, on ne parle pas ici d'un quartz seul (comme on en aura l'usage pour l'UART), mais d'un composant intégrant effectivement un quartz, fournissant directement un signal carré. Ceux-ci sont facilement reconnaissables puisqu'ils se présentent généralement sous la forme d'un petit boîtier métallique à 4 pattes, et non 2 comme le quartz qu'on trouve sur les premières générations de cartes Arduino, par exemple, ou sur les modules RTC. La fréquence de cet oscillateur est importante, car même s'il est possible, en fonction du modèle de Z80, de monter jusqu'à 8 MHz, un assemblage sur platine à essais implique de calmer ses ardeurs et de rester raisonnable. 1 MHz est généralement un bon choix dans ces conditions et 4 MHz pourra être envisagé avec des platines de très bonne qualité ou, plus sérieusement, dans le cas d'un montage sur plaque pastillée (*perfboard*), impliquant donc de la soudure. Quoi qu'il en soit, rien ne vous empêche d'opter tout d'abord pour 1 MHz et de remplacer ce composant par la suite, voire de permuter au besoin de l'un à l'autre (attention tout de même si votre code implique des temporisations et des boucles pour induire des délais).

Parlant de platines à essais et de fréquences d'horloge, je ne saurais que trop vous recommander d'opter pour des produits de qualité, via des distributeurs réputés (Mouser, RS, Farnell, etc.) et non des modèles bas de gamme chinois (eBay, AliExpress, etc.). Il ne s'agit pas simplement de qualité des matériaux et d'usure, mais également d'intégrité des signaux, en particulier lorsqu'on monte en fréquence. Les produits bas de gamme, certes peu chers,

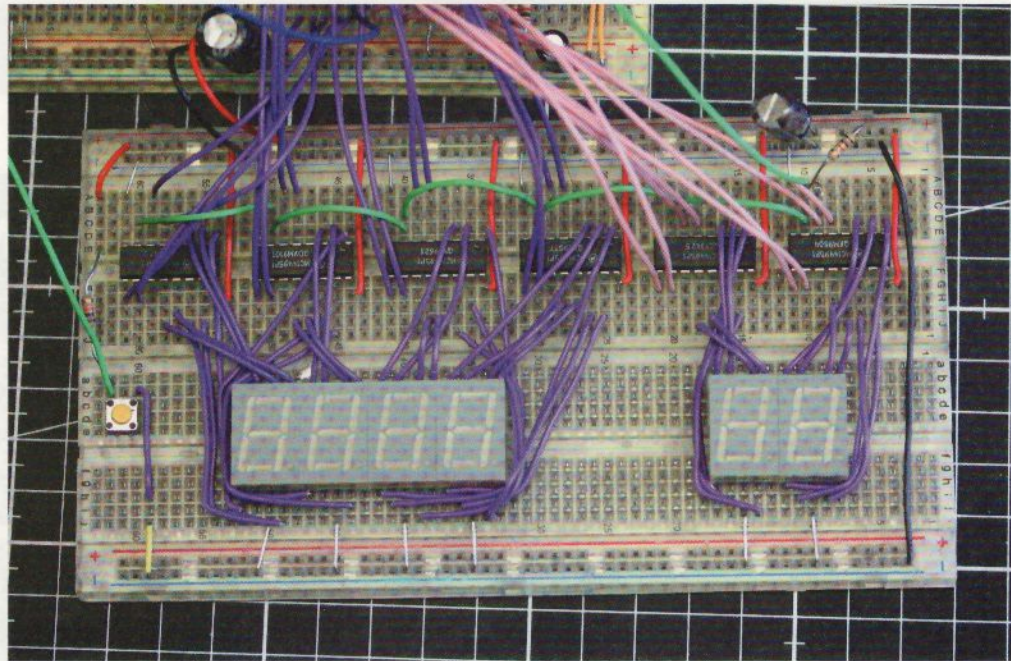
Choisir un oscillateur avec une jolie fréquence bien ronde (1 MHz, 2 MHz, 8 MHz, etc.) semble logique. Ceci facilite en effet les calculs liés aux cycles processeur et aux temps d'exécution des instructions. Mais il existe également des oscillateurs avec des fréquences plus « exotiques », car adaptées à différents usages, qu'il s'agisse de gestion du temps (comme 32,768 kHz pour une RTC) ou de génération de signaux (PAL, NTSC, VGA, etc.).

Il en va de même pour les communications série et les UART, utilisant une fréquence de base et plusieurs diviseurs pour arriver à obtenir des débits standard comme 2400 bps, 4800 bps, 9600 bps, 14400 bps, etc. Les fréquences utilisées pour que ces divisions « tombent justes » sont souvent 1,8432 MHz, 3,072 MHz ou encore 18,432 MHz.

Rien ne vous empêche de faire d'une pierre deux coups, en optant par exemple pour un oscillateur 1,8432 MHz fournissant un signal à la fois au processeur et à l'UART. On économise des composants, on simplifie le circuit, mais les calculs de délai, de temporisation et de temps d'exécution deviennent sensiblement plus difficiles. De plus, augmenter la fréquence pour gagner en performance côté CPU ne sera pas forcément compatible avec l'UART, à moins de tolérer un certain pourcentage d'erreur dans les communications.

Ici, nous préférons avoir deux sources d'horloge pour des raisons de modularité que nous verrons une autre fois...

Le prototype initial utilise une platine supplémentaire connectée aux bus d'adresses et de données. Les 6 Motorola MC14495 sont des décodeurs hexa vers afficheurs LED 7 segments et n'ont pas besoin d'autres composants pour faire le travail. C'est très joli lorsque la fréquence du système est basse, mais à 1 MHz, ceci ne sert pas à grand-chose, puisque tout est parfaitement illisible.



soulèvent des problèmes de capacités parasites et de faux contacts pouvant conduire à des erreurs difficiles à détecter, comprendre et corriger. Une excellente vidéo (en anglais) de Ben Eater détaille parfaitement ces problèmes et pourquoi certaines platines sont plus chères que d'autres [11]. Personnellement, **énormément** de choses sont rentrées dans l'ordre d'elles-mêmes en basculant le projet de platines économiques à des modèles de meilleure facture et de marque (PBS, acheté chez Mouser). C'est certes deux fois plus cher (~10 € pour une 830 points) qu'une platine d'eBay venant de Shenzhen, mais ça fonctionne à merveille !

Mais revenons à nos moutons. Pour que le processeur ait quelque chose à faire, nous avons besoin d'un endroit pour stocker notre programme. Le Z80 dispose de 16 lignes d'adresses, A0 à A15, permettant donc d'adresser 65536 octets de mémoire (2 puissance 16). Nous pourrions parfaitement utiliser une EPROM de cette taille, ou moins, et nous satisfaire de cela, mais l'objectif n'est pas de nous limiter dans ce que nous pouvons vraiment faire. Qui dit ROM dit « lecture seule », du point de vue du processeur et donc impossibilité d'écrire en mémoire, d'utiliser une pile (*stack*) et donc de faire des appels à des sous-routines de

façon efficace (comprendre « non-spaghetti »). De plus, si nous voulons utiliser autre chose que de l'assembleur (du C, donc), nous **devons** avoir une pile et donc de la RAM.

La solution, qui n'a rien de bien excentrique, consiste à diviser l'espace d'adressage en deux. D'une part, 32 Kio de ROM s'étendant de l'adresse **0x0000** à **0x7FFF** et de l'autre, 32 Kio de RAM de **0x8000** à **0xFFFF**. Matériellement, le processeur ne sait pas que cette division existe et nous devons donc gérer cela dans notre montage. Nous reviendrons en détail sur le circuit, mais ceci implique la mise en œuvre d'un composant supplémentaire :

LS, HC ET HCT

Les circuits logiques de la série 74 se déclinent en plusieurs versions, indiquées dans leurs désignations. Un 7400, par exemple, peut être un 74LS00, un 74HC00 ou un 74HCT00. Ces lettres précisent la technologie utilisée par le composant, qui peut être :

- LS : TTL (*Transistor Transistor Logic*) Low Schottky ;
- HC : CMOS (*Complementary Metal Oxide Semiconductor*) ;
- HCT : CMOS compatible TTL.

La différence entre TTL et CMOS, en dehors de la technologie de fabrication elle-même, a une incidence sur le courant utilisé, la tension d'alimentation, les seuils de tension (haut et bas) pour les états logiques 0 et 1, ainsi que sur les délais de propagation des signaux. Autre point important, une broche laissée « en l'air » (ou « flottante ») sur un circuit TTL sera automatiquement au niveau haut, alors qu'en CMOS, il sera impératif de relier une entrée inutilisée à la masse ou à la tension d'alimentation via une résistance de rappel d'une dizaine de k Ω .

Ce dernier point est le seul réellement important dans notre cas puisque le Z80 est cadencé suffisamment lentement pour limiter les problèmes de propagation de signaux et que l'ensemble du circuit est alimenté intégralement en +5 V.

un 74HCT00. Celui-ci contient quatre portes logiques *NAND* (*NON-ET*) à deux entrées qui nous permettent d'activer ou non la ROM ou la RAM en fonction de l'état de la ligne d'adresses A15 et de celle du signal /MREQ du Z80.

ROM comme RAM peuvent être de différents types, mais nous allons aller au plus simple et au plus pratique. Côté ROM, nous allons opter pour une 28C256 effaçable et reprogrammable électriquement, c'est donc une EEPROM (*Electrically-Erasable Programmable Read-Only Memory*). Ceci par opposition à une 27C256 qui peut également être (re)programmée électriquement (avec une tension de 12,75 V), mais qui doit être tout d'abord effacée par une exposition à un rayonnement ultraviolet. On parle alors d'UVPROM (*Ultra-Violet Programmable Read-Only Memory*). Je ne vous cache pas que le côté *vintage* et la petite fenêtre transparente laissant voir la puce sont vraiment très attrayants en ce qui me concerne, mais que cette caractéristique devient très secondaire après plusieurs cycles code/test/erreur/correction. De plus, les 28C256 peuvent être programmées avec une carte Arduino et deux registres à décalage 74HC595 (voir [12]), sans avoir à investir dans un programmeur coûteux.

Côté RAM, et bien que le Z80 soit en mesure de gérer les mémoires dynamiques (nécessitant un cycle de rafraîchissement après accès), nous optons pour une RAM statique de 256 kbit (32k x 8, et donc 32 Kio) de type 62256 comme une CY62256N de Cypress, une AS7C256B de *Alliance Memory* ou encore une 71256SA de *Renesas Electronics* (pour ne citer que ce qui est encore distribué neuf). Vous trouverez facilement votre bonheur également sur eBay, en cherchant simplement « SRAM 62256 ».

La dernière partie active du système est l'UART qui ici est un PC16550DN. Comme dit précédemment, n'importe quel UART fera l'affaire, mais il existe suffisamment de 16550, fabriqués par divers constructeurs, pour que

l'approvisionnement via eBay ne soit pas réellement un problème. Encore une fois, une simple recherche de « 16550 uart dip40 » remontera beaucoup de résultats avec une gamme de prix très variée allant d'une dizaine d'euros (port inclus) à quelque 20 € (au-dessus, c'est exagéré). En plus de ce composant, vous aurez besoin d'éléments complémentaires, à commencer par un quartz d'une fréquence bien précise, divisible par 16, permettant de cadencer les échanges de données avec un débit normalisé (2400 bps, 4800 bps, 9600 bps, 19200 bps, etc.). Mes expérimentations ont débuté par un quartz de 14,7456 MHz, mais rencontrant des problèmes de communication probablement liés aux capacités parasites de la platine à essais, j'ai réduit cette fréquence à 1,8432 MHz en optant pour un quartz de cette valeur. À ce composant s'ajoute une résistance de 1 MΩ, une de 1,5 kΩ et deux condensateurs céramiques de 22 pF et 47 pF, pour être parfaitement en accord avec la *datasheet* du constructeur. Tout cela accompagné, bien entendu, d'un convertisseur USB/série pour la connexion au PC.

À tout cela s'ajoute une belle collection de résistances de 10 kΩ pour tirer les entrées inutilisées (du CPU, de l'EEPROM et de l'UART) à la tension d'alimentation, une poignée de condensateurs électrolytiques entre 200 et 400 μF pour filtrer l'alimentation et quelques condensateurs de découplage tantale goutte (idéalement) ou à film métallisé entre 100 nF et 1 μF. Enfin, un bouton pression pour le *reset* sera fort pratique ainsi qu'une source d'alimentation 5 volts stable, qu'il s'agisse d'un bloc secteur ou d'un module à placer sur la platine (ou tout simplement les 5 V fournis par l'USB). Des LED peuvent également être intéressantes pour visualiser quelques signaux et l'état de l'alimentation, sans que cela soit réellement obligatoire.

Tous ces composants, CPU, ROM, RAM, etc., doivent être choisis en version *through-hole* (ou « composants de type trous traversants » en bon français) et non en composants de surface (CMS, SMD, etc.) pour pouvoir être montés sur la platine à essais. Rien ne vous empêche de décliner le montage entièrement en version « composants de surface » en confectonnant un circuit, le Z80 existe en boîtier (*package*) QFP-44 par exemple, mais il n'en va pas de même pour l'EEPROM ou le 16550. De plus, ceci sort totalement du cadre du présent article qui se concentre précisément sur l'aspect platine à essais (ou éventuellement plaque pastillée) du système.

Notez enfin qu'on trouve, sur eBay, des « kits » très complets autour des 50 € contenant des composants (CPU, UVPROM, RAM, régulateur, etc.) en plusieurs exemplaires (x5, généralement), en provenance de Chine. Les jeux de composants ne correspondent pas exactement à notre système, mais les éléments essentiels sont là et en particulier le CPU, la RAM et l'EPROM (UVPROM généralement). Il peut être intéressant d'opter pour cette solution, à condition de comprendre le risque, puisque les contrefaçons sont légion, en particulier lorsque l'affaire semble vraiment trop intéressante.

1.2 Les outils

Vous l'aurez compris, cette fois, nous travaillons avec un montage bien réel, *old school* et sans assistance d'un microcontrôleur. Il nous faudra donc pouvoir programmer l'EEPROM, sans doute à répétition, afin de faire évoluer notre création (ceci fait précisément partie du charme de l'exercice). Dans ce sens, le premier outil nécessaire sera un programmeur d'EEPROM, quel que soit le modèle, il faut simplement qu'il puisse gérer les 28C256. Personnellement, j'utilise un TL866CS (alias Minipro de

Xgecu/AutoElectric/Haikou Xingong Electronics) acquis il y a fort longtemps, qui semble être aujourd'hui un produit en fin de vie. Il existe également un modèle plus performant, le TL866-II+, encore en production mais beaucoup plus cher. On parle ici d'un budget entre 150 € et 200 € et c'est un investissement conséquent qui devra être justifié par d'autres usages que la simple programmation d'une unique EEPROM 28C256 pour ce projet-ci.

En effet, cette famille de périphériques Xgecu est capable de gérer plus de 15000 composants allant des UVPRM, aux micro-contrôleurs, en passant par les EEPROM série (SPI & i²c), la vérification des circuits logiques type 74* ou encore la programmation des PAL/GAL (circuits logiques programmables, ancêtres des FPGA, forts pratiques pour la construction de systèmes 8 bits comme le nôtre). J'utilise ce programmeur avec l'outil en ligne de commandes **minipro** de David Griffith [13], disponible pour bien des distributions GNU/Linux, et aussi pour [Free | Open | Net]BSD.

Notez qu'il existe également deux autres programmeurs du même constructeur [14], les T48 (alias TL866-3G) et T56,



bien plus abordables (~60 € pour le T48) et plus récents, mais pour l'heure non supportés par **minipro**, car totalement différents du point de vue de leur conception. Ces produits restent parfaitement utilisables, mais uniquement sous Windows, avec le logiciel propriétaire du constructeur (Xgpro).

Si cela vous paraît excessif, car vous n'aurez pas d'autre usage pour ce matériel, il reste la possibilité de construire votre programmeur avec un Arduino UNO ou Nano. C'est, encore une fois, Ben Eater qui est à l'origine de cette idée [15], mais celle-ci a été reprise et étendue par quelques développeurs, dont Pierpaolo Pravatto [12]. Il vous suffira d'ajouter deux 74HC595 en plus de la carte Arduino et vous vous retrouverez avec un programmeur de 28C256 à un prix dérisoire sur platine à essais. Bien entendu, ceci ne permettra de programmer **que** ce composant et aucun autre, mais d'un point de vue « budget », vous ne trouverez pas mieux.

Ceci n'est pas un Saleae Logic, mais une pathétique contrefaçon initialement destinée à fonctionner avec le firmware et l'application du fabricant ayant créé l'analyseur logique original. Aujourd'hui, il est possible d'utiliser ce matériel légalement avec un firmware et un outil open source (SigRok/PulseView). L'utilisation du nom « Saleae » en revanche reste parfaitement illégale.



Le projet utilise une EEPROM 28C256, mais il aurait été possible d'utiliser également une 27C256. Avec un brochage quasi similaire, ce composant UVPROM présente un avantage cosmétique évident (c'est bôôôôôô), mais reste malheureusement très pénible à utiliser lorsqu'on expérimente et développe intensivement, puisqu'une exposition systématique aux UV est nécessaire (~15 min) pour effacer la mémoire avant programmation.

Un autre outil très intéressant, mais qui n'est pas totalement nécessaire est l'analyseur logique. Celui-ci vous permettra de lire les signaux présents sur les bus et autres lignes de contrôle du processeur ou des périphériques et vous fournira une vue précise des états en fonction du temps. Il vous sera également possible de décoder les données, comme les adresses, directement et donc d'inspecter le fonctionnement du système.

Plusieurs matériels peuvent être utilisés et se distinguent généralement par trois caractéristiques notables :

- le nombre de lignes (ou canaux) pouvant être observés simultanément ;
- la vitesse de capture ou plus exactement la fréquence d'échantillonnage ;
- et la taille des captures.

Selon le type de produit, ces deux dernières caractéristiques sont liées. Soit votre matériel capture les signaux et les envoie directement au PC pour analyse, soit les signaux sont stockés dans la mémoire de l'outil puis transférés au PC. La vitesse de capture est l'élément clé ici, car la communication entre le périphérique et le PC (en USB généralement) ne peut excéder un certain débit. Un outil supportant un stockage interne sera en mesure de capturer des signaux à une fréquence importante de l'ordre de plusieurs centaines de mégahertz, alors qu'un produit plus modeste et sans stockage se limitera au débit du bus USB (2.0 = 480 Mbit/s) et donc quelques mégahertz tout au plus. Pour les

analyseurs logiques à stockage, étant donné la limite physique en présence (mémoire interne), plus la fréquence sera importante et moins la durée de capture sera longue.

Les appareils performants, comprenant « supportant des fréquences importantes et ayant un nombre important de canaux », utilisent généralement des FPGA accompagnés d'une RAM interne. Ceci en fait des périphériques souvent très coûteux, mais qui s'avèrent fort utiles dans bien des situations où il est nécessaire de mettre au point un montage impliquant d'importants débits ou des bus très larges (16 ou 32 canaux). Une excellente solution, aujourd'hui quasi introuvable était l'*Openbench Logic Sniffer* (OLS pour les intimes) qui, accompagné d'un module complémentaire, permettait de capturer 32 canaux à une fréquence importante (jusqu'à 200 MHz), pour environ 70 €, le tout avec un support logiciel (incluant le *gateway* pour le FPGA Xilinx) entièrement *open source*.

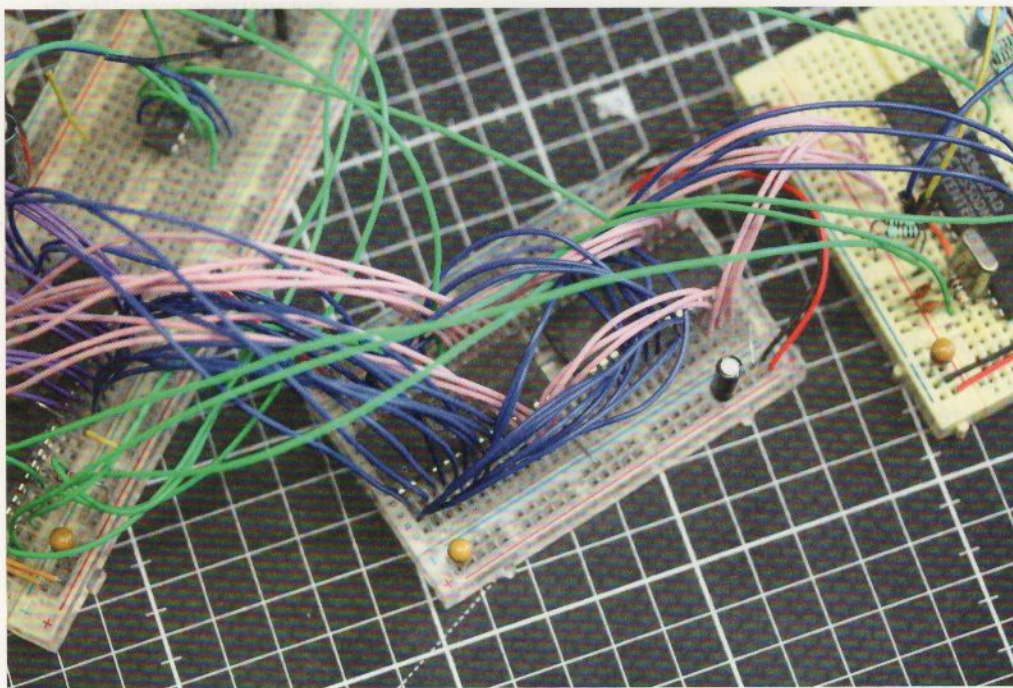
Une alternative possible, mais dans une tout autre gamme de prix et non *open hardware* est aujourd'hui les LA5016 (16 canaux, 256 Mio, pour ~270 €) et LA5032 (32 canaux,

512 Mio, pour ~400 €, faciles à trouver sur des sites comme AliExpress [16] [17]. Ces deux produits, contrairement à l'OLS, ne découlent pas d'un projet communautaire comme SUMP [18], mais sont cependant supportés par le socle logiciel *open source* Sigrok (voir Hackable 43 [19]). Il est donc possible de les utiliser avec l'application PulseView, exactement comme c'était le cas pour l'OLS, ou le produit, bien plus abordable pour le hobbyiste, sans nom, initialement vendu comme un clone de Saleae Logic (premier du nom).

Ce mystérieux périphérique ne vous coûtera qu'une poignée d'euros (~8 €, voire moins parfois), mais s'accompagne de limitations importantes. Le nombre de canaux est limité à 8, la qualité de fabrication

est au niveau du prix et la fréquence de capture est de 24 MHz maximum. Son histoire est assez douteuse puisqu'il s'agit ni plus ni moins que d'une contrefaçon pure et simple d'un produit tout à fait officiel commercialisé il y a plusieurs années par Saleae [20] qui a depuis revu sa gamme et la conception de ses produits. Ce qui était alors un « faux Saleae Logic », livré avec une copie pirate du logiciel officiel, est aujourd'hui un produit générique, parfaitement utilisable avec Sigrok/PulseView, très économique et modeste.

La magie de ce matériel, souvent vendu comme un « Analyseur Logique USB » sans nom [21] repose sur le microcontrôleur Cypress CY7C68013 (alias FX2) ou CY7C68013A (FX2LP), capable de transférer, en masse, l'état de ses entrées/sorties directement en USB. Il n'y a pas de *buffer*/RAM interne, le flux de données est continu et il n'y a donc pas de limitation de durée de capture. Ces clones utilisaient initialement,



Les platines à essais permettent de créer rapidement des montages afin de valider, en pratique, une idée ou un concept. Le problème cependant, en particulier avec des bus de 16 bits, c'est qu'on se retrouve rapidement avec une salade de câbles, intransportable et fragile.

et illégalement, le firmware propriétaire de Saleae, mais depuis, un *firmware open source* a été développé et amélioré au fil du temps (fx2lafw) permettant une utilisation totalement légale du produit (combo matériel + *firmware open source* + logiciel *open source*). Pas d'inquiétude concernant le *firmware* lui-même et sa programmation, celui-ci est chargé automatiquement par Sigrok/PulseView sans que vous ayez à intervenir.

Sur la base du même microcontrôleur Cypress existe également une version 16 canaux, vendue sous la forme d'un simple module [22] dans la même gamme de prix. C'est un clone du *Lcsoft Mini Board* (lui-même ressemblant au *devkit* Geeetech CY7C68013) se différenciant par le nombre d'entrées (16 au lieu de 8), mais faisant l'impasse sur le *buffer* (NXP 74LVC245A) protégeant le microcontrôleur. Attention donc, car sans cette protection, il est déconseillé de brancher directement le produit à un montage, car celui-ci ne tolère que des tensions de maximum 5 volts. La page du wiki Sigrok [23] recommande l'ajout de résistances et de diodes Zener, ou tout simplement deux *buffers* 74HC241 ou deux *latches* 74HCT573 en guise de protection, ce qui augmente sensiblement le prix, mais vous doublez le nombre de canaux par rapport au « faux Saleae Logic » (il serait plaisant que quelqu'un donne un jour un vrai nom à ce matériel, c'est pénible).

En ce qui concerne notre système et la fréquence de base de 1 MHz pour cadencer le Z80, ce genre de produit sera suffisant. Certes, vous ne pourrez analyser que 8 signaux en même temps (ou 16 avec la version EZ-USB/Lcsoft) ce qui ne permet pas de surveiller 16 bits d'adresses, 8 bits de données et quelques signaux complémentaires (/MREQ, /RD, /RW, /M1, etc.), mais ce sera déjà une aide précieuse en cas de dysfonctionnement ou de comportement douteux. Dans d'autres situations, et je pense en particulier à une petite carte industrielle équipée d'un Z180 cadencé à 18,432 MHz dont je vous parlerai prochainement (note de moi-du-futur : dans ce même numéro, en réalité), un analyseur performant était nécessaire et j'étais très heureux d'avoir en ma possession un OLS en état de marche pour analyser de fonctionnement de l'ensemble. Mais avec un Z80 à 1 MHz sur platine à essais, un produit économique à base de CY7C68013 est déjà un atout de taille.

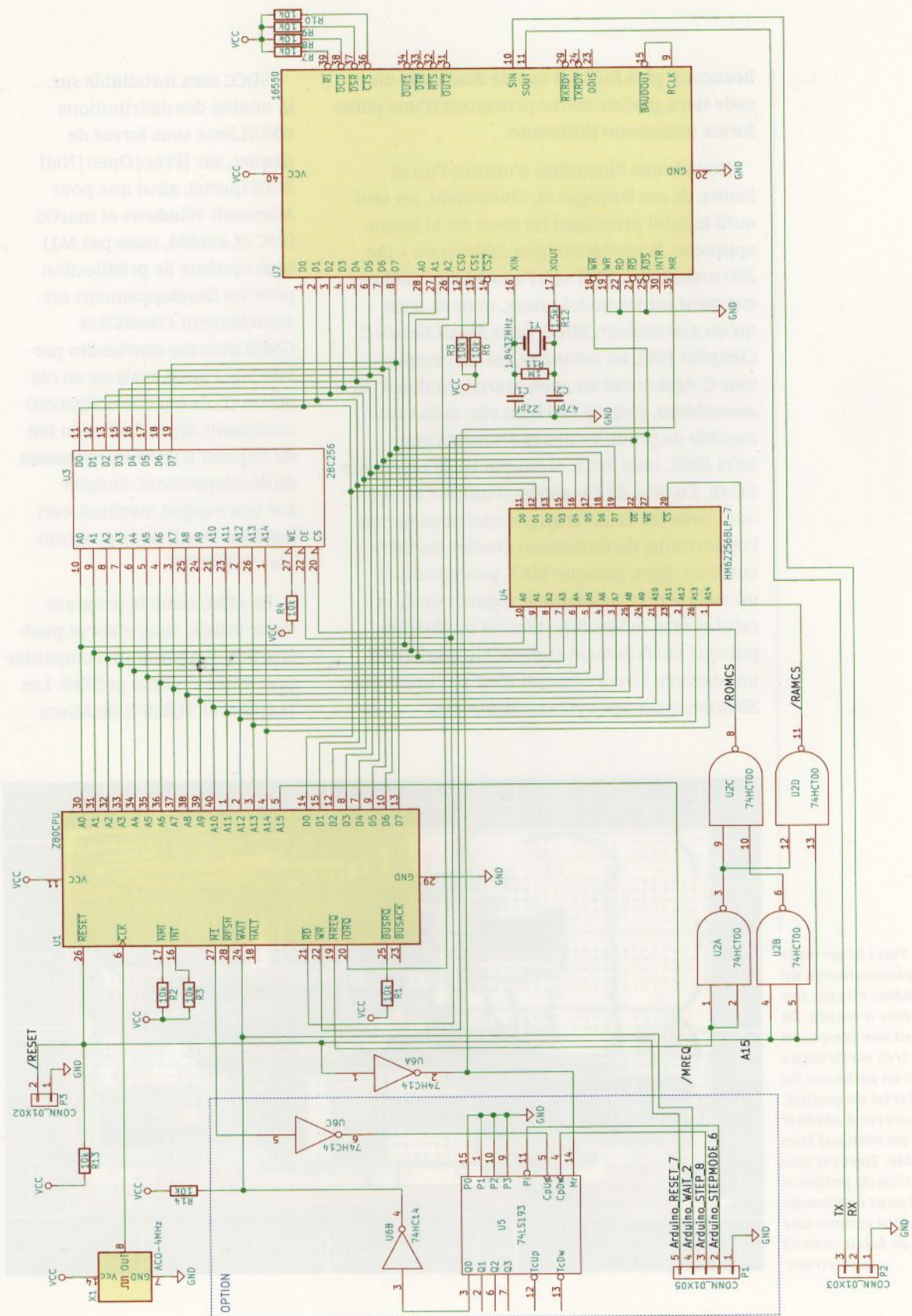
De manière générale, vous l'aurez compris, dès lors qu'on ajoute ROM et RAM, les choses deviennent en tantinet plus « sérieuses » financièrement et demandent un investissement

non négligeable. Certains d'entre vous, chers lecteurs, possèdent peut-être déjà un tel équipement, qui s'avère généralement assez générique pour tout un tas d'autres activités, de conception ou de réparation. Le programmeur et l'analyseur logique viennent souvent compléter l'équipement standard juste après le multimètre, l'alimentation de laboratoire et l'oscilloscope.

1.3 Côté logiciel

Plusieurs solutions, plus ou moins complètes/complexes, existent pour produire du code binaire exécutable par un Z80. Le langage principalement utilisé est l'assembleur, offrant une maîtrise complète du fonctionnement du système et une compacité optimale quant au code machine produit. L'assembleur est cependant un langage souvent considéré comme difficile à apprendre, mais en réalité, c'est davantage une question d'habitude et de disposition mentale qu'autre chose. Il suffit de penser comme la machine et tout devient subitement plus clair. L'autre langage utilisé est le C, offrant un niveau d'abstraction plus important et favorisant la portabilité. La proximité avec le matériel est moindre, mais il devient

- Zilog Z80 : le retour du fils de la vengeance ! -



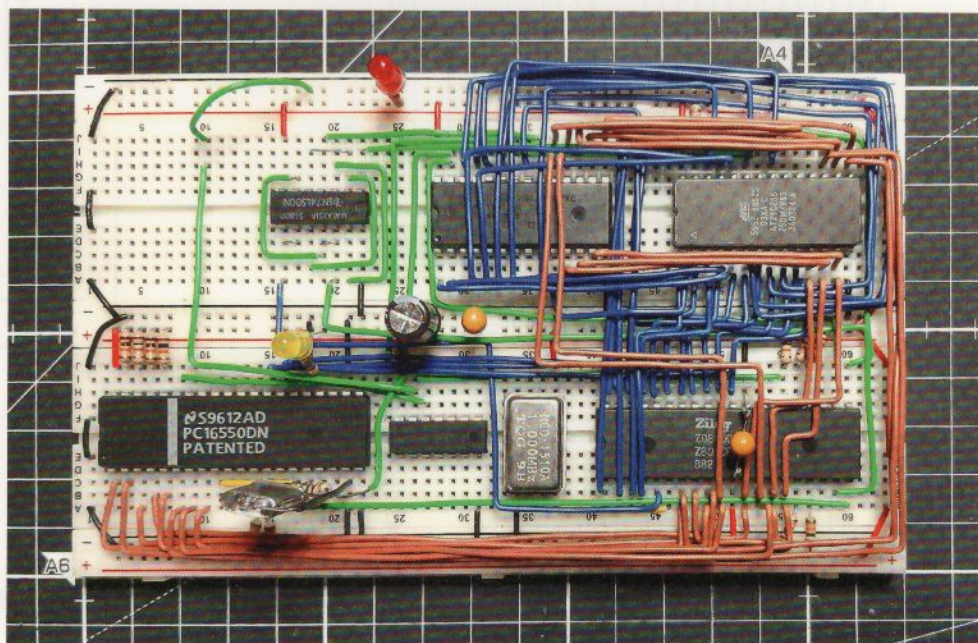
beaucoup plus facile et rapide d'adapter un code tiers, parfois même provenant d'une plateforme totalement différente.

Nous avons l'intention d'utiliser l'un et l'autre de ces langages et, idéalement, un seul outil logiciel proposant les deux est la bonne approche. Il semblerait que *Z80asm* ou « *the Z80 assembler* » [24] soit l'assembleur plébiscité pour initier un tel projet, mais ce n'est qu'un assembleur. SDCC, pour *Small Device C Compiler* [25], en revanche, est un compilateur C, également en *open source*, doublé d'un assembleur, pour le Z80 bien sûr, mais aussi capable de produire des exécutables pour MOS 6502, Intel 8051, Motorola HC08 ou encore STM8. En plus de l'avantage d'utiliser un seul outil pour les deux langages, ceci nous donne l'opportunité de facilement étudier les liens entre les deux, puisque SDCC peut produire un code source assembleur à partir du C, et celui-ci sera même directement modifiable puisque parfaitement compatible pour créer un nouveau binaire (ce qui n'est pas le cas avec *Z80asm*, qui a une syntaxe différente).

SDCC sera installable sur la totalité des distributions GNU/Linux sous forme de paquet, sur [Free | Open | Net] BSD] (ports), ainsi que pour Microsoft Windows et macOS (PPC et amd64, mais pas M1). Mon système de prédilection pour les développements est actuellement FreeBSD et GNU/Linux me conviendra parfaitement aussi, mais est en réalité un choix (ou une obligation) totalement dépendant(e) du fait de disposer d'un environnement de développement complet sur une unique machine, ceci incluant l'outil pour programmer l'EEPROM.

En effet, selon le programmeur utilisé, vous n'aurez peut-être pas d'application compatible avec votre système préféré. Les TL866CS et TL866-II de XGecu

Voici la seconde implémentation du même circuit sur platine à essais. En prenant son temps, en restant très méticuleux et en utilisant du matériel de qualité, on arrive à obtenir un résultat très acceptable. Tout est une question de patience et ceci peut également être vu comme une activité de loisir créatif antistress.

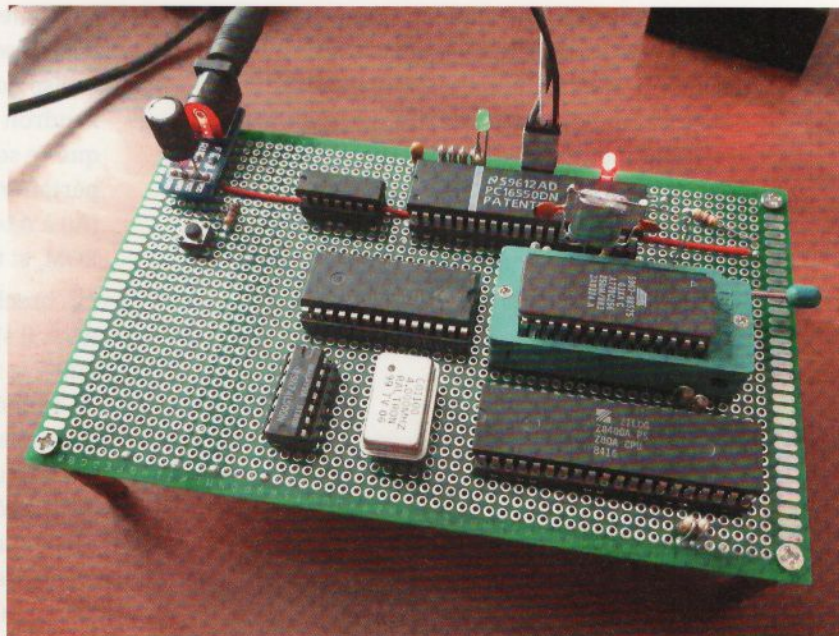


peuvent s'utiliser avec **mini-pro**, en ligne de commande (ce qui convient à merveille pour s'intégrer dans un **Makefile**), mais les nouveaux modèles comme le T48 ou le T56 du même constructeur ne sont pas supportés (il en va de même pour le RT809H qui semble également gagner en popularité, en ce moment). Là, Windows est obligatoire et il serait sans doute préférable (« malheureusement » selon vos préférences personnelles) d'utiliser SDCC sur le même OS, éventuellement via WSL [26]. L'option « programmeur Arduino », quant à elle, est suffisamment générique pour fonctionner partout (Arduino IDE + script Python).

Nous nous occuperons plus en détail de l'aspect logiciel et de la programmation la prochaine fois, le présent article concernant avant tout l'aspect matériel et vous laissant le temps de vous procurer tous les composants et éventuellement les outils (programmeur et analyseur logique).

2. LE CIRCUIT

L'ensemble du circuit se résume au schéma présenté en pleine page dans le présent article. Vous pouvez ignorer la partie encadrée en bleu puisque celle-ci est



optionnelle et un héritage direct des expérimentations précédentes (que je ne suis pas arrivé à me convaincre de supprimer). Ce circuit se compose des éléments suivants :

- 1 x Zilog Z80 ;
- 1 x oscillateur quartz 1 MHz ;
- 1 x EEPROM 28C256 ;
- 1 x SRAM 62256 ;
- 1 x 74HCT00 ;
- 1 x 74HC14 ;
- 12 x résistances 10 k Ω ;
- 1 x UART PC16550DN ;
- 1 x quartz 1,8432 MHz ;
- 1 x résistance 1 M Ω ;
- 1 x résistance 1,5 k Ω ;
- 1 x adaptateur USB/série 5 V ;
- 1 x condensateur céramique 22 pF ;
- 1 x condensateur céramique 47 pF ;
- quelques condensateurs électrolytiques entre 47 et 400 μ F et tantale (ou film) entre 100 nF et 1 μ F pour stabiliser et filtrer l'alimentation (découplage).

Une fois l'étape de la platine à essais validée, il est possible de s'attacher à quelque chose de plus définitif, comme la plaque pastillée. L'avantage de cette approche est la pseudorobustesse (tout est relatif), mais aussi le fait que tout semble propre et ordonné...

Notez que la partie alimentation n'est pas représentée ici et, plus ou moins, laissée à votre discrétion. Il est toujours intéressant d'ajouter quelques condensateurs de filtrage, mais mes essais ont montré qu'avec une alimentation assez stable comme le +5 V d'un port USB et des platines de qualité, ceci ne semble pas être une absolue nécessité (j'entends d'ici les électroniciens de métier hurler).

2.1 Processeur

Le Z80, dans la version utilisée ici, se présente sous la forme d'un composant de 40 broches dont la nomenclature est parfaitement décrite dans sa documentation [27] qui deviendra rapidement votre livre de chevet, si vous prenez goût au sujet. Faisons rapidement le tour de ce que nous utilisons :

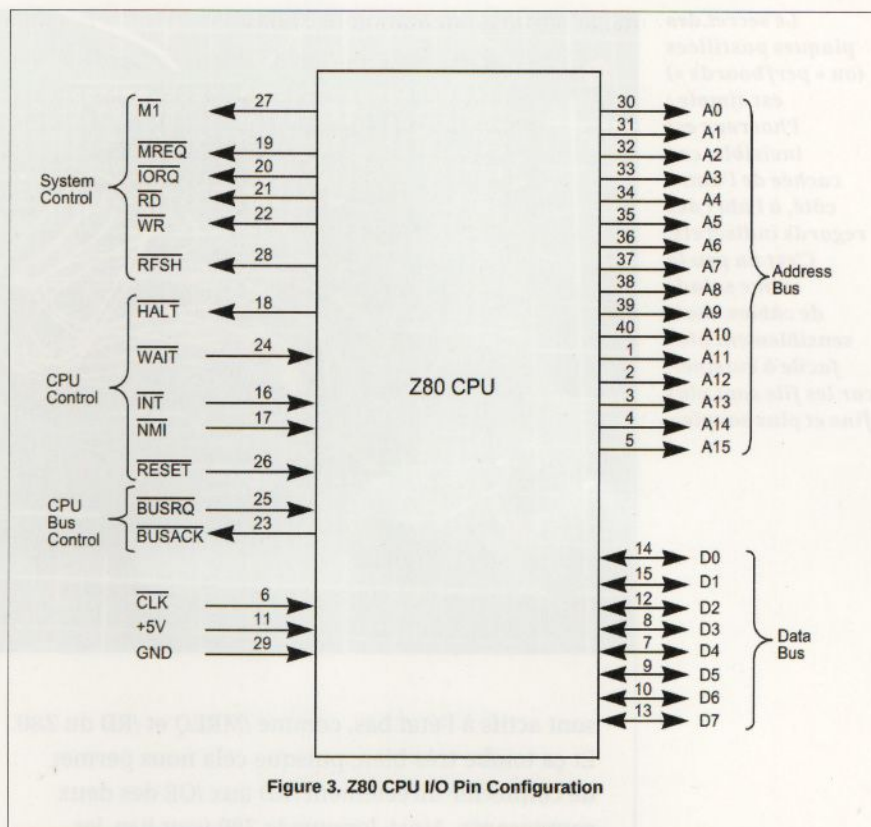
- Les broches 11 (+5 V) et 29 (GND) sont respectivement la source d'alimentation du processeur et la masse. C'est au plus proche de ce point que nous devrions placer un condensateur tantale ou film pour garantir une tension propre et stable.
- /RESET (26) est une entrée permettant de procéder à une (ré)initialisation du processeur. Plusieurs points sont importants ici. Premièrement, le Z80 n'a pas de circuit d'autoreset, ce qui signifie qu'à la mise sous tension, il se trouve dans un état indéfini. Il est alors nécessaire de procéder à un *reset* en mettant cette broche à la masse pendant au moins trois cycles d'horloge avant de la ramener à +5 V via une résistance de 10 kΩ. On dit que ce signal est **actif à l'état bas**, car ce n'est pas la mise à Vcc (+5 V) qui déclenche l'action mais la mise à la masse. /RESET n'est pas le seul signal, en entrée comme en sortie, à utiliser cela et la barre de fraction (« / ») placée devant le nom « RESET » précise cette caractéristique. Il en va de même pour tous les autres signaux de ce type au travers de la documentation de ce processeur (et de bien d'autres).
- Les lignes A0 à A15 forment le bus d'adresses. Ce sont des sorties que le Z80 utilise pour indiquer au reste du système l'endroit précis où il souhaite accéder, que ce soit en RAM, en ROM ou dans un périphérique (registres). 15 de ces lignes (A0 à A14) sont reliées à la RAM et la ROM, et trois (A0 à A2) à l'UART.
- Les broches D0 à D8 sont le bus de données. Il s'agit d'entrées/sorties permettant à la fois au processeur de lire 8 bits de données lorsque l'état des broches est imposé par un autre composant, ou d'écrire une valeur lorsque c'est à lui de contrôler le bus. Ces 8 signaux sont reliés à la ROM, la RAM et l'UART. Les bus d'adresses et de données sont... des bus. Ceci signifie que ces signaux connectent la quasi-totalité des composants du système. Pour arbitrer leur utilisation (savoir quand le processeur lit ou écrit), plusieurs autres signaux/broches sont présents.
- /MREQ (19) et /IORQ (20), actifs à l'état bas, sont des sorties qui permettent au processeur d'indiquer s'il souhaite accéder à la mémoire, *Memory REQuest* ou à un périphérique d'entrée/sortie, *I/O Request*. C'est alors au reste du circuit de réagir en conséquence en activant ici, soit l'une des mémoires, soit l'UART, et donc de prendre en compte, via le bon composant, l'état des différentes lignes du bus d'adresses.
- /RD (21) et /WR (22), également actifs à l'état bas et également des sorties, indiquent respectivement au reste du système si le Z80 souhaite lire ou écrire à l'adresse présentée sur le bus. Là encore, le composant concerné utilise l'un ou l'autre signal pour savoir comment réagir. En cas de lecture, c'est lui qui impose des états au bus de données en réponse, mais en cas d'écriture, le Z80 définit les bits et le composant les utilise.

D'autres signaux sont également présents sur le Z80, mais ne sont pas présentement utilisés. Certaines de ces broches sont des entrées comme /WAIT (24), /INT (16), /NMI (17) et /BUSRQ (25), et doivent être ramenées au +5 V avec des résistances de 10 kΩ pour imposer un état haut (inactifs, donc). D'autres, /HALT (18), /M1 (27), /RFSH (28) et /BUSACK (23) sont des sorties qui sont alors simplement laissées non connectées dans notre circuit.

Une dernière broche nécessite notre attention, c'est /CLK (6), tantôt appelée « Phi ». C'est le signal d'horloge cadencant le processeur et nous y connecterons la sortie de notre oscillateur de 1 MHz (ou 4 si vous voulez tenter le diable). Cet oscillateur est alimenté, comme le reste du système, avec +5 V et une connexion à la masse et devra de préférence se trouver au plus proche du processeur, pour assurer l'intégrité du signal.

2.2 RAM et ROM

Les deux mémoires que sont l'EEPROM 28C256 et la RAM (SRAM, pour être précis) utilisent un brochage très similaire, et pour cause, ce n'est que le caractère « volatile » des données stockées qui change. Les



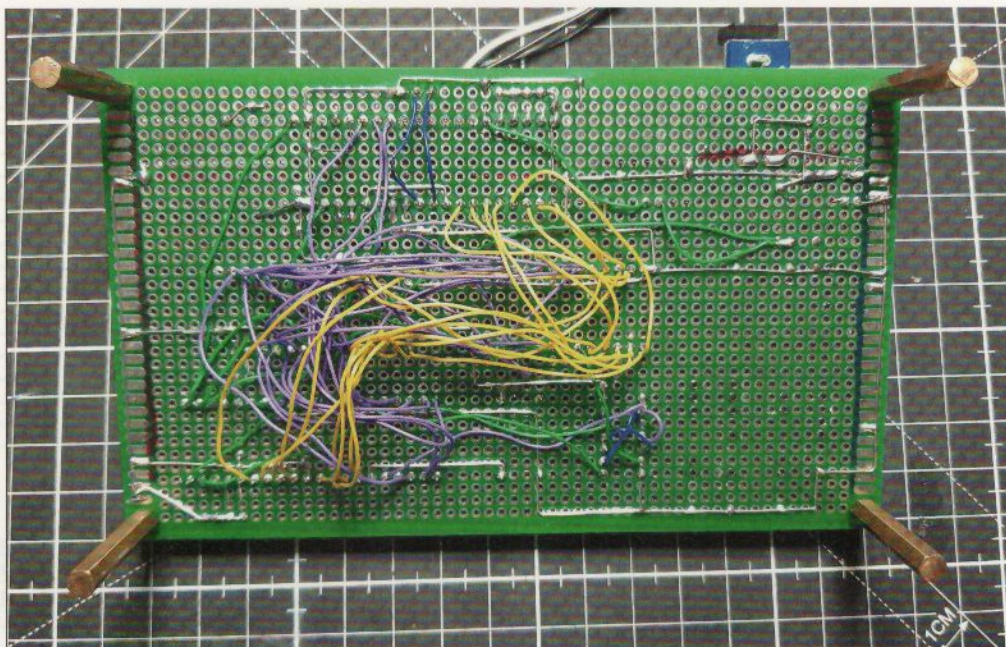
deux composants sont en boîtier DIP28 et on retrouve, sans surprise, les mêmes signaux des deux côtés.

Nous avons tout d'abord le bus d'adresses de A0 à A14 (15 bits pour adresser 32768 emplacements mémoires de 8 bits) et le bus de données de 8 bits. Comme décrit précédemment, RAM et ROM voient ces deux séries de broches connectées au bus d'adresses et au bus de données du Z80.

Deux broches spécifiques permettent respectivement d'activer le composant et de lui demander de présenter les données sur le bus de données, c'est /CE pour *Chip Enable*, parfois appelé *Chip Select* et donc /CS (les deux désignations sont équivalentes et interchangeables) et /OE pour *Output Enable*. Remarquez que ces deux signaux, qui sont des entrées de la ROM et la RAM,

**Brochage du
processeur Zilog
Z80.**

Le secret des plaques pastillées (ou « perfboards ») est simple : l'horreur est invisible, car cachée de l'autre côté, à l'abri des regards indiscrets. C'est un peu la même salade de câbles, mais sensiblement plus facile à cuisiner, car les fils sont plus fins et plus souples.



sont actifs à l'état bas, comme /MREQ et /RD du Z80. Et ça tombe très bien, puisque cela nous permet de connecter directement /RD aux /OE des deux composants. Ainsi, lorsque le Z80 veut lire, les mémoires reçoivent, toutes deux, l'ordre de présenter les données correspondant à l'adresse présentée via A0 à A14. Mais ce n'est que le composant dont /CE (ou /CS) est à la masse qui répondra.

De la même manière, /WE (Write Enable), actif à l'état bas toujours, ordonne à la mémoire de prendre en compte l'information présentée sur le bus de données et de la stocker à l'adresse correspondant à l'état de A0 à A14. Ceci ne concerne, bien entendu, que la RAM et le /WR du Z80 est donc connecté directement à /WE du composant. Le /WE de l'EEPROM, utilisé que pour la programmation, est tiré à Vcc avec une résistance de 10 kΩ.

Nous reste donc le signal /CE, qui permet à une mémoire « de se sentir concernée » si on veut (c'est la broche « c'est à toi que je cause ») et là, nous avons un peu de magie à faire. En effet, pour décider si c'est la ROM ou la RAM qui est accédée pour une adresse donnée, selon notre organisation mémoire décrite précédemment, nous prenons en

compte les sorties /MREQ et /A15 du Z80 et, en fonction de leurs états, activons le /CE de la ROM (alias /ROMCS) ou de la RAM (alias /RAMCS). La logique est la suivante :

- si /MREQ est à la masse et /A15 à la masse, alors le /CE de la ROM est à la masse et celui de la RAM à +5 V ;
- si /MREQ est à la masse et /A15 à +5 V, alors le /CE de la RAM est à la masse et celui de la ROM à +5 V ;
- si /MREQ est à +5 V, peu importe l'état de /A15, les /CE de la RAM et la ROM sont à +5 V.

Nous pouvons accomplir cela à l'aide du circuit logique 74HCT00, intégrant quatre portes logiques

NON-ET (NAND), c'est-à-dire quatre portes logiques ET dont la sortie est inversée. Ce composant est représenté, sur le schéma, sous la forme des quatre portes logiques U2A, U2B, U2C et U2D.

Si on suit simplement la logique booléenne et du fait que le signal /MREQ est connecté aux deux entrées de la première porte (U2A), et /A15 à celles de la seconde (U2B) ainsi qu'à la quatrième (U2D), on se rend compte que si /MREQ est à +5 V, alors la porte ET correspondante est *VRAI* et devient *FAUX* après inversion. De ce fait, la porte ET de U2C ne peut jamais être *VRAI*, peu importe la seconde entrée, mais la sortie l'est après inversion. On se retrouve donc avec /ROMCS (/CE de la ROM) et /RAMCS (/CE de la RAM) à l'état haut et ROM et RAM sont inactifs, quel que soit l'état de A15.

Si /MREQ est à la masse, en revanche, la sortie de la U2A est *VRAI* et si tel est le cas, deux cas de figure se présentent. Si A15 est à la masse, U2B est *VRAI* (ET est *FAUX*, mais on inverse), et comme l'autre entrée de U2C est également *VRAI* à cause de /MREQ, alors le ET est *VRAI*. Et comme nous inversons, ceci devient *FAUX* et la sortie /ROMCS

est à la masse, activant la ROM dont le /CE se retrouve également à la masse. Dans le même temps, comme A15 est à la masse, l'une des deux entrées de U2D l'est également, rendant le ET forcément *FAUX*, puis inversé en *VRAI*. /RAMCS se retrouve alors à +5 V et la RAM n'est pas activée.

Enfin, si A15 est à +5 V, l'inverse se produit. La sortie de U2A est toujours à *VRAI* du fait de /MREQ à la masse, mais la sortie de U2B est *FAUX*. La porte ET de U2C est donc *FAUX* et inversée en *VRAI*. /ROMCS est donc à +5 V et la ROM n'est pas activée. Mais U2D voit ses deux entrées à *VRAI* et donc sa sortie à *FAUX*, mettant /RAMCS à la masse et donc le /CE de la RAM également. Celle-ci est alors active.

Il est possible d'arriver au même résultat avec d'autres portes logiques et inverseurs (comme un OU entre /MREQ et A15 pour /ROMCS qu'on inversera pour /RAMCS), mais sauf erreur de ma part, la quadruple porte NAND est la seule solution permettant de le faire avec un unique composant (à l'exclusion des PAL/GAL). Notez que nous avons déjà un sextuple inverseur dans le circuit (74HC14) et pourrions éventuellement nous en servir, avec un 74HC32 (quadruple porte OU à deux entrées).

2.3 UART

Après cette description des interactions entre le processeur et les mémoires, difficile de ne pas faire le pendant avec les périphériques et donc ici le seul en présence : l'UART PC16550. Également connecté au bus de données et à une toute petite partie du bus d'adresses, ce composant n'est pas interfacé très différemment de la RAM ou la ROM. Il ne dispose que de trois lignes d'adresses, A0 à A2, permettant donc d'adresser seulement 8 emplacements mémoires, correspondant aux 8 registres dont il dispose en interne. Il est donc possible de voir ce périphérique, comme tout autre du même genre, comme une sorte de minuscule mémoire.

Comme la ROM ou la RAM, le 16550 dispose d'une broche /CS, mais celle-ci est appelée /CS2, car il existe également CS0 et CS1. Notez l'absence de « / » devant le

nom de ces signaux, car ils sont, eux, actifs à l'état haut. En reliant ces deux broches (12 et 13) à Vcc via des résistances, nous les rendons actives et /CS2 seul détermine alors l'activation de tout le composant, exactement comme pour les mémoires. Inutile ici d'avoir recours à un circuit logique, puisqu'il nous suffit de connecter /CS2 à /IORQ du Z80, qui ne passera à la masse que si le processeur s'adresse à un périphérique d'entrée/sortie.

POURQUOI TROIS CHIP SELECT ?

Sur cet UART, comme bien d'autres périphériques, on trouve plusieurs lignes *Chip Select* (ou *Chip Enable*), certaines actives à l'état haut et d'autres non. Le 16550 est assez typique de l'utilité de ces broches avec /CS2 pouvant être relié à /IORQ et, par exemple, CS0 et CS1 respectivement à A3 et A4. Il faudra alors que ces deux lignes d'adresses se trouvent à +5 V pour que l'UART soit active (ou « sélectionnée »). De ce fait, les 8 registres ne seront plus accessibles aux adresses 0x00 (00000000 en binaire) à 0x07 (00001111), car le composant serait désactivé, mais de 0x18 (00011000) à 0x1f (00011111).

Comme il est possible de lire et d'écrire dans les registres de l'UART, les lignes /RD et /WR sont connectées aux signaux du même nom dans le reste du montage. Notez ici aussi la présence de RD et WR, qui sont des signaux actifs à l'état haut, équivalents de /RD et /WR. Mais attention, ceci ne fonctionne pas comme CS0, CS1 et /CS2 qui doivent tous être actifs pour activer le 16550. Ici, l'une **ou** l'autre broche permet de lire ou d'écrire : mettre /RD à la masse **ou** RD à Vcc active une lecture. Comme nous utilisons /RD et /WR, nous connectons RD et WR à la masse, pour qu'ils soient **toujours** inactifs et que seuls /RD et /WR comptent.

Parmi les autres broches et comme vous pouvez le constater sur le schéma, certaines sont des entrées mises à l'état haut, car inutilisées, /RI (*Ring Indicator*), /DCD (*Data Carrier Detect*), /DSR (*Data Set Ready*), CTS (*Clear To Send*), ou des sorties non connectées, /DTR (*Data Terminal Ready*), /RTS (*Request To Send*), DDS, etc. Elles sont toutes en rapport

avec le contrôle de flux ou des signaux utilisés pour les communications modem et ne les utilisons pas.

Seules MR (*Master Reset*), RCLK et /BAUDOUT nécessitent encore notre attention. La première est le *reset* de l'UART que nous connectons, via un inverseur (il n'y a pas de « / » devant « MR ») pour être réinitialisé en même temps que le Z80. /RCLK pour *Receiver Clock* est un signal d'horloge (de 16 fois la vitesse de communication) en entrée pour la partie réception de l'UART, comme nous n'utilisons pas un circuit d'horloge distinct pour cela, nous connectons cette broche à /BAUDOUT qui est une sortie correspondant au signal généré par la partie émission de l'UART à partir du circuit d'horloge et du quartz. La fréquence générée est également 16 fois celle utilisée pour la communication et est réglable via une configuration des registres. Cette connexion implique que nous utilisons toujours la même vitesse et le même débit en émission comme en réception.

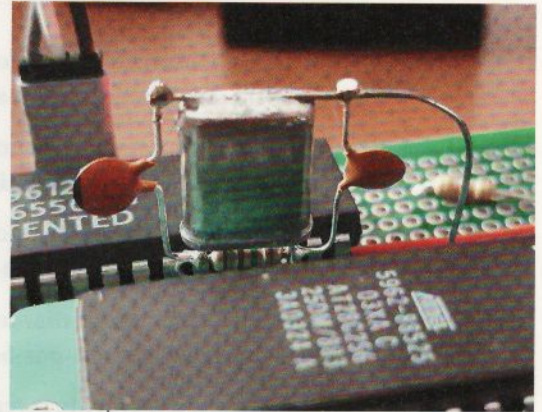
Ce qui nous amène justement à la partie concernant l'oscillateur intégré à l'UART. Celle-ci utilise un quartz connecté

sur les broches XIN (16) et XOUT (17), en parallèle d'une résistance de 1 M Ω avec XOUT relié à une résistance de 1,5 k Ω . Cette partie du circuit est strictement celle décrite dans la *datasheet* [28] du PC16550DN de *National Semiconductor* (page 16). Les condensateurs sont donnés à titre indicatif avec celui entre XIM et la masse entre 10 et 30 pF, et l'autre entre 40 et 60 pF, et le document précise qu'il s'agit de valeurs approximatives et que ceci est dépendant des spécifications du quartz (que nous n'avons pas) et des caractéristiques du circuit. La plupart des circuits à base de Z80 mettant en œuvre cet UART semblent totalement ignorer la différence de valeur entre l'un et l'autre condensateur et/ou les résistances. D'autres préfèrent connecter un oscillateur (comme celui pour le Z80) à 1,8432 MHz directement sur XIN (ceci est décrit par la *datasheet*) et donc éviter ces problématiques.

Enfin, la connexion avec l'adaptateur USB/série se fera par les broches SIN (*Serial In*) et SOUT (*Serial OUT*), en s'assurant que les tensions soient bien 0/+5 volts et non 0/+3,3 volts. La plupart des adaptateurs ont un interrupteur permettant de choisir l'un ou l'autre.

CONCLUSION

Voilà qui nous amène à la fin de cette première (re)mise en jambe et jette les bases de notre ordinateur 8 bits nouvelle génération. Dans le prochain volet, déjà implémenté (ainsi qu'au moins deux ou trois des suivants), nous nous pencherons sur la partie logicielle avec une prise en main des outils et la programmation elle-même. Nous avons là une solide base, très traditionnelle, matérialisée deux fois sur platine à essais et une fois sur plaque pastillée. Nous reviendrons peut-être (mais j'en doute) sur l'interaction avec une carte Arduino dans l'avenir, mais je dois vous avouer que cette nouvelle approche classique et autonome fait que je ne suis pas motivé outre mesure pour partir dans cette direction. Il est bien plus satisfaisant de voir le système « vivre » de lui-même ainsi... **DB**



Lorsqu'on ne réfléchit pas suffisamment avant de s'attaquer au montage sur prefboard, on se retrouve parfois dans une situation où il faut se montrer créatif, faute de place. Dans cette catégorie, je vous présente le « quartz à oreilles » ! Allez savoir, avec les horreurs dégénérées qu'on trouve dans les musées d'art moderne actuellement, quelqu'un est bien capable de penser que c'est une œuvre...

RÉFÉRENCES

- [1] <https://connect.ed-diamond.com/Hackable/hk-020/creez-votre-ordinateur-8-bits-sur-platine-a-essais-le-processeur>
- [2] <https://connect.ed-diamond.com/Hackable/hk-021/creez-votre-ordinateur-8-bits-sur-platine-a-essais-la-memoire>
- [3] <https://connect.ed-diamond.com/Hackable/hk-022/votre-ordinateur-8-bits-sur-platine-de-la-ram-une-pile...-cap-vers-le-c>

- [4] <https://connect.ed-diamond.com/Hackable/hk-023/votre-ordinateur-8-bits-sur-platine-assez-joue-on-passe-au-c>
- [5] <https://connect.ed-diamond.com/Hackable/hk-024/et-si-on-faisait-communiquer-notre-z80>
- [6] <https://connect.ed-diamond.com/Hackable/hk-027/ordinateur-8-bits-z80-on-prend-les-memes-et-on-recommence>
- [7] <https://connect.ed-diamond.com/Hackable/hk-028/z80-une-execution-pas-a-pas-qui-fonctionne>
- [8] <https://www.retrobrewcomputers.org/forum/index.php?t=msg&th=511&goto=8063>
- [9] <https://rc2014.co.uk/what-is-rc2014/>
- [10] <https://zeal8bit.com/>
- [11] <https://www.youtube.com/watch?v=fCbAafKLqC8>
- [12] <https://github.com/ppravatto/28C256>
- [13] <https://gitlab.com/DavidGriffith/minipro>
- [14] http://www.autoelectric.cn/en/tl866_main.html
- [15] <https://github.com/beneater/eeprom-programmer>
- [16] <https://fr.aliexpress.com/item/1005004740111095.html>
- [17] <https://fr.aliexpress.com/item/10000279631856.html>
- [18] <https://www.sump.org/projects/analyzer/>
- [19] <https://connect.ed-diamond.com/hackable/hk-043/instrumentez-votre-analyseur-logique-avec-lbsigrok>
- [20] <https://www.saleae.com/>
- [21] <https://fr.aliexpress.com/w/wholesale-Analyseur-Logique-USB.html>
- [22] <https://fr.aliexpress.com/item/32896748758.html>
- [23] https://sigrok.org/wiki/Circuits_for_barebone_boards
- [24] <http://savannah.nongnu.org/projects/z80asm>
- [25] <https://sdcc.sourceforge.net/>
- [26] <https://learn.microsoft.com/fr-fr/windows/wsl/install>
- [27] <https://www.zilog.com/docs/z80/um0080.pdf>
- [28] <https://www.alldatasheet.com/datasheet-pdf/pdf/9302/NSC/PC16550DN.html>