# HACKABLE MAGAZINE

DÉMONTEZ | COMPRENEZ | ADAPTEZ | PARTAGEZ

N° 34

JUILLET / AOÛT
SEPTEMBRE 202

FRANCE MÉTRO. : 12,90 € - BELUX : 13,90 € - CH : 20,70 C
ESP/IT/PORT-CONT : 13,90 € - DOM/S : 13,90 € - TUN : 38 TN
MAR : 145 MAD - CAN : 21,99 $C

L 19338 - 34 H - F: 12,90 € - RD


**ESP32/SDK**

Découvrez les nouveautés majeures de l'environnement de développement officiel ESP32 version 4.0

**NFC/E-PAPER**

Prise en main en C du Waveshare NFC ePaper, un écran e-ink entièrement piloté et alimenté en NFC

**CODE/AFFICHEUR**

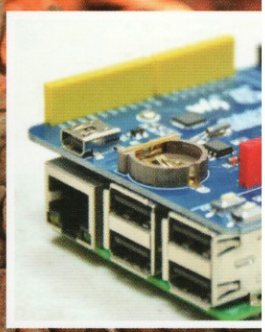
Optimisation de code Arduino : l'exemple de l'octuple afficheur 7 segments à registre à décalage

Raspberry Pi / Chaudière / IoT :

## TRANSFORMEZ UN POÊLE À GRANULÉS EN SYSTÈME CONNECTÉ ET INTELLIGENT

# DOMOTISEZ VOTRE CHAUFFAGE !





- S'interfacer avec le poêle à granulés
- Automatiser et installer des capteurs
- Ajouter des services web RESTful

**AVR/DEBUG**

Émulez fidèlement un circuit complet intégrant un Microchip/Atmel AVR grâce à SimAVR

**NINTENDO/8-BITS**

Écrivez votre premier code assembleur pour la NES : gestion graphique et scrolling

# L'électronique pérenne et innovant

Bureau d'études en électronique et informatique embarquée, Agilack vous accompagne dans la conception, le prototypage et l'industrialisation de vos produits.

AGILACK

## Cowlab

Des outils plus simples, des développements plus rapides avec Cowlab

## OpenHardware

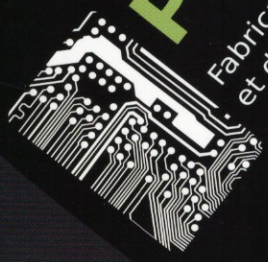
Apprenez à contribuer tout en protégeant votre propriété intellectuelle

## CAO

Des logiciels Libres et des formats ouverts pour une pérénité maximale

## Prototype

Fabrication des premières unités et des petites séries



## Software

Développement de firmwares, drivers, BSP, ...

www.agilack.fr contact@agilack.fr

# ÉDITO

« J'ai un nouveau projet... »

Je ne compte plus le nombre de vidéos YouTube débutant par ces mots (en anglais le plus souvent) en guise d'introduction, et s'avérant n'être finalement, après maints détours inutiles (pour allonger la vidéo et faire plaisir aux algos de la plateforme), que l'exposé d'une quelconque opération ne nécessitant pas la moindre créativité ou innovation.

Jugez-moi un rien naïf, mais voyez-vous, lorsque quelqu'un me dit que « son nouveau projet » est de remplacer tel élément de son ordinateur 8 bits vintage par un FPGA, je m'attends tout naturellement à ce que la personne ait développé et conçu la chose... Et non qu'elle soude simplement le bidule qui, soit dit en passant, est un vrai projet, mais de quelqu'un d'autre.

Le problème est que ce genre de démonstration (ou de déballage de produit), élevé au rang de « projet » et présenté comme tel, a tendance à se généraliser. Ouvrir l'exemple Blink dans l'IDE Arduino et le flasher dans le microcontrôleur est tout autant un « projet » que la cuisson des pâtes de supermarché est de la « cuisine ».

Ne vous y trompez pas, je n'ai absolument rien contre le fait de présenter des circuits/modules/kits et de fournir deux ou trois explications utiles au passage, bien au contraire (d'autant que je serai mal placé pour critiquer). Mais faire passer cela pour un projet est, pour moi, rien d'autre qu'une façon de s'attribuer indirectement une partie du crédit dû à la personne qui a réellement travaillé dur pour obtenir un résultat facilement réutilisable par tous.

L'humilité est une base essentielle pour pouvoir apprendre. En survalorisant une simple manipulation, on se met dans une position où le faux sentiment d'accomplissement devient suffisant, et on cesse alors automatiquement de vouloir en apprendre davantage. Encore une fois, expliquer comment cuire des pâtes n'est pas un problème, mais dire que c'est de la cuisine est anti-pédagogique, car celui ou celle qui le croit n'apprendra peut-être jamais à vraiment cuisiner...

Sur ces belles paroles, je vous laisse découvrir ce numéro et en particulier l'article principal décrivant très exactement ce que j'appelle un projet !

Denis Bodor


**Hackable Magazine**
est édité par Les Éditions Diamond

LES ÉDITIONS DIAMOND
STIMULANT DE MATIÈRE GRISE

10, Place de la Cathédrale - 68000 Colmar -
Tél. : 03 67 10 00 20 – Fax : 03 67 10 00 21
E-mail : lecteurs@hackable.fr -
Service commercial : cial@ed-diamond.com
Sites : www.hackable.fr - www.ed-diamond.com
Directeur de publication : Arnaud Metzler
Rédacteur en chef : Denis Bodor
Réalisation graphique : Kathrin Scali
Responsable publicité : Tél. : 03 67 10 00 27
Service abonnement : Tél. : 03 67 10 00 20
Impression : pva, Landau, 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 : 09 53 15 21 77
IMPRIMÉ en Allemagne - PRINTED in Germany
Dépôt légal : À parution
N° ISSN : 2427-4631
Commission paritaire : K92470
Périodicité : trimestriel - Prix de vente : 12,90 €




Suivez-nous sur Twitter
@hackablemag

Alsace

**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.

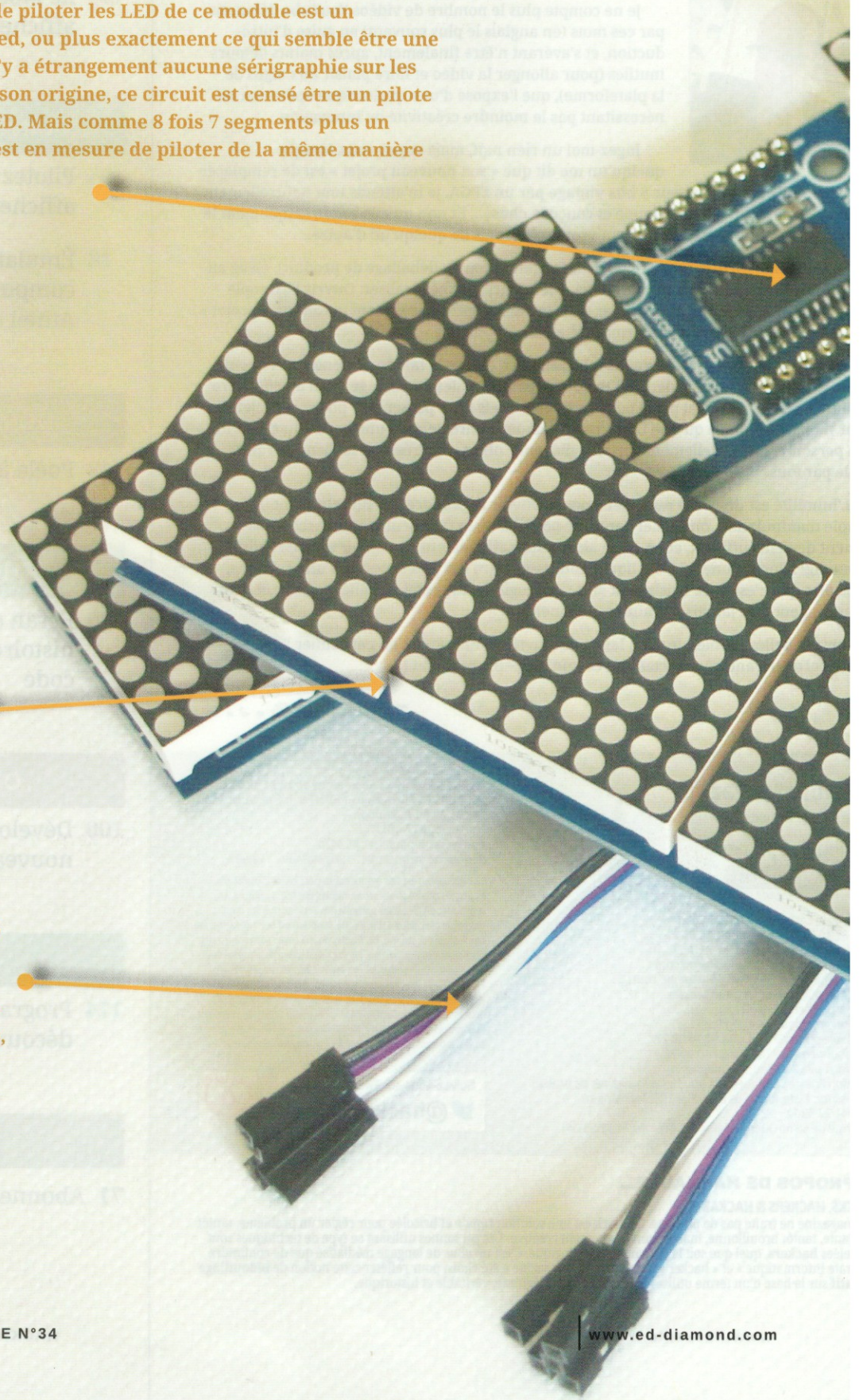


# SOMMAIRE

# LE MODULE DU MOMENT

**Le circuit intégré chargé de piloter les LED de ce module est un MAX7219 de Maxim Integrated, ou plus exactement ce qui semble être une copie compatible puisqu'il n'y a étrangement aucune sérigraphie sur les composants. Quelle que soit son origine, ce circuit est censé être un pilote d'afficheur huit chiffres à LED. Mais comme 8 fois 7 segments plus un point nous donne 64 LED, il est en mesure de piloter de la même manière une grille de 8 par 8 LED.**

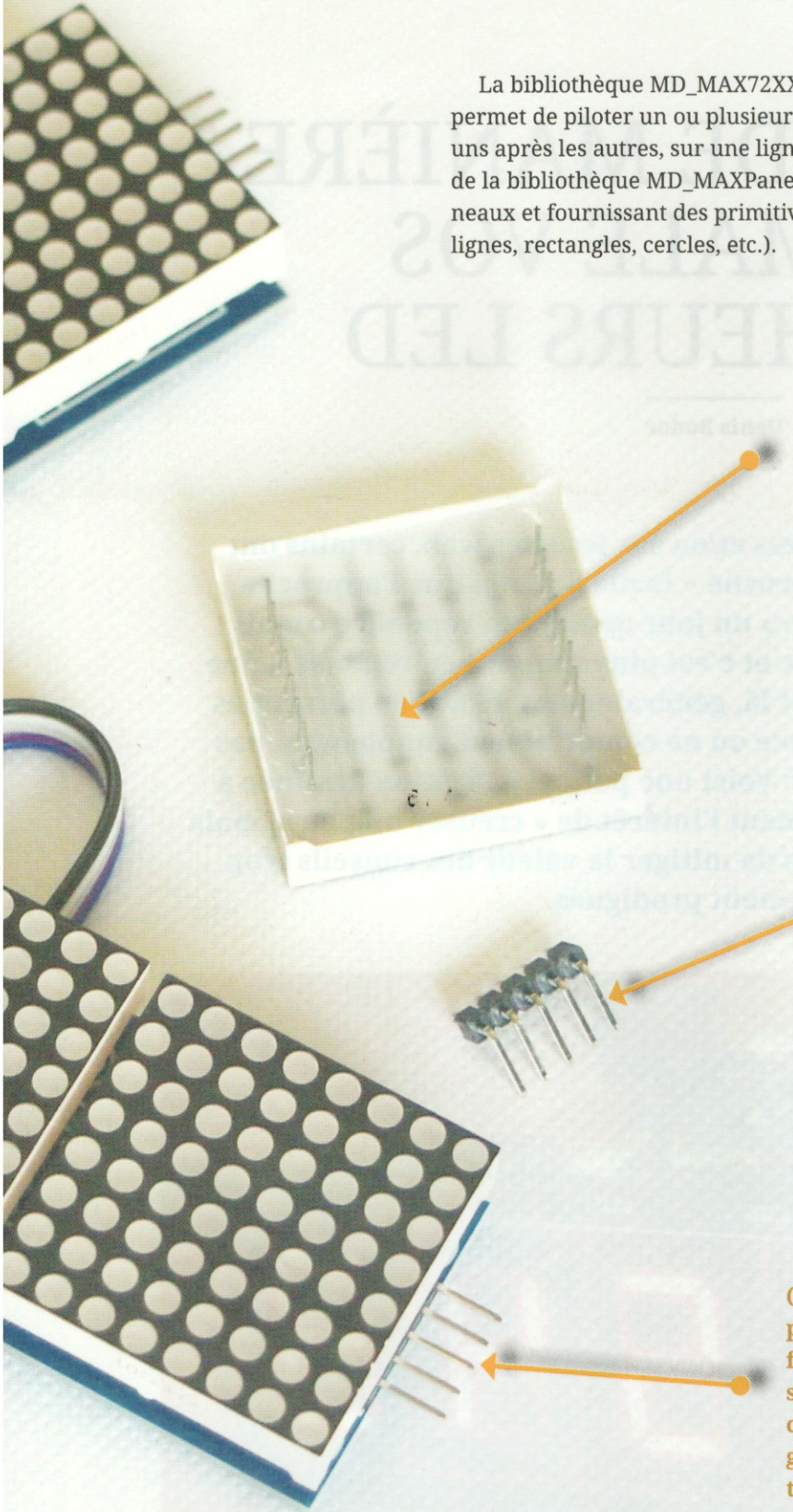
Seul bémol pour ce module, il peut arriver que l'espacement entre les blocs de 64 LED ne soit pas toujours identique et que les blocs eux-mêmes ne soient pas parfaitement parallèles. C'est un point de détail, mais qu'il est important de connaître, car cela peut être problématique selon le projet que vous aurez en tête.



**Chose relativement rare, en particulier pour cette catégorie de prix (moins de 5€ pour le modèle rouge chez alice1101983 sur eBay), les câbles Dupont femelle-femelle sont fournis, et le tout est livré proprement dans un emballage antistatique.**

# AFFICHEUR MATRICE LED 8X32

La bibliothèque MD_MAX72XX de Marco Colli (alias MajicDesigns) permet de piloter un ou plusieurs modules en configuration linéaire (les uns après les autres, sur une ligne), mais elle forme également la base de la bibliothèque MD_MAXPanel supportant des agencements en panneaux et fournissant des primitives graphiques intéressantes (points, lignes, rectangles, cercles, etc.).



**Les matrices de LED ne sont pas soudées sur le module et il sera donc possible de mélanger les couleurs par paquet de 64. Le produit se décline en version rouge, vert, bleu et jaune, mais rien ne vous empêche d'en commander plusieurs et de faire des permutations ensuite.**

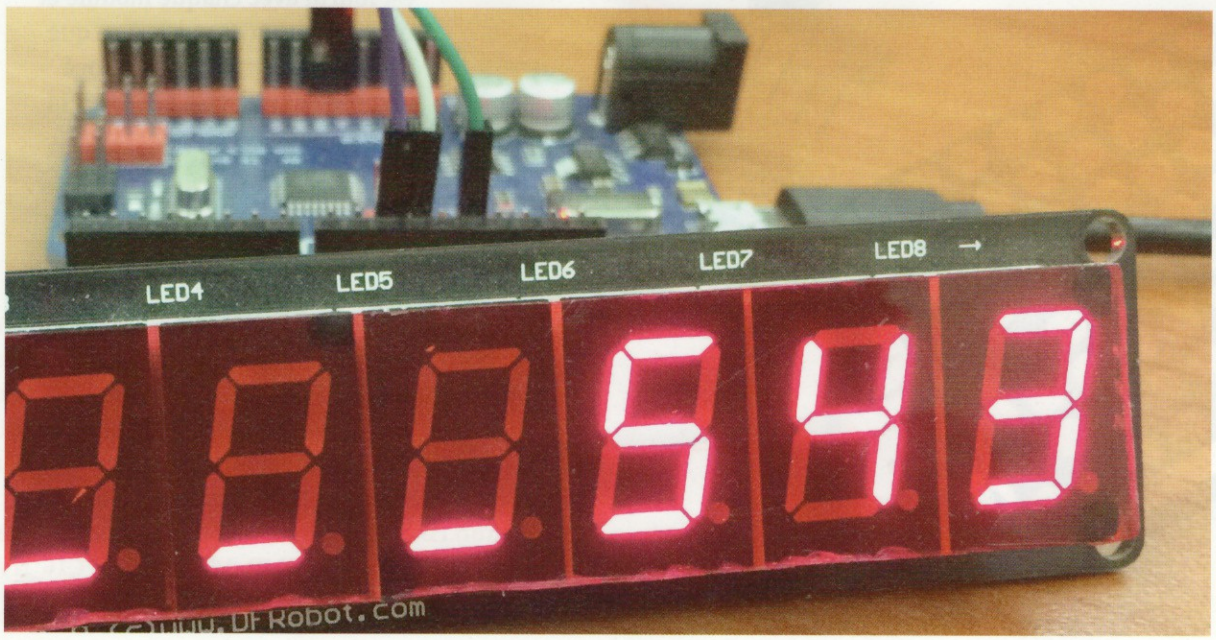
Les MAX7219 sont « chaînés » sur le circuit imprimé et il est possible de chaîner les modules eux-mêmes de façon à en piloter un ensemble. Un connecteur à souder supplémentaire (sortie) est fourni avec chaque module et il faudra prendre garde à ce que celui-ci ne se trouve pas coincé entre le circuit et les LED durant le transport.

**Un module se pilote avec seulement 5 connexions (Vcc, masse, Data in, CS et clock). Il semble exister plusieurs types de ces afficheurs, se déclinant en fonction de l'orientation des LED, mais fort heureusement, une bibliothèque est disponible prenant en charge parfaitement chaque modèle connu : https://github.com/MajicDesigns/MD_MAX72XX. Le modèle testé ici est le type « MD_MAX72XX::FC16_HW ».**

# PILOTEZ DE MANIÈRE OPTIMALE VOS AFFICHEURS LED

Denis Bodor

**Trop souvent, dans les forums et/ou sur les sites web, certains ont tendance à conseiller l'approche « facile » plutôt que l'approche « efficace ». Qui n'a jamais vu un jour quelqu'un répondre « mais utilises donc xxx(), ça marche et c'est plus simple » en réponse à une problématique précise ? C'est là, généralement, le fait de personnes qui n'ont que peu d'expérience ou ne comprennent simplement pas la motivation du demandeur. Voici une petite réalisation destinée à mettre en évidence non seulement l'intérêt de « creuser un peu », mais également une bonne raison de mitiger la valeur des conseils trop rapidement prodigués.**



Pilotez de manière optimale vos afficheurs LED

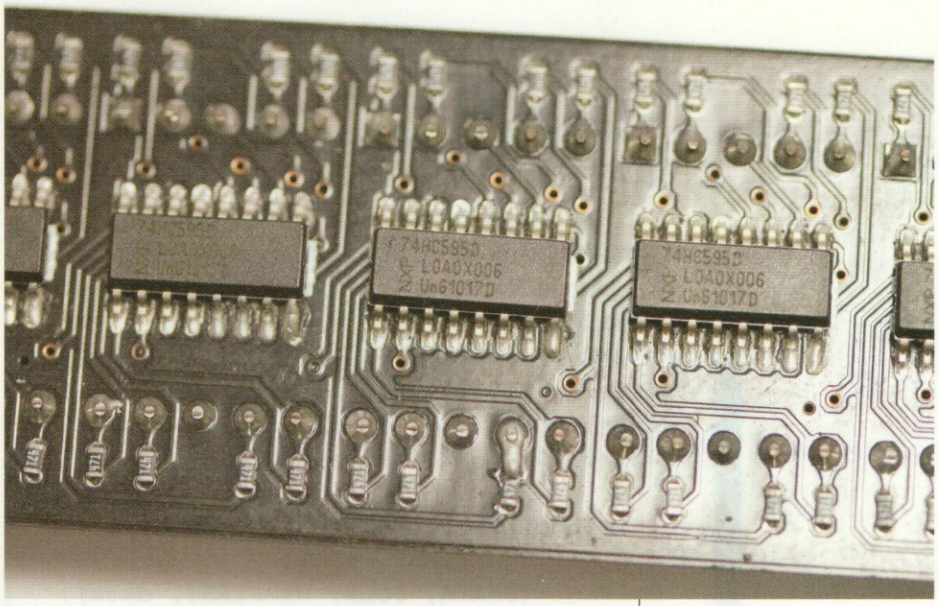
Notre problématique, pour cet article, sera de faire fonctionner un afficheur de conception relativement simple. Celui-ci, initialement vendu par DFRobot, se compose de huit afficheurs LED 7 segments, chacun piloté par un registre à décalage 8 bits 74HC595. Ces composants sont chaînés sur le circuit imprimé et forment ainsi un énorme registre de 64 bits, où chaque bit correspond à un segment de LED (7 barres du « 8 » plus le point en bas à droite).

L'ensemble est interfacé via cinq connexions :

- une masse ;
- une alimentation 5V ;
- une entrée série permettant de présenter un bit ;
- un signal d'horloge permettant de valider un bit ;
- et un *latch* permettant d'appliquer le contenu du registre aux sorties correspondantes.

Le 74HC595 dispose également d'une broche de *reset* pour réinitialiser le registre et une broche *output enable* dont l'état active/désactive les sorties (pratique pour utiliser la PWM afin de varier l'intensité lumineuse de l'ensemble des LED), mais celles-ci ne sont pas accessibles sur ce circuit.

Le tout est relativement classique et on retrouve ce style de conception dans d'autres modules disponibles un peu partout sur le Web, aussi bien pour contrôler des afficheurs 7 segments que pour du 16 segments ou même des ensembles de LED (barres, vu-mètre, etc.).




*Le registre à décalage 74HC595 est un classique. Il équipe ici le module d'affichage en huit exemplaires, mais c'est un circuit intégré utilisé de longue date pour toutes sortes d'usage. Ce type de circuit logique discret est en voie de disparition, remplacé par des puces plus avancées intégrant généralement une « intelligence » plus importante.*

## 1. PREMIER CROQUIS

Le pilotage du module est également très simple, puisqu'il suffit de présenter les bits correspondant à chaque segment de chaque afficheur les uns à la suite des autres, tout en les validant avec le signal d'horloge et en utilisant le signal *latch* pour pousser cela vers les sorties/LED. Pour afficher un premier chiffre sur l'afficheur, il suffit donc d'identifier quel bit, ou plutôt quelle position de chaque bit correspond à chaque segment. Après quelques tâtonnements et s'être rendu compte que les bits sont inversés (un 0 correspond à un segment allumé), nous pouvons établir une correspondance entre une série de 8 bits et un symbole dessiné correspondant à un chiffre. Nous pouvons alors résumer tout cela sous la forme d'un tableau :

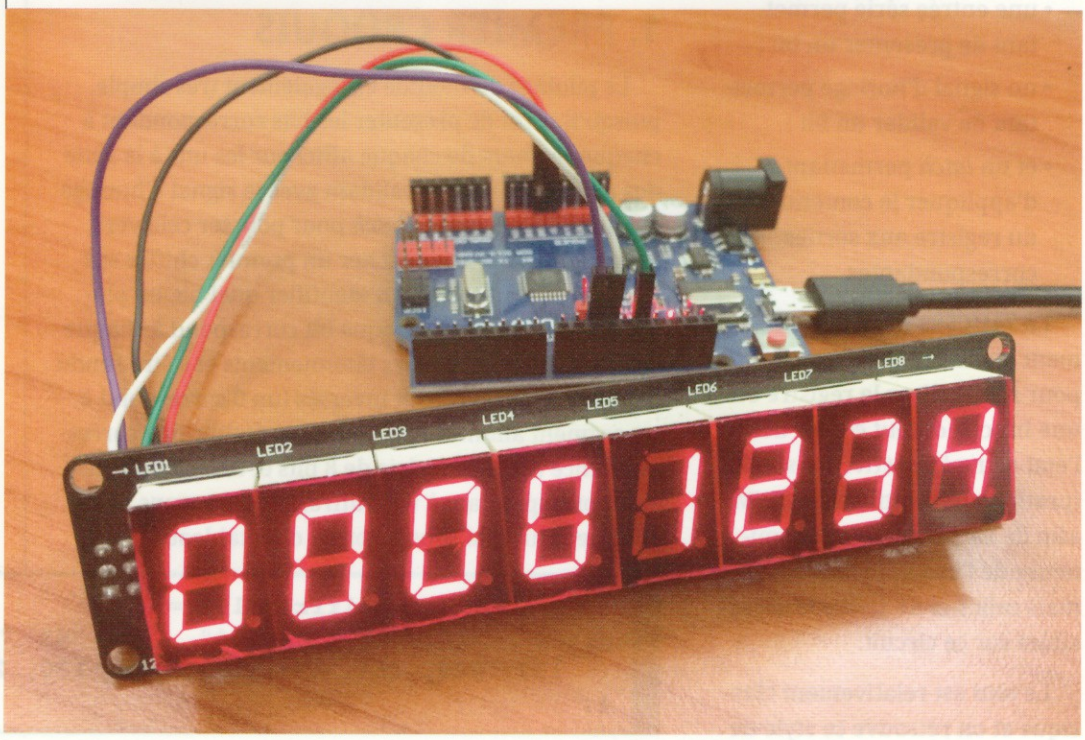
```
unsigned char chiffres[10] = {
// "0"  "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"
   192, 249, 164, 176, 153, 146, 130, 248, 128, 144
};
```

La position du chiffre correspond au nombre devant être affiché. Ainsi `chiffre[2]` contient `164` (ou `0xA4`) correspondant à l'activation des 5 segments dessinant un « 2 ».

Pour simplifier les choses, nous définissons trois macros utilisées en lieu et place des numéros des sorties à utiliser et nous pouvons alors créer une fonction pour envoyer un chiffre à l'afficheur :

```
#define PLAT  10  // latch
#define PCLK  13  // horloge
#define PDATA 11  // bit/data

void sendsingle(unsigned char data) {
  digitalWrite(PLAT, LOW);
  shiftOut(PDATA, PCLK, MSBFIRST, data);
  digitalWrite(PLAT, HIGH);
}
```

La broche *latch* « pousse » sur la sortie les bits contenus dans le registre à décalage sur le front montant (passage de l'état bas à l'état haut, autrement dit de la masse vers +5V). On passe donc cette sortie à `LOW` avant de fournir les 8 bits d'un chiffre via la fonction `shiftOut()` généralement recommandée, pour enfin repasser `PLAT` à `HIGH`. `shiftOut()` permet de « sérialiser » l'envoi de données. On précise en argument la broche à utiliser pour présenter les bits, la broche d'horloge, l'ordre des bits et l'octet à transmettre. `MSBFIRST` signifie que nous souhaitons envoyer le bit de poids fort en premier, celui le plus à gauche dans une représentation binaire d'une valeur (l'ordre inverse est spécifié par la macro `LSBFIRST`). Il n'y a pas de convention particulière à ce niveau, il faut simplement être cohérent dans l'ensemble du croquis (les valeurs dans `chiffres[]` sont en `MSBFIRST`).




*Obtenir un résultat avec ce type d'afficheur à LED est chose facile et rapide. Mais plutôt que de se reposer sur ses lauriers au premier essai réussi, c'est là une excellente opportunité pour prendre le temps d'optimiser son code.*

À présent que nous disposons d'une fonction qui s'occupe de dialoguer avec le matériel, nous pouvons procéder traditionnellement en ajoutant des couches d'abstraction. Nous avons une fonction qui envoie un chiffre, l'afficheur en a 8, créons donc la fonction qui envoie les 8 sur la base d'un nombre :

```
void sendnum(unsigned long val, int len) {
  if(len<=0)
    return;
  for(int i=0;i<len;i++)
    sendsingle(chiffres[(val%pow10(i+1))/pow10(i)]);
}
```

Cette fonction prend en argument une valeur entière, ainsi qu'une taille correspondant au nombre de chiffres à afficher. De manière générale, il vaut mieux éviter de prendre le temps de produire un code pour un cas spécifique qui nécessiterait une réécriture, si quelque chose change. Le module utilisé se compose de huit registres à décalages chaînés, mais il est possible d'également chaîner les modules eux-mêmes pour multiplier les afficheurs 7 segments, tout en n'ayant que 3 signaux de contrôle. Mieux vaut donc prévoir nos fonctions pour gérer un nombre quelconque de chiffres à afficher et non simplement 8. C'est pour cette raison que `len` est présent en argument, et parce que cela nous permet aussi de préfixer la valeur à afficher avec des zéros.

Remarquez la technique permettant d'extraire chaque chiffre du nombre en utilisant l'opérateur modulo et le `i` de la boucle `for` dans le calcul. Pour déduire les dizaines, les centaines, etc., de `i` nous élevons 10 à la puissance `i`, mais en utilisant une fonction développée spécifiquement :

```
long pow10(int expo) {
  unsigned long ret = 1;
  for(int i=1; i<=expo; i++) {
    ret = ret * 10;
  }
  return(ret);
}
```

Nous ne pouvons pas en effet utiliser `pow()`, car cette fonction retourne un `double` et non une valeur entière. Le problème qui se pose alors est celui de la précision lors de l'utilisation d'un `double` en `int`. Ainsi, *caster* `pow(10,3)` en `int` (avec `(int)pow(10,3)`) nous donne 999 et non 1000, comme espéré. De plus, éviter d'utiliser des fonctions mathématiques de ce type allégera notre croquis compilé.

Préfixer systématiquement le nombre affiché de zéros n'est pas toujours souhaitable, ne serait-ce que pour des raisons de lisibilité. « 00000453 » est parfois moins adapté qu'afficher simplement « 453 » aligné à droite. Cependant, ceci ne signifie pas, au regard de l'afficheur lui-même, que les trois chiffres n'ont pas de préfixe. Au contraire, il est impératif d'envoyer 5 symboles n'allumant aucune LED pour avoir ce résultat.

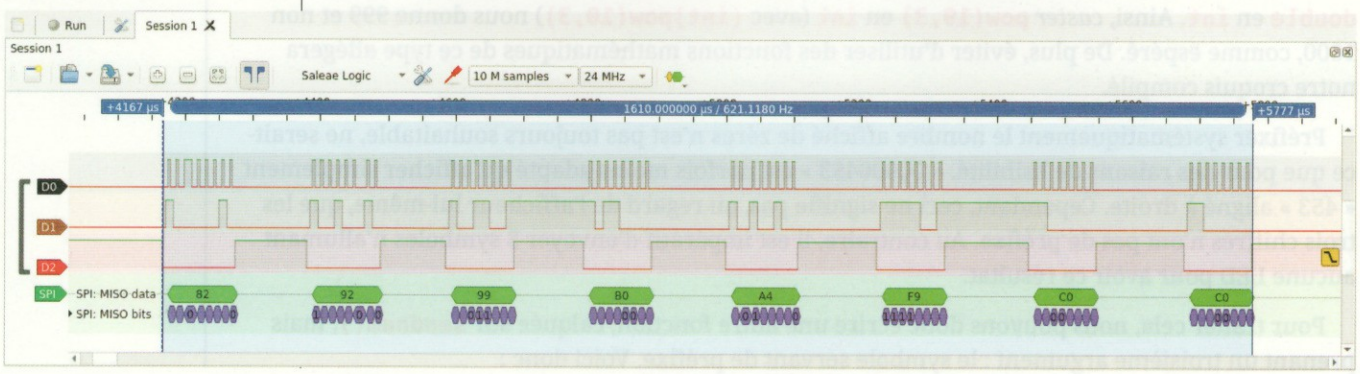
Pour traiter cela, nous pouvons donc écrire une autre fonction, calquée sur `sendnum()`, mais prenant un troisième argument : le symbole servant de préfixe. Voici donc :

```
void sendnum(long val, int len, unsigned char pref) {
  int nbr = 0;

  if(len<=0)
    return;

  while(val-pow10(nbr) > 0)
    nbr++;
  for(int i=0;i<len;i++) {
    if(i<nbr)
      sendsingle(chiffres[(val%pow10(i+1))/pow10(i)]);
    else
      sendsingle(pref);
  }
}
```

Remarquez que le nom de la fonction est la même que la précédente et c'est là toute la magie de la surcharge en C++. L'une ou l'autre fonction, identiquement nommée, sera appelée en fonction des arguments utilisés. Pour préfixer notre valeur sur 8 chiffres de « vide », il nous suffit alors d'appeler `sendnum(453,8,255)` et le tour est joué. Là encore, comme avec le nombre de chiffres à gérer, nous nous laissons la porte ouverte à d'autres éventualités, comme préfixer avec un autre symbole comme un point ou un trait de soulignement, par exemple.

Il ne nous reste plus maintenant qu'à assembler tout cela directement dans la fonction `setup()` (`loop()` étant vide) :

```
void setup() {
  digitalWrite(PLAT, HIGH);
  pinMode(PLAT, OUTPUT);
  pinMode(PDATA, OUTPUT);
  pinMode(PCLK, OUTPUT);

  sendnum(1234567,8);
}
```




*Un analyseur logique, ou plus exactement une suite d'analyse de signal électronique comme Sigrok vous permet relativement simplement et à peu de frais d'observer les données qui circulent entre la carte Arduino et le module d'affichage. Nous pouvons voir ici que le délai entre les envois de 8 bits ralentit énormément la communication.*

## 2. ON MESURE PLUSIEURS FOIS, MAIS ON NE COUPE QU'UNE SEULE

Ou plus exactement, on *clock* plusieurs fois, mais on ne *latch* qu'une seule !

En effet, pourquoi envoyer les bits par paquet de 8 ? Il ne s'agit pas vraiment d'octets et il n'existe aucun protocole nous imposant cet état de fait. Nous pouvons parfaitement envoyer 64 bits et ne pas activer le *latch* entre chaque paquet de 8 bits. Nous gagnerons certainement du temps en procédant de cette manière, mais notre fonction `sendsingle()` envoyant un chiffre n'est plus du tout valable. Il nous faut gérer cela directement dans une fonction envoyant un paquet de 64 bits, et recevant donc en argument les 8 chiffres. Mais rappelez-vous, nous avons créé le premier croquis en songeant qu'il était possible de chaîner deux modules ou plus. Les 8 chiffres pourraient donc être au nombre de 16, 24, 32 ou pourquoi pas 128 (ce qui soulève un autre problème et suggère déjà une troisième évolution) !

La fonction `sendsingle()` doit donc être réécrite et son nom changé par la même occasion. J'en profite ici pour vous confirmer que, oui, j'aime mes noms de fonction en anglais et, contrairement à ce dont un lecteur a osé nous accuser un jour, non, ce n'est pas parce que nous utilisons des articles traduits. Tous nos articles sont originaux, mais je trouve normal en écrivant dans un langage mondialement utilisé de généralement nommer mes fonctions dans une langue tout aussi mondialement utilisée, surtout dans le domaine informatique. Si Arduino utilisait le langage de Windev (sic) peut-être que je raisonnerai différemment, mais ce n'est pas le cas, heureusement.

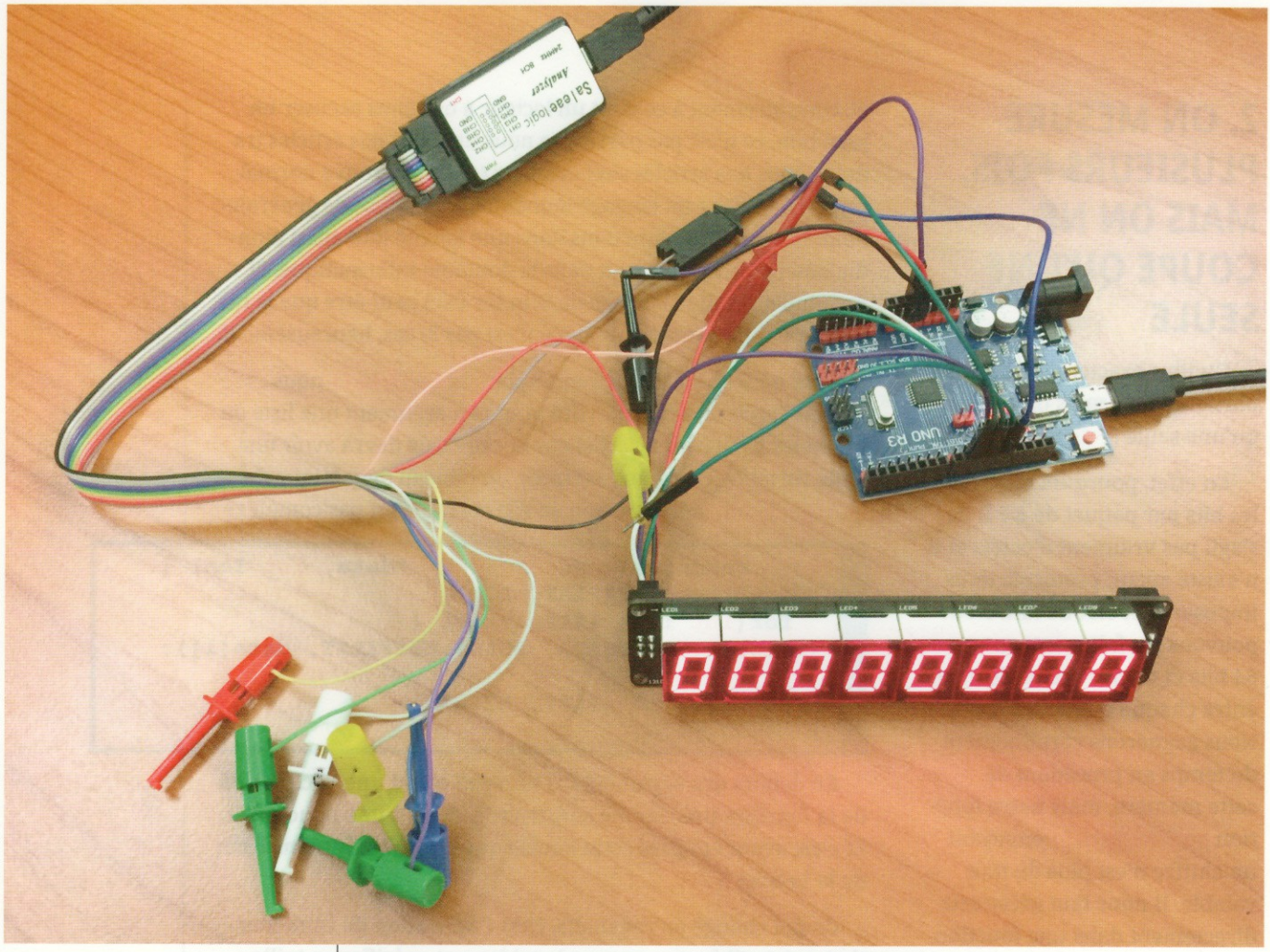
Quoi qu'il en soit, pour remplacer `sendsingle()` nous devons écrire une fonction qui prend en argument une liste de symboles à envoyer à l'afficheur et cette liste devra avoir une taille variable. En d'autres termes, un tableau dynamique de `unsigned char`. La fonction en elle-même n'est pas complexe :

```
void sendnsym(unsigned char *data, int len) {
  digitalWrite(PLAT, LOW);
  for(int i=0; i<len; i++) {
    shiftOut(PDATA, PCLK, MSBFIRST, data[i]);
  }
  digitalWrite(PLAT, HIGH);
}
```

En dehors des arguments, il ne s'agit que d'une déclinaison de la fonction précédente construite autour d'une boucle `for`. Et c'est avant et après cette boucle que nous manipulons l'état de la broche `PLAT`.

Ce qui change en revanche, c'est l'utilisation de cette fonction puisqu'il faut composer le tableau et spécifier sa taille avant de l'appeler. Ceci impacte alors notre `sendnum()` qu'il faut réécrire ainsi :

```
void sendnum(unsigned long val, int len) {
  unsigned char *data = NULL;

  if(len<=0)
    return;

  if((data = malloc(len)) == NULL) {
    Serial.println("Erreur malloc!");
    return;
  }

  for(int i=0;i<len;i++)
    data[i]=chiffres[(val%pow10(i+1))/pow10(i)];

  sendnsym(data,len);
  free(data);
}
```



*L'utilisation d'un analyseur logique finit généralement ainsi : en salade de câbles. Cependant, cet effort permet de vérifier et surtout mesurer l'efficacité de son code. Un moindre mal lorsqu'on sait que ce genre d'équipement, basé sur un Cypress CY7C68013A et compatible avec la suite open source Sigrok, ne vous coûtera qu'une dizaine d'euros.*

Dans les grandes lignes, c'est la même chose, mais nous complétons `data[]` à chaque tour dans la boucle plutôt que d'appeler `sendsingle()`. Cependant, pour utiliser ce tableau, nous devons tout d'abord le créer.

Deux solutions s'offrent à nous :

- Avoir un tableau de taille arbitrairement choisie et utiliser `len` (une longueur) pour préciser la taille des données utiles dans ce tableau.
- Créer le tableau de la taille des données utiles et utiliser `len` pour la préciser.

Dans le premier cas, quelle que soit la taille choisie, nous avons soit un tableau trop petit pour supporter une éventuelle démultiplication des modules d'affichage, soit un tableau trop grand et occupant inutilement de la mémoire. C'est pour cette raison que j'ai préféré choisir l'allocation dynamique.

À l'entrée de la fonction, `data` n'est qu'un pointeur nul, il occupe la taille d'un pointeur sur des `unsigned char`. On utilise ensuite `malloc()` pour allouer un morceau de mémoire de la taille dont nous avons besoin et cette fonction nous retourne un pointeur vers cette espace. Si `malloc()` retourne `NULL`, c'est que quelque chose n'aura pas fonctionné.

Nous pouvons alors remplir le tableau, appeler `sendnsym()` puis libérer la mémoire. Ainsi, lorsque nous utilisons la fonction avec 8 afficheurs, nous occupons 8 octets, mais en chaînant trois modules, nous en occuperions 24. Souple, efficace, modulaire, adaptable...

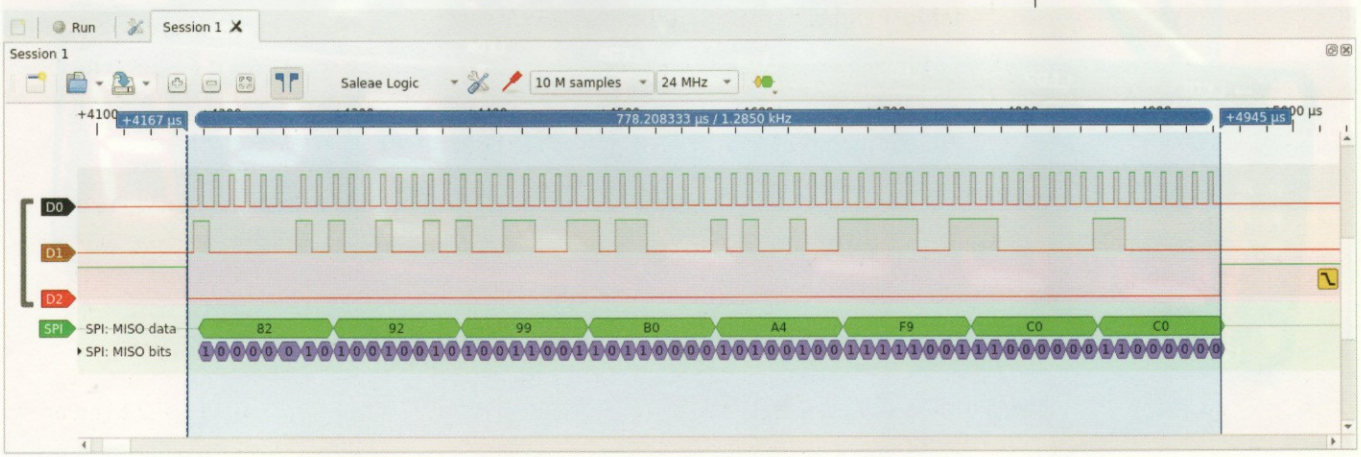
Bien entendu, l'autre version de `sendnum()` est également modifiée en conséquence :

```
void sendnum(long val, int len, unsigned char pref) {
  unsigned char *data = NULL;
  int nbr = 0;

  if(len<=0)
    return;
  if((data = malloc(len)) == NULL) {
    Serial.println("Erreur malloc!");
    return;
  }

  while(val-pow10(nbr) > 0)
    nbr++;

  for(int i=0;i<len;i++) {
    if(i<nbr)
      data[i]=chiffres[(val%pow10(i+1))/pow10(i)];
    else
      data[i]=pref;
  }
  sendnsym(data,len);
  free(data);
}
```

Notre `setup()` ne change pas.

Si nous procédons à quelques tests avec un analyseur logique, comme le merveilleux et open source Sigrok/PulseView, le fruit de notre effort est directement mesurable. La première version du croquis procède à l'envoi des 64 bits des 8 symboles en 1610 µs. Presque autant de temps est utilisé à laisser `PLAT` à l'état bas qu'à l'état haut entre deux salves de 8 bits.

*En supprimant la manipulation du signal latch à chaque octet, nous arrivons à diviser par deux le temps de transmission d'un groupe de 8 chiffres. Quelque 700 µs peuvent paraître une durée relativement courte, mais il est important de penser aux évolutions possibles. Augmentez le nombre de modules et vous arriverez rapidement à une durée qui sera perceptible visuellement.*



Le simple fait de n'utiliser un changement d'état de **PLAT** qu'en début et fin de la transmission des 64 bits nous fait gagner énormément de temps. La transaction complète passe à 778 µs. Ceci peut sembler anodin, mais imaginez-vous piloter 10 modules chaînés et vous voilà en train de comparer 16 ms et 7 ms, une différence qui sera clairement visible à l'œil nu lors d'un rafraîchissement des valeurs.

## 3. « POURQUOI UTILISER SPI PUISQU'IL Y A SHIFTOUT() ? »

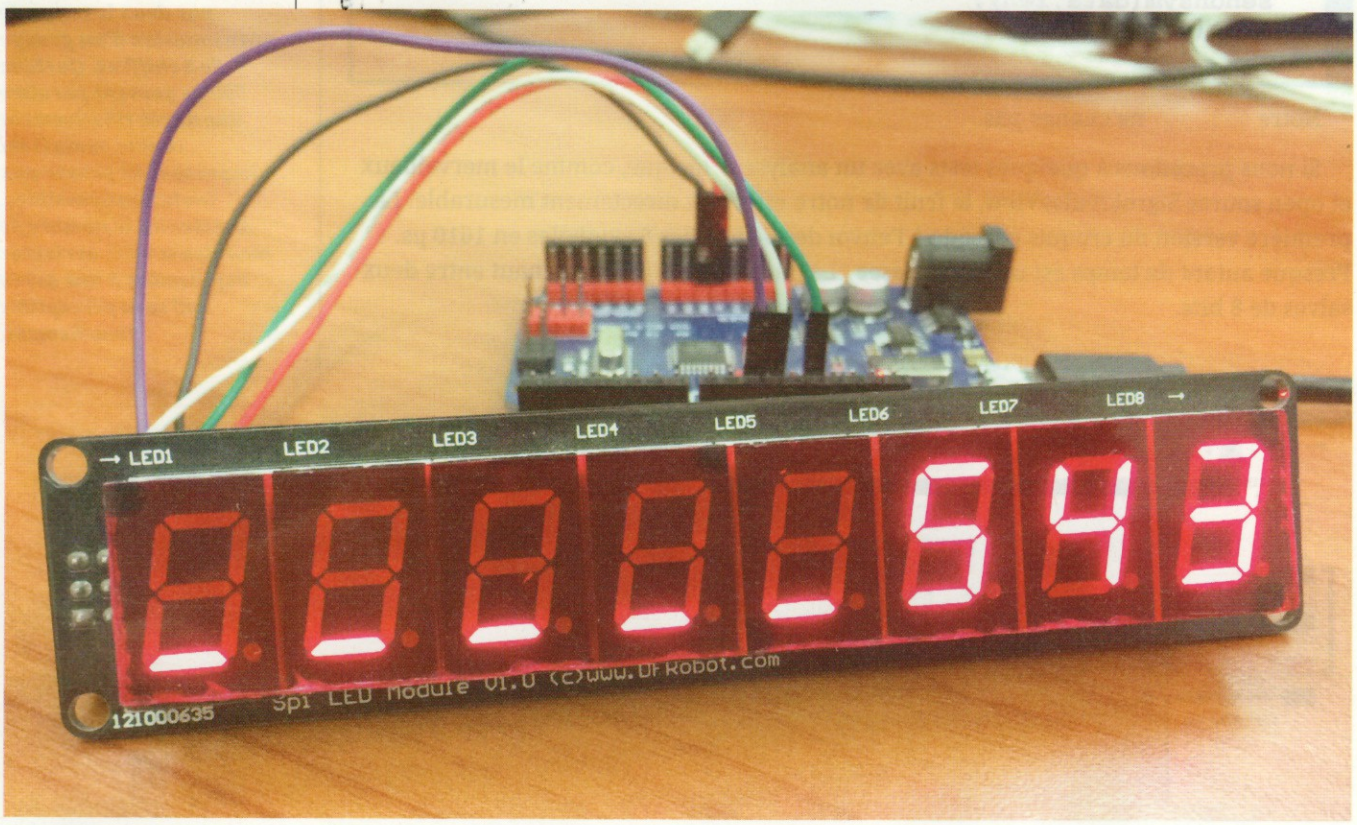
À chaque fois que j'ai affaire avec une fonctionnalité ou un protocole qui ressemble à un autre, en particulier matériellement pris en charge par un microcontrôleur, je me demande s'il n'est pas possible de gérer la chose ainsi. Et c'est exactement le cas ici.

Devinette : qu'est ce qui utilise au moins un signal d'horloge, un signal de donnée d'un maître vers un esclave et valide l'asservissement avec un signal en logique inversée ? Réponse : SPI (entre autres).

En effet, en faisant abstraction du signal MISO de l'esclave vers le maître, nous avons précisément ici une configuration SPI :

- **PCLK** est le signal d'horloge géré par le maître, CLK ;
- **PDATA** est MOSI ;
- et **PLAT** n'est autre que CS.

Cela tombe fort bien, une carte Arduino comme la UNO dispose d'un microcontrôleur




*Le rendu de ce genre d'afficheur 7 segments à LED est généralement grandement amélioré en ajoutant un filtre de couleur destiné à masquer le blanc des segments éteints. Pour le rouge, inutile de chercher bien loin, le plastique d'une boîte de « Mon Chéri » fait parfaitement l'affaire avec, en prime, la parfaite excuse pour manger tous les chocolats.*

qui sait parfaitement gérer le protocole SPI nativement, directement sur le silicium. Et comble de la coïncidence (suis-je vraiment crédible ?), nous utilisons déjà les bonnes broches puisque `PCLK` = CLK = 13, `PDATA` = MOSI = 11 et `PLAT` = CS = 10. Il nous est donc relativement facile d'adapter notre croquis pour utiliser la fonctionnalité SPI en lieu et place de `shiftOut()`.

Pourquoi, alors que cette fonction est précisément faite pour cela, que c'est plus simple et que tout est déjà fait ? La réponse arrivera le moment venu, mais pour l'instant, ajoutons ce qu'il nous faut, à commencer par l'inclusion de l'en-tête adéquate :

```
#include <SPI.h>
```

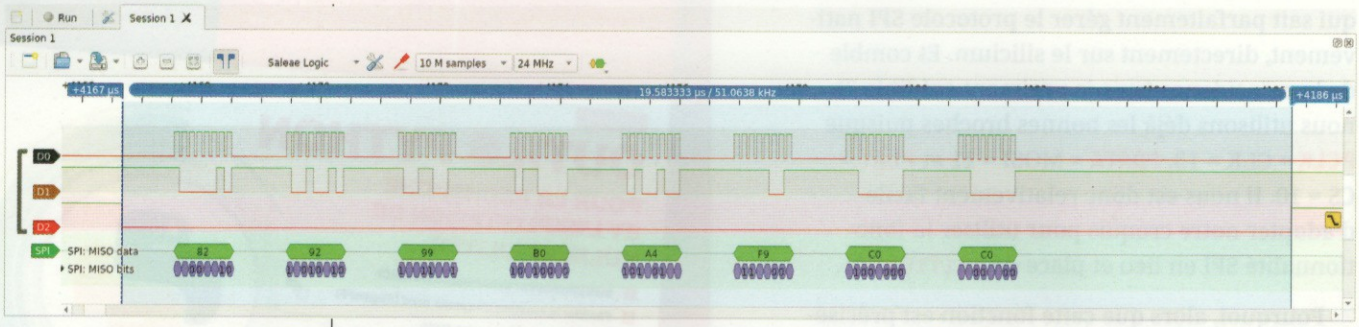
Nous avons maintenant à disposition de quoi configurer le bus SPI dans `setup()` :

```
SPI.beginTransaction(
  SPISettings(20000000,
MSBFIRST, SPI_MODE0)
);
```

Vous reconnaissez sans doute `MSBFIRST` qui a ici le même sens que précédemment. Le `20000000` correspond à la fréquence d'horloge souhaitée pour envoyer les bits (ici, le maximum) et `SPI_MODE0` fait référence à la polarité et à la phase du signal d'horloge. Le mode SPI 0 correspond précisément à ce que nous avons fait précédemment avec `shiftOut()`.

Les méthodes `beginTransaction()` et `endTransaction()` permettent de signifier respectivement le début de l'utilisation du bus SPI et sa fin. Ceci est généralement utilisé pour permettre à d'autres fonctions (bibliothèques) d'utiliser le bus à leur tour et à leur façon, sans qu'il y ait de collision. Ceci n'a pas d'importance ici, il n'y aura donc pas d'appel à `endTransaction()`. Mais si vous faites de votre code une bibliothèque, pensez-y.

Le bus étant configuré, nous n'avons plus qu'à adapter notre `sendnsym()` :




*Voici, en une image, l'argument ultime au bénéfice de l'utilisation du bus SPI face à celle de shiftOut(). Avec un temps de transmission de moins de 20 µs pour 64 bits, il devient possible de « penser plus grand », voire d'envisager des choses plus atypiques comme... faire de la PWM en envoyant des bits ?*

```
void sendnsym(unsigned char *data, int len) {
  digitalWrite(PLAT, LOW);
  for(int i=0; i<len; i++) {
    SPI.transfer(data[i]);
  }
  digitalWrite(PLAT, HIGH);
}
```

Vous voyez que cela n'a pas été bien compliqué, mais le bénéfice à la clé est impressionnant. Nous avions seulement divisé par deux le temps de transmission en utilisant plus judicieusement `PLAT`, mais cette nouvelle modification fait tomber le temps complet pour 8 chiffres à... 19,58 µs, soit une transmission complète 40 fois moins longue !

## CONCLUSION

Moralité de l'histoire, méfiez-vous donc des conseilleurs qui ne sont pas les payeurs, lorsqu'ils donnent de faux bons conseils et vous incitent à limiter vos recherches, prétextant que telle ou telle fonction « est précisément faite pour cela ». Ce n'est pas parce qu'une fonction répond à **un** besoin générique qu'elle répondra forcément à **votre** besoin précis. À titre d'exemple, les bibliothèques hautes performances comme FastLED permettant de piloter les LED adressables type WS2812b reposent (ESP32) sur le bus I2S (*Integrated Interchip Sound*) destiné initialement à la communication entre un support audio numérique et un DAC. Pas vraiment quelque chose de « fait pour cela »... N'est-ce pas ?

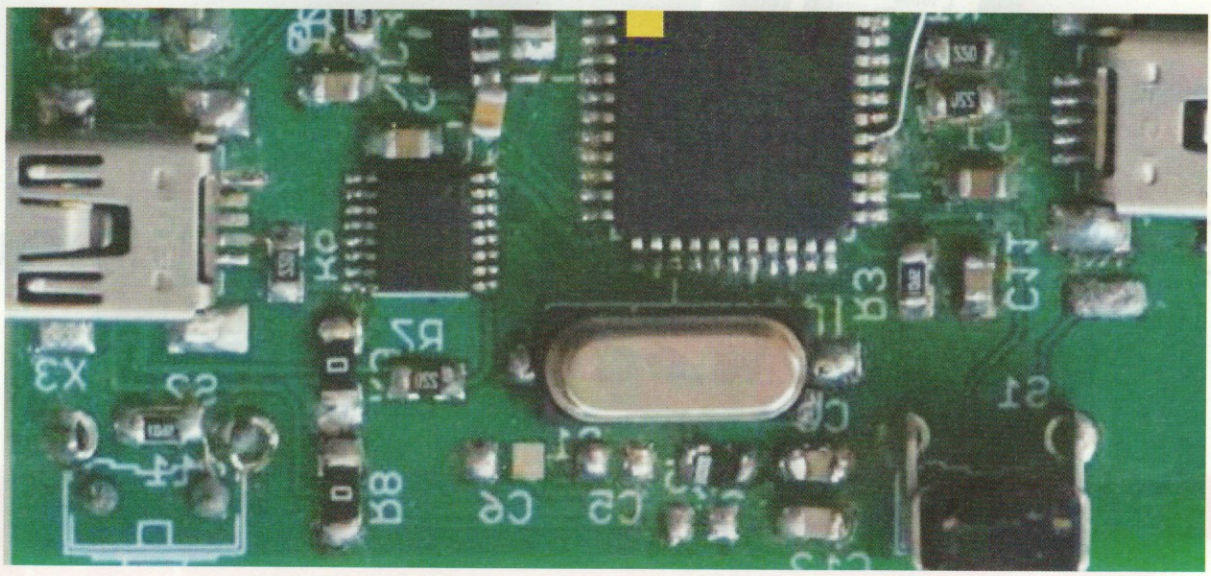
Notez que ce croquis, de pure démonstration, est loin de satisfaire toutes les possibilités offertes par un tel module d'affichage. Il serait relativement simple de prendre en charge les valeurs négatives par exemple, ou encore de décliner `sendnum()` une nouvelle fois afin de gérer des valeurs en virgule flottante. Mais tel n'était pas l'objet de l'article et ceci vous sera donc cordialement laissé en exercice, si le cœur vous en dit...

Le point important à ne pas oublier ici se résume à : pour optimiser, il faut oser expérimenter hors des sentiers battus ! **DB**

# ÉMULATION D'UN CIRCUIT COMPORTANT UN PROCESSEUR ATMEL AVEC SIMAVR

J.-M Friedt, Enseignant-chercheur à l'Université de Franche-Comté à Besançon

**Il existe de nombreux cas où le matériel n'est pas disponible pour développer un système embarqué, que ce soit parce que la carte commandée n'a pas encore été livrée, parce que le collègue chargé de la conception du circuit imprimé a fait une erreur ou est en retard, ou parce qu'un virus interdit l'accès aux salles de travaux pratiques de l'Université (Fig. 1). Pour toutes ces raisons, nous désirons appréhender le développement d'un système embarqué sur un émulateur, c'est-à-dire un logiciel capable de fournir une représentation fidèle du comportement du dispositif réel, incluant ses latences et temporisations.**



Nous avions proposé **[1]** l'utilisation d'émulateurs pour appréhender le développement du code embarqué (firmware) sur divers microcontrôleurs, de la petite architecture 8 bits de l'Atmega de Atmel (maintenant Microchip) aux gros ARM, RISC-V et MIPS capables d'exécuter GNU/Linux. Dans tous ces cas, nous nous étions arrêtés au firmware, sans considérer l'émulation du circuit imprimé qui entoure le microcontrôleur et lui injecte potentiellement des stimuli, que ce soit sous forme de signaux binaires (GPIO pour General Purpose Input Output) ou messages (série sur bus USART ou parallèle sur un port de GPIO). Nous allons combler cette lacune en appréhendant l'émulateur de petits microcontrôleurs de la gamme Atmega `simavr` non plus au niveau du cœur du processeur, mais des périphériques qui l'entourent. En particulier, nous constaterons que l'utilisation des interruptions devient naturelle dès qu'il y a interaction du microcontrôleur avec son environnement. Par ailleurs, nous verrons que l'accès au temps émulateur – et non pas au temps du système d'exploitation de l'hôte sur lequel s'exécute l'émulateur – permet d'injecter des signaux avec une grande précision temporelle dans le *référentiel de l'émulateur*, une propriété fondamentale pour résoudre le problème qui nous intéressera, à savoir l'asservissement de l'oscillateur cadençant le microcontrôleur sur la référence supposée exacte qu'est le signal de temporisation 1-PPS issu d'un récepteur GPS (virtuel) **[2]**.

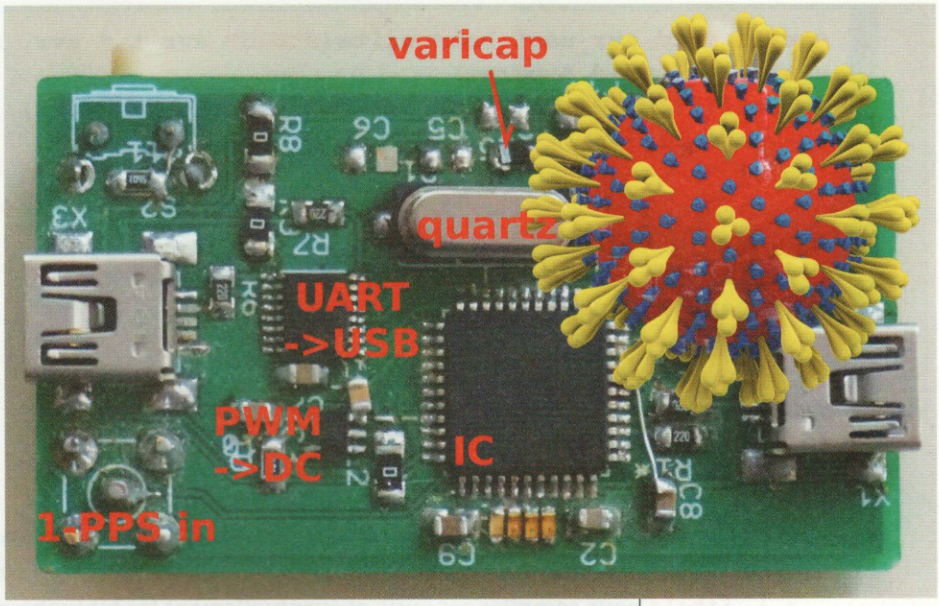



*Figure 1 : Circuit de contrôle d'un oscillateur à quartz sur GPS. Contaminé par un virus, il nous a fallu recourir à une solution d'émulation pour pallier l'absence de matériel pour achever le développement logiciel.*

Nous supposerons que le lecteur, travaillant sous GNU/Linux, a acquis les sources de `simavr` à **https://github.com/buserror/simavr** et l'aura compilé par `make` dans sa racine après les quelques modifications suivantes, qui seront l'occasion de voir que les sources de l'émulateur sont limpides à analyser. Afin de nous faciliter la visualisation de l'état des ports, nous complétons la fonction `avr_ioport_write()` de `simavr/sim/avr_ioport.c` avec :

```
static void avr_ioport_write(struct avr_t * avr, avr_io_addr_t addr, uint8_t v,
void * param)
{printf("\nSIMAVR: IOPORT @0x%x<-0x%x\n",addr,v);fflush(stdout);
 avr_ioport_t * p = (avr_ioport_t *)param;
 [...]
```

et de la même façon nous pouvons observer l'initialisation du port série dans `simavr/sim/avr_uart.c` en complétant :

```
static void avr_uart_baud_write(struct avr_t * avr, avr_io_addr_t addr,
uint8_t v, void * param)
{printf("SIMAVR: BAUDRATE 0x%x\n",v);fflush(stdout);
 p->io = _io;
 [...]

void avr_uart_reset(struct avr_io_t *io)
{printf("\nSIMAVR: UART RESET\n");fflush(stdout);
 avr_uart_t * p = (avr_uart_t *)io;
 [...]

void avr_uart_init(avr_t * avr, avr_uart_t * p)
{printf("\nSIMAVR: UART INIT\n");fflush(stdout);
 p->io = _io;
 [...]
```

Aucune subtilité de compilation, si ce n'est d'avoir installé la version de développement de **libelf** pour compiler cet outil, et éventuellement les bibliothèques OpenGL pour profiter des interfaces graphiques des exemples. La compilation des *firmware* des exemples de **simavr** impose d'avoir installé le cross-compilateur **avr-gcc** (paquet **gcc-avr** dans Debian GNU/Linux avec sa dépendance **avr-libc** qui peut être pratique).

## 1. GPIO

Le cas du GPIO en sortie ne présente à peu près aucun intérêt puisque **simavr** affiche l'état des registres manipulés, y compris ceux chargés de définir le statut des ports d'entrée-sortie, mais fournit l'opportunité d'introduire l'arborescence du projet, le fichier de définition d'un circuit imprimé et les nomenclatures à respecter pour exploiter les **Makefile** de **simavr**.

Soit le programme trivial suivant pour faire clignoter périodiquement une LED connectée à la broche 5 du port B :

```
#define F_CPU 16000000UL
#include <avr/io.h>
#include <util/delay.h>  // _delay_ms

int main(void)
{DDRB |=1<<PORTB5;
 PORTB |= 1<<PORTB5;
 while(1)
  {PORTB ^= (1<<PORTB5);
   _delay_ms(200);
  }
}
```

qui se compile avec `avr-gcc` par `avr-gcc -mmcu=atmega32u4 -O2 prog.c` en supposant que le code source est stocké dans le fichier `prog.c` et s'exécute sur un Atmega32U4. Ce programme définit le bit 5 du port B en sortie (DDRB) et fait clignoter la LED connectée à ce port (PORTB) avec une période de 400 ms. Nous n'avons pas besoin de rechercher dans la *datasheet* l'emplacement des ports, le fichier d'en-tête `io.h` et ses dérivés dans `/usr/lib/avr/include/avr` définissant un certain nombre de constantes en cohérence avec la nomenclature d'Atmel dans ses documentations.

Cependant, en tentant d'exécuter ce programme dans `simavr` par `simavr/run_avr a.out`, nous nous faisons insulter : la nature du processeur n'a pas été renseignée, `avr_make_mcu_by_name: AVR '' not known`, car `simavr` ne sait pas quel cœur émuler. Nous pouvons tricher en explicitant ces paramètres par `run_avr -m atmega32u4 -f 16000000`, mais ce n'est pas l'approche sélectionnée dans cet article, qui visera à décrire un circuit imprimé équipé d'un processeur de modèle connu.

Que faut-il ajouter pour exécuter ce fichier depuis un circuit imprimé embarquant le microcontrôleur et décrit selon les préceptes de `simavr` ? Premièrement, si le circuit imprimé est décrit dans le fichier `intro.c` – et ce programme sera exécuté sur le processeur exécutant `simavr`, donc probablement un processeur compatible Intel x86 et non pas une architecture AVR – alors la structure des `Makefiles` des exemples de `simavr` impose d'appeler le firmware correspondant – qui lui s'exécutera sur le microcontrôleur – avec un nom de la `forme atXXX_intro.c`. Nous constatons ceci en lisant un des `Makefile` des exemples contenus dans `examples/board*` et en observant la règle `firm_src = ${wildcard at*${target}.c}` avec `target= intro` dans ce cas. Ainsi, le nom `firm_src` du logiciel embarqué se déduit de target préfixé de `at` et, pour faciliter la lecture, le nom du processeur cible. Nous reprenons donc notre `prog.c` pour le renommer `atmega32u4_intro.c` : contrairement à l'habitude sous UNIX, cette fois les noms de fichiers ont une importance (on verra que ce n'est pas la dernière fois). Ensuite, il faut compléter le programme avec quelques définitions qui seront exploitées par le compilateur pour savoir quel processeur appréhender : ces instructions n'affecteront *pas* le firmware flashé à terme dans le vrai microcontrôleur, mais sont contenues dans l'exécutable au format ELF fourni à `simavr`. Nous allons préciser que nous travaillons sur Atmega32U4 en ajoutant, après les en-têtes, les lignes :

```
#include "avr_mcu_section.h"
AVR_MCU(F_CPU, "atmega32u4");
```

Ce programme se compile toujours de la même façon en ajoutant (option `-I`) le répertoire de `simavr` contenant le fichier d'en-tête, soit :

```
avr-gcc -mmcu=atmega32u4 -O2 -Isimavr/sim/avr atmega32u4_intro.c
```

et s'exécute de la même façon, en faisant appel à `simavr` au travers de la commande :

```
simavr/run_avr a.out
```

pour cette fois convenablement s'exécuter et indiquer :

```
Loaded 232 .text at address 0x0
Loaded 0 .data
SIMAVR: UART INIT
SIMAVR: UART RESET
SIMAVR: IOPORT @0x25<-0x20
SIMAVR: IOPORT @0x25<-0x0
SIMAVR: IOPORT @0x25<-0x20
```

Ça y est, la simulation est bien partie, mais est toujours faite en explicitant **run_avr** et non pas en passant par la description d'un circuit imprimé.

Nous rédigeons donc un fichier **intro.c** qui contient :

```
#include <stdlib.h>
#include <stdio.h>
#include "sim_elf.h"

int main(int argc, char *argv[])
{avr_t * avr = NULL;
 elf_firmware_t f;
 const char * fname =  "atmega32u4_intro.axf"; // a.out if compiled manually
 elf_read_firmware(fname, &f);
 printf("fw %s f=%d mmcu=%s\n", fname, (int)f.frequency, f.mmcu);
 avr = avr_make_mcu_by_name(f.mmcu);
 if (!avr) {fprintf(stderr, "%s: AVR '%s' not known\n", argv[0],
f.mmcu);exit(1);}
 avr_init(avr);
 avr_load_firmware(avr, &f);
 while (1) {avr_run(avr);}
}
```

qui décrit le circuit imprimé... qui n'est ici fait de rien d'autre qu'un microcontrôleur dont le firmware est chargé après avoir été lu au format ELF. Nous avons choisi de conserver la nomenclature par défaut des makefiles proposés dans **examples/board*** de nommer le fichier ELF du nom du code source avec l'extension **axf**, mais il est tout à fait envisageable de compiler à la main et fournir a.out comme nom d'exécutable.

La simulation s'exécute dans la boucle infinie faisant appel à **avr_run()** : cela sera important pour la suite de la discussion et la génération des stimuli.

Évidemment, nous désirons automatiser la compilation par **Makefile** : partant de l'exemple fourni dans **examples/board_ledramp** que nous copions dans le répertoire de travail, nous modifions le nom de target et tentons de compiler par **make**.

Tout ceci étant fait, nous émulons le *circuit imprimé* (sans appeler explicitement run_avr) par **./obj-x86_64-linux-gnu/intro.elf** qui donne le même résultat que précédemment.

Ici, nous devons passer par une étape fort peu élégante pour contourner un dysfonctionnement du **Makefile**. Soit nous créons à la main le répertoire **obj-x86_64-linux-gnu**, et dans ce cas **make** est satisfait, soit nous remontons au sommet de l'arborescence de **simavr**, nous nettoyons le projet par **make clean**, et nous le recréons par **make** qui fabrique le répertoire manquant dans notre exemple *avant* de peupler le répertoire examples/parts, cause de l'erreur **fatal error: opening dependency file obj-x86_64-linux-gnu/intro.d: No such file...**.

Dans ce second cas, nous aurons pris soin de nommer le répertoire contenant le projet d'un nom commençant par board, par exemple **board_intro**, pour qu'il soit compilé, puisque c'est selon ce critère que le **Makefile** de examples balaie les répertoires d'après la règle **for bi in ${boards}; do $(MAKE) -C $$bi; done**.

Qu'avons-nous gagné si le résultat est le même qu'avant ? Nous avons maintenant la possibilité d'instancier des périphériques ! Ajoutons par exemple un bouton-poussoir connecté au port C3 :

```
#include <stdlib.h>
#include <stdio.h>
#include "sim_elf.h"
#include "avr_ioport.h"
#include "button.h"
volatile int appui=0;

int main(int argc, char *argv[])
{avr_t * avr = NULL;
 elf_firmware_t f;
 button_t button;                    // add a button
 const char * fname = "atmega32u4_intro.axf";
 elf_read_firmware(fname, &f);
 avr = avr_make_mcu_by_name(f.mmcu);
 avr_init(avr);
 avr_load_firmware(avr, &f);
 // ajout pour la definition du bouton
 button_init(avr, &button, "button"); // inits our 'peripheral'
 avr_connect_irq(                    // connect the button to the port pin
    button.irq + IRQ_BUTTON_OUT,
    avr_io_getirq(avr, AVR_IOCTL_IOPORT_GETIRQ('C'), 3));
 avr_raise_irq(button.irq + IRQ_BUTTON_OUT, 1); // raise = pull up
 while (1) {avr_run(avr);
            if (appui==1) {printf("PCB: pushed\n"); // level to 0
                           appui=0;button_press(&button, 300000);}

  }
}
```

Ce programme définit la structure `button_t`, l'initialise et lui associe des événements liés à C3, se compile et s'exécute pour donner strictement le même résultat qu'avant, puisque évidemment appui ne change jamais d'état. On notera que dans la nomenclature de `simavr`, appuyer sur un bouton signifie le passer à l'état bas, et son état par défaut (équivalent à une résistance de tirage en pull-up) est défini par `avr_raise_irq(button...,1)`. L'utilisation des termes « irq » est quelque peu trompeuse, car fait référence au transfert d'événements asynchrones – qui peuvent se déclencher à tout moment – entre le circuit imprimé (PCB – *Printed Circuit Board*) et le firmware exécuté par le cœur, mais la détection de l'état du port se fera en mode *polling* sans activer les interruptions de changement d'état de broche PCINT, ou de niveau INT (nous ferons cela plus tard). Cette affirmation est confirmée par le commentaire « *IRQ stands usually for Interrupt Request, but here it has nothing to do with AVR interrupts* » de **http://fabricesalvaire.github.io/simavr/doxygen/group__sim__irq.html**.

C'est ici que nous découvrons l'importance d'avoir la simulation cadencée dans la boucle infinie appelant `avr_run()` : il nous faut une seconde boucle infinie pour cadencer les événements extérieurs. Il y a plusieurs religions :

- sûrement par souci d'égayer leurs simulations avec des interfaces graphiques (nous ferons de même à la fin de ce document), les auteurs de `simavr` ont choisi de faire appel à GLUT, l'OpenGL Utility Toolkit. Dans ce cas, OpenGL se charge de générer les événements (appui de touche, de souris ou de timer) de façon asynchrone, et `avr_run()` est relégué à son propre thread ;
- fort de cet enseignement, nous pouvons créer notre propre POSIX-thread (`pthread_create()`), y placer soit `avr_run()`, soit le séquenceur d'événements, et ainsi voir les deux boucles infinies tourner en parallèle ;
- soit faire appel aux signaux d'UNIX qui se déclencheront en arrière-plan de la boucle infinie qui exécute `avr_run()`.

Bien qu'aucune de ces solutions ne soit la bonne, tel que nous le découvrirons plus tard (section 2.1) en termes d'exactitude temporelle, nous poursuivons cette description, car elle permet de générer des stimuli complexes depuis l'émulateur du circuit imprimé vers le microcontrôleur, si la temporisation relative au quartz qui cadence le microcontrôleur n'a pas d'importance.

Nous choisissons ici le dernier cas, qui nécessite le moins d'investissement intellectuel puisque `SIGALRM` se déclenche toutes les secondes par `signal(SIGALRM, handle_alarm); alarm(1);` et que le gestionnaire d'alarme `handle_alarm()` réenclenche le compte à rebours. Ainsi, nous ajoutons au code précédent :

```
#include <signal.h>
void handle_alarm(int xxx) { appui=1; alarm(1); }      // 1 PPS on PC side (board)
void init_alarm(void) { signal(SIGALRM, handle_alarm); alarm(1);} // 1 PPS signal
```

Comme nous ne savons pas passer d'argument au gestionnaire de signal `handle_alarm()`, nous appliquons les préceptes des gestionnaires d'interruptions d'exceptionnellement nous autoriser une variable globale (appui), de type `volatile` pour interdire au compilateur de l'optimiser, qui partage l'information entre le *handler* et la boucle exécutant `avr_run()`.

Finalement, il reste à voir si le bouton est bien appuyé : en l'absence de communication pour le moment, nous modifions le masque d'allumage du port B en fonction de l'appui sur C3 :

```
int main(void)
{int masque=(1<<PORTB5);
 DDRB =0xff;  // port B output
 PORTB=0;     // port C default config = input
 while(1)
  {PORTB ^= masque;
   if ((PINC&(1<<3))==0)  // pressed: button goes low
     {// PORTB=0;         // reset PORTB (later ...)
      if (masque<0x80) masque=masque<<1;
         else masque=0x01;
     }
   _delay_ms(100);
  }
}
```

Ainsi, le bouton-poussoir reste appuyé 300 ms (pour rappel, `if (appui==1) {button_press(b, 300000);appui=0;}`) et se réenclenche toutes les secondes (`alarme(1);`), tandis que le firmware change l'état du port B toutes les 100 ms. Nous nous attendons à voir deux (ou trois) changements consécutifs du masque, puisqu'il reste au même état pendant le reste de la seconde qui s'écoule. Cependant :

```
Loaded 250 .text at address 0x0
Loaded 0 .data
fw atmega32u4_intro.axf f=16000000 mmcu=atmega32u4
SIMAVR: UART INIT
SIMAVR: UART RESET
SIMAVR: IOPORT @0x25<-0x0
[... efface 35 fois la même ligne...]
SIMAVR: IOPORT @0x25<-0x0
PCB: pushed
SIMAVR: IOPORT @0x25<-0x20
SIMAVR: IOPORT @0x25<-0x60
SIMAVR: IOPORT @0x25<-0xe0
button_auto_release
SIMAVR: IOPORT @0x25<-0xe1
SIMAVR: IOPORT @0x25<-0xe0
SIMAVR: IOPORT @0x25<-0xe1
SIMAVR: IOPORT @0x25<-0xe0
[... efface 25 lignes alternant e0 et e1...]
SIMAVR: IOPORT @0x25<-0xe0
SIMAVR: IOPORT @0x25<-0xe1
SIMAVR: IOPORT @0x25<-0xe0
PCB: pushed
SIMAVR: IOPORT @0x25<-0xe1
SIMAVR: IOPORT @0x25<-0xe3
SIMAVR: IOPORT @0x25<-0xe7
button_auto_release
```

```
SIMAVR: IOPORT @0x25<-0xef
SIMAVR: IOPORT @0x25<-0xe7
SIMAVR: IOPORT @0x25<-0xef
[... efface 25 lignes alternant ef et e7...]
SIMAVR: IOPORT @0x25<-0xef
SIMAVR: IOPORT @0x25<-0xe7
SIMAVR: IOPORT @0x25<-0xef
SIMAVR: IOPORT @0x25<-0xe7
PCB: pushed
SIMAVR: IOPORT @0x25<-0xef
SIMAVR: IOPORT @0x25<-0xff
SIMAVR: IOPORT @0x25<-0xdf
button_auto_release
SIMAVR: IOPORT @0x25<-0x9f
SIMAVR: IOPORT @0x25<-0xdf
```

donc nous constatons que le bouton est appuyé (notre gestionnaire de temps sur l'émulation du PCB nous en informe par le message **"PCB: pushed"**) puis relâché (message de **simavr "button_auto_release"**), mais le nombre de messages laisse cependant à désirer. La temporisation imposée par le programme émulant le PCB qui contrôle **simavr** est fausse, il y a clairement plus d'une seconde écoulée entre la fin de l'appui et l'appui suivant.

En effet, comme attendu, nous avons 3 messages entre **"pushed"** et **"auto_release"** correspondant à trois intervalles de temps de 100 ms : les 300 ms de l'émulateur de PCB sont comptabilisées comme 3 fois 100 ms par le microcontrôleur. Cependant, nous avons une trentaine de messages de transitions d'état entre la fin de l'appui et l'appui suivant, soient environ 3 s dans le temps interne du microcontrôleur. Cette valeur est incohérente, notre 1-PPS ne dure pas du tout 1-seconde.

## 1.1 Du temps faux au temps juste

La source de notre erreur apparaît à la lecture de **examples/parts/button.c** et sa fonction **button_press()** dont la temporisation est observée correcte. Nous constatons que la durée d'appui est enregistrée auprès d'un *timer* géré par **simavr** qui appelle, à expiration, la fonction de *callback* qui relâche le bouton. Dans notre approche, la temporisation est prise en charge par le système d'exploitation exécutant le simulateur de PCB, dont le temps n'a aucune raison de s'écouler au même rythme que le temps interne du simulateur (sans parler des fluctuations induites par l'absence de contraintes temps réel de GNU/Linux). Nous reprenons donc l'émulateur de PCB en modifiant l'appel au signal **alarm()** par un appel au gestionnaire de timer interne à **simavr** grâce aux fonctions **avr_cycle_timer_cancel();** et **avr_cycle_timer_register_usec();**. L'émulateur de PCB **intro.c** devient donc :

```
static avr_cycle_count_t PPS(avr_t * avr, avr_cycle_count_t when, void * param)
{button_t * b = (button_t *)param;
 button_press(b, 300000);        // pressed = low
 printf("PCB: pushed\n");
 avr_cycle_timer_cancel(b->avr, PPS, b);
```

```
 avr_cycle_timer_register_usec(b->avr, 1000000, PPS, b);
 return 0;
}

int main(int argc, char *argv[])
{[... initialisation du PCB...]
 avr_raise_irq(button.irq + IRQ_BUTTON_OUT, 1); // raise = pull up
 // 1-PPS timer init
 avr_cycle_timer_cancel(button.avr, PPS, &button);
 avr_cycle_timer_register_usec(button.avr, 1000000, PPS, &button);
 while (1) {avr_run(avr);}
}
```

qui a été épuré au maximum en éliminant le code redondant avec l'exemple précédent. Nous n'avons plus besoin de `signal()` ou `alarm()`, cette fois tous les événements d'appui (*callback* nommé `PPS()`) et de relâchement du bouton sont pris en charge par les timers de `simavr`.

Fort de ces modifications, le résultat devient cohérent avec la temporisation attendue puisque :

```
Loaded 250 .text at address 0x0
Loaded 0 .data
fw atmega32u4_intro.axf f=16000000 mmcu=atmega32u4
SIMAVR: UART INIT
SIMAVR: UART RESET
SIMAVR: IOPORT @0x25<-0x0
[... 8 transitions retirées...]
SIMAVR: IOPORT @0x25<-0x20
SIMAVR: IOPORT @0x25<-0x0
PCB: pushed
SIMAVR: IOPORT @0x25<-0x20
SIMAVR: IOPORT @0x25<-0x60
SIMAVR: IOPORT @0x25<-0xe0
button_auto_release
SIMAVR: IOPORT @0x25<-0xe1
SIMAVR: IOPORT @0x25<-0xe0
SIMAVR: IOPORT @0x25<-0xe1
SIMAVR: IOPORT @0x25<-0xe0
SIMAVR: IOPORT @0x25<-0xe1
SIMAVR: IOPORT @0x25<-0xe0
SIMAVR: IOPORT @0x25<-0xe1
PCB: pushed
SIMAVR: IOPORT @0x25<-0xe0
SIMAVR: IOPORT @0x25<-0xe2
SIMAVR: IOPORT @0x25<-0xe6
button_auto_release
SIMAVR: IOPORT @0x25<-0xee
SIMAVR: IOPORT @0x25<-0xe6
SIMAVR: IOPORT @0x25<-0xee
SIMAVR: IOPORT @0x25<-0xe6
SIMAVR: IOPORT @0x25<-0xee
SIMAVR: IOPORT @0x25<-0xe6
SIMAVR: IOPORT @0x25<-0xee
PCB: pushed
SIMAVR: IOPORT @0x25<-0xe6
```

avec au début 10 transitions de 100 ms chacune avant que le bouton soit appuyé une première fois, puis un appui pendant 3 transitions correspondant à 300 ms, et un bouton relâché pendant 7 transitions ou les 700 ms restantes. Il ne reste plus qu'à décommenter l'ordre `PORTB=0x00` lors de l'appui du bouton pour obtenir un masque cohérent qui avance d'un bit à chaque appui du bouton.

Cette longue introduction sur le cas du GPIO nous a armés pour la suite des hostilités : nous savons créer un PCB, lui assigner une architecture de processeur et son firmware, exciter des signaux extérieurs et constater la cohérence avec les temporisations attendues. Passons aux choses sérieuses que sont les communications et interruptions sur événements timers.

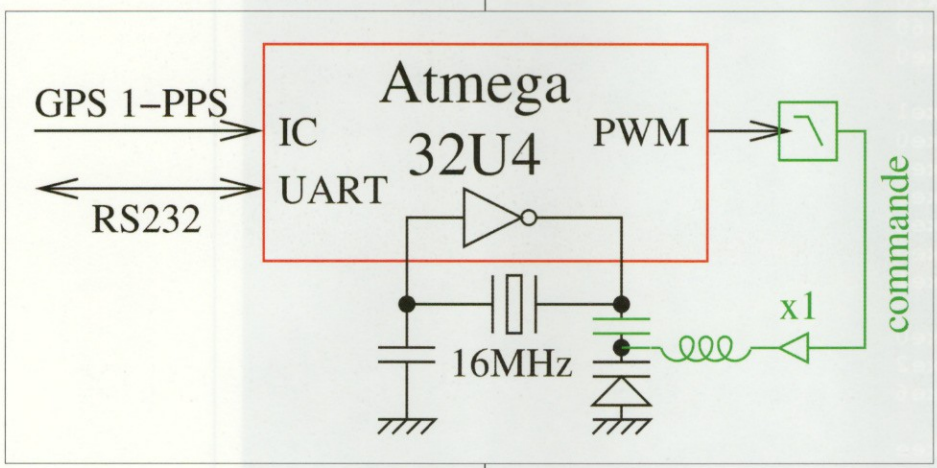



*Figure 2 : Schéma de principe, avec le 1-PPS issu du récepteur GPS qui alimente une entrée input capture (IC) connectée à un compteur (timer). L'écart de fréquence à la valeur nominale est corrigé en polarisant une varicap faisant office de condensateur de pieds d'un oscillateur à quartz comprenant un résonateur et la porte inverseuse se comportant comme amplificateur introduisant un déphasage de 180°. Le filtre passe-bas chargé de lisser la PWM, l'amplificateur suiveur, l'inductance bloquant la fuite du signal radiofréquence et le condensateur de blocage de la commande DC vers l'amplificateur (en vert) ne seront pas explicités dans ce document, qui se focalise sur les éléments numériques de l'asservissement.*

Nous nous proposons de tester `simavr` sur un cas pratique que nous avions exposé dans **[2]** – moins impressionnant que **https://github.com/buserror/simreprap** qui émule une imprimante additive 3D complète et sera source d'inspiration pour appeler certains périphériques – mais qui sert encore de prétexte à appréhender les divers périphériques de processeurs, à savoir l'asservissement d'un oscillateur à quartz sur le signal horaire de référence issu d'un récepteur GPS.

Un récepteur GPS est avant tout un instrument dédié à reproduire une horloge locale suffisamment stable pour dater avec précision les signaux horaires transmis par la constellation de satellites, et éventuellement par trilatération de positionner son propriétaire dans l'espace. L'information de base fournie par un récepteur GPS, avant cette information de position, est une impulsion périodique générée chaque seconde – 1-PPS (1-*Pulse Per Second*) – dont le front montant indique, avec une précision de +/- 100 ns environ **[4, Figs.2–4]**, le début de la seconde GPS.

Toute fluctuation de l'oscillateur local qui cadence le microcontrôleur entre deux fronts montants du 1-PPS s'observe comme une variation du compteur dont une interruption *input capture* est déclenchée par ce front. Trop lent, le décompte sera inférieur à la fréquence nominale du résonateur à quartz, trop rapide et le décompte sera supérieur à cette valeur. En plus d'observer cette dérive de la fréquence

du résonateur à quartz face au 1-PPS de référence, supposé parfait, nous pourrons corriger cette erreur de fréquence en ajustant les conditions d'oscillations dites de Barkhausen : la rotation de phase le long d'un oscillateur doit être multiple de 2π. Étant donné que dans un microcontrôleur l'amplificateur est formé d'une porte inverseuse (NOT) qui introduit donc une rotation de phase de 180°, et que le résonateur n'introduit pas de rotation de phase à la résonance (le circuit résonant formé d'une inductance et d'une capacité en série voit les deux réactances s'annuler à la résonance) et il reste les deux condensateurs de pieds – typiquement quelques dizaines de pF – reliant chaque borne du résonateur à quartz à la masse pour vérifier la condition de Barkhausen sur la phase. En jouant sur ces valeurs de condensateurs, nous pouvons abaisser la fréquence d'oscillation de quelques parties par million (ppm) et ainsi ajuster la fréquence d'oscillation. Un condensateur commandé en tension est une varicap. Comme le microcontrôleur Atmega32U4 ne possède pas de convertisseur analogique numérique, le signal de commande sera généré par une sortie en modulation de largeur d'impulsion PWM suivie d'un filtre passe-bas pour la lisser et fournir une tension continue égale à la valeur moyenne : la commande se traduira donc par une modification du rapport cyclique de la PWM (en vert sur Fig. 2).

Résumons donc les périphériques dont nous aurons besoin (Fig. 2) et que nous simulerons sur `simavr` :

- la **communication** pour interagir avec l'utilisateur au travers du port série de communication asynchrone compatible RS232 ;
- un timer avec une résolution aussi bonne que possible pour **mesurer la fréquence** du quartz entre deux fronts montants du 1-PPS détectés par *input capture* qui se charge de mémoriser l'état du compteur au moment de l'événement ;
- un timer en PWM pour **commander** la varicap et de ce fait la fréquence de l'oscillateur cadençant le microcontrôleur, variable à laquelle nous aurons accès dans la configuration du microcontrôleur équipant le PCB ;
- pour être complet, un **bouton-poussoir déclenché par interruption** pour laisser l'opportunité à l'utilisateur d'interagir avec son instrument.

Nous allons aborder chacun de ces périphériques dans les sections qui suivent.

## 2. COMMUNICATION ASYNCHRONE DU MONDE EXTÉRIEUR VERS LE PROCESSEUR

Commençons par communiquer. Le port de communication série asynchrone, compatible RS232, s'initialise en activant les divers paramètres que sont le débit de communication (*baudrate*), nombre de bits/symboles transmis, présence ou non de bits de parité, des choses très standard :

```
#define USART_BAUDRATE 115200  // avoid wasting too much time talking

void usart_setup()
{unsigned short baud;
 baud  = (((( F_CPU / ( USART_BAUDRATE * 16UL))) - 1));
```

```
 DDRD  |= 0x18;
 UBRR1H = (unsigned char)(baud>>8);
 UBRR1L = (unsigned char)baud;
 UCSR1C = (1<<UCSZ11) | (1<<UCSZ10); // async, no parity, 1 stop, 8 data
 UCSR1B = (1<<RXEN1 ) | (1<<TXEN1 ); // enable TX and RX
}
```

Écrire sur le port série s'obtient en vérifiant que le périphérique est libre, et en stockant dans le registre adéquat l'octet à communiquer, le matériel se chargeant ensuite de générer les signaux :

```
void mytransmit_data(uint8_t data)
{while ( !( UCSR1A & (1<<UDRE1)) );
 UDR1 = data;
}
```

L'émission d'un message d'un périphérique vers le microcontrôleur n'étant pas déterministe, la « bonne » façon de gérer la communication n'est pas d'attendre que le message arrive, mais de déclencher une interruption sur la réception du caractère, le stocker dans un tampon afin de prévenir la boucle du programme principal de gérer le message quand il en a le temps. Le vecteur d'interruption **USART1_RX_vect** correspondant à l'UART1 est appelé quand un tel événement survient sous réserve d'avoir activé la fonctionnalité par le bit **RXCIE1** de **UCSR1B**. Nous proposons par ailleurs d'activer le traçage de toutes les interruptions pour en observer l'évolution dans le temps :

```
#include "avr_mcu_section.h"
AVR_MCU(F_CPU, "atmega32u4");
AVR_MCU_VCD_FILE("trace_file.vcd", 1000);
const struct avr_mmcu_vcd_trace_t _mytrace[]  _MMCU_ = {
        { AVR_MCU_VCD_SYMBOL("PORTB"), .what = (void*)&PORTB, },
        { AVR_MCU_VCD_SYMBOL("UDR1"), .what = (void*)&UDR1, },
};
AVR_MCU_VCD_ALL_IRQ();  // also show ALL irqs running

volatile char rx,flag;
ISR(USART1_RX_vect) {rx=UDR1;flag=1;}
int main(void)
{char c='.';
 DDRB=0xff;
 usart_setup();
 UCSR1B |= (1<<RXCIE1); // add RX interrupt
 flag=0;
 sei();
 while(1)
```

```
    {mytransmit_uart(c);              // transmits current symbol
     if (flag!=0) {flag=0;c=rx+1;} // if received, update c
     PORTB^=0x10;
     _delay_ms(200);
    }
}
```

Du point de vue du PCB, communiquer du microcontrôleur vers l'utilisateur ne présente aucune difficulté puisque **simavr** affiche sur la console tout caractère transmis sur le port de communication asynchrone. Plus difficile, envoyer un symbole de l'utilisateur vers le microcontrôleur, qui impose de connecter un terminal à **simavr** pour en capturer les messages. Ceci s'obtient dans le code d'émulation du PCB par :

```
#include "uart_pty.h"
uart_pty_t uart_pty;

int main(int argc, char *argv[])
{[... initialisation AVR et firmware ELF...]
 uart_pty_init(avr, &uart_pty);
 uart_pty_connect(&uart_pty, '1');
  while (1) {avr_run(avr);}
}
```

pour connecter un terminal à UART1 (dernier argument de **uart_pty_connect()**). Selon le programme proposé plus haut, nous affichons le dernier symbole mémorisé toutes les 200 ms, et mettons à jour le symbole si une communication de l'utilisateur vers le microcontrôleur a eu lieu afin de vérifier le bon fonctionnement de l'interruption.

À l'exécution, simavr nous informe que :

```
uart_pty_init bridge on port *** /dev/pts/9 ***
uart_pty_connect: /tmp/simavr-uart1 now points to /dev/pts/9
note: export SIMAVR_UART_XTERM=1 and install picocom to get a terminal
```

et en nous exécutant, nous obtenons le résultat de la Fig. 3.

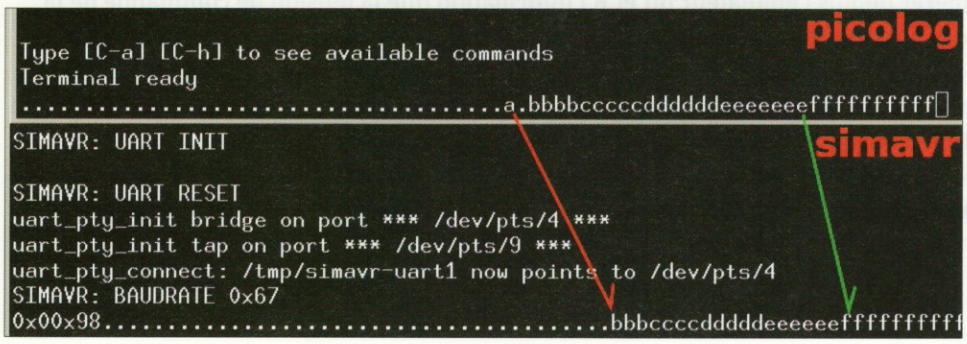



*Figure 3 : Communication depuis picolog vers simavr émulant un Atmega32U4 et répondant avec le caractère suivant de celui fourni. Lorsque nous tapons 'a' dans picolog (rouge), le microcontrôleur répond 'b' qui s'affiche dans le terminal exécutant simavr ainsi que dans picolog. Nous répétons l'opération jusqu'à 'e' qui se traduit par la réponse 'f' (vert).*

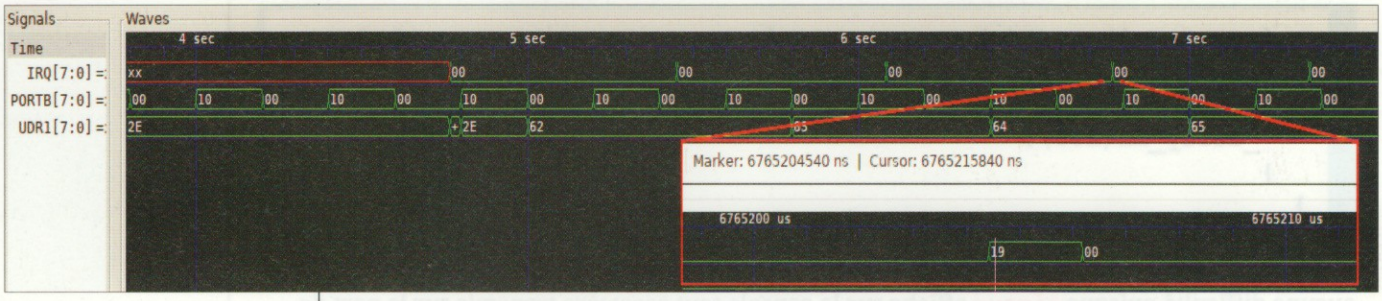



*Figure 4 : Trace des signaux internes au microcontrôleur, incluant le registre de transmission des caractères du port série UDR et l'état des interruptions. Le cadran du bas est un zoom sur la courbe du haut pour identifier le numéro d'interruption déclenchée.*

Très intéressant grâce à la fonction de traçage des signaux internes au microcontrôleur, nous observons avec gtkwave la trace au format VCD (*Value Change Dump*) qui contient l'état des interruptions (Fig. 4) et constatons que l'interruption `0x19=25` s'est déclenchée, en accord avec la documentation technique **[3, p.57]** qui, en indexant les interruptions à partir de 1 et non de 0, indique que `"USART1 Rx Complete"` est l'interruption numéro 26.

## 3. INTERRUPTIONS : INPUT CAPTURE, PWM ET GPIO

Passons désormais aux *timers*. Il s'agit de périphériques forts simples, mais forts utiles : un compteur tourne continuellement, atteint sa valeur maximale qui peut être source d'interruption (*overflow*) pour revenir à 0, et nous pouvons potentiellement déclencher des événements le long du décompte, par exemple un changement d'état de broche pour moduler le rapport cyclique en mode PWM en maintenant une période constante. Plus intéressant dans notre cas, ce compteur peut être mémorisé lors du déclenchement d'un événement externe tel que la transition d'état sur une broche : il s'agit du mode *input capture* qui nous permettra de compter le nombre d'oscillations du circuit cadençant le microcontrôleur entre deux fronts montants du 1-PPS.

### 3.1 PWM

La PWM est sous le contrôle du microcontrôleur, donc notre seul objectif ici est de récupérer depuis l'émulateur du PCB l'information que la tension de commande de la varicap a été modifiée, entraînant une modification de la fréquence d'oscillation du circuit cadençant le microcontrôleur, et modifier les variables d'émulation en conséquence. Nous nous inspirons pour cela de `examples/board_timer_64led/timer_64led.c` qui module l'intensité lumineuse de l'afficheur par le rapport cyclique de la PWM.

```
void pwm_changed_hook(struct avr_irq_t * irq, uint32_t value, void * param)
     { display_pwm = value; pwm_flag=1; }
int main()
{[...]
```

```
 avr_irq_t* i_pwm= avr_io_getirq(avr, AVR_IOCTL_TIMER_GETIRQ('0'), TIMER_IRQ_OUT_
PWM0);
 avr_irq_register_notify(i_pwm, pwm_changed_hook, NULL);
 [...]
 while (1)
 { avr_run(avr);  // avr->frequency: see ../../simavr/sim/sim_avr.c: avr_init()
   if (pwm_flag==1) {avr->frequency=16000000+display_pwm*10;
                     printf("PWM:%d -- freq: %d\n",display_pwm,avr->frequency);pwm_
flag=0;
                    }
 }
}
```

Le pendant dans le firmware exécuté sur le microcontrôleur est de périodiquement modifier la valeur de la PWM qui a été initialisée par :

```
void mypwm0_setup(void)
{TCCR0A =  (1<<COM0A1)|(0<<COM0A0)|(0<<COM0B1)|(0<<COM0B0)
            |(1<<WGM01) |(1<<WGM00);
 TCCR0B =  (0<<FOC0A) |(0<<FOC0B) |(0<<WGM02)
            |(0<<CS01)  |(1<<CS00);
 TCNT0 = 0;
 OCR0A = 127; // initial value
}

int main(void)
{mypwm0_setup();
 while(1) {OCR0A++;_delay_ms(200);}
}
```

Ce code affirme que la PWM s'incrémente toutes les 200 ms et se traduit, dans le code gérant le PCB, par un incrément de la fréquence de cadencement du microcontrôleur (`avr->frequency`) de (arbitrairement) 10 fois la valeur de la PWM, soit quelques kHz sur la gamme d'ajustement du timer, une valeur typique d'un résonateur à quartz. Comment pouvons-nous vérifier si ces affirmations sont exactes ? En analysant la valeur du timer configuré en *input capture* pour mesurer l'intervalle de temps entre deux fronts montants du 1-PPS.

## 3.2 Input Capture

Contrairement à la PWM où seule la simulation du circuit imprimé doit déclarer la gestion de l'événement de variation du rapport cyclique puisque le matériel gère cette configuration du timer dans le microcontrôleur, le cas *input capture* doit être pris en compte et dans le firmware, et dans le PCB.

Côté firmware, une gestion classique d'*input capture* sans originalité :

```
volatile int flag_icp,value_icp;   // globar var for exchanging with ISR

ISR(TIMER3_OVF_vect) {flag_ovf++;} // timer overflow

ISR(TIMER3_CAPT_vect)
{flag_icp  = 1;       // informs main() of an event
 value_icp = ICR3;    // remember timer value
 TCNT3 = 0;           // resets counter to 0 (timer diff)
}

void myicp_setup() // inits (16 bit) timer3
{TCCR3B = 1<<ICES3 | 1<<CS30;     // prescaler = 1
 TIMSK3 = (1<<ICIE3)|(1<<TOIE1); // IC & OVF interrupts
 TIFR3  = 1<<ICF3;
}

int main(void)
{int compteur=0;
 myicp_setup();
 sei();
 flag_icp=0;
 while(1)
  {if (flag_icp!=0)
      {write_short(value_icp); mytransmit_data(' ');
       write_short(flag_ovf);  mytransmit_data('\n');
        compteur++;
        if (compteur==5)
           {OCR0A++;compteur=0;} // update PWM duty cycle
            // -> triggers an event on the PCB
            flag_ovf=0;flag_icp=0;}
  }
}
```

Dans cet exemple, nous comptons grossièrement l'intervalle de temps entre deux fronts montants du 1-PPS en mémorisant le nombre d'*overflows* (interruption `TIMER3_OVF_vect` indiquant que le compteur a atteint sa borne maximale – TOP dans la nomenclature Atmel) et avec précision cet intervalle de temps en mémorisant la valeur du compteur au moment de l'événement par *input capture* par l'interruption `TIMER3_CAPT_vect`. Le compteur est remis à 0 dans ce gestionnaire d'interruption pour mesurer un *écart* de temps entre deux fronts montants, et un drapeau est validé pour informer le programme principal que l'événement a eu lieu. Afin de caractériser la dépendance en boucle ouverte de la fréquence du quartz avec la valeur de la PWM, nous maintenons 5 fois de suite le seuil de transition de la broche correspondante à la même valeur, avant de l'incrémenter pour mesurer la fréquence avec cette nouvelle commande.

Côté PCB, nous générons une fois par seconde un événement représentatif du 1-PPS du GPS. Nous avons vu (section 2) que nous ne devions pas exploiter le temps hôte (ordinateur sur lequel

s'exécute **simavr**), mais les timers de **simavr** pour induire une datation cohérente des événements. Qu'à cela ne tienne : nous utilisons deux *timers* de **simavr**, un pour déclencher le front montant toutes les secondes, et l'autre pour redescendre le signal une centaine de millisecondes après sa montée :

```
volatile int pwm_flag,icp_flag;

void icp_changed_hook(struct avr_irq_t * irq, uint32_t value, void * param) { icp_
flag=1; }

static avr_cycle_count_t PPSlo(avr_t * avr, avr_cycle_count_t when, void * param)
{avr_irq_t* irq=(avr_irq_t*) param;
 printf("PPS: lo\n");
 avr_raise_irq(irq,1);
 return 0;
}

static avr_cycle_count_t PPSup(avr_t * avr, avr_cycle_count_t when, void * param)
{avr_irq_t* irq=(avr_irq_t*) param;
 printf("PPS: up\n");
 avr_raise_irq(irq,0);
 // avr_cycle_timer_cancel(avr, PPSup, irq);
 // useless cf ../../simavr/sim/sim_cycle_timers.c
 avr_cycle_timer_register_usec(avr, 1000000, PPSup, irq);
 // which alread takes care of that
 // avr_cycle_timer_cancel(avr, PPSlo,NULL);
 avr_cycle_timer_register_usec(avr, 200000, PPSlo, irq);
 return 0;
}

int main(int argc, char *argv[])
{[ ... inits AVR and firmware provided in ELF format ...]
 avr_irq_t* irq= avr_io_getirq(avr, AVR_IOCTL_TIMER_GETIRQ('3'), TIMER_IRQ_IN_ICP);
 // avr_irq_t* irq= avr_get_interrupt_irq(avr, 32);
 // same effect than TIMER3+TIMER_IRQ_IN_ICP
 avr_irq_register_notify(irq, icp_changed_hook, NULL);
 // 1-PPS timer init
 avr_cycle_timer_cancel(avr, PPSup, irq);
 avr_cycle_timer_register_usec(avr, 1000000, PPSup, irq);

 icp_flag=0;
 while (1)
 { avr_run(avr);
   if (pwm_flag==1) {avr->frequency=16000000+display_pwm*10;
   // change oscillator frequency
                    printf("PWM:%d -- freq: %d\n",display_pwm,avr->frequency);pwm_
flag=0;}
   if (icp_flag==1) {printf("ICP\n");icp_flag=0;}
 }
 return NULL;
}
```

Le résultat de l'exécution de ce code est de la forme :

```
$ ./obj-x86_64-linux-gnu/PPScontrol.elf > t
23E3 00F4
23E3 00F4
23E3 00F4
23ED 00F5
23ED 00F4
23EB 00F4
23ED 00F4
23EF 00F4
23F6 00F5
23F6 00F4
```

qui indique donc que 244 ou 245 *overflows* se déclenchent, et que le résidu entre deux fronts montants est de l'ordre (première ligne) de 9187. Ainsi, la fréquence du quartz est de l'ordre du nombre d'*overflows* multiplié par la valeur maximale du compteur (ici, 16 bits donc 65536) auquel on ajoute le résidu, soit 244 x 65536 + 9187=15,999971 MHz qui est raisonnablement proche des 16 MHz visés.

Plus important, le PPS résultant de l'utilisation des timers de `simavr` est stable tel que le démontre la Fig. 5 dans laquelle l'incrément d'une unité de la PWM toutes les 5 mesures se traduit bien par une croissance de 10 unités de la fréquence.

## 4. INTERRUPTION GPIO

Afin de proposer un petit exercice sur les pointeurs, nous nous proposons de gérer un dernier type d'événement que sont les interruptions sur appui de bouton connecté à un GPIO.

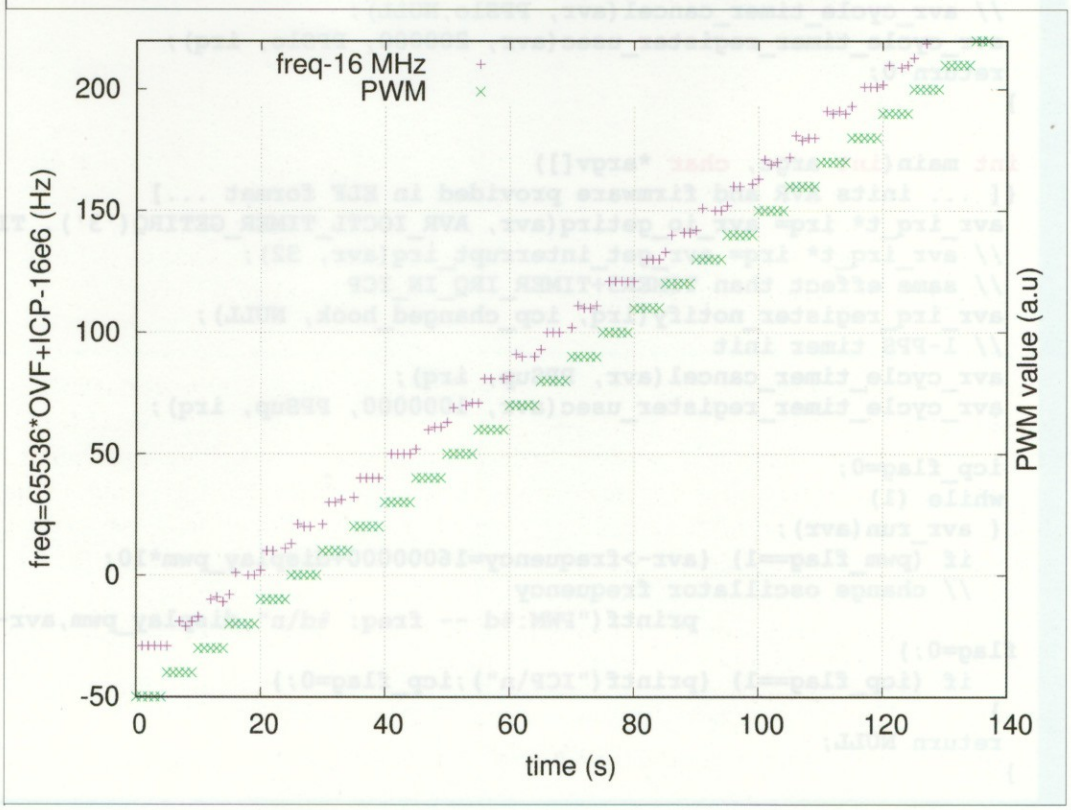



*Figure 5 : Évolution de la fréquence de l'oscillateur cadençant le microcontrôleur (violet), et donc ses timers, en fonction de la tension de commande issue de la PWM filtrée par un passe-bas (vert).*

Nous n'avons pas d'application pratique dans ce contexte, mais nous allons voir que cela donne l'opportunité de nous amuser avec les cast.

Ainsi, nous ajoutons à notre séquence d'interruptions gérées par le firmware un bouton connecté (arbitrairement) à PD2, qui déclenche l'interruption sur niveau INT2 :

```
int main()
{avr_irq_t* irq[2];
 irq[0]= avr_io_getirq(avr, AVR_IOCTL_IOPORT_GETIRQ('D'), 2);
 irq[1]= avr_io_getirq(avr, AVR_IOCTL_TIMER_GETIRQ('3'), TIMER_IRQ_IN_ICP);
 avr_irq_register_notify(irq[1], icp_changed_hook, NULL);
 avr_irq_register_notify(irq[0], pd2_changed_hook, NULL);
 avr_cycle_timer_register_usec(avr, 1000000, PPSup, irq);
}
```

dans lequel nous ne mentionnons que les lignes ajoutées au cas précédent, puisque nous désirons à la fois gérer l'*input capture* du timer et cette interruption GPIO.

Côté simulation du PCB, nous déclarons la fonction de callback liée à l'événement de changement de niveau du GPIO, mais comme nous voudrons passer les *deux* gestionnaires d'interruptions (*input capture* et GPIO) en argument aux gestionnaires de temporisation de `simavr` ou au thread que nous allons déclarer plus tard pour gérer de façon asynchrone `simavr`, tandis que le gestionnaire d'interface graphique monopolise le processeur par sa boucle infinie, il nous faut une unique structure à passer comme argument. Nous allons donc déclarer un tableau de gestionnaires d'interruptions `irq[2]` :

```
int main()
{avr_irq_t* irq[2];
 irq[0]= avr_io_getirq(avr, AVR_IOCTL_IOPORT_GETIRQ('D'), 2);
 irq[1]= avr_io_getirq(avr, AVR_IOCTL_TIMER_GETIRQ('3'), TIMER_IRQ_IN_ICP);
 avr_irq_register_notify(irq[1], icp_changed_hook, NULL);
 avr_irq_register_notify(irq[0], pd2_changed_hook, NULL);
 avr_cycle_timer_register_usec(avr, 1000000, PPSup, irq);
}
```

Ainsi grâce à cette structure, nous pouvons passer l'unique pointeur comme argument aux fonctions appelées, par exemple `avr_cycle_timer_register_usec()` ici, et comprendre la puissance du typage `void*` qui détermine la nature de ce dernier argument. Un `void*` est un pointeur sur n'importe quoi, une zone mémoire contenant donc aussi bien un scalaire (`char`, `short` ou `int`) qu'une structure complexe. Il nous suffit, dans la fonction appelée, de caster ce pointeur vers la nature de la structure passée en argument pour expliciter l'organisation de la mémoire :

```
static avr_cycle_count_t PPSup(avr_t * avr, avr_cycle_count_t when, void * param)
{avr_irq_t** irq=(avr_irq_t**) param;
 avr_raise_irq(irq[0],0);
 avr_raise_irq(irq[1],0);
 avr_cycle_timer_register_usec(avr, 1000000,PPSup, irq); // next rising edge event
```

```
 avr_cycle_timer_register_usec(avr, 200000, PPSlo, irq); // next falling edge event
 return 0;
}

static avr_cycle_count_t PPSlo(avr_t * avr, avr_cycle_count_t when, void * param)
{avr_irq_t** irq=(avr_irq_t**) param;
 avr_raise_irq(irq[0],1);
 avr_raise_irq(irq[1],1);
 return 0;
}
```

La première fonction est un callback d'un timer chargé de déclencher le signal 1-PPS. Lors de son déclenchement, le 1-PPS s'enregistre pour se redéclencher une seconde plus tard, et enregistre un second timer pour s'abaisser 200 ms plus tard en appelant la seconde fonction. Dans ces deux fonctions, nous avons choisi de manipuler simultanément PD2 (connectée à INT2) et *input capture* afin d'illustrer le passage du tableau d'interruptions. Dans les deux cas, l'organisation de la mémoire est obtenue par le cast `avr_irq_t** irq=(avr_irq_t**) param;` qui indique que la zone mémoire sans type `void* param` doit être interprétée comme un tableau de pointeurs vers les gestionnaires d'interruptions. Cette méthode de passage de paramètre est très générale puisque nous la retrouvons dans FreeRTOS (prototype des tâches crées par `xTaskCreate()` de la forme `void vATaskFunction( void *pvParameters)` telle que décrite à **https://www.freertos.org/implementing-a-FreeRTOS-task.html**, ou dans le noyau Linux avec le prototype du gestionnaire d'interruptions :

```
int request_irq(unsigned int irq, irqreturn_t (*handler)(int, void *, struct pt_
regs *),
    unsigned long flags, const char *dev_name, void *dev_id);
```

décrit à **https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch10.html**.

## 5. AJOUT DE L'INTERFACE GRAPHIQUE

Maintenant que les bases du programme fonctionnent, avec la PWM, *input capture*, communication et interruption GPIO actives, nous pouvons retourner aux aspects esthétiques en ajoutant une illustration afin de mettre cette démonstration au niveau des autres projets de `examples/board*`, et profiter de ce fait du passage de paramètres tel que nous venons de l'expliciter lors de la création du POSIX thread qui exécute l'émulateur.

Nous n'entrons pas dans les détails des étapes, nous étant contentés de glaner des informations sur divers forums faute d'expérience sur la programmation en OpenGL, mais GLUT simplifie considérablement la gestion des événements graphiques tels que nous laisserons le lecteur le constater en consultant l'archive **https://github.com/jmfriedt/l3ep/blob/master/board_project/PPScontrol.c**. Nous voulions présenter à l'utilisateur les divers signaux transmis depuis le PCB au microcontrôleur et visualiser les réactions à ces stimuli. Ainsi, le 1-PPS est visualisé par un carré rouge sur le connecteur SMA auquel nous serions susceptibles de connecter la sortie du signal horaire d'un récepteur

GPS, tandis que des carrés de diverses couleurs (jaune pour *input capture* du timer 3 dans l'exemple de la Fig. 6) sur le microcontrôleur représentent le déclenchement de l'interruption correspondante. Il ne reste qu'à rajouter autant de symboles dans la fonction d'affichage de l'image que de signaux à présenter, en conditionnant l'affichage du symbole sur le drapeau correspondant mis en place dans la fonction de *callback* (Fig. 6). La lecture de l'image JPEG en arrière-plan s'appuie sur libdevil telle que décrite à **https://community.khronos.org/t/how-to-load-an-image-in-opengl/71231/6**.

Comme nous l'avions mentionné dans **[1]**, un des avantages d'un tel outil de simulation est la génération de conditions de mesure difficilement reproductibles expérimentalement, voire de systématiser les tests en balayant les signaux physiques comme le fait tout bon développeur avec les tests unitaires. Dans ce cas, il reste donc à induire une variation brusque de la fréquence du quartz telle que nous l'observerions en plaçant la panne d'un fer à souder dessus, et boucler l'asservissement sur le 1-PPS pour corriger cette dérive de fréquence en ajustant de façon appropriée le condensateur de pieds de l'oscillateur. La démarche suivie dépasse le cadre du présent article, mais est largement discutée dans le manuscrit qui

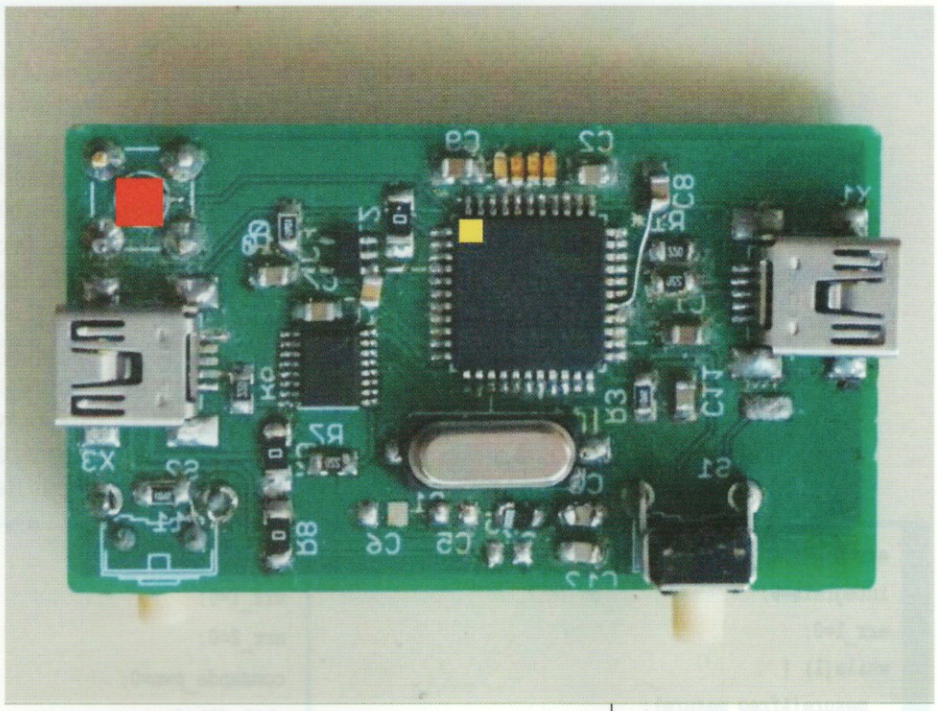


*Figure 6 : Interface graphique permettant de visualiser les stimuli issus du circuit imprimé vers le microcontrôleur. En rouge, le 1-PPS, en jaune l'input capture (timer 3) résultant. Un second input capture (timer 1) peut apparaître en cyan à côté du carré jaune s'il est connecté.*

accompagne le projet de Licence 3 qui a motivé cette étude **[5]** : nous sautons ici au résultat qui vise à démontrer le bon fonctionnement de la boucle d'asservissement fonctionnant sur le principe d'un contrôle proportionnel intégral (PI **[6, 7]**).

Dans ce contexte, une erreur $\varepsilon_n$ entre l'observable (la fréquence de l'oscillateur) à l'instant n et sa consigne est utilisée pour produire une commande $c_n$ formée de la somme de cette erreur multipliée par une constante $K_p$ (terme proportionnel) et l'intégrale de cette erreur multipliée par une constante $K_i$ (terme intégral) avec toute la subtilité du réglage des constantes pour minimiser le temps de convergence, tout en évitant de trop bousculer la commande. Alors qu'en temps continu, la méthode dite de Ziegler et Nichols **[9]** est classique, en temps discret l'approche est un peu différente, avec un point de départ pour régler $K_p$ et $K_i$ fournie par Takahashi **[10, 11]**. Dans ce contexte, sachant que nous avons choisi dans notre émulateur un facteur a=10 entre la commande (sortie de PWM) et la variation de

Le concept de commande proportionnelle, intégrale et dérivée (PID) issue des premières réflexions sur les pilotes automatiques **[8, chap.1.6]** se base sur l'idée de générer une commande sur un actuateur formé de la somme d'un terme proportionnel à la différence entre une observable et une consigne, d'un terme intégral (somme des valeurs passées) de cette différence, et éventuellement de la dérivée de cette différence. Nombre d'ouvrages décrivent la contribution de chaque terme – correction, annulation de l'erreur statique et éviter de trop brusquer la commande, mais ici nous nous intéressons uniquement à l'implémentation. La commande c(t) en temps continu t s'obtient à partir de l'erreur ε entre observable et consigne par $c(t)=K_p\varepsilon(t)+K_i\int_t^0 \varepsilon(\tau)d\tau+K_d d\varepsilon/dt$ avec $K_p$, $K_i$ et $K_d$ des constantes à identifier, pour devenir en temps discret n : $c_n=K_p\varepsilon_n+K_i\sum_{k=0}^{n}\varepsilon_k+K_d(\varepsilon_n-\varepsilon_{n-1})$ en considérant la période d'échantillonnage unitaire. Afin d'éviter de voir la somme de l'intégrale diverger quand n devient grand, il est classique d'utiliser l'équation de récurrence sur $c_n$ en écrivant $c_{n+1}$ et en constatant que $c_{n+1}-c_n=K_p(\varepsilon_{n+1}+\varepsilon_n)+K_i\varepsilon_{n+1}+K_d(\varepsilon_{n+1}-2\varepsilon_n+\varepsilon_{n-1})$ puisque les termes de la somme $\sum_{k=0}^{n}$ s'annulent. Ainsi, nous ne devrions pas écrire la solution de gauche, mais celle de droite :

```
err=0;
integrale=0;
err_1=0;
while(1) {
  mesure(&freq_mesure);
  erreur=consigne-freq_mesure;
  integrale+=err;                 // *dt
  satureintegrale(&integrale); // evite depassement
  derivee=(err-err_1);            // /dt
  cmde_pwm=Kp*err+Ki*integrale+Kd*derivee;
  satureCommande(commande_pwm); // protege cmd
  err_1=err;
}
```

```
err=0;
err_1=0;
err_2=0;
commande_pwm=0;
while(1) {
  mesure(&freq_mesure);
  err_2=err_1;
  err_1=err;
  err=consigne-freq_mesure;
  cmde_pwm+=Kp*(err-err_1)+Ki*err+Kd*(err+err_2-2*err_1);
  saturecommande(&commande_pwm); // noter ci-dessus "+="
}
```

Nous nous imposons de travailler sur des entiers uniquement, l'Atmega n'étant pas équipé d'une unité de calcul flottante, et donc choisirons une expression fractionnaire de $K_p$, $K_i$ et $K_d$ qui nous permet d'exprimer ces coefficients inférieurs à 1 dans le cas de notre système de gain supérieur à 1 (pour rappel, nous avons choisi de bouger la fréquence de 10 Hz pour une commande qui varie d'une unité). Lors de la mise en œuvre de l'algorithme de droite, nous avons constaté que l'erreur statique n'était pas annulée, mais que la commande faisant converger vers une valeur proche de la consigne, mais biaisée. En effet, dans la solution itérative, le terme intégral $K_i\varepsilon_n$ s'annule si $\varepsilon_n$ est plus petit que $1/K_i$ (dont nous rappelons qu'il est l'inverse d'un entier). Au contraire, si l'intégrale est conservée explicitement dans la solution de gauche, alors le terme $\sum_{k=0}^{n}\varepsilon_k$ accumule tout l'historique des erreurs de l'asservissement et sa multiplication par $K_i$ ne s'annule pas même si à un instant $\varepsilon_n$ devient plus petit que $1/K_i$ : la commande continuera à annuler le biais statique jusqu'à ce que la somme devienne plus petite que $1/K_i$, un cas qui se produit tard après que le biais ait été compensé. C'est donc la solution de gauche qui a été implémentée pour fournir les solutions de la Fig. 7.

fréquence, le gain est connu et le retard L supposé nul, car la commande agit immédiatement sur la fréquence sans devoir attendre une période $T_e$ de la boucle d'asservissement. Dans ce contexte, Takahashi prévoit donc $K_p=0,27/a\cdot(L+0,5T_e)2 =\sim 0,11$ et $K_i=0,9/a\cdot(L+0,5T_e)-0,5\ K_i x\ T_e =\sim 0,125$. Comme nous travaillons sur microcontrôleur 8 bits sans unité de calcul en virgule flottante, nous nous imposons de ne travailler qu'avec des entiers et prendrons les inverses de ces coefficients dans l'implémentation de la loi de commande, soit $1/K_p =\sim 9$ et $1/K_i =\sim 8$. De ce point de départ, nous observons l'évolution de la fréquence de l'émulateur sous commande de la loi que nous venons de proposer : les divers régimes sont observés en Fig. 7.

Nous constatons sur cette figure 7 que Takahashi est un peu optimiste et induit un régime d'oscillations qui, sur un système mécanique, se traduirait rapidement par un risque de rupture. En atténuant un peu la loi de commande en abaissant $K_p$ ou $K_i$ (*i.e.* en augmentant leur inverse), nous retrouvons des régimes soit excessivement atténués, soit proche de l'optimum avec un léger dépassement de consigne avant de converger vers la valeur recherchée (rouge sur la Fig. 7). La boucle d'asservissement est donc complètement fonctionnelle dans l'émulateur.

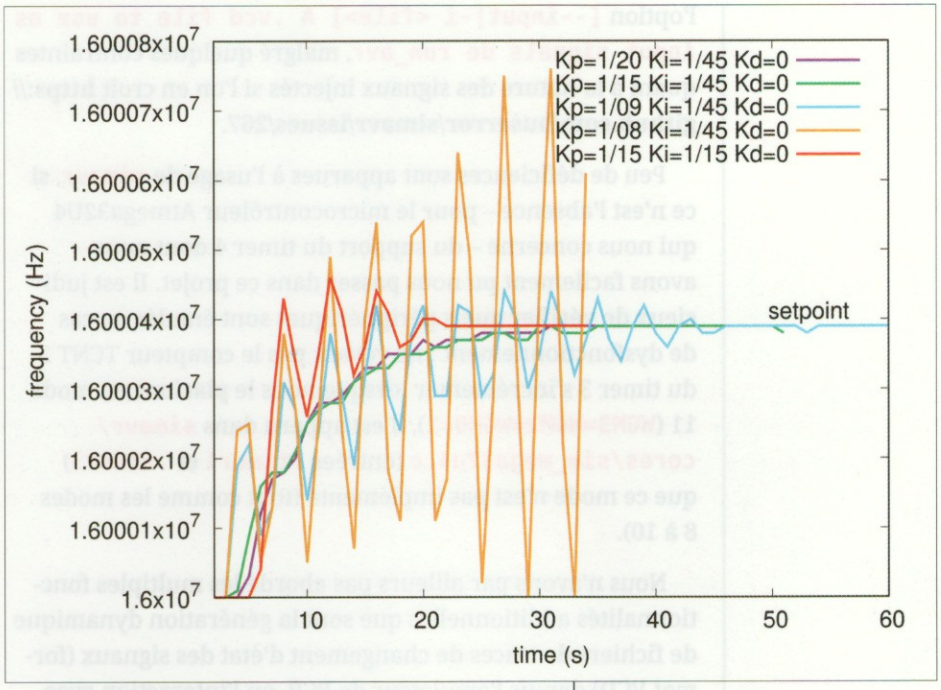



*Figure 7 : Évolution de la fréquence de l'oscillateur en fonction des paramètres de la loi de commande proportionnelle intégrale tels que discutés dans le texte. Les courbes violettes et vertes sont excessivement lentes à converger vers la consigne de 16 MHz+400 Hz, la courbe cyan présente des oscillations dangereuses, mais finit par se stabiliser, tandis que la courbe orange présente un gain excessif et donc des oscillations divergentes. La courbe rouge est un optimal alliant rapidité de convergence sans présenter d'oscillation excessive.*

## CONCLUSION

Un projet conçu pour être aussi expérimental que possible d'asservissement de la fréquence d'un oscillateur à quartz cadençant un microcontrôleur sur le 1-PPS de référence issu d'un récepteur GPS a été rendu virtuel, en nous appuyant sur `simavr` pour émuler le comportement du microcontrôleur Atmega32U4, mais surtout des divers périphériques qui le stimulent. Nous avons vu que grâce aux *timers* proposés par `simavr`, nous obtenons une simulation avec une base de temps commune à celle du microcontrôleur, tandis que les divers stimuli issus du microcontrôleur (gestion d'interruption, changement d'état de broche, communication ou variation de l'état d'une PWM) sont restitués à l'outil de simulation, permettant de fermer la boucle entre le traitement embarqué dans le *firmware* et l'environnement physique émulé. On notera dans ce contexte la capacité à enregistrer des séquences de mesures et à les rejouer comme stimuli tel que nous informe

l'option `[--input|-i <file>] A .vcd file to use as input signals de run_avr`, malgré quelques contraintes quant à la nature des signaux injectés si l'on en croît **https://github.com/buserror/simavr/issues/267**.

Peu de déficiences sont apparues à l'usage de `simavr`, si ce n'est l'absence – pour le microcontrôleur Atmega32U4 qui nous concerne – du support du timer 4 dont nous avons facilement pu nous passer dans ce projet. Il est judicieux de vérifier quels périphériques sont émulés en cas de dysfonctionnement : ne voyant pas le compteur TCNT 3 du timer 3 s'incrémenter lorsque nous le placions en mode 11 (`WGM3=WGM1=WGM0=1`), il est apparu dans `simavr/cores/sim_mega32u4.c` (entrées `.timer1` et `.timer3`) que ce mode n'est pas implémenté (tout comme les modes 8 à 10).

Nous n'avons par ailleurs pas abordé les multiples fonctionnalités additionnelles que sont la génération dynamique de fichiers de traces de changement d'état des signaux (format VCD) depuis l'émulateur de PCB, ou l'interaction avec `gdb` qui est très bien décrite dans **[12]**. L'auteur de `simavr` nous met cependant en garde que les valeurs des compteurs de timers ne sont calculées qu'à leur utilisation par le cœur de processeur, et que `gdb` sondant l'état de ces registres observera une valeur erronée qui ne s'incrémente pas en l'absence de sollicitation par un périphérique.

Le résultat de ce travail est disponible à **https://github.com/jmfriedt/l3ep/** dans le sous-répertoire `board_project` à côté des divers exemples de code qui sont proposés en cours d'introduction à la programmation des microcontrôleurs 8 bits de Licence 3 de l'Université de Franche-Comté à Besançon.

## RÉFÉRENCES


[1] J.-M. Friedt, *Développer sur microcontrôleur sans microcontrôleur : les émulateurs, GNU/Linux Magazine HS 103* (2019), à **https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMFHS-103/Developper-sur-microcontroleur-sans-microcontroleur-les-emulateurs**

[2] J.-M. Friedt & al., *Les microcontrôleurs MSP430 pour les applications faibles consommations – asservissement d'un oscillateur sur le GPS, GNU/Linux Magazine France 98* (2007)

[3] Atmel, *ATmega16U4/ATmega32U4 datasheet*, révision 7766H (06/2014)

[4] U-Blox, *GPS-based Timing Considerations with u-blox 6 GPS receivers – Application Note* (2011) à **https://www.u-blox.com/sites/default/files/products/documents/Timing_AppNote_(GPS.G6-X-11007).pdf**

[5] Guide orientant l'asservissement d'un oscillateur à quartz sur un signal de référence horaire issu d'un récepteur GPS : **http://jmfriedt.free.fr/projet_atmega.pdf**

[6] N. Minorsky *Directional stability of automatically steered bodies*, J. American Society for Naval Engineers 34(2), pp. 280–309 (1922)

[7] K. Åström & T. Hägglund, *PID controllers – 2nd Ed.*, Instrument Society of America (1995)

[8] S. Bennett. *A History of Control Engineering 1930-1955*, IET (1993)

## REMERCIEMENTS

M. *BusError* Pollet, auteur de `simavr`, a amélioré ce manuscrit par ses relectures des versions préliminaires, et motivé la finalisation de l'étude par sa disponibilité sur IRC. G. Cabodevila (enseignant-chercheur à l'École Nationale Supérieure de Mécanique et des Microtechniques) m'a enseigné la mise en œuvre en temps discret de la loi de commande proportionnelle, intégrale et dérivée, et l'identification des coefficients de pondération par la méthode de Takahashi ainsi que l'implémentation de la méthode itérative. Toutes les références bibliographiques qui ne sont pas librement disponibles sur le Web ont été acquises auprès de *Library Genesis* à **http://gen.lib.rus.ec**, une ressource indispensable à nos activités de recherche et d'enseignement. MF

[9] J.G. Ziegler & N.B. Nichols, *Optimum settings for automatic controllers*, Trans. ASME 64 pp. 759-768 (1942)

[10] Y. Takahashi, C.S. Chan & D.M. Auslander, *Parametereinstellung bei linearen DDC-Algorithmen, Automatisierungstechnik* (1971), 237-244

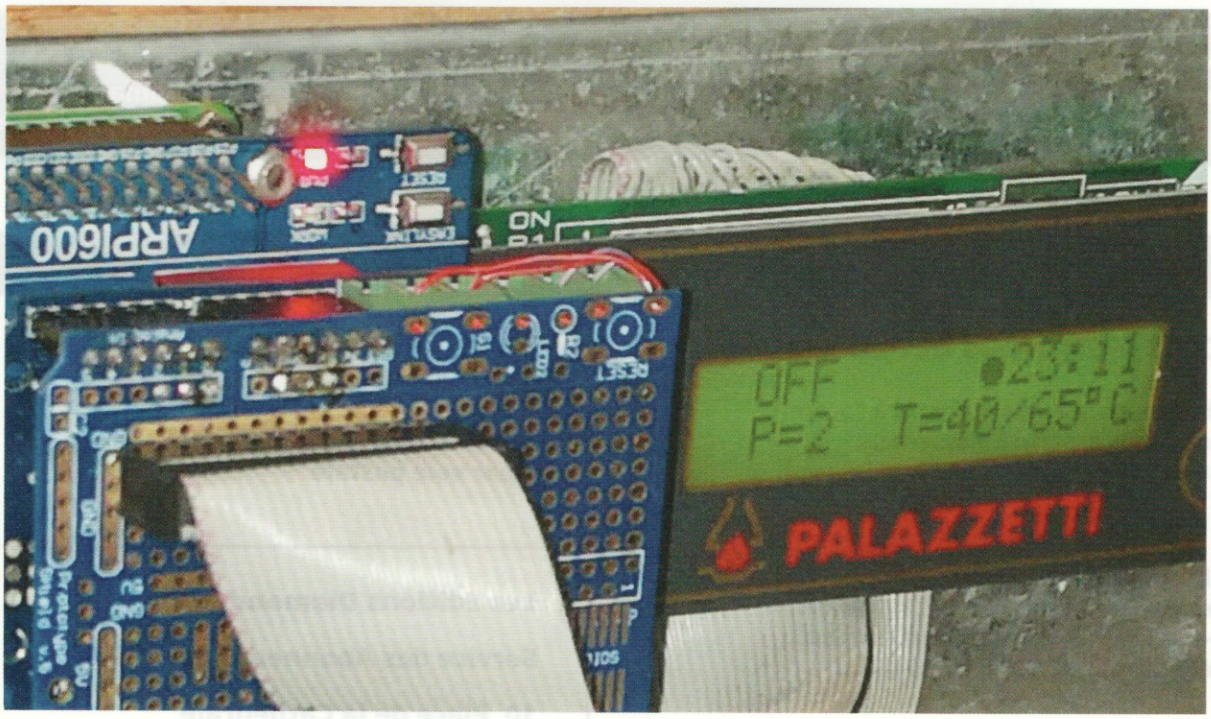
[11] A. Besançon-Voda & S. Gentil, *Régulateurs PID analogiques et numériques*, Tech. de l'ingénieur R7416 (1999), ou sans les fautes, le cours de Gonzalo Cabodevila disponible à **http://jmfriedt.free.fr/Gonzalo_cours1A.pdf**

[12] Manuel de simavr par J. Gruber dans le répertoire doc du projet, ou L. Kellogg-Stedman, *Debugging attiny85* code (2019) à **https://blog.oddbit.com/post/2019-01-22-debugging-attiny-code-pt-1/**

# POÊLE À GRANULÉS CONNECTÉ

Thierry Descombes - Ingénieur en informatique au CNRS
& Abderrahmane Ghimouz - Doctorant en microélectronique à INPG

**Comme la plupart des produits électroniques industriels, les systèmes de chauffage se révèlent souvent très frustrants à l'utilisation, car très fermés. Aucune information technique n'est publique, et l'interopérabilité avec d'autres équipements est difficile. Lequel d'entre nous n'a pas déjà rêvé de pouvoir disposer d'une interface d'accès ouverte sur son système de chauffage, ou d'un mode qui le rendrait un peu plus intelligent et autonome ? Alors c'est parti : nous allons transformer notre poêle à granulés classique en un véritable système connecté IoT, intuitif et intelligent, disposant d'une interface d'accès ouverte.**



Poêle à granulés connecté

Lorsque j'ai fait l'acquisition de ma maison, j'ai aussitôt réalisé l'installation d'un système complet de chaudière « écologique », qui me permet à la fois de me chauffer, mais aussi d'avoir de l'eau chaude sanitaire toute l'année, pour une consommation électrique quasi nulle.

J'ai donc fait l'acquisition (sur Internet, pour 2500 € environ) d'un kit avec un panneau thermique à tubes (sous vide) complet fourni avec un chauffe-eau « tri énergie ». Ce dernier permet de produire de l'eau chaude à partir de 3 sources d'énergie : il est équipé de 2 échangeurs (des serpentins) dans lesquels circulent des liquides caloporteurs, et d'une résistance électrique (que je n'ai jamais eu besoin de raccorder).

Un échangeur est raccordé au circuit du panneau thermique (installé sur mon toit), l'autre devait être raccordé à une chaudière.

Mon choix (motivé par le prix et l'impact écologique) s'est porté sur l'installation d'un poêle à pellets hydraulique, installé au milieu de la maison, afin de maximiser son rendement.

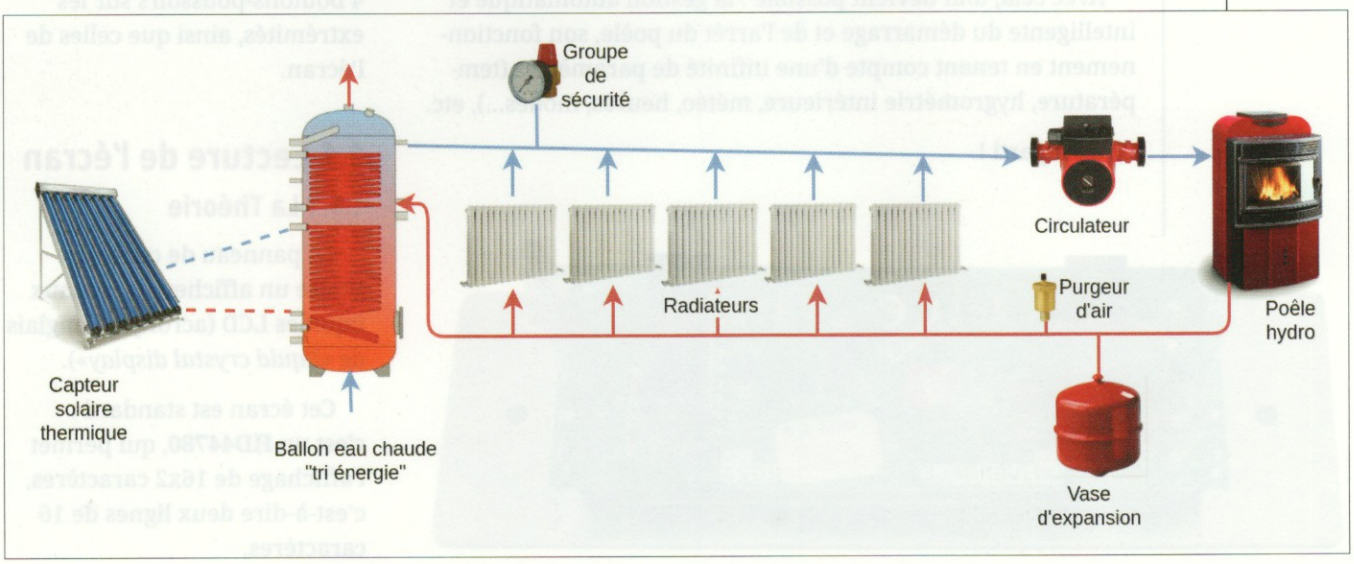
Comme mon budget était serré, j'ai fait l'acquisition d'un vieux poêle d'occasion : un Palazzetti Ecofire Maxi Hydro.

L'installation comporte 2 circuits similaires de liquides caloporteurs : celui de notre chaudière et celui du panneau (je ne l'ai pas détaillé, mais les composants et le principe sont les mêmes).

Le fonctionnement repose sur :

- un circulateur ;
- un groupe sécurité équipé d'un manomètre pour surveiller la pression, et d'un purgeur avec une vanne automatique qui s'ouvre lorsque la pression dépasse 3 bars ;
- un purgeur d'air automatique qui permet d'évacuer l'air du système ;
- des radiateurs ;
- le raccordement au chauffe-eau « tri énergie » ;
- un vase d'expansion qui permet de limiter l'augmentation de pression dans le système lorsque la température de l'eau augmente.

À l'usage, l'ensemble fonctionne parfaitement bien, mais les fonctionnalités du poêle se sont vite avérées limitées ! Elles sont vraiment minimalistes :



- le poêle est incapable de s'arrêter ou de démarrer en fonction de la température extérieure ;
- il est impossible de le faire démarrer à distance ;
- impossible d'avoir les informations de fonctionnement à distance ;
- pas de possibilité d'adapter la puissance. L'utilisateur fixe une valeur de puissance, parmi les 6 possibles. Le poêle conserve celle-ci, quel que soit l'environnement, jusqu'à ce que la température de l'eau ait atteint la consigne. Au mieux, en ajoutant un thermostat externe, il est capable de basculer à la puissance minimale, lorsque la température de consigne est atteinte, mais cela reste très limité ;
- pas de contrôle de la température de déclenchement du circulateur ;
- pas de nettoyage automatique du poêle, qui s'encrasse et finit par ne pas redémarrer au bout de 3 ou 4 cycles de fonctionnement.

Le plus pertinent m'a donc rapidement paru être de s'interfacer au panneau de contrôle et de pouvoir simuler l'utilisation des boutons de commande. En clair : il allait falloir contrôler l'appui sur les 4 boutons du panneau, et parvenir à relire l'afficheur LCD !

Chaque fonctionnalité ne correspond finalement qu'à une combinaison d'appui sur ces 4 boutons et tout devrait pouvoir être facilement accessible à distance via une interface REST standard, avec une réponse dans le format standard JSON.

Avec cela, tout devient possible : la gestion automatique et intelligente du démarrage et de l'arrêt du poêle, son fonctionnement en tenant compte d'une infinité de paramètres (température, hygrométrie intérieure, météo, heures, modes...), etc.

C'est parti !


# 1. CONTRÔLE COMMANDE

Le panneau de contrôle de ce poêle à granulés est commun à une trentaine de modèles de la gamme Palazzetti : les ECOFIRE « FLORA », « FREDDY », « GAJA », « GINEVRA », « GIORGIA », « JENNY », « LARA », « LORA », « MARGHERITA », « MOLLY », « MOLLY-SISSY », « NICOLETTA », « PATTY », « POLLY » et « SISSY ».

Il est basique : 4 boutons pression de 2 fils chacun, et un écran de 16 caractères et 2 lignes.

Le panneau est un composant indépendant du poêle. Il se démonte facilement et il est raccordé à la carte mère du poêle par une nappe. Au dos, sur le circuit électronique, se distinguent clairement les soudures des connecteurs des 4 boutons-poussoirs sur les extrémités, ainsi que celles de l'écran.

## 1.1 Lecture de l'écran

### 1.1.1 La Théorie

Le panneau de contrôle utilise un afficheur à cristaux liquides LCD (acronyme anglais de «*liquid crystal display*»).

Cet écran est standard : c'est un **HD44780**, qui permet l'affichage de 16x2 caractères, c'est-à-dire deux lignes de 16 caractères.

Il existe 2 façons de piloter ce type d'écran :

- soit en utilisant les 16 connexions (mode 8 bits) ;
- soit en utilisant 4 connexions de moins (mode 4 bits).

Ici, les 16 broches du connecteur sont reliées. Il est donc probable qu'il fonctionne en mode 8 bits.

Le standard 44780 requiert 3 broches de contrôle, et 4 ou 8 broches d'E/S pour le bus de données.

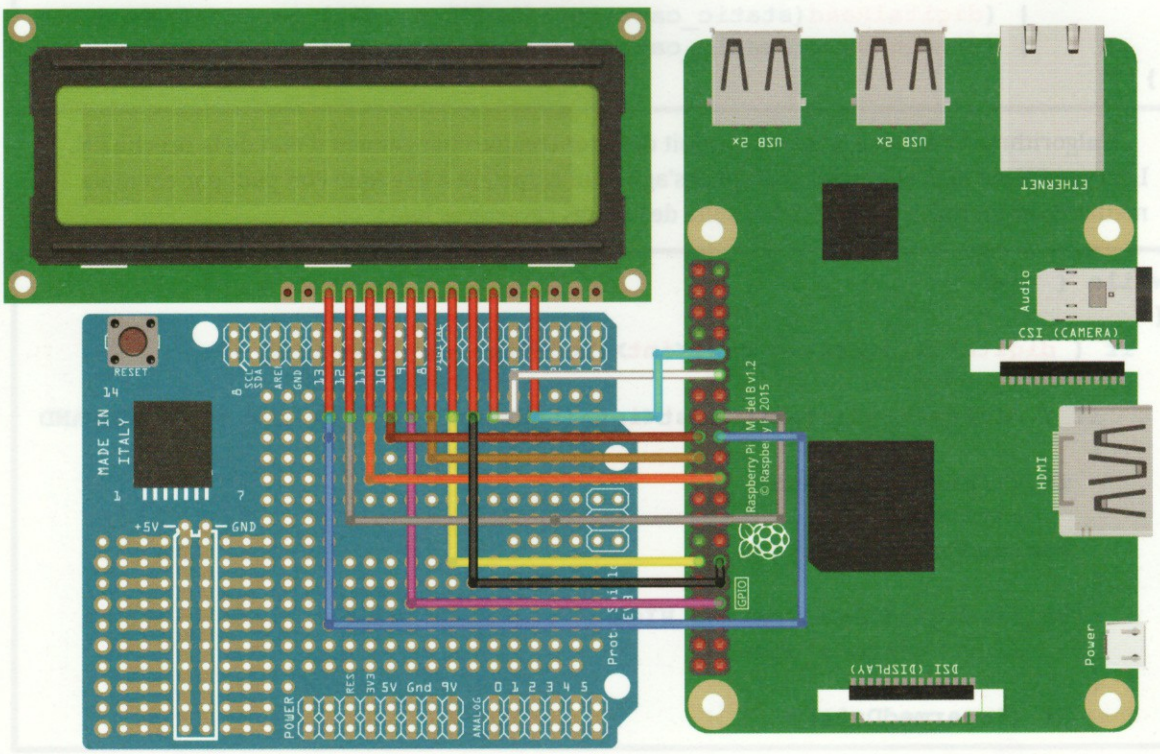
Les informations relatives à la connectivité et à la lecture/écriture de l'écran sont disponibles sur Wikipédia :

**https://fr.wikipedia.org/wiki/HD44780.**

Puisque le bus de données semble sur 8 bits, 11 broches sont donc nécessaires pour l'affichage (3 broches de contrôle, et 8 broches du bus de données).

Les trois broches de contrôle sont appelées EN, RS et RW.

- La broche EN est appelée «*Enable*». Cette ligne de contrôle est utilisée pour indiquer à l'écran LCD qu'il va recevoir des données. Lorsque les autres broches sont prêtes, et qu'elles sont à l'état où elles doivent être écrites ou lues, EN passe à *high* (1). EN revient ensuite à son niveau *low* (0).
- La broche RS correspond au «*Register Select*». Lorsque RS est à *low* (0), les données doivent être traitées comme une commande ou une instruction spéciale (par exemple, l'effacement de l'écran, le déplacement du curseur, etc.). Lorsque RS est *high* (1), les données envoyées sont des données textuelles qui vont être affichées à l'écran.
- La broche RW est la ligne de contrôle «*Lecture / écriture*».
- Lorsque RW est l'état *low* (0), les informations sur le bus de données sont en cours d'écriture sur l'écran LCD.



• Lorsque RW est à l'état *high* (1), le programme interroge (ou lit) efficacement l'écran LCD. Une seule instruction («*Get LCD status*») est une commande de lecture. Toutes les autres sont des commandes d'écriture. Dans ce cas, on peut donc considérer que l'état de **RW ne varie probablement jamais et qu'il est à low (0) en permanence.**

Enfin, le bus de données 8 bits est constitué de 8 broches. Ces lignes s'appellent D0, D1, D2, D3, D4, D5, D6 et D7.

Dans le cas présent, l'écran est déjà piloté par un contrôleur, il n'y a qu'un bus, et nous ne pouvons donc pas utiliser les commandes de lecture pour accéder au contenu de l'écran. La seule possibilité que nous avons est de faire l'acquisition en continu des broches de contrôle et de données.

En considérant que RW est continuellement à l'état « low », nous allons devoir relire en continu l'état de 10 broches : D0 à D7, ainsi que RS et EN.

En mode 8 bits, l'écran utilise un codage très proche de l'ASCII. La fonction de décodage d'un caractère se résume donc à lire les 8 broches et à décaler chaque bit lu en fonction de sa position :

```
char readLcdData()
{
  return  (digitalRead(static_cast<int>(LcdPins::d7)) == HIGH) << 7
        | (digitalRead(static_cast<int>(LcdPins::d6)) == HIGH) << 6
        | (digitalRead(static_cast<int>(LcdPins::d5)) == HIGH) << 5
        | (digitalRead(static_cast<int>(LcdPins::d4)) == HIGH) << 4
        | (digitalRead(static_cast<int>(LcdPins::d3)) == HIGH) << 3
        | (digitalRead(static_cast<int>(LcdPins::d2)) == HIGH) << 2
        | (digitalRead(static_cast<int>(LcdPins::d1)) == HIGH) << 1
        | (digitalRead(static_cast<int>(LcdPins::d0)) == HIGH);
}
```

L'algorithme de lecture, quant à lui, doit détecter chaque changement d'état de la broche EN. Lorsque EN est *high* et que RS est *high*, il s'agit d'un caractère à lire (que l'on peut par exemple mettre dans un buffer `bufText` à la suite des autres caractères lus) :

```
while ( true )
{
  if ( digitalRead(static_cast<int>(LcdPins::en)) == HIGH )
  {
    if ( digitalRead(static_cast<int>(LcdPins::rs)) == LOW ) // COMMAND
    {
      if (!rearmedCommand)
        continue;
      rearmedCommand=false;
    }
    else // digitalRead(rs) == HIGH
         // => write a char
    {
      if (!rearmedData)
```

```
            continue;
          rearmedData=false;

          if ( isprint(data)   && posCurr<MAX_SCREEN_SIZE)
            bufText[posCurr++]=data;
          setLcdMessage(bufText[posLine]);
        }
      }
      else
      {
        rearmedData=true;
        rearmedCommand=true;
      }
    }
```

### 1.1.2 Contraintes et choix du hardware

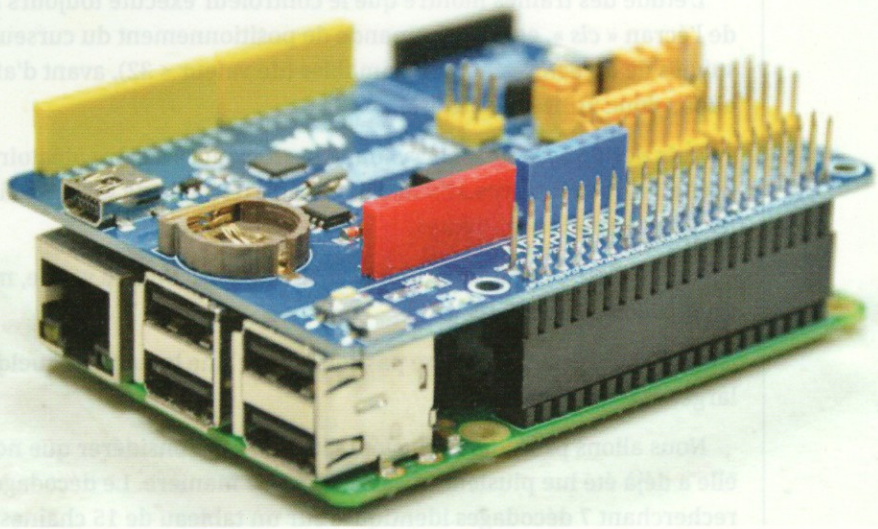
Pour ce projet, le choix de la carte embarquée doit tenir compte de certaines contraintes :

- elle doit disposer d'au moins 10 ports GPIO pour le LCD, et 4 pour commander les boutons ;
- elle doit être rapide et permettre l'acquisition de l'état des 10 broches du LCD en seulement quelques centaines de nanosecondes.

J'ai débuté le projet avec une Arduino UNO WiFi rev.2 et des cartes filles Arduino.

À l'usage, les tests m'ont paru assez laborieux (pas de shell, beaucoup de temps perdu à construire des images et à les transférer) et au final, l'Arduino s'est révélé trop lent : je n'ai jamais réussi à décoder correctement l'afficheur.

J'ai donc rapidement migré le projet sur un Raspberry PI 3A : avec un OS, des outils de développement, une compatibilité avec la plupart des projets GPL/GNU Linux ;-).



Afin de conserver mes cartes filles Arduino, j'ai fait l'acquisition d'une carte ARPI600, que j'utilise de manière très basique comme un bridge (concrètement, les pins Arduino sont routés vers des broches GPIO du Raspberry, via un mapping).

Avec tous ces mapping, il devient difficile de s'y retrouver ;-) :

| LCD pin | Arduino pin | RPI3 GPIO | WiringPi pin |
|---|---|---|---|
| D0 | 2 | P0 | 0 |
| D1 | 3 | P1 | 1 |
| D2 | 8 | P6 | 6 |
| D3 | 9 | P7 | 7 |
| D4 | 10 | CE0 | 10 |
| D5 | 11 | MOSI | 12 |
| D6 | 12 | MISO | 13 |
| D7 | 13 | SCK | 14 |
| EN | A1 | P21 | 21 |
| RS | A2 | P22 | 22 |

### 1.1.3 En pratique

Ce mapping se traduit en C++ par :

```
    enum class LcdPins : int { d0=0, d1=1, d2=6, d3=7, d4=10, d5=12, d6=13,
d7=14, en=21, rs=22 };
```

La détection de la fin de ligne nécessite un peu d'analyse.

L'étude des trames montre que le contrôleur exécute toujours au moins 2 commandes d'effacement de l'écran « *cls* », et une commande de positionnement du curseur à sa position initiale « *returnHome* », suivies de caractères non imprimables (de valeur < 32), avant d'afficher une chaîne de caractères. Nous allons donc utiliser ce critère.

Lors des tests, la Raspberry loupe parfois un caractère aléatoirement, et cela s'avère préjudiciable : la chaîne de caractères affichée devra être décodée de manière fiable pour pouvoir en extraire toutes les informations pertinentes.

Plutôt que de considérer que la chaîne lue est d'office valide, mieux vaut ajouter un peu de statistique à notre algorithme.

Le texte affiché ne change pas souvent, et une latence de quelques centaines de millisecondes est largement acceptable...

Nous allons procéder à plusieurs lectures, et considérer que notre chaîne est valide seulement si elle a déjà été lue plusieurs fois de la même manière. Le décodage se montre ainsi fiable à 100 %, en recherchant 7 décodages identiques sur un tableau de 15 chaînes lues.

L'algorithme C++ complet devient donc (**NB_LINES** vaut 15) :

```
void LcdReader::loop()
{
  for (int i=0; i<NB_LINES; i++)
    bufText[i][0]=0;

  initPinMode();

  while ( !exiting )
  {
    if ( digitalRead(static_cast<int>(LcdPins::en)) == HIGH )
    {
      if ( digitalRead(static_cast<int>(LcdPins::rs)) == LOW ) // COMMAND
      {
        if (!rearmedCommand)
          continue;
        rearmedCommand=false;

        char data = readLcdData() ;
        if (( data & 0x02 )== 0x02 ) //ReturnHome
          nbChome++;

        else
          if (data == 0x01) // cls
            nbCls++;

      }
      else // digitalRead(rs) == HIGH
           // => write a char
      {
        if (!rearmedData)
          continue;
        rearmedData=false;

        char data = readLcdData();
        if ((nbCls>=2 && nbChome) && (data < 0x20 ))
        {
          bufText[posLine][posCurr++]=0;
          int nbEq=0;
          for (int i=0; i<NB_LINES && nbEq < NB_LINES/2; i++)
            if (strcmp(bufText[posLine], bufText[(posLine + i)%NB_LINES] )== 0)
              nbEq++;
          if (nbEq == NB_LINES/2)
            setLcdMessage(bufText[posLine]);
          posLine=(posLine+1)%NB_LINES;
          posCurr=0;
        }
```

```
            nbChome=0;
            nbCls=0;

            if ( isprint(data)  && posCurr<MAX_SCREEN_SIZE)
              bufText[posLine][posCurr++]=data;

          }
        }
        else
        {
          rearmedData=true;
          rearmedCommand=true;
        }
      }
    }
```

La fonction membre `setLcdMessage()` récupère maintenant une chaîne valide, et elle peut en extraire toutes les informations de fonctionnement utiles, en fonction du menu affiché :

- Le mode de fonctionnement du poêle est défini comme pouvant prendre les valeurs suivantes :

```
enum class OperatingMode : int { unknown, on, off, starting, stopping,
cleaning, alertTempFume, alertTermDepr, delayedStart };
```

- la température de l'eau dans le poêle et la consigne ;
- la puissance de fonctionnement : un entier compris entre 1 et 6.

Concrètement :

```
void LcdReader::setLcdMessage(const char *msg)
{
  lock_guard<std::mutex> lk(mutex_lcdMessage);

  if (lcdMessage == msg) return;

  lcdMessage = msg;
  NVJ_LOG->append(NVJ_DEBUG, string ("new Lcd Message :" + lcdMessage));
  currentOperatingMode=OperatingMode::unknown;

  try
  {
    if ( lcdMessage.substr(0, 10) == "set TRAVAI" )
      currentOperatingMode=OperatingMode::on;
    else if ( lcdMessage.substr(0, 10) == "  OFF     " )
    {
if (stopping) // artificial behavior
  currentOperatingMode=OperatingMode::stopping;
```

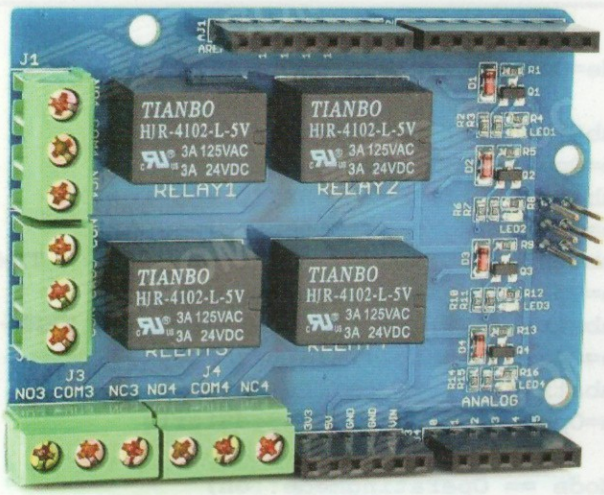
```
        else
          currentOperatingMode=OperatingMode::off;
      }
      else if ( lcdMessage.substr(0, 6) == "COMNC " )
        currentOperatingMode=OperatingMode::starting;
      else if ( lcdMessage.substr(0, 11) == "  NETTOYAGE")
        currentOperatingMode=OperatingMode::cleaning;
      else if ( lcdMessage.substr(0, 28) == "    ALARME          TEMP FUME" )
        currentOperatingMode=OperatingMode::alertTempFume;
      else if ( lcdMessage.substr(0, 28) == "    ALARME        TERM- DEPR." )
        currentOperatingMode=OperatingMode::alertTermDepr;
      else if ( lcdMessage.substr(0, 7) == "ATTENTE")
        currentOperatingMode=OperatingMode::delayedStart;

      if ( (currentOperatingMode == OperatingMode::on)
      || (currentOperatingMode == OperatingMode::off)
      || (currentOperatingMode == OperatingMode::starting) )
      {
        std::size_t found = lcdMessage.find("P=");
        if (found!=std::string::npos)
          power = stoi(lcdMessage.substr(found+2,1));

        found = lcdMessage.find("T=");
        if (found!=std::string::npos)
        {
          tempWater = stoi(lcdMessage.substr(found+2,2).c_str());
          tempWaterConsigne = stoi(lcdMessage.substr(found+5,2).c_str());
        }
      }
    }
    catch(...)
    {
      NVJ_LOG->append(NVJ_INFO, string ("EXCEPTION"));
    }
  }
```

## 1.2 Simuler l'appui sur les boutons de commande

La manière la plus simple et la moins intrusive de simuler l'appui sur un bouton est clairement de le court-circuiter. Pour cela, nous allons utiliser un relais électromécanique.

### 1.2.1 des relais pour simuler l'appui sur un bouton

Un relais électromécanique est un organe électrique qui va nous permettre (grâce à un électroaimant et un système de commutation électrique) de commander l'ouverture et la fermeture d'un circuit électrique par un second circuit, complètement isolé. C'est en fermant le circuit que le relais va donc court-circuiter notre bouton, simulant ainsi une pression.



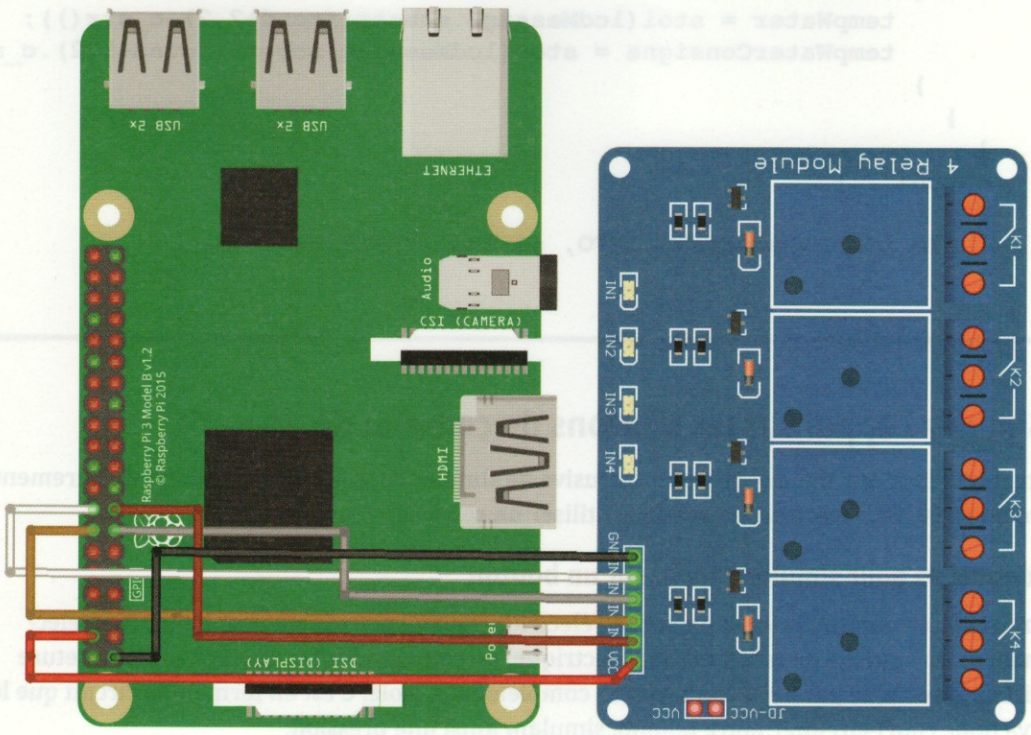

4 boutons sont à contrôler, il faut donc une carte avec 4 relais.

Les tensions ne dépassent pas les 5 V, un relais basique (sans opto-coupleur) est parfaitement adapté. Comme historiquement, le projet devait fonctionner sous Arduino, j'ai donc sélectionné la carte suivante.

La carte utilise les pins 4 à 7 de l'Arduino pour piloter les 4 relais, qui court-circuitent nos 4 boutons.

Comme j'utilise désormais un ARPI600, je dois à nouveau consulter les tableaux de mapping.

| Commande | Broche Arduino | RPI3 GPIO | Broche WiringPi |
|---|---|---|---|
| ON | 6 | P4 | 4 |
| OFF | 4 | P2 | 2 |
| UP | 7 | P5 | 5 |
| DOWN | 5 | P3 | 3 |



De manière analogue, je crée donc un type énuméré C++ :

```
enum class ControlButtons : int { on=4, off=2, up=5, down=3 } ;
```

L'appui sur un bouton se compose de 2 appels à la fonction **setButton** suivante :

```
void ButtonControl::setButton(const ControlButtons b, const int state)
{
  digitalWrite(static_cast<int>(b), state);
}
```

le premier avec **state = HIGH**, et le second avec **state = LOW**.

Il faut aussi tenir compte de la durée de pression de ces boutons, qui peut être courte (une demi-seconde environ), normale (2 secondes environ) ou longue (3 secondes et demie)...

Lors des tests, la durée de la pression « longue » ne suffisait pas toujours. J'ai donc dû ajouter une pression à durée automatique, qui maintient les boutons jusqu'à ce que l'affichage change.

```
enum class ButtonPressionDuration { quick, normal, longer, automatic };
```

La pression d'un bouton est donc simulée par la fonction membre **pressButton** suivante :

```
void ButtonControl::pressButton(const ControlButtons b,
const ButtonPressionDuration duration)
{
  setButton(b, HIGH);

  if (duration == ButtonPressionDuration::automatic)
  {
    string curMsg;
    int i = 0;
    do
    {
      curMsg = lcdReader->getLcdMessage();
      usleep(200000);
      i++;
    }
    while (curMsg == lcdReader->getLcdMessage() && i<50);
  }
  else
  {
    usleep(1000*500);
    if (duration != ButtonPressionDuration::quick) // Add a delay
    {
      usleep(1000*1500);
      if (duration != ButtonPressionDuration::normal) // Add a delay
        usleep(1000*1500);
    }
  }

  setButton(b, LOW);
  usleep(1000*500);
}
```

L'accès à certains menus nécessite la pression de 2 boutons simultanément. Par chance, une durée de 2 secondes convient, dans tous les cas :

```
void ButtonControl::press2Buttons(const ControlButtons b1, const ControlButtons b2)
{
  setButton(b1, HIGH);
  setButton(b2, HIGH);
  delay(2000); // 2seconds
  setButton(b1, LOW);
  setButton(b2, LOW);
  delay(500);
}
```

### 1.2.2 les commandes : parcours des menus

Tous les menus du panneau de commande sont hiérarchisés en 4 sous-niveaux. Le menu principal permet d'avoir accès à la plupart des paramètres de fonctionnement. Dans tous les cas de figure, si le menu actif n'est pas le menu principal, nous le retrouvons après 4 pressions (maximum) sur le bouton off.

```
void ButtonControl::goToMainMenu()
{
  NVJ_LOG->append(NVJ_INFO, "goToMainMenu()");
  // start from main menu
  pressButton(ControlButtons::off);
  pressButton(ControlButtons::off);
  pressButton(ControlButtons::off);
  pressButton(ControlButtons::off);
}
```

Il reste seulement à décomposer l'accès aux menus et sous-menus, depuis le menu principal :

• Pour augmenter ou diminuer la puissance du poêle :

```
void ButtonControl::incPower(short step)
 {
   NVJ_LOG->append(NVJ_INFO, "incPower(" + to_string(step) + ")" );
   // start from main menu
   // increase puissance
   press2Buttons(ControlButtons::up, ControlButtons::down);
   pressButton(ControlButtons::on);
   pressButton(ControlButtons::on);

   for (int i=0; i<abs(step); i++)
     if (step > 0)
       pressButton(ControlButtons::up);
     else
```

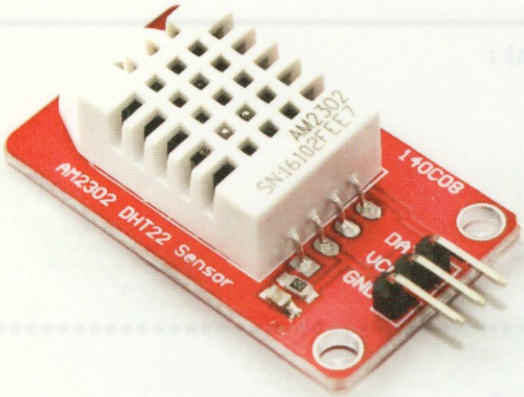
```
    pressButton(ControlButtons::down);

  pressButton(ControlButtons::on);
  pressButton(ControlButtons::on);
  pressButton(ControlButtons::off);
  pressButton(ControlButtons::off);
}

/*****************************************************************/
void ButtonControl::decPower(short step)
{
  incPower(-step);
}
```

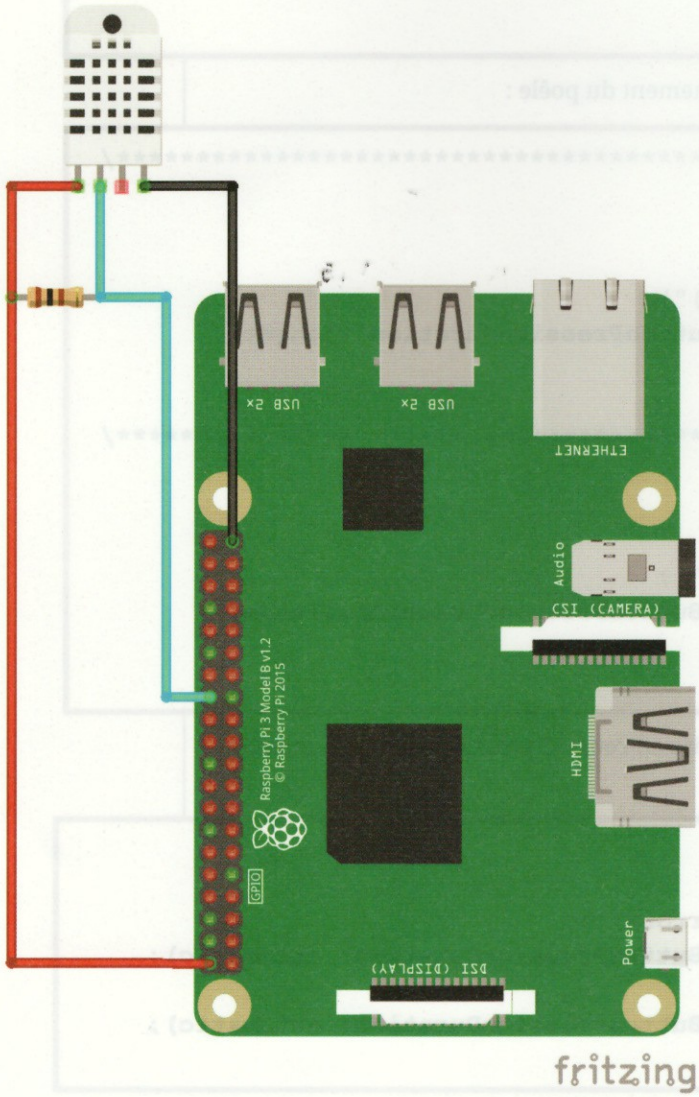
• Pour démarrer ou interrompre un cycle de fonctionnement du poêle :

```
/*****************************************************************/
void ButtonControl::start()
{
  NVJ_LOG->append(NVJ_INFO, "start()");
  pressButton(ControlButtons::on, ButtonPressionDuration::longer);
}

/*****************************************************************/
void ButtonControl::stop()
{
  NVJ_LOG->append(NVJ_INFO, "stop()");
  pressButton(ControlButtons::off, ButtonPressionDuration::longer);
  lcdReader->setStopping();
}
```

• En cas de dysfonctionnement, pour désarmer la remise en route (2 pressions de durée automatique sur le bouton *off*) :

```
void ButtonControl::resetError()
{
  NVJ_LOG->append(NVJ_INFO, "resetError()");
  pressButton(ControlButtons::off, ButtonPressionDuration::automatic);
  delay(2000);
  pressButton(ControlButtons::off, ButtonPressionDuration::automatic);
}
```




# 2. COLLECTE D'INFORMATIONS EXTERNES

## 2.1 Ajout d'un capteur de température/hygrométrie

Il est important que notre système puisse également disposer des informations de température et d'hygrométrie intérieures. En effet, ces données sont essentielles, aussi bien pour le contrôle à distance que pour la gestion automatique du démarrage, du choix de la puissance de fonctionnement, de son extinction, etc.

Après quelques recherches, j'ai décidé d'utiliser un capteur DHT22 (aussi appelé AM2302), car :

- il est économique (de l'ordre de 3 €, port compris) ;
- il est adapté (sa plage de fonctionnement varie de -40 à ~80 °C, et de 20 à 90 % de taux d'humidité) ;
- il a une précision correcte (de l'ordre de +/-0,5 °C et +/-2 % d'humidité) ;
- il a une distance de transmission relativement longue. La longueur du fil qui le relie au Raspberry peut atteindre plus de 20 m sans problème ; or nous cherchons précisément à pouvoir installer ce capteur loin de toute source de chaleur (un capteur utilisant l'I2C n'aurait pas pu être installé à plus de 2 m de notre Raspberry) ;
- sa lecture ne nécessite qu'un seul fil ;
- un seul composant est requis : une résistance *pull-up* de 10 kOhms, à placer entre l'alimentation et le bus de données.

En prime, le DHT22 est un capteur numérique avec convertisseur analogique numérique intégré. Le convertisseur facilite beaucoup la connexion du capteur au Raspberry Pi 3A, car alors traiter des puces supplémentaires s'avère inutile.

Le principal inconvénient de ce capteur est qu'il est plutôt lent. En effet, son taux d'échantillonnage n'est que d'une fois toutes les 2 secondes.

### 2.1.1 Protocole de communication

La communication sur un fil (*Onewire*) est propriétaire. Chaque session dure environ 4 ms.

Par défaut, le bus de données est à l'état haut du côté du DHT.

1. Nous devons maintenir ce bus à l'état bas *low* pendant 18 ms.
2. Puis nous lui envoyons une impulsion haute *high* qui doit durer de 20 à 40 µs.
3. Le capteur qui a reçu ce signal de démarrage place le bus de données à l'état bas *low* pendant 80 µs, puis haut *high* pendant 80 µs en guise d'accusé de réception.
4. Le capteur transfère ensuite les données.

Au total, 40 bits (5 octets) sont transmis. 4 octets correspondent à des données, et le 5e est une somme de contrôle qui permet de vérifier que le message reçu est bien celui qui a été envoyé.

La fonction `read_dht_dat()` qui effectue la lecture est donc :

```
bool DhtReader::read_dht_dat()
{
uint8_t laststate = HIGH;
uint8_t counter = 0;
uint8_t j = 0;

dht_dat[0] = dht_dat[1] = dht_dat[2] = dht_dat[3] = dht_dat[4] = 0;

pinMode( DHTPIN, OUTPUT );

digitalWrite( DHTPIN, LOW );
delay( 18 );
digitalWrite( DHTPIN, HIGH );
    delayMicroseconds( 40 );

pinMode( DHTPIN, INPUT );

for ( int i = 0; i < MAXTIMINGS; i++ )
{
 counter = 0;
 while ( digitalRead( DHTPIN ) == laststate )
 {
  counter++;
  delayMicroseconds( 3 );
  if ( counter == 255 )
   break;
 }
 laststate = digitalRead( DHTPIN );

 if ( counter == 255 )
  break;
```

```
  if ( (i >= 4) && (i % 2 == 0) )
  {
   dht_dat[j / 8] <<= 1;
   if ( counter > 16 )
    dht_dat[j / 8] |= 1;
   j++;
  }
 }

 return ( (j >= 40) &&
      (dht_dat[4] == ( (dht_dat[0] + dht_dat[1] + dht_dat[2] + dht_dat[3]) &
0xFF) ) );

  }
```

### 2.1.2 Décodage du DHT22

Si la lecture est valide (c'est-à-dire que la somme de contrôle est correcte), le DHT22 retourne donc 4 octets de données utiles :

- un taux d'humidité, en dixièmes de pour cent, sous la forme d'un entier non signé codé sur 16 bits (2 octets) ;
- une température en dixièmes de degré, sous la forme d'un entier signé sur 16 bits (2 octets). Le bit de poids fort indique le signe.

```
if (read_dht_dat())
{
   humi = (double)((dht_dat[0] << 8) + dht_dat[1]) / 10;
   temp = (double)(((dht_dat[2] & 0x7F) << 8) + dht_dat[3]) / 10;

   if ( dht_dat[2] & 0x80 )
  temp = -temp;
}
```

## 3. COLLECTE D'INFORMATIONS MÉTÉO

Il est également pertinent que notre système de chauffage ait accès à des informations de météo extérieure.

Ces informations sont, en effet, intéressantes pour plusieurs raisons :

- en cas de trop basse température extérieure, notre système devra retarder l'arrêt du poêle ;
- en cas de fort ensoleillement, mes baies vitrées étant orientées plein sud, notre système devra au contraire anticiper l'arrêt du poêle ;
- il paraît intéressant de pouvoir afficher ces informations sur le futur écran de contrôle du poêle.

Poêle à granulés connecté

Le site OpenWeatherMap offre une solution parfaitement adaptée et gratuite de consultation en ligne de la météo. Une fois enregistré, le site nous permet via une API complète de faire des requêtes (à l'aide d'un APPID fourni).

Pour connaître la météo en temps réel, l'API nous permet notamment de faire des requêtes à l'aide de coordonnées géographiques.

Il suffit pour cela d'ajouter les paramètres `lat` et `lon` qui correspondent aux coordonnées du lieu.

Voici un exemple de requête :

```
https://api.openweathermap.org/data/2.5/weather?lat=45.17&lon=5.72&units=metric&cnt=2&APPID={APIKEY}
```

Il est également possible de récupérer les prévisions météo de 1 à 6 jours en remplaçant `weather` par `forecast` dans l'URL ci-dessus.

Le serveur génère alors une réponse au format JSON contenant entre autres les variables `weather`, `main` et `wind`, qui contiennent les informations qui nous intéressent. Elles sont définies sous cette forme :

```
"weather": [{
 "id": 804,
 "main": "Clouds",
 "description": "overcast clouds",
 "icon": "04n"
}],

"main": {
 "temp": 7.98,
 "pressure": 1009,
 "humidity": 93,
 "temp_min": 4.44,
 "temp_max": 10.56
},
"wind": {
 "speed": 3.1,
 "deg": 250
}
```

Le champ `icon` de `weather` contient un identifiant sous la forme d'une chaîne de caractères

alphanumériques. L'icône correspondante est disponible directement sur le site, à l'URL : **http://openweathermap.org/img/w/{ICON_VALUE}.png** - avec **ICON_VALUE** correspondant à la valeur associée à la variable *icon* : « **04n** » dans l'exemple précédent.

La fonction de récupération des informations météo est donc la suivante :

```
char *resultJson = nullptr;
try
{
  resultJson = send_get_http_query ("api.openweathermap.org",
                "https://api.openweathermap.org/data/2.5/weather",
                "lat=45.37&lon=5.7&units=metric&cnt=2&APPID="+OPENWEATHER_APPID);
}
catch(...)
{
  NVJ_LOG->append(NVJ_ERROR, "OpenWeatherClient failed to send http request" );
  sleep(10);
  continue;
}

//resultJson contains the response (as a character array)
Document document;  // Default template parameter uses UTF8 and MemoryPoolAllocator.
if ( document.Parse<0>( resultJson ).HasParseError()
  || !document.HasMember("weather")
  || !document.HasMember("main")
  || !document.HasMember("wind") )
{
  NVJ_LOG->append(NVJ_ERROR, "OpenWeatherClient bad json document" );

  free(resultJson);
  disconnect();
  continue ;
}
free(resultJson);
disconnect();

temp = document["main"]["temp"].GetDouble();
humi = document["main"]["humidity"].GetInt();
wind_speed = document["wind"]["speed"].GetDouble(),
wind_dir   = document["wind"]["deg"].GetDouble();
icon = document["weather"][0]["icon"].GetString();

// (extrait du fichier src/OpenWeatherClient/OpenWeatherClient.cc)
```

La fonction **send_get_http_query** utilise la libSSL pour générer la requête HTTPS. Elle est incluse dans le code source du projet. Le framework rapidJson **[1]** nous permet ensuite d'accéder simplement aux variables JSON de la réponse générée par le serveur OpenWeatherMap.

# 4. INDICATION DU NIVEAU DE GRANULÉS

L'idée est de pouvoir suivre la quantité de granulés encore disponible dans la trémie du poêle et de pouvoir avertir l'utilisateur lorsque le niveau devient critique. Pour cela, j'ai utilisé un capteur ultrasons qui mesure la distance entre une position référence et la surface des granulés dans le stock, comme l'illustre la figure en haut ci-contre.

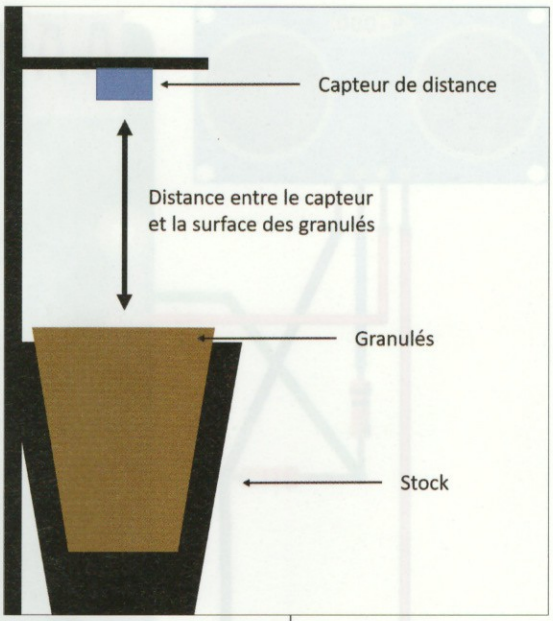
La surface de granulés change de position suivant la consommation et la distance mesurée est quasiment proportionnelle à la quantité disponible de granulés.

J'ai choisi le capteur HC-SR04, il permet de mesurer une distance de 2 à 400 cm et est très abordable (puisque deux modules coûtent environ 8 €, port compris).

## 4.1 Principe de fonctionnement

Un émetteur d'ultrasons envoie un train d'impulsions sonores (8 impulsions à 40 kHz). Elles se réfléchissent sur un obstacle et reviennent vers un récepteur. Connaissant la vitesse du son dans l'air (environ 343 m/s), il suffit de diviser par 2 le temps mis par cette onde pour prendre en considération l'aller-retour et on calcule facilement la distance.

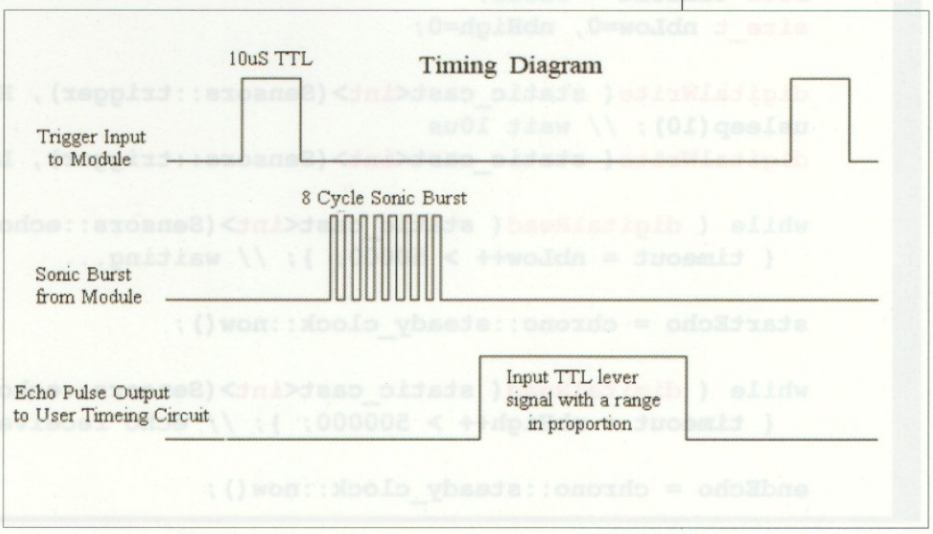
La circuiterie du HC-SR04 facilite les choses à l'utilisateur, puisqu'elle délivre un créneau de tension 5 V qui démarre juste après l'émission du train d'impulsions ultrasonores, pour s'arrêter lorsque le récepteur détecte ces impulsions en retour.

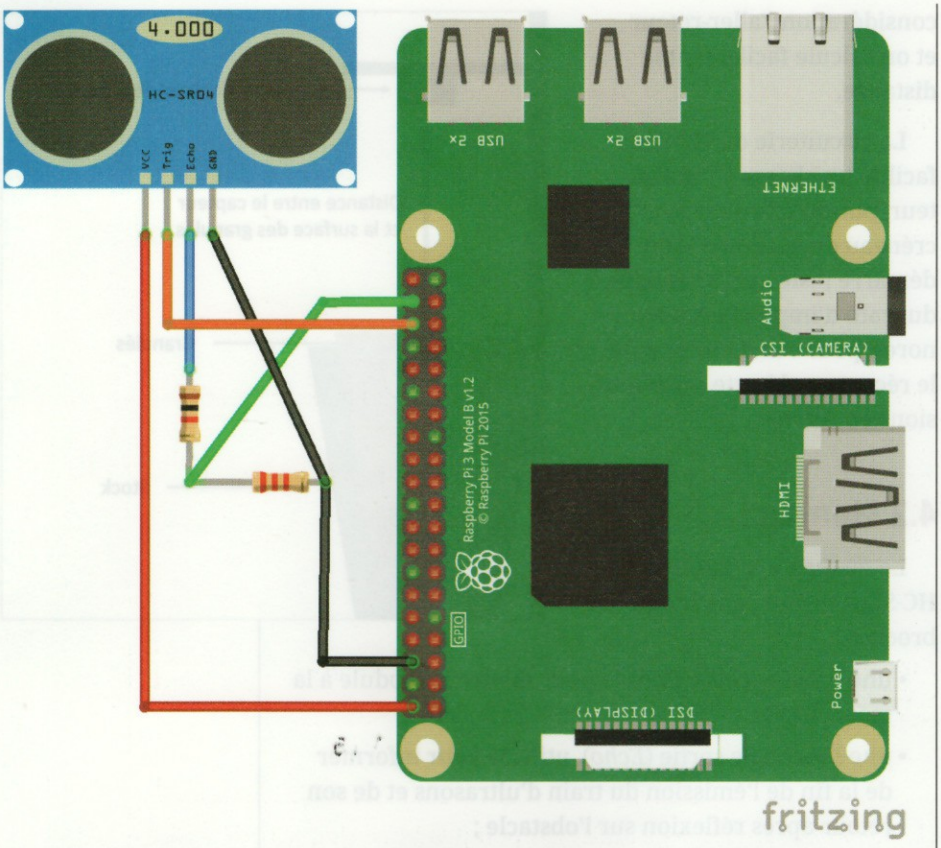



## 4.2 Montage

Le module à ultrasons HC-SR04 utilisé contient quatre broches :

- une broche (*Gnd*), utilisée pour mettre le module à la masse (0 V) ;
- une broche de sortie (*Echo*), utilisée pour informer de la fin de l'émission du train d'ultrasons et de son retour après réflexion sur l'obstacle ;
- une broche d'entrée (*Trig* pour *Trigger*), utilisée pour déclencher l'émission du train d'ultrasons ;
- une broche (*Vcc*), utilisée pour alimenter le capteur en 5 V.






Il est important de savoir que la sortie (*Echo*) de ce module est en 5 V, ce qui ne convient pas au GPIO de la Raspberry (limité à 3.3 V). Pour ceci, un diviseur de tension est utilisé pour passer de 5 V à 3.3 V.

Compte tenu des ports GPIO disponibles, j'ai fait le choix de raccorder le capteur comme ceci :

- Echo => GPIO 27 : pin 38 ;
- Trig => GPIO 28 : pin 36 ;
- VCC => 5 V : pin 2 ;
- GND= > : pin 6.

## 4.3 Algorithme de mesure

Les méthodes sont implémentées dans la classe **Gauge**.

La fonction **readMesure()** permet de mesurer la distance.

```
void Gauge::readMesure()
{
  chrono::steady_clock::time_point startEcho, endEcho;
  double distance = .0;
  bool timeout = false;
  size_t nbLow=0, nbHigh=0;

  digitalWrite( static_cast<int>(Sensors::trigger), HIGH );
  usleep(10); // wait 10us
  digitalWrite( static_cast<int>(Sensors::trigger), LOW );

  while ( digitalRead( static_cast<int>(Sensors::echo) ) == LOW && !timeout)
    { timeout = nbLow++ > 50000; }; // waiting...

  startEcho = chrono::steady_clock::now();

  while ( digitalRead( static_cast<int>(Sensors::echo) ) == HIGH && !timeout )
    { timeout = nbHigh++ > 500000; }; // echo received

  endEcho = chrono::steady_clock::now();
```

```
    if (timeout)
    {
      NVJ_LOG->append(NVJ_INFO, "Gauge: TIMEOUT ! nbLow="+to_string(nbLow)+",
nbHigh="+to_string(nbHigh) );
      return;
    }

    histDistance[nbDistance%GAUGE_NBVAL] = chrono::duration_cast<chrono::
nanoseconds>(endEcho - startEcho).count() * 1e-9 * 17150;

    nbDistance++;
  }
```

Pour diminuer les fluctuations, j'ai choisi de réaliser une mesure toutes les 10 secondes et de calculer la moyenne flottante sur les 10 dernières mesures. Cette valeur est calculée par fonction `getAvgDistance()` qui utilise le tableau `histDistance` contenant les dernières valeurs lues.

```
  double Gauge::getAvgDistance() const
  {
    double sum = .0;
    size_t i = 0;
    if (!nbDistance) return 0;
    size_t nbVal=(nbDistance>GAUGE_NBVAL)?GAUGE_NBVAL:nbDistance;
    for (size_t i=0; i < nbVal; i++)
      sum += histDistance[i];
    return sum / nbVal;
  }
```

La fonction `getLevel()` détermine le poids de granulés disponible, en appliquant une simple règle de 3.

```
  double Gauge::getLevel() const
  {
    const double topLevel = 15.0; // cm
    const double capacity = 60;  // Kg
    const double canisterHeight = 70; //cm
    double res = capacity * (1 - ( getAvgDistance() - topLevel ) / canisterHeight
) ; // in Kg
    if (res > 60 ) return 60;
    if (res < 0 ) return 0;
    return res;
  }
```

# 5. CONCEPTION LOGICIELLE

## 5.1 Analyse

À ce stade, il est nécessaire de bien définir l'application, côté *backend* (c'est-à-dire serveur).

Celle-ci doit être capable de :

- renvoyer les paramètres de fonctionnement du poêle, les paramètres environnementaux (température et hygrométrie intérieures et extérieures), la météo et toutes les informations disponibles ;

- contrôler complètement le fonctionnement du poêle à distance (à l'aide de fonctionnalités plus ou moins évoluées) ;
- activer/désactiver le mode automatique qui en fonction de l'état du poêle, de l'heure, des paramètres environnementaux (température et hygrométrie intérieures et extérieures) et des paramètres de présence est capable de démarrer le poêle, de l'arrêter, et de déterminer de manière pertinente la puissance à utiliser.

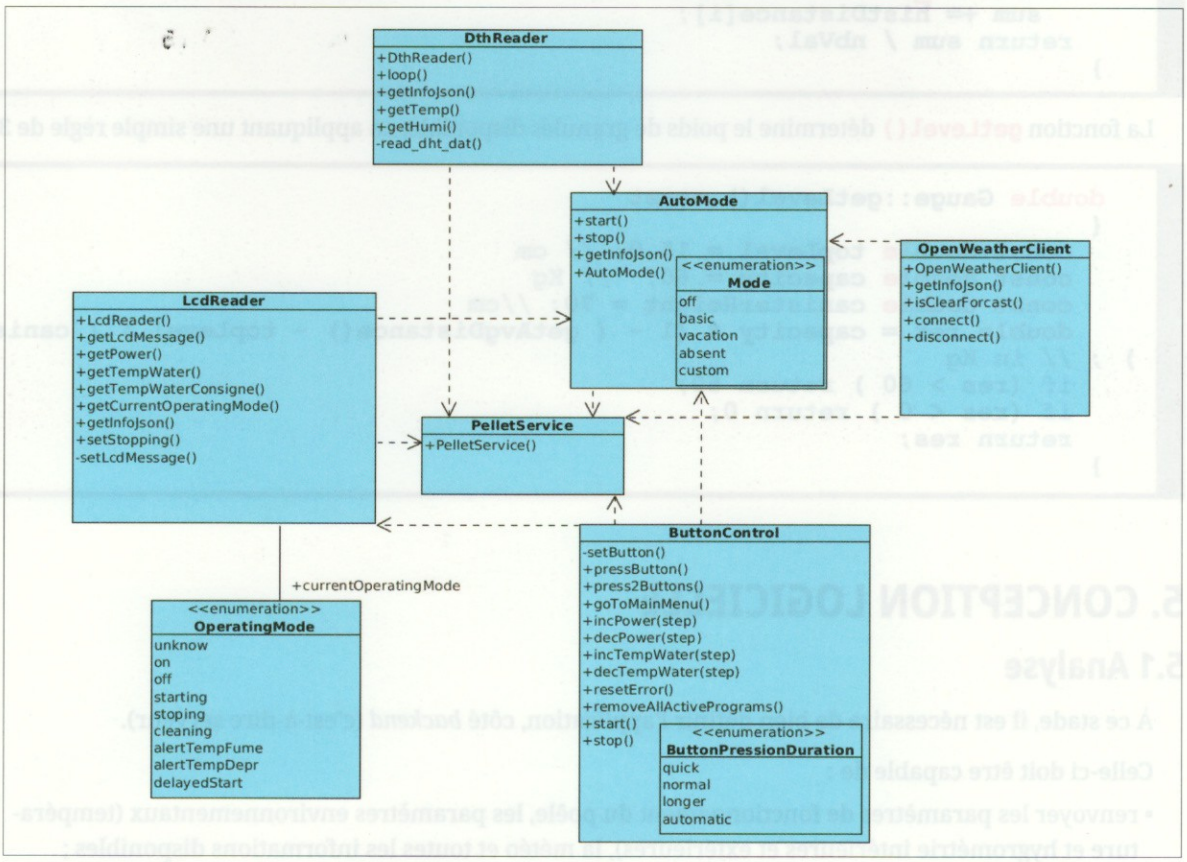
## 5.2 Conception objet

J'ai fait le choix de créer des objets représentant les modules logiques de notre système :

- une classe `LcdReader` pour relire l'écran LCD et d'en extraire certains paramètres ;
- une classe `ButtonControl` pour commander les boutons du panneau ;
- une classe `DhtReader` pour lire la température et l'hygrométrie ;
- une classe `OpenWeatherClient` pour accéder aux données météo du site OpenWeatherMap ;
- une classe `Gauge` pour évaluer la quantité de granulés restante.

J'ai ensuite ajouté 2 classes qui doivent être capables de communiquer avec les précédentes :

- une classe `PelletService` qui implémente un service web et permet l'interrogation et l'exécution de commande à distance ;
- une classe `AutoMode` capable de piloter automatiquement le système en fonction de l'ensemble des paramètres.



## 5.3 Implémentation multi-thread

Afin de ne pas engendrer de latence lors de la consultation des différentes informations et paramètres de fonctionnement, et d'avoir à tout moment accès à des informations à jour, ceux-ci sont lus et décodés en continu.

Les classes `LcdReader`, `DhtReader`, `OpenWeatherClient`, `Gauge` effectuent ainsi des tâches de lecture et le traitement, qui sont répétés indéfiniment. D'une manière un peu inspirée de l'Arduino, j'ai implémenté ces algorithmes dans des fonctions membres `loop()`.

Dans le cas présent (contrairement à l'Arduino), chacune de ces fonctions *loop* est exécutée au sein d'un *thread* indépendant qui effectue infiniment le traitement : tant que l'application fonctionne, c'est-à-dire que la variable condition `exiting` vaut faux.

Ce traitement débute à la création de l'objet, lorsque le constructeur de la classe s'exécute :

```
DhtReader::DhtReader()
{
  exiting = false;
  thread_loop=new thread(&DhtReader::loop, this);
  NVJ_LOG->append(NVJ_INFO, "DhtReader is starting" );
}
```

Et se termine lors de l'appel du destructeur, lorsque l'objet est détruit :

```
DhtReader::~DhtReader()
{
  exiting = true;
  thread_loop->join();
}
```

L'implémentation de la boucle infinie se trouve à l'intérieur de la méthode `loop()` :

```
void DhtReader::loop()
{
  while ( !exiting )
  {
    // ... loop treatment
  }
}

.. (extraits du fichier src/DhtReader/DhtReader.cc)
```

## 5.4 Mode automatique

Le mode automatique est également implémenté dans une classe possédant une fonction membre `loop()` qui exécute une boucle infinie au sein d'un *thread* indépendant.

Grâce aux communications avec les autres objets de notre système, nous pouvons désormais rendre notre poêle intelligent en implémentant un algorithme pertinent.

```
void AutoMode::loop()
{
 bool shutdownPeriod = false;
 bool needCleaning = false;
 sleep(5);

 while ( !exiting )
  {
    if ( ( currentMode == Mode::off )
       || ( lcdReader->getCurrentOperatingMode() == OperatingMode::alertTempFume )
       || ( lcdReader->getCurrentOperatingMode() == OperatingMode::alertTermDepr ) )
    {
       sleep(1);
       continue;
    }

    if ( lcdReader->getCurrentOperatingMode() == OperatingMode::starting )
    {
      sleep(10);
      continue;
    }

    if ( needCleaning && (lcdReader->getCurrentOperatingMode() == OperatingMode::off)
       && (currentMode != Mode::off ) )
    {
      buttonControl->stop();
      needCleaning = false;
      sleep(20*60); // 20 mn
    }

    std::time_t time_temp = std::time(nullptr);
    const std::tm * tm_local = std::localtime(&time_temp);

    shutdownPeriod = ( tm_local->tm_hour < 6 )
// 0 to 6 hours : off dans tous les modes
       || ( currentMode == Mode::basic && tm_local->tm_hour >= 9 && tm_local->tm_hour
< 17 && tm_local->tm_wday !=0 && tm_local->tm_wday !=6 )
// tm_wday (0 à 6) 0 dimanche - 6 samedi
       || ( currentMode == Mode::absent && tm_local->tm_hour >= 8 );
// work only from 6 to 8

    // if the pellet is running...
    if ( lcdReader->getCurrentOperatingMode() == OperatingMode::on )
    {
 // if the temp/hygro has been reached
       if (  ( currentMode == Mode::absent && dhtReader->getTemp() > 14.0 )
          || ( currentMode != Mode::absent && dhtReader->getTemp() > 20.0 )
          || shutdownPeriod )
```

```
        {
          buttonControl->stop();
          needCleaning = true;
          sleep(60*60); // 1h
          continue;
        }
        // else
        short deltaPower = 0;

        if (currentMode == Mode::absent)
          deltaPower = 1 - lcdReader->getPower();
        else
          deltaPower= std::min( 6,(short)(19.0-dhtReader->getTemp())+1)-lcdReader->getPower();

        if (deltaPower != 0)
          buttonControl->incPower(deltaPower);
        sleep(60); // 1 mn
        continue;
      }

      // if the pellet is stopped
      if (  ( lcdReader->getCurrentOperatingMode() == OperatingMode::off )
         && ! shutdownPeriod
  && ( ( dhtReader->getTemp() < ( 19.0 - dhtReader->getHumi() / 30.0 ) )
         && !( openWeatherClient->isClearForcast() && 20.0 - dhtReader->getTemp() <= 2 ) )
        )
      {
        buttonControl->start();
        sleep(20*60); // 20 mn
        continue;
      }
    }
  }
```

## 5.5 Implémentation de l'interface REST

Pour implémenter notre API REST, j'ai utilisé le *framework* libnavajo **[2]**, car il est simple et adapté.

L'implémentation d'un serveur web au sein de notre application ne nécessite que 3 lignes de code :

```
int main()
{
  webServer = new WebServer;
  webServer->startService();
  webServer->wait();
  return 0;
}
```

Un serveur web écoute désormais sur le port 8080, mais ne répond à aucune requête.

Afin que notre serveur libnavajo puisse répondre dynamiquement à des requêtes, nous allons utiliser un dépôt dynamique (`DynamicRepository`) contenant des pages dynamiques (`DynamicPage`) qui seront générées à la demande. L'implémentation de ces pages nous permettra de gérer l'interactivité de notre interface, d'analyser les valeurs des paramètres et d'afficher des informations relatives au fonctionnement de notre application.

Pour simplifier l'implémentation, nous allons définir l'objet `PelletService` comme étant une spécialisation d'un objet `DynamicRepository` :

```
class PelletService : public DynamicRepository
{
//...
};
```

Le démarrage de notre service web contenant notre `DynamicRepository` nécessitera simplement 2 lignes supplémentaires : l'instanciation de notre classe `PelletService` et son ajout en tant que *repository* web au *framework* libnavajo :

```
int main()
{
  webServer = new WebServer;
  PelletService service;
  webServer->addRepository(&service);
  webServer->startService();
  webServer->wait();
  return 0;
}
```

Nous devons maintenant renseigner notre `DynamicRepository`, afin qu'il contienne un certain nombre de pages dynamiques, les `DynamicPage`, qui doivent être instanciées au démarrage de l'application.

Nous avons besoin de 2 pages dynamiques, qui génèrent une réponse au format JSON **[3]** (qui est un format standard, adapté et peu verbeux) :

- une pour récupérer les informations de fonctionnement : `info.json` ;
- une pour envoyer des commandes au poêle : `command.json`.

Ces pages sont donc 2 objets `DynamicPage`, qui doivent être ajoutés à notre objet `DynamicRepository`, grâce à la méthode `add()`. Elles sont référencées par leurs URL absolues.

Ici, nous les ajoutons directement lors de la création de l'objet `PelletService`, dans le constructeur :

```
PelletService::PelletService()
{
// ...
    pelletInfoMonitor = new PelletInfoMonitor(dhtReader, lcdReader,
&openWeatherClient, autoMode);
    add("info.json",pelletInfoMonitor);
```

```
    pelletCommand = new PelletCommand(buttonControl, lcdReader, autoMode);
    add("command.json", pelletCommand);
}
```

Chacune de ces pages `pelletInfoMonitor` et `pelletCommand` doit surcharger la fonction (virtuelle pure) :

```
bool getPage(HttpRequest* request, HttpResponse *response);
```

Afin de pouvoir développer simplement une application *front-end* qui utilise un autre serveur web (sur un autre port, voire un autre ordinateur : avec un clavier, une souris et un vrai écran ;-), nous demandons à libnavajo de renvoyer une réponse qui autorise le Cross Site, grâce à la fonction `setCors()` :

```
/*************************************************************************/

bool PelletInfoMonitor::PelletInfoMonitor::getPage(HttpRequest* request,
HttpResponse *response)
{
  std::string json = "{ \"indoorData\" : " + dhtReader->getInfoJson() + ",";
  json += "\"pelletMonitor\" : " + lcdReader->getInfoJson() + ",";
  json += "\"outdoorData\" : " + openWeatherClient->getInfoJson() + ",";
  json += "\"autoMode\" : " + autoMode->getInfoJson();
  json += " }";
  response->setCORS(/*true, false, "http://192."*/);
  return fromString(json, response);
}
```

Les classes `DhtReader`, `LcdReader`, `OpenWeatherclient` et `AutoMode` possèdent chacune une fonction membre `getInfoJson()`. L'appel à ces fonctions permet de générer une réponse dynamique globale au format JSON :

```
/*************************************************************************/

bool PelletCommand::PelletCommand::getPage(HttpRequest* request, HttpResponse
*response)
{
// traitement et appels aux fonctions de nos sous-systèmes
if ( request->hasParameter( "start" ) )
{ /* ... */ }
else if ( request->hasParameter( "stop" ) )
{ /* ... */ }
//...
  return fromString(json, response);
}
```

La page dynamique `PelletCommand` utilise quant à elle un certain nombre de paramètres (`start`, `stop`...) qui permettent de contrôler complètement le système.

Et voilà !

Nous sommes désormais capables d'interagir avec notre système à distance !!!

```
$ curl localhost:8080/info.json
{ "indoorData" : {"temp":21.5,"humi":57.3},"pelletMonitor" : {"power":1,"tempWater":34,
"tempWaterConsigne":85,"operatingMode":"off"},"pelletGauge" : {"remaining":22.19394859714
285},"outdoorData" : {"temp":10.25,"humi":61,"wind_speed":5,"wind_dir":0,"icon":"http://
openweathermap.org/img/w/04d.png"},"autoMode" : {"mode":"basic"}

$ curl "http://192.168.8.10:8080/command.json?getLcdMsg"
{"success":true,"result":"  OFF      18:02  P=1  T=33/85C"}

$ curl "http://192.168.8.10:8080/command.json?decPower"
{"success":true}
```

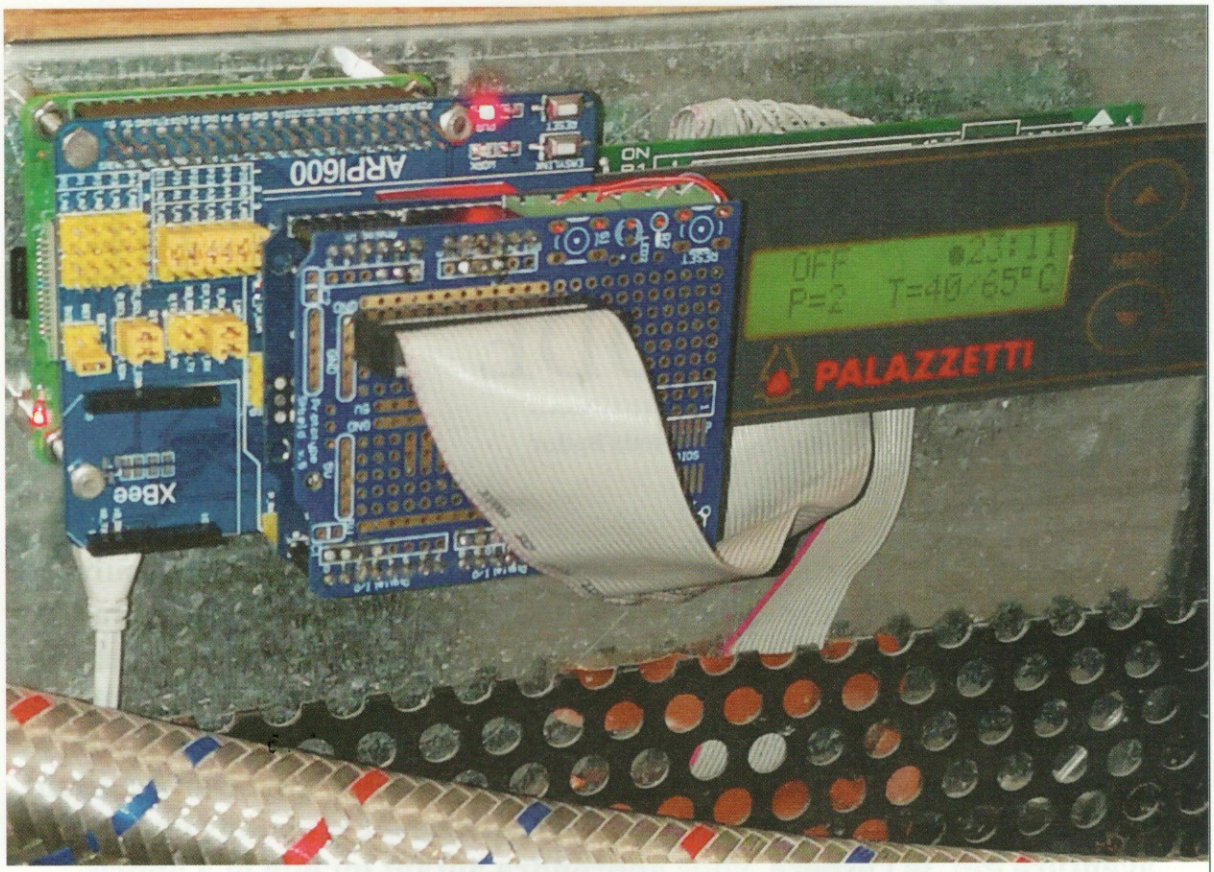
## CONCLUSION

En n'utilisant que des composants « grand public » et peu onéreux, il nous a été possible :

- de s'interfacer complètement avec le panneau de commande du poêle à granulés ;
- de relire les informations affichées sur l'écran LCD commandé par la carte mère du poêle ;
- L'utilisation de relais a permis de simuler l'appui sur les boutons de commande ;
- d'ajouter un capteur de température et d'hygrométrie ;
- d'interroger un site gratuit de météo en ligne ;
- d'intégrer un nouveau capteur de lecture du niveau de granulés.

Nous avons réussi à construire un logiciel qui intègre tous ces éléments, et notre poêle à granulés est devenu un véritable système connecté IoT, intuitif et intelligent.

Dans un développement qui dépasse un peu le cadre de cet article, j'ai également raccordé la Raspberry à un écran LCD tactile de 7 pouces, qui affiche une interface web (développée en AngularJS). Pour les plus courageux que cela intéresse, les sources sont disponibles sur GitHub **[3]**. **TD & AG**




## RÉFÉRENCES

[1] **https://rapidjson.org**

[2] **https://github.com/titi38/libnavajo**

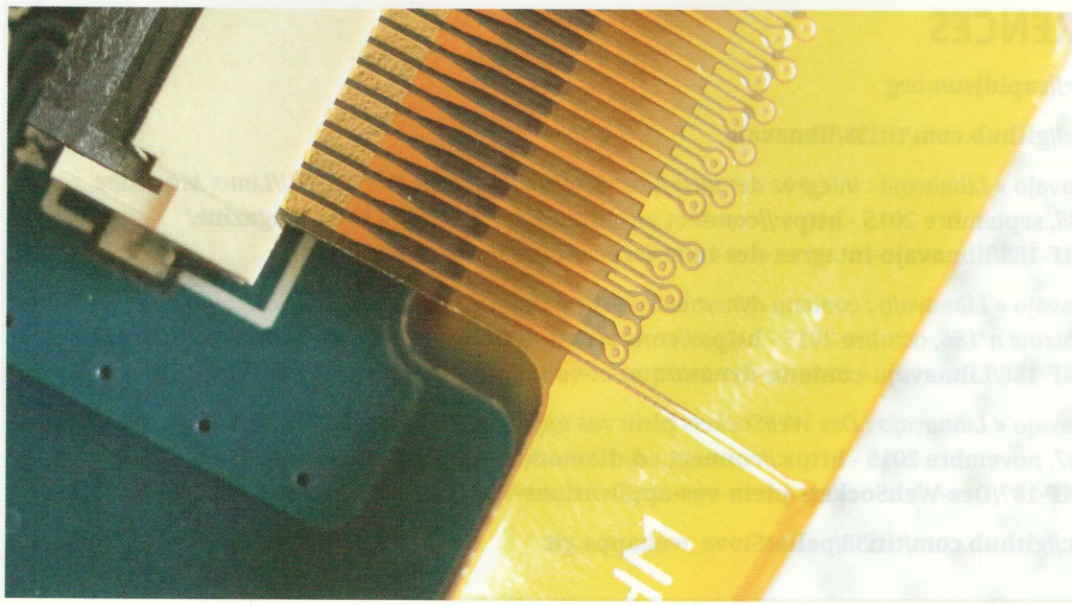
- libnavajo « *Libnavajo : intégrez des interfaces Web à vos projets C++* », *GNU/Linux Magazine n°185*, septembre 2015 - **https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-185/libnavajo-integrez-des-interfaces-Web-a-vos-projets-C**
- libnavajo « *Libnavajo : contenu dynamique et conception d'applications* », *GNU/Linux Magazine n°186*, octobre 2015 - **https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-186/Libnavajo-contenu-dynamique-et-conception-d-applications-Web**
- libnavajo « *Libnavajo : Des WebSockets plein vos applications web* », *GNU/Linux Magazine n°187*, novembre 2015 - **https://connect.ed-diamond.com/GNU-Linux-Magazine/GLMF-187/Des-WebSockets-plein-vos-applications-Web**

[3] **https://github.com/titi38/pelletStove_webapps.git**

# ÉCRAN E-PAPER NFC : UNE HISTOIRE D'EXPLORATION ET DE CODE

Denis Bodor

**Il est difficile de trouver un titre adéquat pour le contenu qui va suivre, car le matériel dont il sera question est tout aussi atypique qu'absolument captivant en termes de fonctionnement. Pire encore, ce n'est pas tant le matériel qui importe que l'approche nécessaire pour obtenir exactement le comportement attendu. Il sera donc ici question de NFC, de papier électronique (e-paper), de C et de la maxime voulant « qu'à cœur vaillant, rien d'impossible ». Sans attendre, embarquez avec moi dans une petite aventure qui, je l'espère, vous apprendra autant qu'elle m'a appris…**



Écran e-paper NFC : une histoire d'exploration et de code

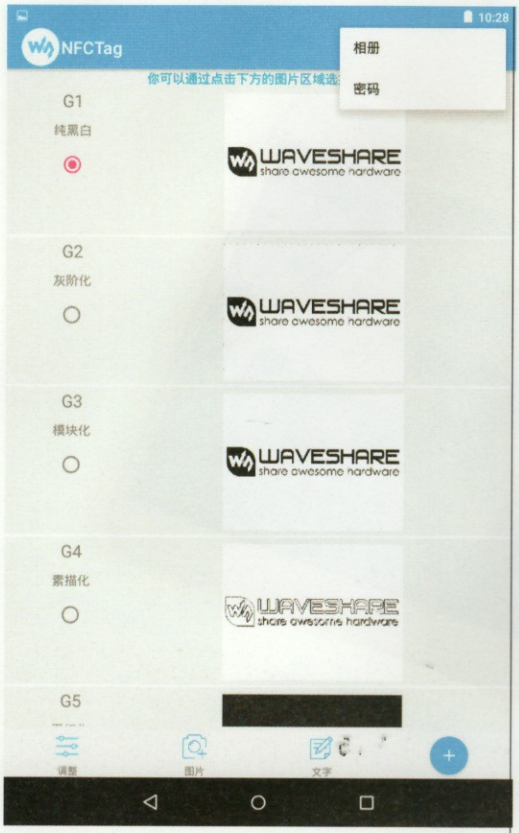
Notre histoire commence début janvier 2020 avec un simple message posté sur Slack, me révélant la disponibilité d'un nouveau produit développé par WaveShare. Nous connaissons déjà cette société de Shenzhen dans ces pages puisque nous nous étions frottés, il y a quelque temps, à leurs écrans *e-paper* tricolores (blanc, noir et jaune/rouge) pilotés par une carte ESP8266 (voir Hackable 27). WaveShare produit et commercialise une quantité impressionnante de modules, cartes, périphériques et accessoires en mesure de satisfaire bon nombre de fantasmes, idées et envies de tout amateur d'électronique et de systèmes embarqués (IoT, FPGA, écrans, *shields* Arduino, périphériques et HAT RPi, robots, Wi-Fi, etc.).

Mais le produit en question présente des caractéristiques vraiment captivantes. Jugez un peu : il s'agit d'un écran *e-paper* de 4,2 pouces, bicolore (noir/blanc), de 400 par 300 pixels, dans un boîtier plastique ABS très « propre », interfacé en NFC et... sans alimentation, ni connecteur ni accu intégré !

Comme vous le savez, le papier électronique présente une caractéristique très intéressante dans le sens où, une fois l'image affichée, aucune alimentation n'est nécessaire pour maintenir l'état des pixels. Ainsi, en plus de la facilité de lecture, plus naturelle, avec ces écrans, ce mode de fonctionnement permet de minimiser les besoins en courant. L'alimentation n'est nécessaire que pour changer l'état des pixels, si rien ne change, l'alimentation est inutile.

*L'écran e-paper NFC de WaveShare se présente comme un produit fini et non comme un kit de développement. La vidéo de présentation du constructeur décrit des applications types d'étiquetage pour les commerces, les séminaires, les stocks et les vitrines. Mais le matériel est bien plus polyvalent que cela.*

Couplez cela (sans mauvais jeu de mots) avec le fait que la technologie RFID/NFC permet d'alimenter un tag passif afin qu'il puisse réagir, et vous obtenez le concept derrière ce produit de WaveShare. Il n'a, en effet, pas besoin d'alimentation filaire ou d'accu, car c'est le champ magnétique produit par le périphérique NFC qui fournit le courant, aussi bien pour assurer la communication que pour rafraîchir l'écran avec de nouvelles informations. Le résultat est un matériel totalement passif pouvant être contrôlé avec un simple smartphone disposant de fonctionnalités NFC.




*L'application Android accompagnant l'écran est assez complète, mais son utilisation est relativement difficile en raison de grosses lacunes de traduction. Elle permet de composer des images à partir de textes et de photos, puis se charge de mettre à jour l'image sur l'écran via NFC.*

## 1. PREMIÈRE EXPLORATION

Bien entendu, devant autant de fonctionnalités innovantes et troublantes, impossible de résister à la tentation. Une commande eBay s'en suit donc pour 40 € l'unité (port offert), directement auprès de WaveShare et la livraison depuis Shenzhen se fait étrangement en un temps record (moins d'une semaine).

La page wiki du constructeur (**https://www.waveshare.com/wiki/4.2inch_NFC-Powered_e-Paper**) détaille des informations assez sommaires comme résolution (400×300), angle de vision (170°), taille de l'écran (84,8 par 63,3 mm) ou encore taille de l'ensemble (105 par 94,1 par 9,9 mm). Il n'y a aucune information concernant le protocole utilisé pour communiquer en NFC, pas la moindre trace de documentation technique ou de code. Rien, si ce n'est une application Android appelée « NFCTag », à télécharger directement sous la forme d'un fichier ZIP sur le site de WaveShare, et donc à installer manuellement sur un smartphone ou une tablette. Quelle déception !

Fort heureusement, je ne suis pas le seul à enquêter sur la bête et l'un des contributeurs Proxmark, Eric Betts, a déjà commencé son exploration en fouillant dans l'application Android. En effet, un fichier APK n'est rien d'autre qu'une archive ZIP contenant le code Java compilé, ainsi que les différents éléments utilisés par l'application. Or, en l'absence de techniques d'obfuscation destinées à cacher le fonctionnement du programme, il est relativement aisé d'obtenir quelque chose de très proche du code source originale en Java.

Dans l'archive APK/ZIP `NFCTag_EN.apk` se trouve un fichier `classes.dex`. Celui-ci contient une masse d'octets destinée à la machine virtuelle Dalvik exécutant les applications dans Android. Contrairement à une utilisation classique de Java, reposant également sur une machine virtuelle, il ne s'agit pas de *bytecode* Java, mais d'un format spécifique. La première étape consiste donc à retrouver un *bytecode* Java à partir du fichier DEX afin de pouvoir utiliser des outils de décompilation pour Java. Ceci se fait très simplement en utilisant, par exemple, un outil comme `dex2jar` (**https://github.com/pxb1988/dex2jar**) et nous obtenons alors un fichier `classes.jar`.

Ce nouveau fichier pourra ensuite être utilisé avec un décompilateur Java, comme JD-GUI (**http://java-decompiler.github.io/**) ou Procyon (**https://bitbucket.org/mstrobel/procyon**) pour obtenir un code source qu'il sera possible d'étudier. Le résultat obtenu est fonction de la qualité de l'outil et, bien souvent, vous devrez en

utiliser plusieurs pour arriver à reconstruire le fonctionnement du code dans votre esprit. En effet, la décompilation n'est pas une activité « normale » et les différents outils utilisés seront plus ou moins performants dans certaines tâches ou avec certaines parties de code.

Dans les pseudo-sources obtenues, on reconnaît assez facilement les méthodes et classes utilisées pour la communication NFC (fichier `MainActivity.java`), ainsi que des tableaux contenant ce qui semble être des commandes envoyées au tag (l'écran). La méthode `canSendPic()` par exemple semble être utilisée pour envoyer les données d'une image, après avoir initialisé l'écran, à grands coups de `this.tntag.transceive()`. Faire le tri dans quelque chose qui n'est clairement pas un vrai code source correctement structuré n'est pas chose facile et, alors que je défrichai la chose et tentai de reproduite la procédure en C, le contributeur Proxmark précédemment cité a publié un morceau de code en Node.js (**https://gitlab.com/bettse/wne_writer**).

Malheureusement, encore une fois, le portage en C de ce code, qui est plus un POC (*Proof Of Concept* ou preuve de concept en français) qu'autre chose, ne semblait apporter qu'un résultat très partiel. Certes, la communication avec le tag/écran via un adaptateur NFC USB ACR122 (ou SCL3711) fonctionnait en partie, et une part importante des détails de configuration s'éclaircissait, mais des erreurs de communication arrivaient systématiquement. Le simple fait que ces erreurs étaient à la fois systématiques, mais aléatoires me laissait penser que quelque chose n'allait pas en termes de temporisation. Ajouter quelques délais ne faisait que déplacer le problème d'une étape de la communication à une autre.

L'absence de code officiel ou de documentation ne permettait que dans une certaine mesure de comprendre et donc de respecter le protocole de communication, les messages à envoyer, l'interprétation des réponses et la cadence des échanges. Et c'est à ce stade que d'autres projets et tâches (comme préparer Hackable 33) ont fini par prendre le pas sur ces expérimentations ressemblant à une course d'orientation sans boussole, dans une forêt brumeuse, une nuit sans lune...

## 2. LA RÉVÉLATION

Mi-mars, alors que ce projet était en pause, WaveShare annonça une version 7,5 pouces du même produit ainsi qu'un outil modestement appelé « *ST25R3911B NFC Board* ». Ce kit de développement NFC, basé sur le microcontrôleur STM32F103RBT6 (ARM Cortex-M3 de chez STMicroelectronics),

*Confronté à une application générique NFC, l'écran se présente comme étant un tag NFC Forum Type 2. Avec NFC Tools PRO sous Android par exemple, il se fait passer pour un tag NXP MIFARE Ultralight, mais il s'agit en réalité d'une émulation.*

se présente comme un outil de lecture/écriture de tags NFC (ISO18092, ISO14443A, ISO14443B, ISO15693, FeliCa) équipé d'un écran OLED, de quelques boutons, de SRAM, d'un emplacement microSD et bien entendu, d'un circuit intégré ST25R3911B chargé de la partie NFC.

Cette plateforme de développement, facile à acquérir, là aussi, sur eBay pour quelque 25 € est très intéressante en soi, mais la page du wiki WaveShare (**https://www.waveshare.com/wiki/ST25R3911B_NFC_Board**) recèle une perle de choix : les sources du *firmware* de démonstration préinstallé dans la mémoire du microcontrôleur STM32, `ST25R3911B-NFC-Demo.zip`. Or, cette démonstration concerne précisément la mise à jour de l'image d'un écran *e-paper* NFC de WaveShare et ces sources contiennent, forcément, une implémentation juste du protocole permettant d'initier la communication et mettre à jour l'image affichée.

Et, effectivement, un petit coup d'œil dans le fichier `ST25R3911B-NFC-Demo/epd-demo/User/Browser/Browser.c` révèle l'information tant recherchée :

```
uint8_t step=0,progress=0;
uint8_t step0[2]={0xcd,0x0d};
uint8_t step1[3]={0xcd,0x00,10}; //select e-paper type and reset e-paper
                                 //4:2.13inch e-Paper
                                 //7:2.9inch e-Paper
                                 //10:4.2inch e-Paper
                                 //14:7.5inch e-Paper
uint8_t step2[2]={0xcd,0x01};    //e-paper normal mode  type£°
uint8_t step3[2]={0xcd,0x02};    //e-paper config1
uint8_t step4[2]={0xcd,0x03};    //e-paper power on
uint8_t step5[2]={0xcd,0x05};    //e-paper config2
uint8_t step6[2]={0xcd,0x06};    //EDP load to main
uint8_t step7[2]={0xcd,0x07};    //Data preparation
uint8_t step8[123]={0xcd,0x08,0x64};    //Data start command
                                        //2.13inch(0x10:Send 16 data at a time)
                                        //2.9inch(0x10:Send 16 data at a time)
                                        //4.2inch(0x64:Send 100 data at a time)
                                        //7.5inch(0x78:Send 120 data at a time)
uint8_t step9[2]={0xcd,0x18};    //e-paper power on
uint8_t step10[2]={0xcd,0x09};   //Refresh e-paper
uint8_t step11[2]={0xcd,0x0a};   //wait for ready
uint8_t step12[2]={0xcd,0x04};   //e-paper power off command
//uint8_t step13[2]={0xcd,0x0b}; //Judge whether the power supply is turned off successfully
//uint8_t step14[2]={0xcd,0x0c}; //The end of the transmission
```

Ces lignes proviennent de la fonction `Start_Dawring(void)` qui, bien entendu, est chargée de mettre à jour l'image sur l'écran. Cette déclaration de tableaux implique clairement une succession d'étapes permettant de configurer le périphérique de manière à le préparer pour recevoir les données. Quelques lignes plus loin, on découvre une boucle `while(1)` utilisant une variable `step` contenant notre position dans la succession d'étapes et déterminant quelle est celle qui doit être accomplie. Dans les déclarations, on peut également constater que les modèles 4,2 pouces et 7,5 pouces ne sont pas les seuls de la famille. Des versions 2,13 et 2,9 pouces semblent également de la partie (mais pour l'heure, introuvables).

Ce code nous présente donc le protocole « complet » que nous pouvons réimplémenter à notre façon (la structure composée d'une myriade de `if`/`else` n'est pas très à mon goût). L'architecture globale du protocole est la suivante :

- initialisation ;
- configuration ;
- mise en route de l'écran ;
- envoi des données ;
- rafraîchissement de l'écran avec les nouvelles données ;
- attente (rafraîchir un écran *e-paper* prend du temps) ;
- et coupure de l'alimentation.

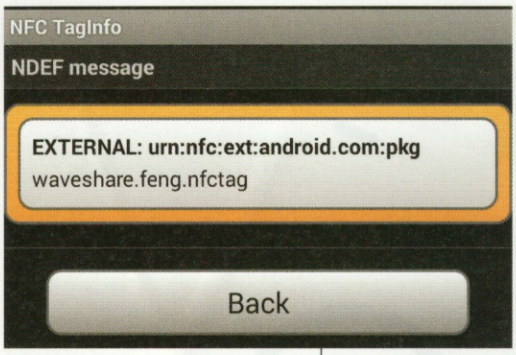
On remarque qu'une étape de la configuration ainsi que la façon d'envoyer les données sont dépendantes du modèle utilisé. Le dernier octet de `step1[]` change en fonction de la taille de l'écran et l'octet en position 2 de `step8[]` détermine le nombre d'octets d'un lot de données envoyé en une fois (100 pour le modèle 4,2 pouces). Nous avons ici un écran de 400×300 pixels, soit 120000 bits et donc 15000 octets. Pour satisfaire le périphérique, nous devons donc envoyer 150 fois 100 octets de données préfixés de la commande `0xcd 0x08 0x64`, soit des messages d'une taille totale de 103 octets. Le tableau déclaré dans le code de démonstration fait 123 octets, de manière à satisfaire les besoins du modèle 7,5 pouces (charge utile de 120 + 3 octets de commande).


*Pour communiquer avec l'écran sur PC ou Raspberry Pi, il est nécessaire de disposer d'un périphérique NFC comme ce très classique et populaire ACR122U d'ACS. Très économique, il vous permettra tout aussi bien de communiquer avec l'écran que de lire, écrire et tester toutes sortes de tags NFC et/ou RFID HF.*

L'autre point important du code concerne l'étape 11 : l'attente de la fin du rafraîchissement de l'écran. Ceci est important, et sans doute la source des nombreux problèmes dans les expérimentations basées sur l'étude du code de l'application Android, car en cessant la communication avec l'écran trop tôt, le champ électromagnétique disparaît et l'écran perd son alimentation, avant d'avoir pu faire quoi que ce soit. Il ne s'agit donc pas simplement de temporisation juste, mais d'obtenir confirmation qu'il est possible de cesser la communication sans risque.

## 3. L'IMPLÉMENTATION « MAISON »

Avant d'entrer techniquement dans le vif du sujet, peut-être devrais-je expliquer pourquoi chercher à réimplémenter l'algorithme et le protocole utilisé. En premier lieu, bien entendu, nous avons la simple




*Le tag de type 2 qui est émulé par l'écran contient des données NDEF. Il s'agit d'un format d'enregistrement standardisé permettant d'interagir avec les applications installées dans votre smartphone. Ici, l'objectif semble être de tout simplement lancer l'application NFCtag de WaveShare.*

curiosité technologique permettant d'appréhender pleinement une technologie, tout en gagnant de l'expérience dans des domaines satellites. Mais nous avons également une démarche plus pragmatique. En effet, l'utilisation d'un smartphone pour mettre à jour l'image sur l'écran semble la plus naturelle, si l'on fait abstraction du fait que l'application n'est que partiellement traduite, que son fonctionnement reste obscur et que le fait qu'elle ne soit pas diffusée par une plateforme standard présente un risque (les *stores* d'applications vérifient la qualité et le fonctionnement des applications validées).

L'objectif est donc de sortir du cadre imposé et d'avoir à disposition un code fonctionnel portable, d'une plateforme à une autre. Et ceci inclut, potentiellement, le fait d'utiliser ce nouveau code dans des projets personnels à base d'Arduino, d'ESP8266 ou 32, ou encore de Raspberry Pi. Pour obtenir cette souplesse, le langage à choisir est forcément celui pouvant être utilisé avec toutes ces plateformes, le C donc, et la première implémentation se fera avec un environnement offrant un maximum de souplesse de développement, et donc une machine GNU/Linux (PC ou RPi) disposant de tout le nécessaire (matériel, bibliothèques diverses, sources d'inspiration, fichiers de test, système de gestion de révision, etc.). Une fois que nous disposerons d'un outil pleinement fonctionnel permettant d'envoyer une image à l'écran, l'expérience ainsi acquise pourra être utilisée sereinement sur des plateformes moins adaptées à l'expérimentation.

Ce premier développement reposera, en premier lieu, sur l'utilisation de la LibNFC nous permettant de communiquer avec l'écran par l'intermédiaire d'un adaptateur NFC. Le plus courant de ces périphériques est l'ACR122U de *Advanced Card Systems* (ACS) et celui-ci se trouve très facilement, un peu partout, pour une vingtaine d'euros. Bien entendu, ce matériel n'est en rien spécifique à cet usage précis et pourra être utilisé avec toutes sortes de tags, de codes et d'expérimentations NFC par la suite, si vous n'en disposez pas déjà.

## 3.1 Communiquer avec le matériel

En premier lieu, nous avons besoin d'un squelette, une base qui pourra être utilisée pour n'importe quel développement autour du NFC. Nous voulons ici simplement initialiser la bibliothèque, accéder au périphérique, le configurer, détecter un tag (cible) et clore les opérations. Tout ceci commence par l'inclusion du fichier d'en-tête et la déclaration de variables globales :

```
#include <nfc/nfc.h>

// contexte LibNFC (options, référence, pointeur, etc.)
nfc_context *context;
// périphérique NFC
nfc_device *pnd;
```

Notez que je fais ici abstraction des éléments « normaux » d'un code en C pour ne traiter que de l'essentiel, afin de ne pas surcharger l'article. L'ensemble du code finalisé (qui continuera sans doute à évoluer après publication) est rendu public (**https://github.com/0xDRRB/wsNFCepaper**) et vous permettra de compléter tout cela. C'est aussi pour cette raison que les différents messages sont en anglais, c'est un choix personnel dès lors qu'un de mes projets est destiné à être ouvert.

Le reste se passera dans la fonction `main()` que je vous livre ici dans son ensemble commenté :

```
int main(int argc, char** argv)
{
    // le tag détecté
    nfc_target nt;

    // configuration modulation
    const nfc_modulation nmMifare = {
        .nmt = NMT_ISO14443A,
        .nbr = NBR_UNDEFINED, //NBR_106,
    };

    // initialisation LibNFC
    nfc_init(&context);
    if (context == NULL) {
        printf("Unable to init libnfc (malloc)\n");
        exit(EXIT_FAILURE);
    }
    // ouverture du périphérique
    pnd = nfc_open(context, NULL);
    if (pnd == NULL) {
        fprintf(stderr, "Error: %s\n", "Unable to open NFC device.");
        nfc_exit(context);
        exit(EXIT_FAILURE);
    }
    // initialisation en tant que lecteur
    if (nfc_initiator_init(pnd) < 0) {
        nfc_perror(pnd, "nfc_initiator_init");
        exit(EXIT_FAILURE);
    }

    printf("NFC reader: %s opened\n", nfc_device_get_name(pnd));

    // Désactivation du champ
    nfc_device_set_property_bool(pnd, NP_ACTIVATE_FIELD, false);
    usleep(200*1000);

    // configuration du périphérique
    nfc_device_set_property_bool(pnd, NP_EASY_FRAMING, false);
```

```
    nfc_device_set_property_bool(pnd, NP_HANDLE_PARITY, true);
    nfc_device_set_property_bool(pnd, NP_HANDLE_CRC, true);
    nfc_device_set_property_bool(pnd, NP_INFINITE_SELECT, false);

    // Réactivation du champ
    nfc_device_set_property_bool(pnd, NP_ACTIVATE_FIELD, false);

    // Attente d'un tag
    printf("Polling for target...\n");
    while (nfc_initiator_select_passive_target(pnd, nmMifare, NULL, 0, &nt) <= 0);
    printf("Target detected!\n");

    /********************/
    /* Faire des choses */
    /********************/

    // Fermeture et fin
    nfc_close(pnd);
    nfc_exit(context);
    exit(EXIT_SUCCESS);
}
```

Ceci est une configuration assez classique que l'on retrouve dans de nombreux exemples de la LibNFC. Notez cependant la ligne concernant la propriété **NP_EASY_FRAMING** passée à **false**, qui est indispensable pour l'écran WaveShare. L'*easy framing* est une facilité offerte par la LibNFC et la puce PN53x du périphérique pour dialoguer avec des tags standard, mais ici nous devrons communiquer de façon brute (*raw*) avec le tag/écran. Cette fonctionnalité, activée par défaut, perturberait nos échanges avec le matériel. Ce qui nous amène justement au cœur du sujet. Nous n'avons pas ici affaire à un tag NFC standard, mais à un matériel communiquant à l'aide de commandes envoyées sous forme d'octets et de réponses nous parvenant en retour. Un certain nombre de ces commandes sont standardisées, comme par exemple le fait de demander à lire un bloc de 16 octets avec un tag compatible NFC Forum Type 2, en envoyant **48** et **4** (**0x30 0x04**). C'est ce qu'on retrouve, par exemple, dans le **MainActivity.java** (ligne 361 à 363) de l'application Android.

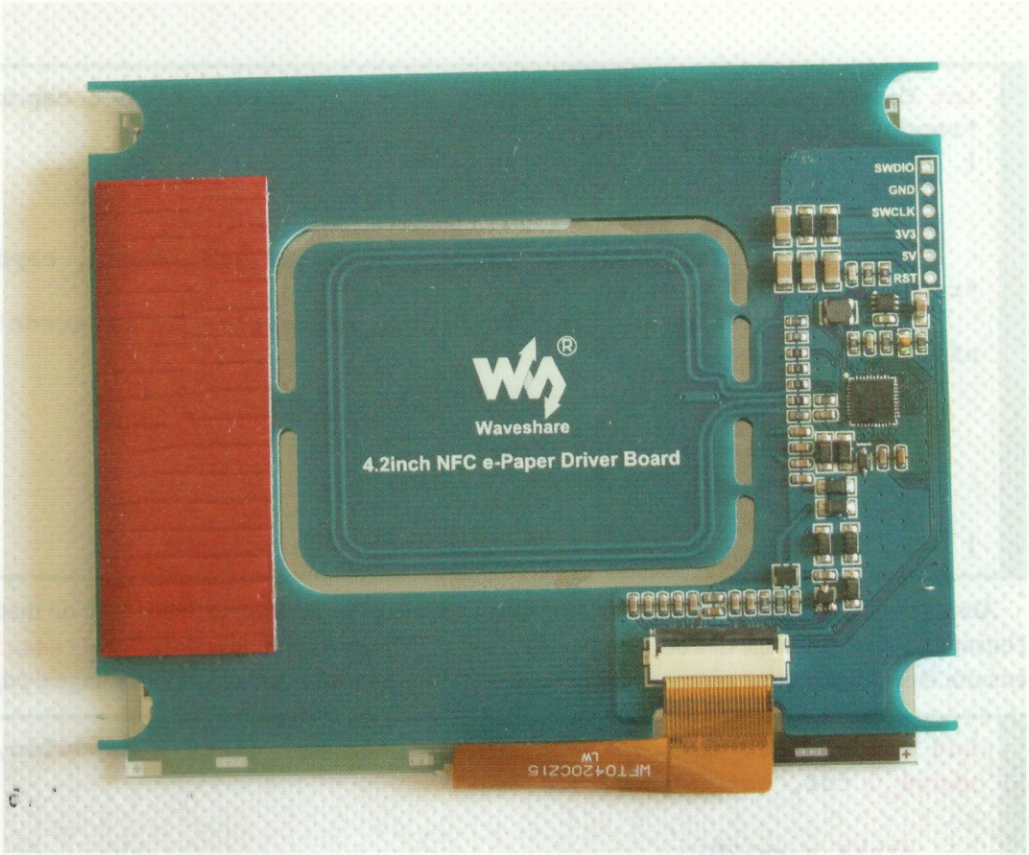
Mais d'autres commandes ne sont pas standard et c'est le cas de celles utilisées pour les fonctionnalités spécifiques à l'écran WaveShare. Ce mélange de standard et de non-standard est ce qui permet à l'écran d'apparaître comme un tag NFC Forum Type 2 contenant un message NDEF devant permettre le lancement de l'application (**waveshare.feng.nfctag**), tout en pouvant être contrôlé pour gérer l'image affichée. Ces commandes « hors normes » sont celles présentes dans les tableaux **step*** des sources du *firmware ST25R3911B-NFC-Demo* et que nous allons utiliser.

Pour transmettre ces commandes au tag/écran, la LibNFC nous fournit la fonction **nfc_initiator_transceive_bytes()**. Cette fonction de très bas niveau sera utilisée pour créer une fonction plus pratique pour notre code :

```
int CardTransmit(nfc_device *pnd, uint8_t *capdu, size_t capdulen, uint8_t
*rapdu, size_t *rapdulen)
{
    int res;

    if ((res = nfc_initiator_transceive_bytes(pnd, capdu, capdulen, rapdu,
*rapdulen, 500)) < 0) {
        fprintf(stderr, "nfc_initiator_transceive_bytes error! %s\n",
nfc_strerror(pnd));
        return -1;
    } else {
        *rapdulen = (size_t)res;
        return 0;
    }
}
```

Absentes ici, des lignes supplémentaires sont ajoutées dans le code distribué afin de mettre au point l'ensemble via l'affichage des octets envoyés et reçus. Cette nouvelle fonction `CardTransmit()` peut ensuite être utilisée pour gérer spécifiquement les commandes concernant l'affichage *e-paper* :

```
int sendcmd(nfc_device *pnd, uint8_t *capdu, uint8_t capdulen, uint8_t rb0,
uint8_t rb1, int nretry, int msec)
{
    uint8_t rx[20];
    size_t rxsz = sizeof(rx);
    int failnum = 0;

    rx[0] = rb0+1;
    rx[1] = rb1+1;

    while(1) {
        CardTransmit(pnd, capdu, capdulen, rx, &rxsz);

        // Are reply bytes ok ?
        if(rx[0]==rb0 && rx[1]==rb1) {
            return(0);
        } else {
            failnum++;
            usleep(msec*1000);
            if(failnum > nretry) {
                return(-1);
            }
        }
    }
}
```




*À l'intérieur du boîtier en plastique ABS se trouve un unique circuit imprimé avec, en son centre, l'antenne/bobine permettant à la fois l'alimentation et la communication. Sur la gauche, l'énorme masse rouge n'est pas un accu, mais un simple « rembourrage » en mousse permettant d'aligner le circuit dans le boîtier.*

Les arguments utilisés par cette fonction sont les suivants :

- `nfc_device *pnd` : notre périphérique NFC ;
- `uint8_t *capdu` : un pointeur sur un tableau d'octets à envoyer (le terme APDU est ici peu opportun en réalité) ;
- `uint8_t capdulen` : le nombre de ces octets ;
- `uint8_t rb0` et `uint8_t rb1` : deux octets attendus comme réponse de la part du tag/écran ;
- `int nretry` : le nombre de tentatives pour obtenir ces deux octets ;
- `int msec` : le délai en millisecondes entre deux tentatives.

Ces arguments découlent directement de la succession de `if`/`else` présente dans le code de démonstration pour STM32. Une logique qui est ici reprise, mais sous la forme d'appels à notre fonction et non plus d'une énorme boucle `while(1)` contrôlée par le contenu d'une variable `step`. `CardTransmit()` reprend une partie des arguments, dont le pointeur vers les données et leur taille, et ajoute un pointeur vers un tableau pour la réponse, ainsi qu'un pointeur vers la taille de cette dernière.

De cette réponse, seuls les deux premiers octets nous intéressent, `rx[0]` et `rx[1]`, qui doivent correspondre à `rb0` et `rb1`, pour confirmer l'exécution correcte de la commande. Enfin, la temporisation est prise en charge par la fonction `usleep()` prenant en argument un nombre de microsecondes (d'où le `*1000`).

Il nous reste une brique de base à créer pour pouvoir regrouper tous les appels dans `main()` et c'est celle permettant l'envoi des données de l'image, les fameux 150 segments de 100 octets :

```
int sendimg(nfc_device *pnd)
{
    int i;
    uint8_t rx[20];
    size_t rxsz = sizeof(rx);
    uint8_t segment[103] = { 0xcd, 0x08, 0x64 };

    for(i = 0; i<150; i++) {
        memcpy(segment+3, imgbuf+(i*100), 100);
        CardTransmit(pnd, segment, 103, rx, &rxsz);
        printf("Sending: %d %%\r", (i+1)*100/150);
        fflush(stdout);
    }
    printf("\n");

    return(0);
}
```

`imgbuf[]` sera déclaré globalement ainsi :

```
#define WIDTH   400
#define HEIGHT  300
#define BUFSIZE WIDTH*HEIGHT/8

uint8_t imgbuf[BUFSIZE] = { 0 };
```

Ce *buffer* devra, à terme, contenir l'image qui sera envoyée à l'écran, mais pour l'instant, simplement l'initialiser avec des `0x00` fera parfaitement l'affaire. Un bit à zéro correspond à un pixel noir et nous devrions donc, en phase de test avec ces données, obtenir un écran entièrement noir. Cette fonction `sendimg()` ne traite et ne traitera aucun fichier graphique. C'est, là encore, une fonction intermédiaire se contentant de prendre le contenu du *buffer*, le découpant (merci `memcpy()`) en 150 paquets de 100 octets et envoyant cela, accompagné de `0xcd 0x08 0x64`, à l'écran via `CardTransmit()`. Notez d'ailleurs que cette opération n'utilise pas les deux octets de réponse en guise de vérification ni dans l'application Android ni dans le code de démonstration STM32 (ce qui ne veut pas dire que le protocole ne l'impose pas, en principe).

Tout est prêt désormais pour compléter notre `main()`, là où nous avions laissé le commentaire « *Faire des choses* » :

```
    printf("Step 0 : init ?\n");
    if(sendcmd(pnd, step0, 2, 0, 0, 10, 0) != 0)
        errorexit("init failed!");
    usleep(200*1000);

    printf("Step 1 : select e-paper type and reset\n");
    if(sendcmd(pnd, step1, 3, 0, 0, 10, 0) != 0)
        errorexit("reset failed!");
    usleep(10*1000);
```

```
    printf("Step 2 : e-paper normal mode\n");
    if(sendcmd(pnd, step2, 2, 0, 0, 50, 0) != 0)
        errorexit("mode failed!");
    usleep(100*1000);

    printf("Step 3 : e-paper config1\n");
    if(sendcmd(pnd, step3, 2, 0, 0, 10, 0) != 0)
        errorexit("config1 failed!");
    usleep(200*1000);

    printf("Step 4 : e-paper power on\n");
    if(sendcmd(pnd, step4, 2, 0, 0, 10, 0) != 0)
        errorexit("power on failed!");
    usleep(500*1000);

    printf("Step 5 : e-paper config2\n");
    if(sendcmd(pnd, step5, 2, 0, 0, 30, 0) != 0)
        errorexit("config2 failed!");
    usleep(10*1000);

    printf("Step 6 : EDP load to main\n");
    if(sendcmd(pnd, step6, 2, 0, 0, 10, 0) != 0)
        errorexit("EDP load failed!");

    printf("Step 7 : Data preparation\n");
    if(sendcmd(pnd, step7, 2, 0, 0, 10, 0) != 0)
        errorexit("data preparation failed!");

    printf("Step 8 : Data start command\n");
    sendimg(pnd);

    printf("Step 9 : e-paper power on\n");
    if(sendcmd(pnd, step9, 2, 0, 0, 10, 0) != 0)
        errorexit("power on failed!");

    printf("Step 10 : Refresh e-paper\n");
    if(sendcmd(pnd, step10, 2, 0, 0, 10, 0) != 0)
        errorexit("refresh failed!");
    usleep(200*1000);

    printf("Step 11 : wait for ready\n");
    if(sendcmd(pnd, step11, 2, 0xff, 0, 70, 100) != 0)
        errorexit("wait for ready failed!");

    printf("Step 12 : e-paper power off command\n");
    if(sendcmd(pnd, step12, 2, 0, 0, 1, 0) != 0)
        errorexit("power off failed!");

    printf("E-paper Update OK\n");
```

On retrouve ici, dans les grandes lignes, la même logique que dans l'enchaînement de `if`/`else` de `Browser.c`, mais, je pense, implémentée de façon un peu moins « code spaghetti »...

La compilation du code et son exécution découlent, comme on peut s'y attendre, sur un succès. On remarque que la majeure partie du temps consacré à l'opération est dédiée à l'étape 11 consistant à attendre la fin du rafraîchissement de l'écran *e-paper*. C'est précisément là, et dans le fait que les commandes s'enchaînaient trop rapidement, que les tentatives précédentes échouaient lamentablement.

Nous pouvons procéder à quelques autres essais sur la même base, en jouant un peu avec le contenu de `imgbuf[]` avant l'appel à `sendimg()`. Avec quelque chose comme :

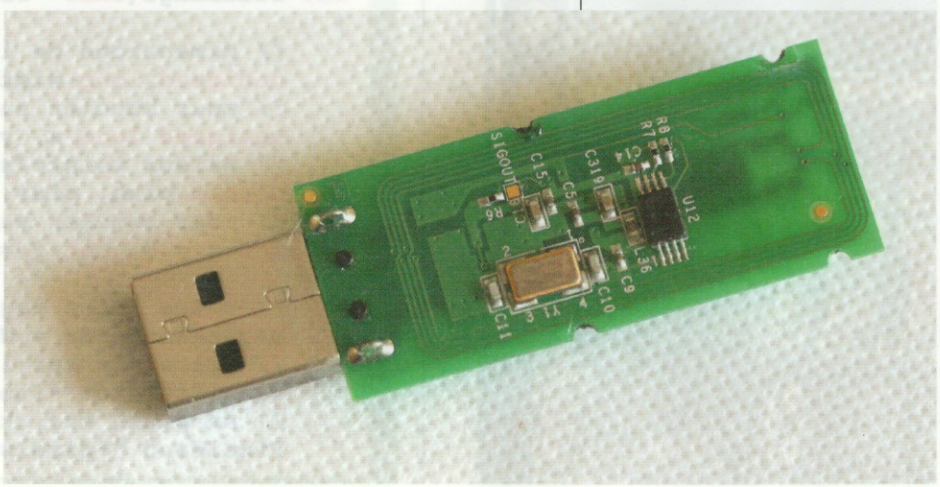
```
for(int i=0; i < BUFSIZE; i++) {
    imgbuf[i] = rand();
}
```

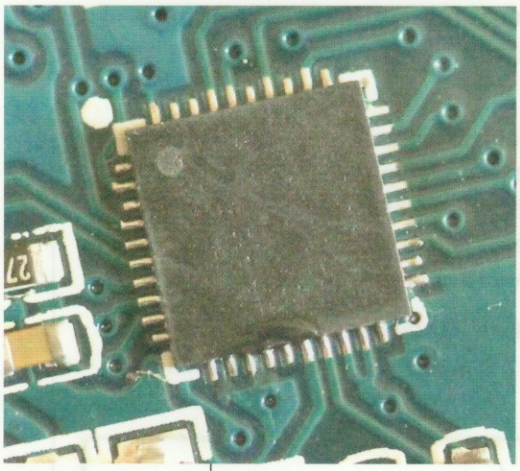
Et nous devrions avoir un résultat composé de pixels aléatoirement choisis. Si nous remplaçons `rand()` par une valeur arbitraire comme `0xf0` (11110000 en binaire), le résultat sera plus ordonné, avec une succession de lignes verticales noires de 4 pixels, espacées de 4 pixels blancs. On peut ainsi procéder à quelques tests rapides et simples, mais la vraie finalité nécessite une fonction supplémentaire...

*Les amateurs de développement et d'expérimentation NFC recommandent généralement l'utilisation du SCL3711 de SCM plutôt que celle de l'ACR122U, en raison d'un certain nombre de bugs d'implémentation matérielle. Dans le cas de l'écran WaveShare, c'est plutôt l'inverse qui s'applique. La petite antenne du SCL3711 semble avoir beaucoup de mal à maintenir une communication stable sans un placement ultra précis contre le boîtier.*

## 4. GÉRER UNE VÉRITABLE IMAGE

Afin de remplir notre `imgbuf[]` avec de réelles données provenant d'un fichier graphique, nous devons être en mesure d'en lire et d'en interpréter le contenu. Je vous disais précédemment que de défricher le terrain dans un environnement adapté facilitait grandement les choses et en voici le parfait exemple. Sous GNU/Linux ou tout autre système d'exploitation complet (comprendre « non léger » comme FreeRTOS ou RIOT), nous disposons d'un nombre important de solutions et de bibliothèques. Nous ne sommes donc pas tenus d'utiliser un format graphique simple à gérer comme BMP ou XPM, ni même d'utiliser une bibliothèque ne supportant qu'un unique format comme JPEG ou PNG.



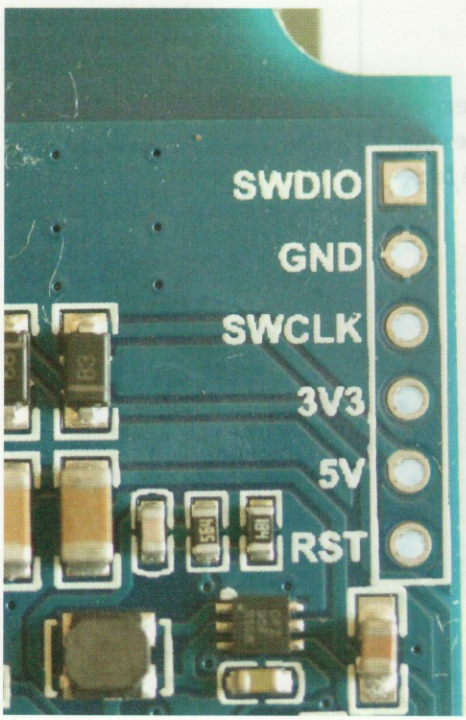


*Le circuit de l'écran ne comprend que deux puces. La plus massive d'entre elles est, sans l'ombre d'un doute, le microcontrôleur chargé du protocole de communication et de la gestion de l'écran. On peut voir que WaveShare ne semble pas vouloir que l'on sache de quel modèle il s'agit précisément, puisque la surface du composant a été poncée pour supprimer le marquage.*

Bien au contraire, nous allons utiliser une bibliothèque nous offrant non seulement la possibilité de lire de multiples formats et de connaître la valeur de chaque pixel, mais également de transformer l'image pour l'adapter aux limitations de l'écran (taille, couleurs, etc.). MagickWand, qui n'est rien de moins que l'API permettant d'utiliser les fonctionnalités d'ImageMagick depuis un code en C, est un choix idéal.

Le code qui va suivre est quelque peu simplifié par rapport à la version que vous trouverez en ligne, afin de gagner un peu de place dans cet article déjà long. La rotation de l'image, si besoin, ainsi que les tests d'usage sur les valeurs retournées par les fonctions MagickWand ne sont donc pas présents ici, mais le sont dans le code final. Bien d'autres manipulations d'images peuvent être ajoutées pour obtenir un rendu de qualité, centrer l'image, réduire le nombre de couleurs différemment, offrir davantage d'options à l'utilisateur, mais ceci sort totalement du cadre du présent article. Nous voulons juste rapidement lire une image.

*Dans un coin du circuit imprimé se trouvent 6 emplacements de connexion dont la sérigraphie laisse penser qu'il s'agit d'une interface de programmation SWD (Serial Wire Debug) permettant de programmer le microcontrôleur. SWD est une alternative matérielle au JTAG faisant circuler le même protocole. Ceci est typique d'un microcontrôleur à cœur ARM. Un STM32 peut-être...*




Notre fonction `readimage()` prendra en argument une chaîne de caractères constituant le nom de fichier à utiliser et remplira `imgbuf[]` avec les données adéquates. Celle-ci débute tout naturellement avec les déclarations qui s'imposent :

```
#include <wand/magick_wand.h>
[...]

int readimage(char *filename)
{
    // dimensions de l'image
    unsigned long width, height;
    // pour boucler sur les pixels
    unsigned long x, y;
    // valeur teinte/saturation/niveau (HSL)
    double pixh, pixs, pixl;
    // pour boucler sur les octets du buffer
    int i = 0; int j = 0;

    // notre image MagickWand
    MagickWand *mw = NULL;
    // itérateur sur les pixels
    PixelIterator *iterator = NULL;
    // un pixel MagickWand
    PixelWand **pixels = NULL;
```

On passe ensuite directement aux différentes étapes de traitement : initialisation, ouverture du fichier, adaptation des dimensions et réduction des couleurs :

```
// initialisation MagickWand
MagickWandGenesis();
// notre image
mw = NewMagickWand();

// ouverture du fichier
printf("MagickWand: load file\n");
if(MagickReadImage(mw,filename) == MagickFalse) {
    fprintf(stderr, "Error loading image file!\n");
    return(-1);
}

// récupération des dimensions (pour test)
width = MagickGetImageWidth(mw);
height = MagickGetImageHeight(mw);

// redimensionnement
printf("MagickWand: resize\n");
MagickResizeImage(mw, WIDTH, HEIGHT, LanczosFilter,1);

// dithering
printf("MagickWand: posterize\n");
MagickPosterizeImage(mw, 4, FloydSteinbergDitherMethod);

// réduction du nombre de couleurs
printf("MagickWand: set type to BW\n");
MagickSetImageType(mw, BilevelType);

// récupération des dimensions finales
width = MagickGetImageWidth(mw);
height = MagickGetImageHeight(mw);
```

La technique de *dithering* ou de diffusion d'erreurs en bon français permet d'améliorer le rendu des images, lors d'une réduction drastique du nombre de couleurs. En effet, notre écran ne peut afficher que des pixels noirs ou blancs et la notion de niveau de gris n'existe pas. Le *dithering* permet de simuler les nuances en éparpillant une quantité finie de pixels noirs sur un fond blanc, par exemple. Plus la densité de pixels noirs est importante, plus on obtient l'impression de se rapprocher d'une teinte 100 % noire. Nous partons ici du principe que l'image fournie n'a pas été préparée spécifiquement pour l'écran, mais le code final inclus un certain nombre de tests pour conditionner ces opérations.

Une fois le format d'image adapté, nous pouvons passer à son analyse et au remplissage du *buffer* en itérant sur l'état des pixels :

```
// Get a new pixel iterator
iterator=NewPixelIterator(mw);
for(y=0; y < height; y++) {
    // Get the next row of the image as an array of PixelWands
    pixels=PixelGetNextIteratorRow(iterator,&width);
    for(x=0; x < width; x++) {
        PixelGetHSL(pixels[x], &pixh, &pixs, &pixl);
        if(pixl > 0)
            imgbuf[j] |= (128 >> (i-(j*8))); // reversed bits order
        i++;
        if(i%8 == 0) j++;
    }
}
```

La double boucle est assez classique, nous traitons ligne par ligne l'image et pixel par pixel pour chaque ligne. L'ordre des bits dans l'octet est important, puisque les pixels sont représentés conformément à leur position à l'écran. Le premier pixel de l'image correspond au bit le plus à gauche du premier octet. C'est pour cette raison que nous décalons 128 (10000000 en binaire) en fonction de la position du pixel dans l'image. Ainsi le huitième pixel est le bit de poids le plus faible dans le premier octet, car nous décalons 7 fois 128 vers la droite.

Pour obtenir l'état d'un pixel, nous utilisons la fonction `PixelGetHSL()` nous retournant la teinte, la saturation et le niveau (*level*) de chaque pixel. Si le niveau est supérieur à 0, le pixel n'est pas totalement noir (et donc blanc), mais comme les bits sont inversés, nous changeons à 1 le bit correspondant dans l'octet. Inversement, si le niveau est bien 0, nous avons un pixel noir et le bit en question reste 0.

*Le papier électronique le plus courant ne sait afficher que deux couleurs, noir et blanc. Pour obtenir un rendu simulant des niveaux de gris, une technique utilisable est celle du dithering : on joue sur la densité de pixels noirs pour obtenir une nuance de gris plus ou moins prononcée. Ceci est une fonctionnalité mise à notre disposition par la bibliothèque MagickWand.*

Une fois cette étape passée, tout ce que nous avons à faire est de libérer la mémoire et de cesser proprement l'utilisation de MagickWand :

```
    if(mw) mw = DestroyMagickWand(mw);
    MagickWandTerminus();

    return(0);
}
```

Il ne nous reste plus alors qu'à utiliser cette nouvelle fonction en début de `main()` pour charger notre image. Dans la version finale/publique de ce code, `getopt()` est utilisé pour récupérer le nom de fichier en argument sur la ligne de commande. Nous pouvons alors, une fois le tout

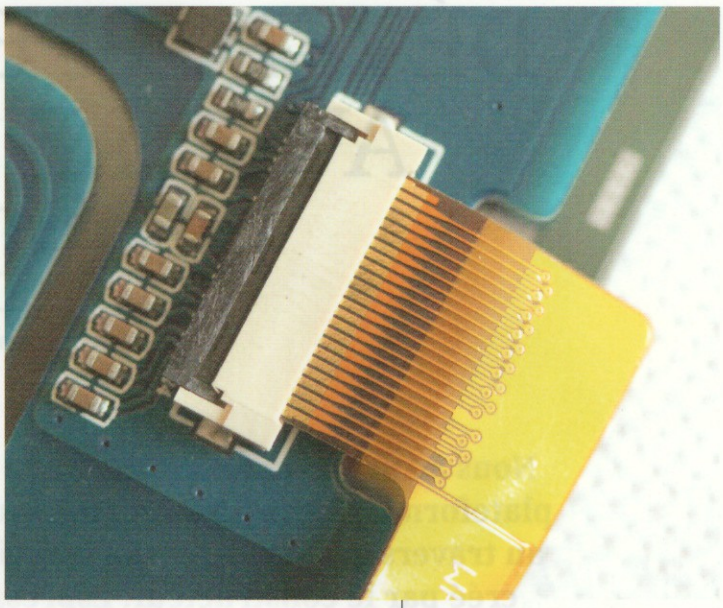
compilé, utiliser le programme en spécifiant n'importe quel type de fichier supporté par ImageMagick, à n'importe quelle taille et indépendamment du nombre de couleurs présentes.

## 5. LA MORALE DE L'HISTOIRE

Cette petite aventure sur fond d'exploration et de réimplémentation nous aura appris plusieurs choses. La documentation technique est l'élément clé de la popularité d'un nouveau produit, en particulier lorsqu'il est destiné aux amateurs de code et d'électronique. Il aurait été bien plus efficace pour WaveShare d'accompagner le matériel d'une page wiki détaillant, même vaguement, le protocole utilisé plutôt que de fournir une application Android clé en main. L'effort aurait été moindre et quelqu'un aurait forcément produit l'application en question, ainsi que toute une gamme d'outils ou de bibliothèques.

En l'absence de documentation, nous apprenons également qu'il y a plus d'une façon d'obtenir ces informations. La rétro-ingénierie est une approche possible, mais comme nous l'avons vu, son succès est fortement dépendant de l'expérience du développeur (c'est un métier) et laisse toujours des zones d'ombre concernant bon nombre de détails. Une autre solution est d'attendre et d'espérer qu'une implémentation différente, accompagnée de sources, se fasse jour, comme ici avec `ST25R3911B-NFC-Demo.zip`. On peut alors s'inspirer du code pour produire sa propre version expérimentale.

Enfin, et c'est peut-être le plus important, aucune tentative n'est réellement vaine, dans le sens où l'on gagne systématiquement en expérience. Via ce projet, nous avons constaté que décompiler une application Android peut être très simple, nous avons appris à communiquer avec un périphérique NFC, nous avons découvert MagickWand et nous avons jeté les bases d'un code aisément portable sur n'importe quelle plateforme.



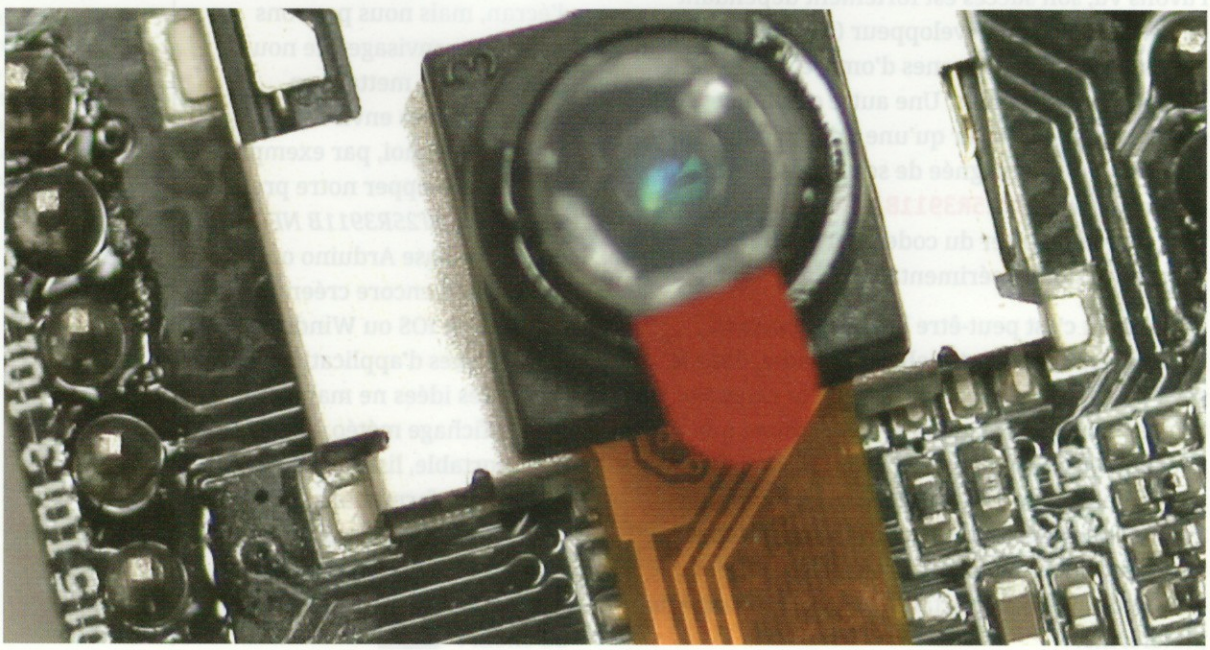
*L'écran e-paper est connecté au circuit imprimé via un connecteur flatflex. Après inspection, il s'avère que l'écran lui-même est disponible au détail et compatible avec les modules WaveShare ESP8266 que nous avions précédemment étudiés. Il existe également une déclinaison trois couleurs (avec jaune ou rouge en plus) de ce modèle et il devrait être possible de « mettre à jour » le produit dans ce sens... à condition de pouvoir développer un firmware adapté.*

À ce stade, plusieurs options s'offrent à nous. Nous pouvons continuer de développer notre outil (chose que je compte faire) pour ajouter des fonctionnalités et prendre en charge les autres modèles d'écran, mais nous pouvons également envisager de nouveaux projets mettant en œuvre d'autres environnements. Pourquoi, par exemple, ne pas développer notre propre version de *ST25R3911B NFC Board* sur base Arduino ou ESP8266 ou encore créer une application iOS ou Windows ? Et en termes d'applications pratiques, les idées ne manquent pas : affichage météo, mémo transportable, listes diverses, relèves d'informations, cadre photo électronique, affichage de QRcode, étiquette électronique... Il n'y a que l'embarras du choix ! **DB**

# DÉVELOPPEMENT ESP32 AVEC LE NOUVEAU ESP-IDF 4.0

Denis Bodor

**Nous avons précédemment traité du développement sur la fantastique plateforme ESP32, héritière du non moins délectable ESP8266, aussi bien au travers de l'IDE Arduino que via l'environnement de développement créé par le constructeur, Espressif Systems, répondant au doux nom de ESP-IDF. Le 11 février dernier était annoncée la version 4.0 de cet environnement, majoritairement compatible avec la précédente version 3.3.1, mais apportant un lot majeur d'améliorations et quelques changements très intéressants dans le système de construction/compilation. Il est donc temps de revisiter la bête et de tester tout cela...**



Beaucoup d'utilisateurs connaissent indirectement ESP-IDF, dans le sens où une version spécifique et préconfigurée forme la base du support ESP32 pour Arduino. Dans le numéro 24, nous avions vu qu'il était possible, grâce à l'ESP-IDF, de non seulement développer des croquis Arduino sans l'IDE dédié, mais également de disposer de cette possibilité sur une plateforme embarquée comme une Raspberry Pi. Pour ce faire, il suffit d'utiliser l'ESP-IDF, après avoir installé ou compilé bon nombre de dépendances, et d'adjoindre dans son projet un *composant* logiciel fournissant les bibliothèques standard Arduino, ainsi que l'architecture spécifique propre aux sources en langage dit « Arduino » (fondamentalement, du C++). C'est cet ensemble basé sur l'ESP-IDF, en version légèrement révisée et structurée, qui est utilisé en guise de support installable via le gestionnaire de carte de l'IDE Arduino.

Ici, il ne sera pas question d'installer un support Arduino. Non seulement à cette date il n'y a pas de version intégrant l'ESP-IDF V4.0 (la dernière version est 1.0.4 utilisant l'IDF 3.2), mais ceci ne suffirait pas à supporter un article entier. Il suffit, en effet, d'une ligne dans les préférences de l'IDE et de quelques clics de souris pour ajouter le nécessaire pour supporter une carte ESP32. Non, ici nous parlons de l'utilisation directe de l'IDF, ou *Integrated Development Framework*, développé par Espressif. Contrairement à ce qui a été expliqué dans le numéro 24, nous n'allons pas non plus construire ce qui est directement utilisable pour les systèmes d'exploitation et plateformes courantes (GNU/Linux x86, ARM et Windows). La chaîne de compilation *xtensa-esp32-elf* n'était pas disponible pour RPi, mais c'est à présent le cas, aussi bien pour les architectures ARM 32 bits (alias armel ou armhf) et 64 bits. Je n'ai cependant pas testé la chose, la RPi n'étant une solution viable que si l'on se trouve dans l'incapacité d'installer l'environnement ESP-IDF sur une machine plus puissante (typiquement, un PC).

Cette nouvelle version de l'ESP-IDF intègre un processus d'installation complet, aussi bien pour GNU/Linux que pour Windows. Il n'est donc pas nécessaire de procéder à des manipulations préalables (autres que les invariables dépendances sous GNU/Linux).

## 1. INSTALLATION D'ESP-IDF SOUS GNU/LINUX

Le processus s'est grandement simplifié pour ce système, mais il est toujours nécessaire de disposer d'éléments logiciels complémentaires. Ceci aussi bien pour le processus d'installation lui-même que pour l'utilisation de l'environnement, une fois en place. Ne vous y trompez pas, ceci est une bonne chose, car il n'y a pas d'intérêt à embarquer des outils ou des bibliothèques dans un environnement, si celles-ci sont déjà disponibles dans le système. Là où nous devons installer des dépendances sous GNU/Linux, nous devons tout installer sous Windows. Différents systèmes, différents utilisateurs, différentes caractéristiques et différente philosophie...

Nous commençons donc par installer tous les prérequis, avec ici un système Debian GNU/Linux (ou Ubuntu) :

```
$ sudo apt-get update
$ sudo apt-get install git wget libncurses-dev flex\
bison gperf python python-pip python-setuptools\
python-serial python-click python-cryptography\
python-future python-pyparsing python-pyelftools\
cmake ninja-build ccache libffi-dev libssl-dev
```

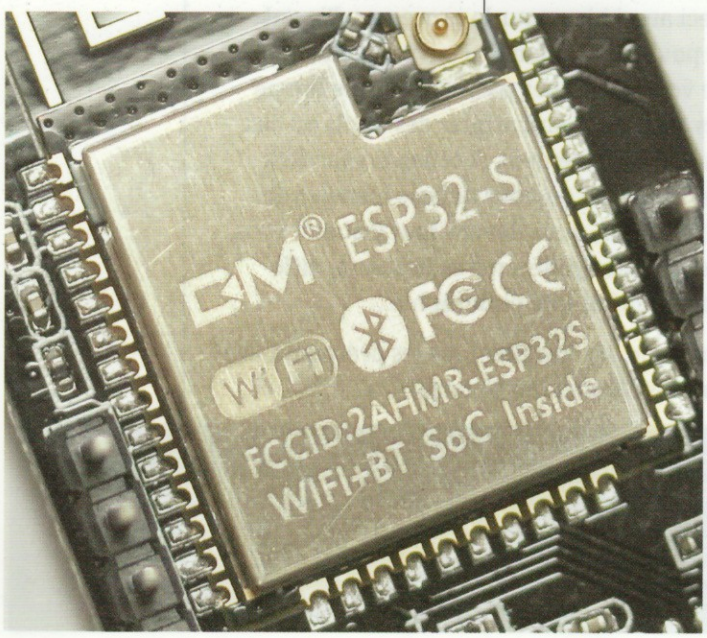
Des équivalences pour CentOS et Arch Linux existent et sont spécifiées dans la documentation officielle. Je ne les détaille pas ici, partant du principe que, si vous êtes utilisateur Raspberry Pi et GNU/Linux sur PC, la logique suppose une correspondante Raspbian/Debian toute naturelle.

Ceci fait, nous pouvons entrer dans le vif du sujet en récupérant l'IDF lui-même directement depuis le dépôt GitHub officiel, mais en prenant soin d'utiliser la bonne version :

```
$ cd
$ mkdir ESP32
$ cd ESP32
$ git clone -b v4.0 --recursive https://github.com/espressif/esp-idf.git
```

Les sources sur GitHub sont celles de développement, mais la toute nouvelle version stable est étiquetée « v4.0 ». C'est l'objet même de l'option `-b` présente dans cette commande. L'option `--recursive` nous permet de récupérer les sous-modules nécessaires à l'utilisation de l'ensemble des fonctionnalités de la plateforme. En effet, l'IDF est un environnement complet intégrant non seulement le strict minimum pour compiler et construire un code à destination de l'ESP32, mais également les bibliothèques supportant des fonctionnalités matérielles et logicielles embarquées (le Bluetooth, les outils pour manipuler la flash, le support MQTT, la gestion du format JSON, le support Wi-Fi, etc.).

*L'ESP32 est sans doute la plateforme microcontrôleur la plus intéressante du moment : deux cœurs, 520 Ko de SRAM, flash externe jusqu'à 16 Mo, 16 sorties PWM, 4 SPI, ADC 18 canaux 12 bits, DAC 2 canaux, MAC Ethernet, Wi-Fi, Bluetooth v4.2 BR/EDR et BLE, cryptographie matérielle, interface SD/eMMC/SDIO... Avec l'arrivée récente de la version 4.0 du framework de développement, l'ensemble atteint un niveau de maturité tout à fait remarquable.*




À ce stade l'installation est, pour ainsi dire, une coquille vide. Nous avons les éléments qui constituent l'environnement, mais nous n'avons rien pour compiler le code, et encore moins pour en faire une image à placer dans la flash et pour accéder au matériel. Compléter l'installation revient à simplement lancer un script shell fourni :

```
$ cd esp-idf
$ bash install.sh
```

Ce script va :

- télécharger les binaires pour le système en cours d'utilisation (le compilateur `xtensa-esp32-elf`, les outils pour binaires `esp32ulp-elf` et l'outil de débogage `openocd-esp32`) ;
- extraire ces trois éléments dans le répertoire `.espressif` de votre répertoire personnel (`$HOME`) ;
- créer un environnement Python pour les outils ;
- installer les modules Python nécessaires dans ce même environnement ;

• et vous affichez un message vous invitant à modifier vos variables d'environnement :

```
[...]
All done! You can now run:

  . ./export.sh
```

Notez que l'installation de l'ensemble se fera automatiquement dans **~/.espressif**, mais qu'il est possible de préciser un répertoire de votre choix via la variable d'environnement **IDF_TOOLS_PATH**. Si tel est le cas, cette variable devra contenir le même chemin avant toute utilisation de **export.sh**.

Le contenu du répertoire **~/.espressif** contient tous les éléments :

• **dist/** : les fichiers téléchargés ;

• **tools/** : les outils pour les binaires (compilateur, *binutils*, etc.) ;

• **python_env/** : l'environnement Python.

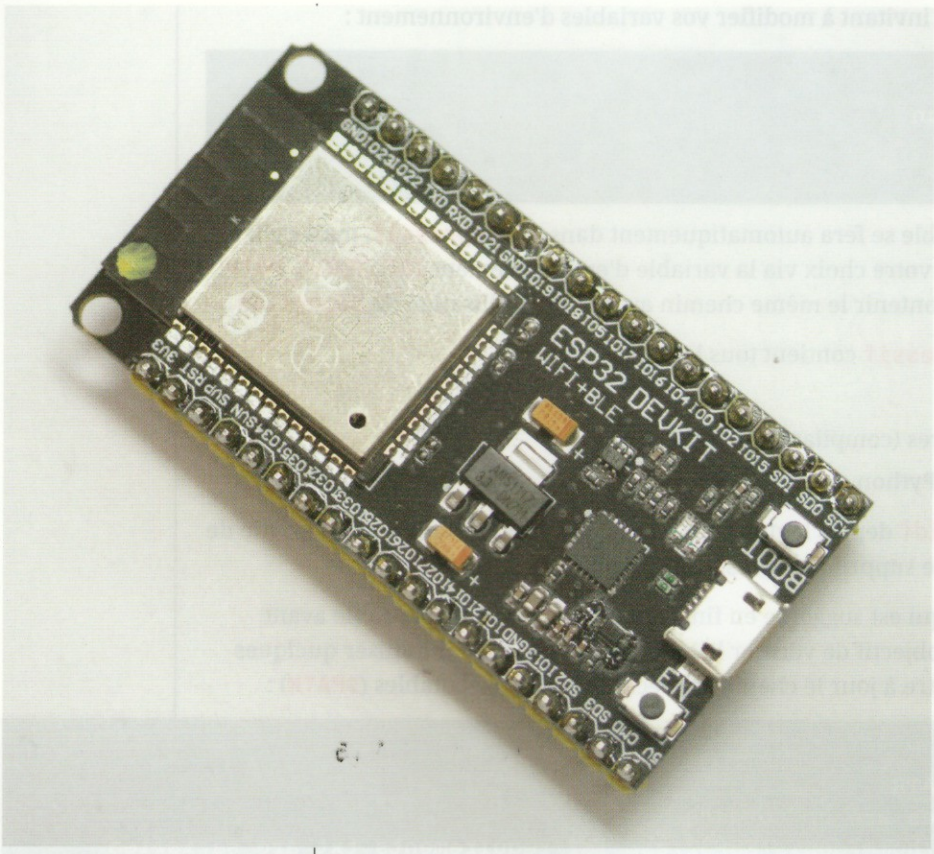
Attention, le sous-répertoire **esp-idf** depuis lequel vous avez procédé à l'installation fait partie de l'environnement, vous ne devez pas le supprimer après avoir invoqué **install.sh**.

La dernière ligne de commandes qui est suggérée en fin d'installation est indispensable avant toute utilisation de l'IDF. Elle a pour objectif de vérifier l'état de l'installation, d'initialiser quelques variables d'environnement et de mettre à jour le chemin de recherche des exécutables (**$PATH**) :

```
$ . export.sh
Adding ESP-IDF tools to PATH...
Checking if Python packages are up to date...
Python requirements from /home/denis/ESP32/esp-idf/requirements.txt are satisfied.
Added the following directories to PATH:
  /home/denis/ESP32/esp-idf/components/esptool_py/esptool
  /home/denis/ESP32/esp-idf/components/espcoredump
  /home/denis/ESP32/esp-idf/components/partition_table/
  /home/denis/.espressif/tools/xtensa-esp32-elf/esp-2019r2-8.2.0/xtensa-esp32-elf/bin
  /home/denis/.espressif/tools/esp32ulp-elf/2.28.51.20170517/esp32ulp-elf-binutils/bin
  /home/denis/.espressif/tools/openocd-esp32/v0.10.0-esp32-20190313/openocd-esp32/bin
  /home/denis/.espressif/python_env/idf4.0_py2.7_env/bin
  /home/denis/ESP32/esp-idf/tools
Done! You can now compile ESP-IDF projects.
Go to the project directory and run:

  idf.py build
```

Parmi les variables d'environnement utilisées, on retrouve naturellement les chemins vers les différents éléments de l'IDF :

```
$ env | grep IDF
IDF_PYTHON_ENV_PATH=/home/denis/.espressif/python_env/idf4.0_py2.7_env
IDF_PATH=/home/denis/ESP32/esp-idf
IDF_TOOLS_EXPORT_CMD=/home/denis/ESP32/esp-idf/export.sh
IDF_TOOLS_INSTALL_CMD=/home/denis/ESP32/esp-idf/install.sh
```




*La plupart des cartes et modules à base d'ESP32 se présentent sous cette forme, à l'instar des environnements/ plateformes Arduino, Ti ou STM32. Il existe des cartes réputées et officielles, mais également des modèles bien plus économiques aux qualités variables, dans une plage de tarifs allant de 3 € à 16 €.*

C'est généralement une bonne idée de déclencher l'évaluation de `export.sh` directement depuis l'un des fichiers d'évaluation de votre shell (`~/.bashrc`, `~/.bash_profile`, etc.). Ceci se fera comme précédemment dans un shell interactif (ou éventuellement avec `source`), mais en spécifiant, bien entendu, le chemin complet vers le fichier).

À ce stade, l'environnement est prêt à être utilisé pour vos projets.

## 2. INSTALLATION SOUS WINDOWS

L'installation de l'ESP-IDF sous Windows est un processus relativement aisé et étrangement plus simple que sous GNU/Linux. Il vous suffira en effet de télécharger l'outil d'installation, le « *ESP-IDF Tools Installer* », depuis le site d'Espressif à l'adresse **https://dl.espressif.com/dl/esp-idf-tools-setup-2.2.exe**. Une fois les quelque 400 Mo du fichier téléchargés, une simple exécution fera l'affaire.

Le processus d'installation :

- installe Python 3.7 (c'est étrange, car la documentation en ligne précise que Python 2.7 est préférable) ;
- installe Git 2.21.0 ;
- télécharge ESP-IDF ;
- vous permet de choisir la version de l'ESP-IDF : « v4.0 (release version) » ;
- vous demande de préciser le répertoire d'installation : `C:\Users\Denis\.espressif` par défaut ;
- vous permet d'automatiser la création d'une entrée du menu ***Démarrer***, le raccourci sur le Bureau et l'ajout des outils ESP-IDF parmi les programmes non couverts par *Windows Defender* (ceci accélère grandement la compilation des projets).

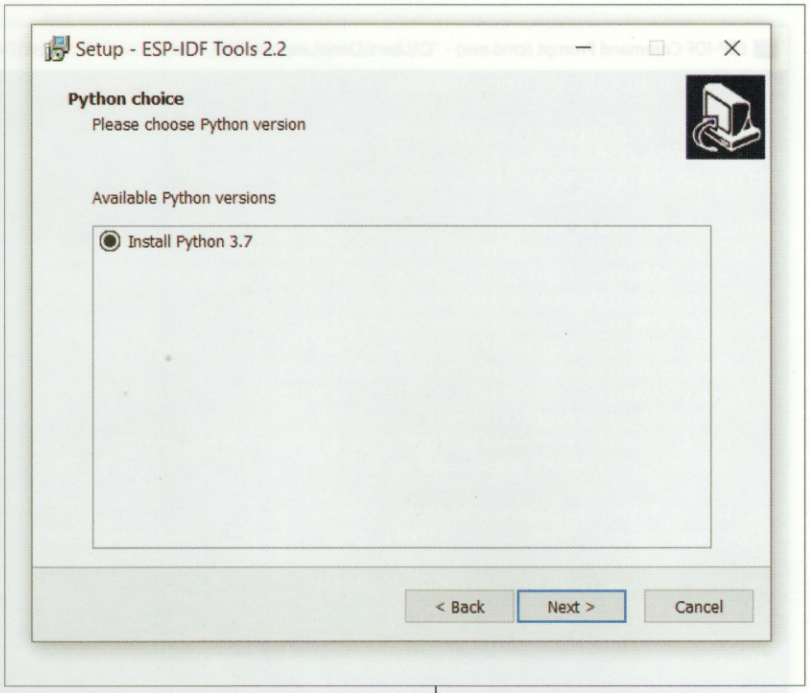
Tout est fait à votre place dans une succession d'étapes appliquées individuellement. Après téléchargement des éléments, le processus procédera à :

- l'installation de Git pour Windows ;
- la décompression de l'archive ESP-IDF ;

- l'installation du langage Python avec un environnement virtuel et ses modules ;
- l'exécution d'un script PowerShell ;
- la création des raccourcis ;
- et enfin, l'ouverture d'un terminal avec les éléments ESP-IDF correctement initialisés.

Pourquoi préciser ces étapes ? Tout simplement, car il est possible que vous rencontriez un problème comme ça a été le cas pour moi. Ici, une erreur est survenue lors de la création d'un environnement Python avec Virtualenv, outil destiné à créer une copie isolée du langage (afin de ne pas perturber une installation précédente). Le Virtualenv utilisé et installé automatiquement avec le langage semble être plus récent que ne le suppose le script `tools/idf_tools.py`. En effet, ligne 1183, le sous-processus `virtualenv` est lancé avec une option `--no-site-packages` qui n'existe pas/plus, puisque celle-ci est devenue obsolète du fait qu'elle demande un comportement qui est désormais celui adopté par défaut.

Si vous butez sur ce problème et que l'installation échoue, la façon la plus simple de le corriger consiste à modifier le fichier `tools/idf_tools.py` du répertoire `esp-idf` créé par défaut sur le Bureau. Prenez un éditeur de texte (le Bloc-Notes fera l'affaire) et supprimez la chaîne `'--no-site-packages'` en ligne 1183 invoquant `virtualenv`. Enregistrez alors le fichier et relancez l'installateur. Comme vous le constaterez, celui-ci se rendra compte, par exemple, de la présence de Python et de Git qu'il ne vous sera pas alors nécessaire de réinstaller. À une étape suivante, vous pourrez spécifier le répertoire `esp-idf` comme un ESP-IDF déjà installé (*Use an existing ESP-IDF directory*), plutôt que de procéder à un nouveau téléchargement (*Download ESP-IDF*). Grâce à la modification de la ligne 1183, l'installation pourra alors arriver à son terme sans problème. Notez que ce problème sera potentiellement corrigé au moment où vous lirez cet article, un ticket étant ouvert à ce propos sur GitHub.

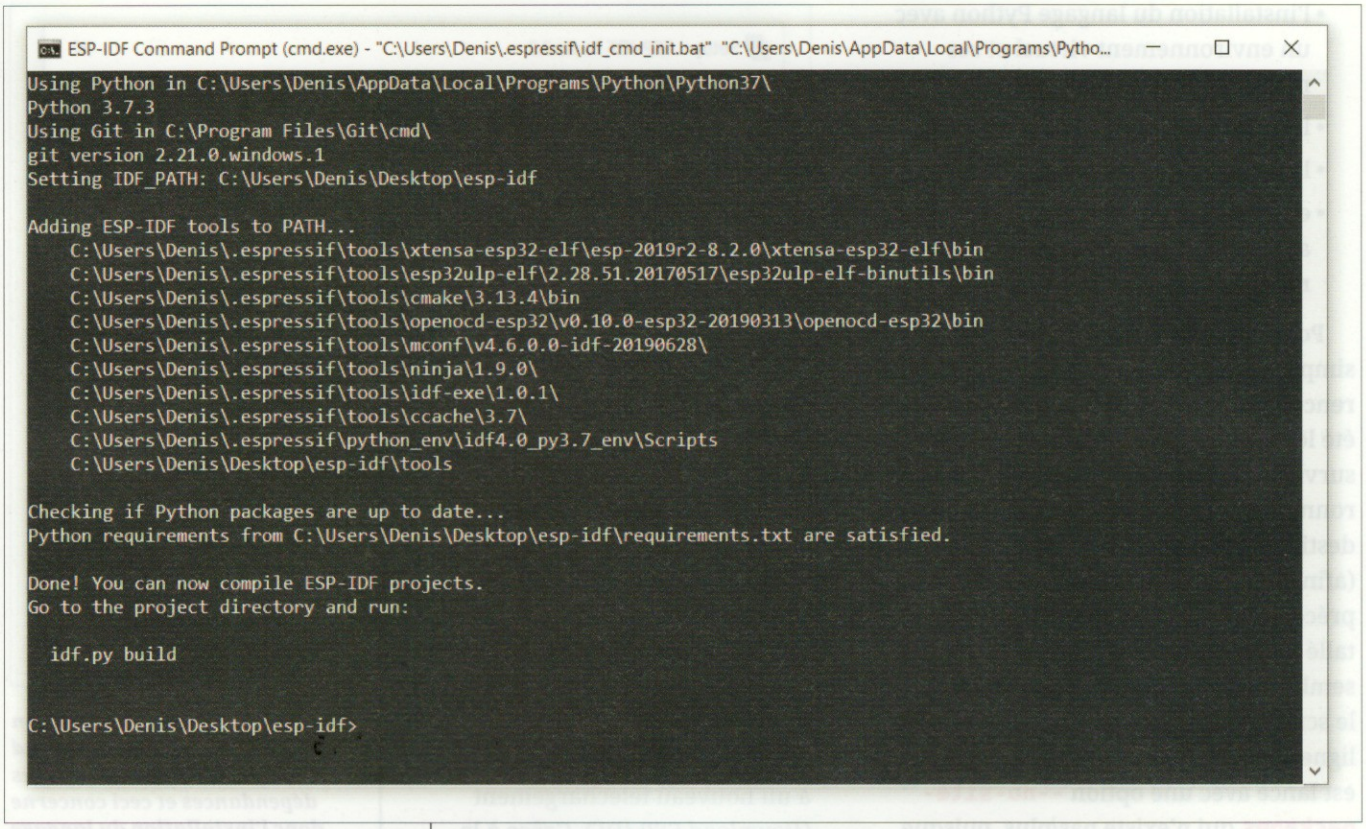



*La procédure d'installation pour Windows 10 comprend l'installation de toutes les dépendances et ceci concerne donc l'installation du langage Python. Seul problème, c'est Python 3.7 qui sera installé automatiquement, alors que la version 2.7 est recommandée dans la documentation (à jour) de l'ESP-IDF. Ceci peut être une source de problèmes...*

Votre environnement est maintenant prêt à être utilisé.

## 3. STRUCTURE DES PROJETS ET CHANGEMENTS

Avant toute chose, il est important de préciser qu'à cette date, le composant *Arduino* pour l'ESP-IDF n'est pas disponible. En effet, celui-




*Après installation, ici relativement hasardeuse, on dispose dans Windows du même environnement de développement que sur les autres systèmes d'exploitation supportés. L'utilisation de WSL peut également être une option intéressante et fonctionnelle, puisque l'ESP32 se programme via un convertisseur USB/série qui est un des rares matériels USB actuellement supportés dans cet environnement. Ceci n'est cependant pas couvert par la documentation Espressif.*

ci, disponible sur **https://github.com/espressif/arduino-esp32** ne supporte que l'IDF en version 3.2 et un gros travail reste à faire par les développeurs pour basculer sur la version 4.0. Rappelons que ce composant permet de développer des croquis Arduino en fournissant les bibliothèques de base et un système de compilation/construction adapté, et ce, aussi bien pour une intégration dans l'IDE Arduino que pour d'autres solutions comme PlatformIO, ou encore le développement de croquis via l'ESP-IDF directement (comme nous l'avons vu dans le numéro 24).

Néanmoins, lorsqu'on parle de développer pour l'ESP32, les deux approches les plus populaires sont soit l'utilisation dans l'IDE Arduino avec le « langage » Arduino soit un développement « natif », avec l'ESP-IDF, en se passant des « facilités » Arduino. La voie mitoyenne à ses avantages (cf. Hackable 24), mais honnêtement, quitte à utiliser l'IDF, autant le faire sans roulettes et à la mode locale.

L'une des grandes nouveautés de la version 4.0 est le passage de GNU Make au duo CMake/Ninja en guise de système de construction. GNU Make est toujours utilisable pour assurer une compatibilité, mais les nouveaux projets doivent reposer sur ce nouveau système. CMake est un système de configuration et de construc-

tion logicielle multiplateforme. Sa tâche est de déterminer et de configurer les dépendances nécessaires à la construction d'un projet et de définir l'ordre de compilation des éléments. Bien qu'étant déjà vastement plébiscité, CMake gagne chaque jour en popularité, car il est très efficace et simple à utiliser, en particulier en comparaison avec de simples `Makefile` écrits manuellement ou d'autres systèmes, comme les *Autotools*.

CMake configure la construction, mais ne procède pas à la compilation ou l'édition de liens. Il génère par défaut des fichiers pour GNU Make, qui se charge de ce travail. Cependant, comme d'autres projets en ce moment, Espressif a décidé de s'écarter de GNU Make, au bénéfice de Ninja. Cette alternative, issue originellement (2010) du développement du navigateur Chrome, met l'accent sur la rapidité de compilation et l'efficacité de la planification (dépendance entre sources). Dès 2011, Ninja est supporté par CMake et ce duo est adopté par de nombreux projets open source, au détriment des classiques `autoconf`/`automake`/`make`.

Une autre caractéristique intéressante de Ninja, en dehors de son nom bien sympathique, tient dans le fait que la syntaxe de ses fichiers de configuration n'est pas initialement destinée aux humains. Ces fichiers ne sont donc pas faits pour être composés manuellement, mais plutôt générés par un outil, comme CMake. Ceci peut sembler frustrant pour un développeur, mais c'est le prix à payer pour un gain de performance notable.

En plus du passage à CMake/Ninja, l'ESP-IDF v4.0 fait évoluer un outil de plus haut niveau, `idf.py` permettant une gestion facilitée des projets. Cet outil se charge alors d'utiliser à votre place CMake, Make/Ninja et `esptool.py` permettant de flasher votre ESP32. Il reste cependant possible d'utiliser ces éléments individuellement et de vous passer totalement de `idf.py`.

Le développement d'un code pour l'ESP32 (ou l'ESP32S2) repose sur la notion de projet. Un

projet est un répertoire dans lequel se trouvent vos codes sources et différents fichiers nécessaires à la construction. L'ESP-IDF ne fait pas partie du projet, pas plus que la chaîne de compilation. L'emplacement du projet/répertoire n'est donc pas dépendant de l'endroit où se trouvent ces éléments. Un projet est constitué d'un ou plusieurs composants et votre code source est l'un d'entre eux. C'est le composant **main** (par défaut) et également le nom du sous-répertoire où placer vos fichiers sources.

Un projet minimal se compose ainsi :

```
monProjet/
    CMakeLists.txt
    sdkconfig
    main/
        CMakeLists.txt
        premier.c
    build/
```

**sdkconfig** est le fichier contenant la configuration du projet contenant, entre autres choses, les fonctionnalités à activer pour faire fonctionner votre code. Le principe de fonctionnement est le même que pour la compilation d'un noyau Linux, avec une interface permettant le choix des fonctionnalités et éléments de configuration générant un fichier. Ce fichier n'existe pas dans le répertoire d'un projet fraîchement créé et sera produit par le script **idf.py**.

**build/** n'existe pas non plus dans un premier temps. Ce répertoire est destiné à accueillir l'ensemble des fichiers produits par le processus de construction et de compilation. Là encore, c'est **idf.py** qui se chargera de le créer ou, éventuellement vous si vous décider d'utiliser directement CMake, Ninja et **esptool.py**.

Le fichier source **premier.c** est ici, tout simplement, notre code très basique à exécuter sur l'ESP32 :

```
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "esp_system.h"
#include "esp_spi_flash.h"

void app_main()
{
    printf("Coucou Monde !\n");

    for (int i = 10; i >= 0; i--) {
        printf("Reboot dans %d seconde(s)...\n", i);
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
    printf("REBOOT !\n");
    fflush(stdout);

    esp_restart();
}
```

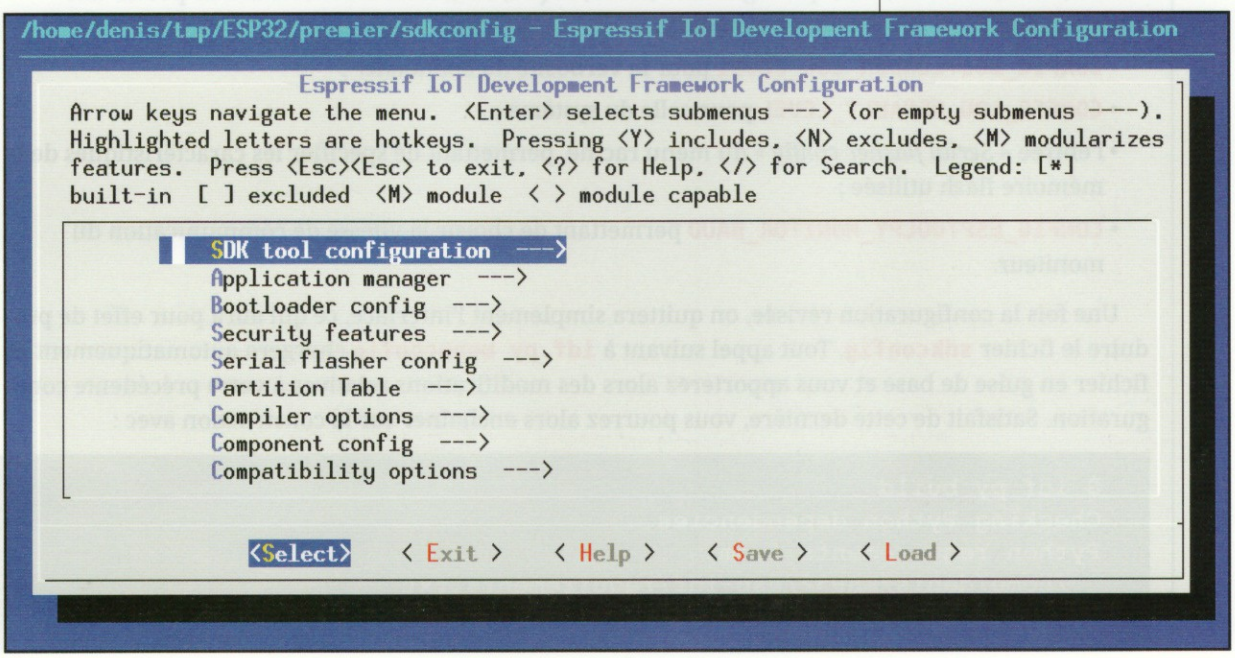
Celui-ci se contente d'afficher un message sur l'interface série, puis décompte 10 secondes avant de provoquer un redémarrage de la plateforme. Ces lignes de C ne sont ici pas importantes. Ce qui l'est, en revanche, est le fichier **CMakeLists.txt** présent en compagnie des sources. Il est destiné à CMake et contient :

```
idf_component_register(SRCS "premier.c"
                       INCLUDE_DIRS "")
```

Ce fichier est le *CMakeLists* du composant **main** et il vous en faudra un pour chaque composant que vous ajouterez à votre projet (comme par exemple, le futur support Arduino pour ESP-IDF en version 4.0). Notez que ceci ne concerne pas les composants nativement inclus dans l'ESP-IDF, mais uniquement ceux que vous ajoutez manuellement ou que vous aurez créé vous-même. L'objectif de ce fichier **CMakeLists.txt** est d'enregistrer le composant dans le système de construction, comme le suggère la directive **idf_component_register**.

Sa syntaxe est relativement claire dans sa plus simple expression. On précise avec **SRCS** le ou les fichiers sources du composant et, via **INCLUDE_DIRS**, une liste de répertoires où rechercher des fichiers *headers* si ce composant est utilisé par d'autres (ici aucun, puisque nous n'avons qu'un source et qu'un composant). C'est généralement une bonne idée de définir **INCLUDE_DIRS** à **"."** dans l'éventualité où le projet évolue par la suite.

*La configuration de vos projets passe par l'utilisation d'une interface qui n'est pas sans rappeler celle de la configuration des sources d'un noyau Linux ou du système de construction Buildroot. Celle-ci vous permet d'activer ou de désactiver des fonctionnalités spécifiques et de configurer avec précision différentes caractéristiques du futur système reposant sur FreeRTOS.*



Enfin, l'autre **CMakeLists.txt**, placé à la racine du répertoire, est le *CMakeLists* du projet. Dans sa version la plus simple, comme ici, il se compose de trois lignes indispensables :

```
# La version minimum de CMake pour construire le projet
cmake_minimum_required(VERSION 3.5)
# inclus la configuration CMake provenant de l'ESP-IDF
include($ENV{IDF_PATH}/tools/cmake/project.cmake)
# Le nom de projet utilisé pour les fichiers .elf et .bin
project(premier)
```

Notez que les lignes débutant par **#** sont de simples commentaires, qu'il est généralement conseillé de placer lorsqu'on débute avec l'ESP-IDF.

Il n'en faut pas davantage pour enchaîner sur la configuration avec la commande **idf.py menuconfig**. L'interface en mode texte qui se présente à vous permet de naviguer dans une série d'options avec les flèches du clavier (haut/bas pour les entrées et gauche/droite pour les boutons du bas), la touche Entrée et la touche Échap pour revenir en arrière. Les options sont activables/désactivables avec la barre d'espace ou les touches « Y » (oui) et « N » (non).

Une aide contextuelle sommaire est disponible via la touche « ? » et il est également possible de rechercher un élément de configuration spécifique en utilisant « / ». Ces deux fonctions sont complémentaires dans le sens où l'aide vous affiche également la directive qui sera utilisée dans le fichier **sdkconfig** généré (**CONFIG_BOOTLOADER_LOG_LEVEL**, **CONFIG_FLASHMODE_DIO**, **CONFIG_ESP32_SPIRAM_SUPPORT**, etc.) et qui peut être facilement recherchée avec « / », en cas d'erreur ou d'avertissement dans la phase de compilation. De plus, lors d'une recherche, les résultats se présentent préfixés d'un nombre. Il vous suffit de le saisir pour sauter directement à l'entrée correspondante.

La configuration par défaut active la quasi-totalité des fonctionnalités de l'ESP32 aussi bien matérielles (Wi-Fi, Ethernet, etc.) que logicielles (HTTP, MQTT, etc.), mais il conviendra de porter une attention particulière à quelques éléments :

- **CONFIG_BOOTLOADER_LOG_LEVEL** pour la verbosité du *bootloader* ;
- **CONFIG_LOG_DEFAULT_LEVEL** pour celle du système ;
- l'entrée « *Serial flasher config* » du menu racine, permettant de spécifier les caractéristiques de la mémoire flash utilisée ;
- **CONFIG_ESPTOOLPY_MONITOR_BAUD** permettant de choisir la vitesse de communication du moniteur.

Une fois la configuration révisée, on quittera simplement l'interface, ce qui aura pour effet de produire le fichier **sdkconfig**. Tout appel suivant à **idf.py menuconfig** chargera automatiquement ce fichier en guise de base et vous apporterez alors des modifications relatives à votre précédente configuration. Satisfait de cette dernière, vous pourrez alors enchaîner sur la construction avec :

```
$ idf.py build
Checking Python dependencies...
Python requirements from
  /home/denis/ESP32/esp-idf/requirements.txt
  are satisfied.
```

```
Executing action: all (aliases: build)
[...]
esptool.py v2.8
Generated /home/denis/tmp/ESP32/premier/build/premier.bin
Project build complete. To flash, run this command:
[...]
```

Vous obtiendrez alors, sauf erreur de compilation, non seulement un fichier **build/premier.bin** dont le nom correspond à celui du projet spécifié dans le **CMakeLists.txt** à la racine, mais également le *bootloader* **build/bootloader/bootloader.bin** ainsi que la description des partitions de la mémoire flash **build/partition_table/partition-table.bin**.

Ces fichiers seront utilisés pour inscrire la mémoire flash en se pliant d'une nouvelle utilisation de **idf.py** :

```
$ idf.py -p /dev/ttyUSB0 flash
Checking Python dependencies...
Python requirements from /home/denis/ESP32/esp-idf/requirements.txt are satisfied.
Adding flash's dependency "all" to list of actions
Executing action: all (aliases: build)
Running ninja in directory /home/denis/tmp/ESP32/premier/build
Executing "ninja all"...
[1/3] Performing build step for 'bootloader'
ninja: no work to do.
Executing action: flash
[...]
esptool.py v2.8
Serial port /dev/ttyUSB0
Connecting.....
Detecting chip type... ESP32
Chip is ESP32D0WDQ6 (revision 1)
Features: WiFi, BT, Dual Core, Coding Scheme None
Crystal is 40MHz
MAC: 30:ae:a4:1e:99:88
Uploading stub...
Running stub...
Stub running...
Changing baud rate to 460800
[...]
Hash of data verified.

Leaving...
Hard resetting via RTS pin...
Done
```

Nous sommes ici avec un système GNU/Linux et l'argument passé avec l'option **-p** permettant de spécifier le port série à utiliser est adapté en conséquence. Avec un système Windows, cette dénomination sera par exemple **COM1** et avec macOS, ce serait **/dev/cu**. Une fois le *firmware* chargé, l'ESP32

redémarre automatiquement et nous pouvons connecter un moniteur série pour voir les messages envoyés par notre petit code. Il n'est cependant pas nécessaire de faire appel à un outil tiers, puisque le script `idf.py` propose une telle fonctionnalité :

```
$ idf.py -p /dev/ttyUSB0 monitor
[...]
I (71) boot: Chip Revision: 1
I (71) boot_comm: chip revision: 1, min. bootloader chip revision: 0
I (41) boot: ESP-IDF v4.0 2nd stage bootloader
I (41) boot: compile time 17:10:50
I (41) boot: Enabling RNG early entropy source...
I (45) boot: SPI Speed      : 40MHz
I (49) boot: SPI Mode       : DIO
[...]
I (189) cpu_start: Pro cpu up.
I (192) cpu_start: Application information:
I (197) cpu_start: Project name:     premier
I (202) cpu_start: App version:      1
I (206) cpu_start: Compile time:     Feb 13 2020 07:10:43
I (212) cpu_start: ELF file SHA256:  ee391a21f8d83c03...
I (218) cpu_start: ESP-IDF:          v4.0
I (223) cpu_start: Starting app cpu, entry point is 0x40080fdc
[...]
Coucou Monde !
Reboot dans 10 seconde(s)...
Reboot dans 9 seconde(s)...
Reboot dans 8 seconde(s)...
Reboot dans 7 seconde(s)...
Reboot dans 6 seconde(s)...
Reboot dans 5 seconde(s)...
Reboot dans 4 seconde(s)...
Reboot dans 3 seconde(s)...
Reboot dans 2 seconde(s)...
Reboot dans 1 seconde(s)...
Reboot dans 0 seconde(s)...
REBOOT !
ets Jun  8 2016 00:22:57
rst:0xc (SW_CPU_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
[...]
```

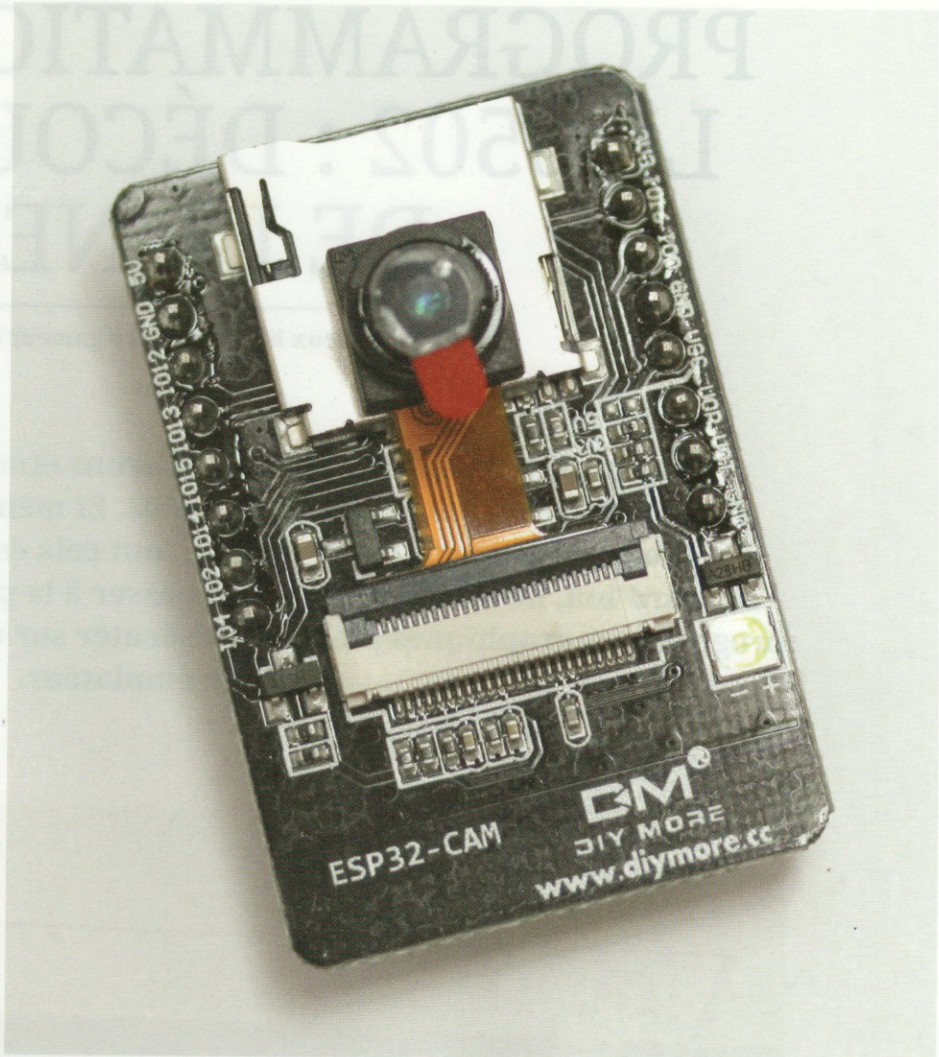
Pour quitter ce moniteur série, vous devrez utiliser les raccourcis Ctrl+T puis Ctrl+X. Chose très intéressante, ce moniteur permet également de déclencher la construction et l'inscription en flash. Il ne vous est donc pas nécessaire de multiplier les terminaux à l'infini ou de quitter le moniteur pour mettre à jour votre code. La séquence Ctrl+T puis Ctrl+F construira et flashera le projet dans son ensemble (*bootloader* inclus), et Ctrl+T puis Ctrl+A (ou simplement « A ») construira et flashera l'application seule (votre code). Notez qu'il est également possible de spécifier plusieurs « ordres » à `idf.py`, comme par exemple `idf.py -p /dev/ttyUSB0 clean build flash monitor` pour effacer les

fichiers des constructions, reconstruire le tout, flasher dans l'ESP32 et lancer le moniteur. Bien sûr, `build` est ici purement démonstratif, car inutile, puisque `flash` provoque une reconstruction si elle est nécessaire (changement des sources, par exemple).

## 4. POUR FINIR

Le but de cet article n'était pas de faire un tour d'horizon de toutes les améliorations et évolutions associées avec cette mise à jour majeure de l'ESP-IDF, mais de vous permettre de prendre en main les nouvelles spécificités de l'environnement. Au-delà de ce que nous venons de voir, une grande majorité des nouveautés concernent le *framework* lui-même et vous trouverez une liste des changements directement sur la page GitHub annonçant la disponibilité de cette version : **https://github.com/espressif/esp-idf/releases/tag/v4.0**.

Je vous recommande, en attendant la mise à disposition d'un nouveau support pour l'IDE Arduino, de jeter un œil à l'ESP-IDF, car ce qu'il est possible de faire avec Arduino ne représente qu'une version très édulcorée du réel potentiel d'un ESP32. Peut-être, en vous familiarisant avec l'ESP-IDF et les nombreux exemples qu'il contient, en viendrez-vous à ne plus envisager une utilisation de l'IDE Arduino pour des projets conséquents et en phase de maturation. Pour faire connaissance avec la bête après cette brève introduction, une technique intéressante consiste à tenter de porter vos projets Arduino/ESP32/ESP8266 existant vers ESP32/ESP-IDF. Vous apprendrez ainsi à vous passer, petit à petit, des structures, fonctions et mécanismes propres à Arduino et donc à rendre génériques et portables vos développements ou, du moins, votre expérience du développement sur microcontrôleur. DB




*La dernière plateforme ESP32 à la mode est l'ESP32-CAM. Intégrant une caméra Omnivision de 2 Megapixels, une puce de SRAM externe et un emplacement pour carte microSD, le tout pour moins de 5 €, ce matériel est parfaitement pris en charge par l'ESP-IDL 4.0 et offre des opportunités techniques presque sans limites.*

# PROGRAMMATION AVEC LE 6502 : DÉCOUVERTE DE LA NES

David Odin, vieux barbu et rétro gameur à mi-temps

**Dans les articles précédents, nous avons étudié de près le langage d'assemblage du microprocesseur 6502. Et même si j'ai essayé d'étayer le tout avec beaucoup d'exemples, tout cela est resté très théorique. Aujourd'hui, nous allons vraiment passer à la pratique en réalisant des programmes graphiques pouvant s'exécuter sur une véritable console NES ou sur un émulateur.**

La NES (*Nintendo Entertainment System*) a été l'une des premières consoles grand public. Et c'est en tout cas la toute première proposée par Nintendo. Sa résolution graphique était de 256x240 pixels, avec assez peu de couleurs : un maximum de 25 à choisir parmi une palette fixe de 56. Ses capacités sonores étaient également très limitées.

Les jeux se présentaient sous la forme de cartouches assez massives que l'on insérait directement dans la console. Ces cartouches contenaient le code et les données du jeu dans des ROM directement accessibles par le processeur, ce qui fait que les temps de chargement étaient quasi nuls. Les joueurs pouvaient interagir généralement par le biais de deux joypads, mais également avec le célèbre pistolet optique orange de Duck Hunt, et plein d'autres interfaces ont vu le jour avec le temps.

# 1. L'ARCHITECTURE GÉNÉRALE DE LA NES

L'architecture de la NES est très simple pour une console permettant de faire tourner autant de jeux. Il n'y a en effet que très peu de composants actifs sur sa carte électronique.

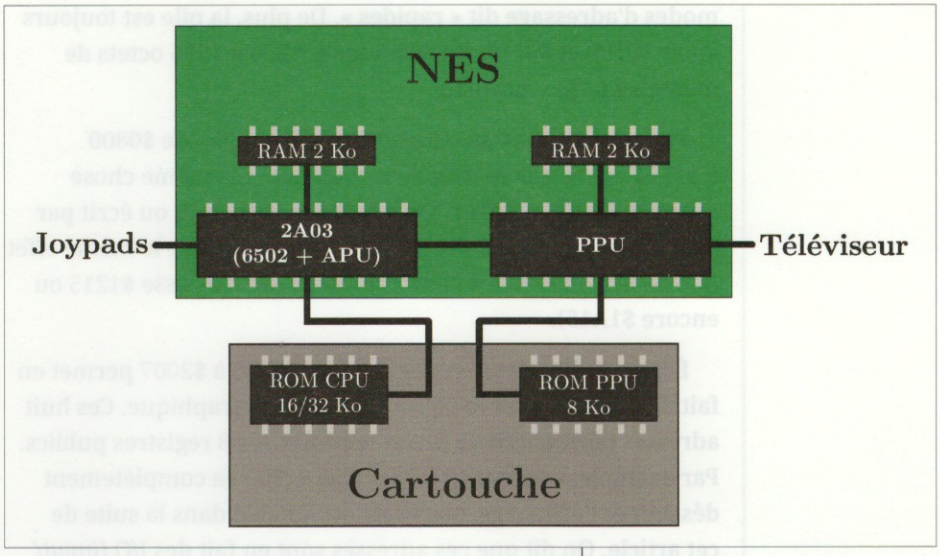



*Figure 1 : Architecture électronique de la NES (nombre de pattes non contractuel).*

Les deux principaux sont le PPU, le processeur graphique qui va nous intéresser aujourd'hui et le Ricoh 2A03, qui est une puce rassemblant un 6502 et un générateur de sons assez rustique nommé APU (voir **[2]**). On trouve également deux boîtiers de 2 Ko de RAM : un pour le 6502 et un pour le PPU, ainsi qu'une puce d'identification des cartouches (non représentée), une sorte de DRM de l'époque, évidemment obsolète. La figure 1 montre tout cela de manière schématique.

## 1.1 L'espace d'adressage de la NES

Le 6502 peut adresser 64 Ko de mémoire, mais la NES ne dispose que de 2 Ko de RAM (sur la console elle-même) et de 16 Ko ou 32 Ko de ROM (sur les cartouches de jeu). Cela laisse donc de la place pour autre chose, et c'est là que les concepteurs de cette console ont été très malins. Comme on peut le voir sur la figure 2, la RAM est placée au début de la mémoire. On pourra donc lire et écrire dans les adresses de $0000 à $07FF, soit les deux premiers Ko de la mémoire. Cela est une bonne idée, car les 256 premiers octets sont un peu particuliers (voir les épisodes précédents) puisqu'ils sont utilisés dans certains

modes d'adressage dit « rapides ». De plus, la pile est toujours entre $0100 et $01FF. Il reste donc 6 * 256 = 1536 octets de mémoire RAM « normale ».

Pour des raisons de simplicité électronique, de $0800 à $1FFF, on retrouve trois fois exactement la même chose qu'entre $0000 et $07FF. C'est à dire que si on lit ou écrit par exemple à l'adresse $0A15, cela aura exactement le même effet que de lire ou écrire à l'adresse $0215 (ou l'adresse $1215 ou encore $1A15).

Écrire ou lire dans les adresses de $2000 à $2007 permet en fait de discuter avec le PPU, le processeur graphique. Ces huit adresses permettent de lire et modifier ses 8 registres publics. Par exemple, écrire un 0 à l'adresse $2000 va complètement désactiver l'affichage, mais cela est détaillé dans la suite de cet article. On dit que ces adresses sont en fait des I/O (*input/output*).

Comme pour la RAM, la zone des registres du PPU est dupliquée (1024 fois !!!) de $2008 à $200F, de $2010 à $2017, etc. jusqu'à de $3FF8 à $3FFF. Là aussi, cela a été fait pour simplifier l'électronique de la NES.

Viennent ensuite 18 registres de l'APU, de $4000 à $4017, qui sont encore des I/O, permettant cette fois de gérer le son et... les joypads (oui, cela peut sembler curieux, mais ce genre d'association un peu contre nature est en fait assez courante : sur Atari ST, c'était le processeur sonore qui gérait quelle était la face active du lecteur de disquette).

Il y a une partie vide entre $4018 et $7FFF : écrire dans cette zone n'aura aucun effet et lire depuis cette zone pourra renvoyer n'importe quoi. Certaines cartouches de jeux utiliseront cette zone pour ajouter des possibilités à la NES.

| Début | Fin | Utilisation |
|---|---|---|
| $0000 | $07FF | RAM CPU : 2048 octets |
| $0800 | $1FFF | Miroir de la RAM |
| $2000 | $2007 | Registres PPU |
| $2008 | $3FFF | Miroir des registres PPU |
| $4000 | $4017 | APU et joypads |
| $4018 | $7FFF | Rien |
| $8000 | $BFFF | ROM optionnelle 16 Ko |
| $C000 | $FFFF | ROM obligatoire 16 Ko |

*Figure 2 : résumé du mapping mémoire de la NES.*

Et on retrouve finalement la ROM, c'est à dire le contenu du jeu (code et données, hormis les données purement graphiques) soit de $8000 à $FFFF (pour les cartouches de 32 Ko) soit de $C000 à $FFFF (pour les cartouches de 16 Ko). Donc si on n'a que 16 Ko, ils sont forcément à la fin.

## 1.2 Les cartouches

Les cartouches de jeux pour NES sont elles aussi très simples, même si elles sont devenues de plus en plus complexes au fil du temps. Dans leur configuration de base, elles ne sont composées que de deux puces de type ROM : une de 16 Ko ou 32 Ko contenant les données et le code qui sera exécuté par le 6502 et utilisé comme nous venons de le voir, et une de 8 Ko contenant les données graphiques destinées au PPU qui se chargera d'y accéder directement lui-même. Le PPU « verra » cette ROM aux adresses de $0000 à $1FFF dans sa mémoire à lui.

Beaucoup de jeux, dont l'immense majorité de ceux publiés lors des premières années d'exploitation de la NES, utilisèrent exactement ce genre de cartouches. Nous reviendrons sur les cartouches plus complexes dans un prochain article.

## 2. PRÉSENTATION DU PPU

Le *Pixel Processor Unit* (PPU) est un véritable processeur, avec ses registres, son horloge, ses instructions, etc., et qui accède à de la mémoire ROM et RAM pour produire les pixels que l'on verra à l'écran. Cependant, la ROM (de 8 Ko) et la RAM (de 2 Ko) ne peuvent contenir que des données graphiques, mais pas de code. Ainsi, le PPU n'est pas vraiment programmable par nous, on ne pourra donc utiliser ses registres que pour le paramétrer.

### 2.1 La composition des images de fond sur NES

Mais avant cela, il nous faut comprendre comment sont représentées les images sur la NES. L'écran de la NES fait 256 pixels de large sur 240 de haut, soit 61440 pixels. Or, la VRAM (la RAM du PPU) n'est que de 2 Ko, soit 16384 bits. On a donc 4 fois plus de pixels que de bits dans la RAM ! Il est donc impossible d'accéder à chaque pixel comme dans un *framebuffer* classique, même en considérant les 8 Ko de ROM (il y aurait tout juste la place pour une seule image statique monochrome).

Au lieu de cela, on utilise un système à base de tuiles. Chaque tuile est un bloc de 8x8 pixels, où chaque pixel occupe deux bits et peut donc prendre 4 valeurs (00, 01, 10, 11). Chaque tuile occupe donc 8x8x2 = 128 bits soit 16 octets. Et ce sont ces tuiles (et uniquement elles) que l'on va retrouver dans la

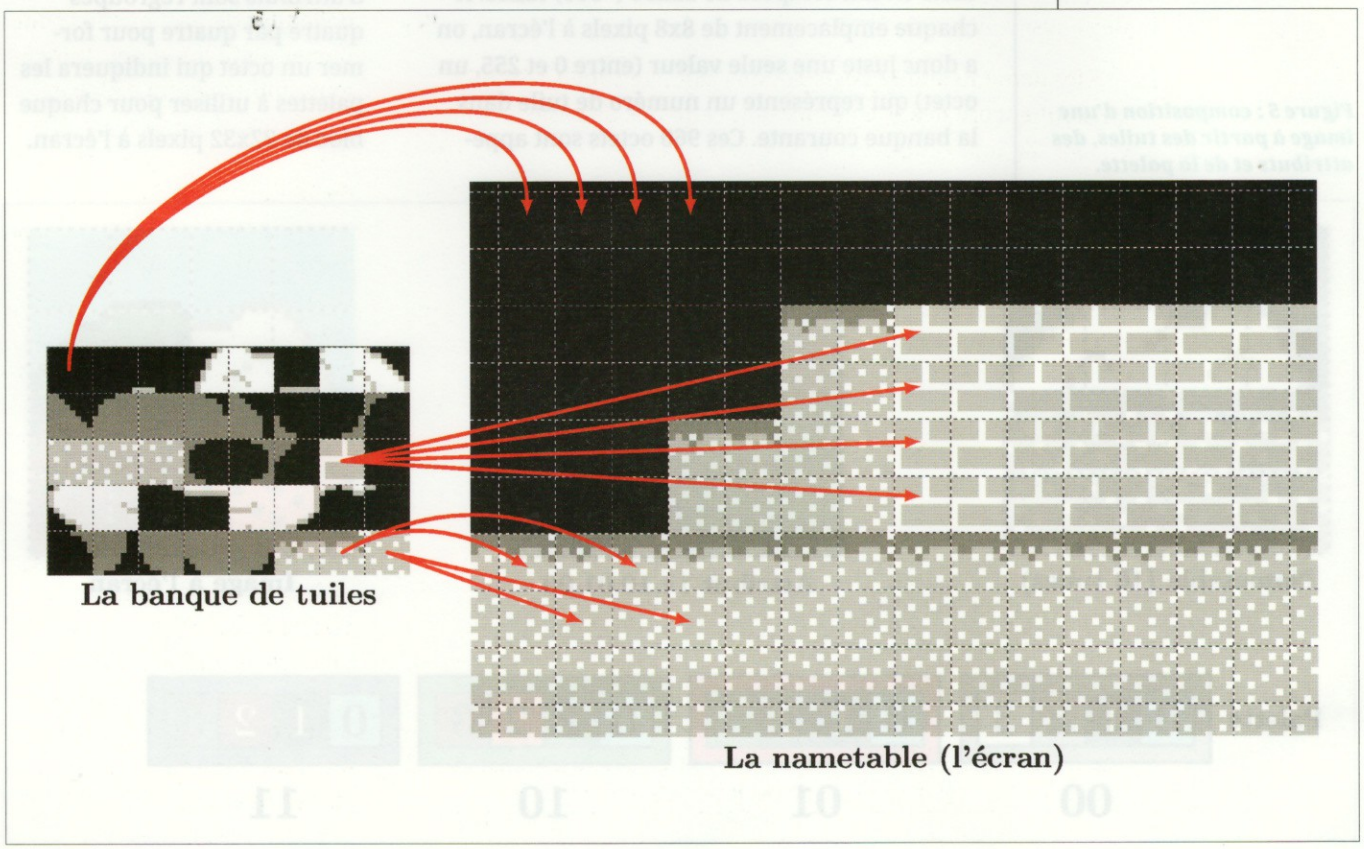



*Figure 3 : composition d'une image avec le système de tuiles.*

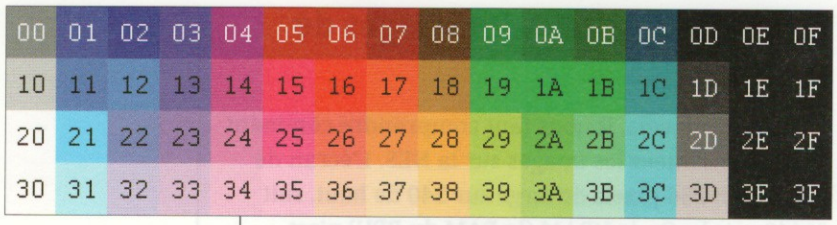



*Figure 4 : La palette des couleurs de la NES.*

mémoire ROM du PPU, sur la cartouche. Cette mémoire est de taille imposée et fait 8 Ko, on va donc avoir 8192 / 16 = 512 tuiles disponibles, réparties en 2 banques de 256 tuiles. À chaque instant, une seule banque de 256 tuiles est disponible pour les tuiles du fond de l'écran, les 256 autres tuiles sont utilisées pour les *sprites*. Dans cet article, on s'intéressera uniquement au dessin du fond d'écran, l'utilisation des *sprites* fera l'objet du prochain article.

Le fond de l'écran, qui fait 256x240 pixels, est donc en fait composé de 32x30 (=960) tuiles. À chaque emplacement de 8x8 pixels à l'écran, on a donc juste une seule valeur (entre 0 et 255, un octet) qui représente un numéro de tuile dans la banque courante. Ces 960 octets sont appelés *Nametable* (voir figure 3). Comme il y a presque 4 fois plus d'emplacements à l'écran que de tuiles disponibles, chaque tuile est généralement utilisée plusieurs fois.

Cependant, avec seulement 2 bits par pixels, on ne peut a priori représenter que 4 couleurs. Aussi, pour ajouter un peu de variété, Nintendo a utilisé la notion d'attributs. Chaque portion de 16x16 pixels (soit 2x2 tuiles) à l'écran se voit affecter un attribut de 2 bits. Ces deux bits peuvent représenter 4 valeurs qui sont des numéros de palette de 4 couleurs chacune. Ces bits d'attributs sont regroupés quatre par quatre pour former un octet qui indiquera les palettes à utiliser pour chaque bloc de 32x32 pixels à l'écran.

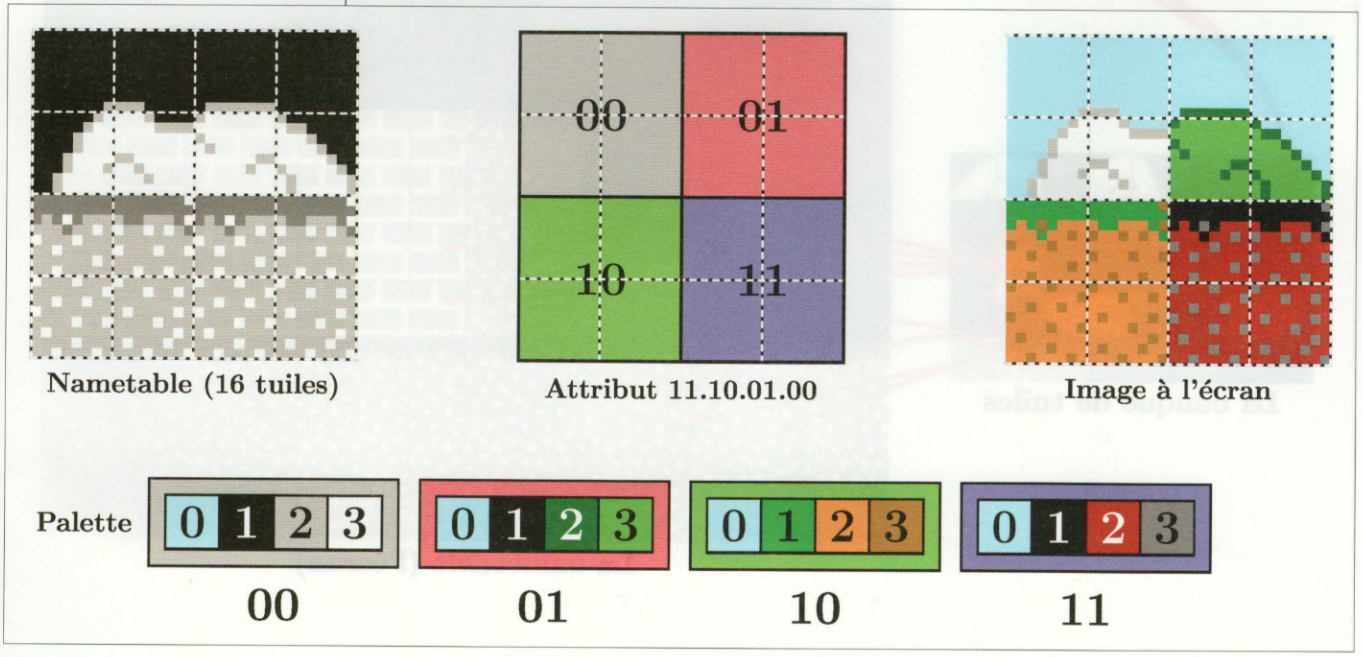



*Figure 5 : composition d'une image à partir des tuiles, des attributs et de la palette.*

Et comme 8*32 = 256, il y a en tout 8x8 = 64 octets d'attributs pour un écran. Si l'on ajoute les 960 octets désignant les tuiles, on a exactement 1024 octets (= 1 Ko) pour un écran. La RAM du PPU étant de 2 Ko, on peut donc y stocker exactement deux écrans (dont un seul est visible à la fois, évidemment).

Les palettes sont à leur tour composées de numéros de couleurs entre 0 et 64, mais seulement 56 sont différentes (voir figure 4). À noter une contrainte supplémentaire : dans toutes les palettes, la première couleur doit être la même ! On a donc une couleur commune plus quatre fois trois couleurs, soit 13 couleurs pour tout le fond d'écran. Et pas plus de 4 couleurs différentes par groupe de 16 pixels. Il existe aussi une seconde palette de 16 couleurs, avec les mêmes restrictions, utilisée pour dessiner les *sprites*. Ces deux palettes de couleurs sont stockées directement dans la RAM interne du PPU et pas dans une puce à part.

Tout cela peut paraître complexe, et ça l'est ! La figure 5 devrait cependant vous aider à y voir plus clair. On voit un bloc de 32x32 pixels, soit 4x4 tuiles. Sur la gauche, en noir et blanc, chaque tuile est représentée par un nombre entre 0 et 255, un index dans la banque des tuiles. Au milieu, un attribut avec ici 4 valeurs différentes pour chaque groupe de 16x16 pixels. Et sur la droite, on voit le résultat à l'écran, avec le coin supérieur gauche dessiné en utilisant la première sous-palette de 4 couleurs (la 00), le coin supérieur droit avec la seconde sous-palette (la 01), etc.

## 2.2 La mémoire du PPU

Comme indiqué précédemment, le PPU est le cœur de la partie graphique de la NES. Il consiste en un système indépendant, et il a donc son propre mappage mémoire.

Il est important de noter que la mémoire vue par le PPU est totalement distincte de celle vue par le CPU. Elle est stockée dans des puces différentes, gérée différemment et une même valeur d'adresse pour l'une ou pour l'autre représentera des données totalement différentes.

La figure 6 montre comment le PPU voit le monde qui l'entoure. La définition des tuiles est en ROM, donc figée une fois pour toutes pour un jeu donné. On trouve deux banques de 256 tuiles chacune vues par le PPU entre $0000 et $1FFF. Puis quatre « écrans » de 1024 octets chacun : 30*32=960 numéros de tuiles et 64 octets d'attributs sont présents dans la RAM du PPU aux adresses $2000, $2400, $2800 et $2C00. Enfin, les deux palettes courantes de 16 couleurs chacune sont stockées aux adresses $3F00 à $3F1F.

| Début | Fin | Type | Utilisation |
|---|---|---|---|
| $0000 | $0FFF | ROM (Cartouche | Première banque de 256 tuiles |
| $1000 | $1FFF | ROM (Cartouche | Deuxième banque de 256 tuiles |
| $2000 | $23FF | RAM (NES) | Premier écran (N0) |
| $2400 | $27FF | RAM (NES) | Deuxième écran (N1) |
| $2800 | $2BFF | RAM (NES) | Troisième écran (N2) |
| $2C00 | $2FFF | RAM (NES) | Quatrième écran (N3) |
| $3F00 | $3F1F | RAM interne PPU | Palette de couleurs |

*Figure 6 : l'espace d'adressage du PPU.*

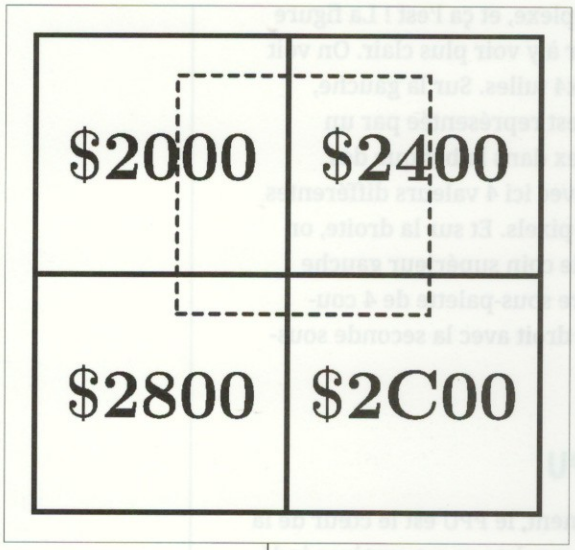



*Figure 7 : l'agencement des écrans.*

Les quatre écrans stockés dans la RAM sont agencés comme indiqué sur la figure 7. En temps normal, c'est le premier écran ($2000) qui est visible. Mais on verra à la fin de cet article qu'il est possible de demander au PPU de se décaler dans cet agencement et de montrer une partie de chaque écran, comme par exemple la partie pointillée sur la figure 7.

Les lecteurs attentifs auront remarqué que la RAM externe du PPU est de 2 Ko alors qu'elle est censée contenir quatre écrans de 1 Ko chacun. Ça dépasse. En fait, une cartouche de jeu peut choisir entre deux réglages possibles avec à chaque fois seulement deux écrans différents :

- soit on utilise $2000 et $2400, pour les jeux à scrolling horizontal, et dans ce cas $2800 est une copie de $2000 et $2C00 est une copie de $2400 ;
- soit on utilise $2000 et $2800, pour les jeux à scrolling vertical, et dans ce cas $2400 est une copie de $2000 et $2C00 est une copie de $2800.

Cela peut sembler tordu, mais cela s'avère assez pratique à l'usage et cela a permis à certaines cartouches de jeux de proposer 2 Ko de RAM PPU supplémentaires au prix de quelques magouilles électroniques. Cela a été utilisé notamment par le jeu Gauntlet afin de proposer un scrolling multidirectionnel particulièrement impressionnant à l'époque.

## 2.3 Transfert de données

Les données qu'on peut envoyer au PPU sont soit des indices de tuiles pour un endroit donné (de 8x8 pixels) de l'écran (*nametable*), soit un attribut indiquant les 4 sous-palettes de couleurs utilisées pour un bloc de 32x32 pixels de l'écran, soit une couleur de la palette.

Mais on ne peut pas accéder à la RAM du PPU directement

| Nom | Adresse | Rôle |
|---|---|---|
| PPUCTRL | $2000 | Contrôl général |
| PPUMASK | $2001 | Réglage de l'affichage |
| PPUSTATUS | $2002 | État du PPU (lecture seule) |
| PPUOAMADDR | $2003 | Registre d'adresse pour les sprites |
| PPUOAMDATA | $2004 | Registre de données pour les sprites |
| PPUSCROLL | $2005 | Gestion du décalage de l'écran |
| PPUADDR | $2006 | Registre d'adresse |
| PPUDATA | $2007 | Registre de données |

*Figure 8 : les registres du PPU.*

depuis le CPU, il va falloir passer par le biais des registres du PPU. La figure 8 montre comment les registres du PPU sont vus par le CPU (à quelle adresse) et à quoi ils servent.

Mais les registres du PPU sont un peu particuliers. On ne pourra pas les additionner, les comparer, les incrémenter, etc. Au contraire, chacun de ces registres a un rôle très précis et doit être utilisé de la bonne façon. Par exemple, on écrira dans le registre **PPUDATA** (à l'adresse $2007) pour transmettre une donnée au PPU pour qu'il la range dans sa RAM.

Et pour savoir où il doit ranger la donnée qu'on lui transmet ainsi, le PPU consultera son registre **PPUADDR**. Ce registre est sur 16 bits alors que les écritures du 6502 ne peuvent être que de 8 bits à la fois. Aussi, On pourra modifier **PPUADDR** en écrivant deux fois de suite à l'adresse $2006, en commençant par le poids fort.

Par exemple, l'extrait de code suivant permet de placer la donnée 42 à l'adresse $2000 de la RAM du PPU (et donc de changer la tuile du tout premier bloc de 8x8 pixels de l'écran).

```
LDA #$20  ; poids fort de #$2000
STA $2006 ; PPUADDR
LDA #$00  ; poids faible de #$2000
STA $2006 ; PPUADDR
LDA #42   ; notre donnée
STA $2007 ; PPUDATA
```

C'est un peu long pour écrire un seul octet ! Heureusement, le PPU regorge d'astuces et il incrémente tout seul **PPUADDR** à chaque écriture dans **PPUDATA**, ce qui permet d'accélérer grandement les écritures successives. Par exemple, pour écrire 42 à l'adresse $2000 et 82 à l'adresse $2001, on pourra utiliser le bout de code suivant :

```
LDA #$20  ; poids fort de #$2000
STA $2006 ; PPUADDR
LDA #$00  ; poids faible de #$2000
STA $2006 ; PPUADDR
LDA #42   ; notre première donnée
STA $2007 ; PPUDATA
LDA #82   ; notre seconde donnée
STA $2007 ; PPUDATA
```

Ceci dit, même ainsi, écrire dans la RAM du PPU s'avère particulièrement lent, ce qui explique que les jeux sur NES ne modifient pas beaucoup les décors d'une *frame* à l'autre.

## 2.4 Contrôler le fonctionnement du PPU

Le fonctionnement général du PPU est contrôlé via deux autres de ses registres. Premièrement, **PPUCTRL** auquel on accède en écrivant à l'adresse $2000. Chaque bit de ce registre a un sens particulier, permettant d'activer ou non l'interruption verticale, de choisir la taille des *sprites*, de définir dans lequel des quatre écrans l'affichage commence ou encore de choisir si les tuiles pour le fond doivent être prises dans la première ou deuxième banque. Tout

cela est bien complexe et on détaillera chaque fonctionnalité au moment où l'on en aura besoin.

Pour l'instant, on gardera simplement à l'esprit qu'en écrivant 0 dans ce registre, on suspend une bonne partie de l'activité du PPU, alors que la valeur 144 (%10010000 en binaire) le remettra en ordre de marche.

Deuxièmement, le PPU est contrôlé par le registre **PPUMASK** accessible à l'adresse $2001. Là encore, chaque bit a un sens permettant d'indiquer si on veut activer l'affichage du fond et/ou des *sprites*, si la toute première colonne doit être affichée ou non et si on veut rendre les couleurs plus vives ou au contraire passer toute l'image en niveau de gris.

Ici aussi, écrire 0 dans ce registre désactivera le fonctionnement du PPU (il n'affichera plus rien à l'écran) et une valeur qui réactive le tout peut être 62 (%00011110 en binaire).

Ainsi, avant d'écrire massivement (et cacher momentanément les changements), on désactivera le PPU avec la séquence suivante :

```
LDA #0
STA $2000 ; PPUCTRL
STA $2001 ; PPUMASK
```

Et on réactivera le tout avec l'extrait suivant :

```
LDA #%10010000 ; réactivation NMI
STA $2000 ; PPUCTRL
LDA #%00011110 ; affichage fond d'écran & sprites
STA $2001 ; PPUMASK
```

Notez que l'on n'aura pas besoin de faire ça très souvent normalement. Ce sera surtout utile au moment de l'initialisation ou lorsque l'on voudra redessiner tout un écran, par exemple.

Le dernier registre du PPU qui va nous être indispensable pour notre premier programme est **PPUSTATUS** ($2002). Ce registre n'est accessible qu'en lecture, et seul son bit 7 nous intéresse pour l'instant. L'*opcode* **BIT** qui permet justement de tester ce bit semble vraiment être fait pour surveiller ce registre.

Le PPU met à 1 ce bit lorsqu'il a fini de dessiner un écran et le remet à 0 lorsqu'il recommence à dessiner la première ligne de l'écran suivant. Cela nous sera utile pour synchroniser nos opérations avec le dessin. En effet, il n'est possible d'écrire dans un des registres du PPU que lorsqu'il n'est pas occupé à dessiner l'écran. Le meilleur moment pour cela est donc le moment où il a fini de dessiner la dernière ligne de l'écran jusqu'à l'instant où il devra commencer à dessiner la première ligne de l'écran lors de la *frame* suivante. On appelle ce moment la VBL pour *Vertical BLank* (à ne pas confondre avec la notion d'intervalle VBLANK d'un signal vidéo, qui est liée, mais purement électronique).

# 3. NOTRE PREMIER PROGRAMME POUR NES

Avec tout ceci, vous devez être impatient de mettre tout ça en œuvre pour créer votre premier programme qui affiche quelque chose sur une NES (ou un émulateur). Mais avant ça, il nous reste quelques petites choses à savoir.

## 3.1 Les interruptions

Lors du premier article de cette série, nous avons passé en revue tous les *opcodes* du 6502. Parmi ceux-ci, il y avait **RTI** qui permet de terminer une routine de traitement d'interruption,

ce qui laisse supposer que ce processeur sait gérer les interruptions. En fait, cela est assez rudimentaire avec le 6502, mais suffisant pour nos besoins. Il y a 3 seulement trois types d'interruptions disponibles :

- **RESET**, qui arrive quand on (re)démarre la NES ou que l'on presse le bouton du même nom. La routine associée initialise le programme et ne se termine jamais vraiment.
- **NMI** (*Non Maskable Interrupt*, interruption non masquable). Sur la NES, cette interruption est générée par le PPU lorsqu'il est en mode normal (non désactivé) à chaque fois qu'il a fini de dessiner un écran. La routine associée à cette interruption sera donc le bon endroit pour écrire dans la RAM du PPU pour modifier l'affichage du prochain écran, par exemple. On appelle souvent cette routine VBL (Vertical BLank).
- **IRQ** (*Interrupt ReQuest*, requête d'interruption). Cette interruption n'est pas utilisée par défaut sur la NES elle-même. Certaines cartouches de jeux assez évoluées l'utilisent pour certains effets, en l'associant à une routine qui sera appelée à chaque fin de ligne (tout comme la VBL est appelée à chaque fin d'écran).

J'indique que l'on peut associer des routines (des morceaux de code assembleur) à ces interruptions. Mais comment faire cela ? Sur le 6502, c'est assez simple, il suffit de placer leurs adresses à la fin de la mémoire (et donc en ROM pour nous). À l'adresse $FFFA (et $FFFB, une adresse est sur 16 bits), on placera l'adresse de l'interruption associée à la NMI. En $FFFC, on mettra l'adresse de la routine d'initialisation (RESET). En $FFFE, on mettra l'adresse de la routine IRQ, donc rien, puisqu'on ne l'utilise pas !

Ces 6 octets (3 adresses 16 bits) à la fin de l'espace mémoire du 6502 sont appelés vecteurs d'interruption.

## 3.2 L'assembleur ASM6

Si dans les épisodes précédents un assembleur/émulateur en ligne était suffisant pour tester les petits exemples proposés, nous allons avoir besoin d'un assembleur un peu plus costaud aujourd'hui.

Il y a plein d'assembleurs pour le 6502, chacun avec ses avantages et inconvénients. Vous en trouverez une bonne liste sur le wiki de NesDev ici **[3]**.

J'ai choisi d'utiliser ASM6 dans mes exemples pour sa simplicité d'utilisation et l'ensemble des possibilités qu'il offre pour organiser le code. Vous le trouverez sur le lien **[4]**. L'archive contient un .exe pour Windows, et le fichier source C permettant de le compiler pour les autres plateformes. Il se compile avec un simple :

```
$ gcc asm6.c -o asm6
```

Un assembleur permet de convertir notre code (un simple fichier texte qu'on écrira avec ce qu'on veut, de Notepad à Vi pour les plus aventureux) en un fichier exécutable sur un émulateur, par exemple. On pourrait par la suite transformer ce fichier pour le placer dans de vraies ROM, sur une vraie cartouche de jeu pour l'utiliser dans une vraie NES, branchée à une vraie télé cathodique, avec de vrais joypads manipulés par un vrai joueur, mais je m'égare.

Le fichier source contenant notre code sera donc la suite des *opcodes* qui forment notre programme, mais pas uniquement. Les assembleurs acceptent généralement d'autres mots clefs qui ne seront pas retranscrits

directement en code machine, mais qui permettent d'écrire beaucoup plus simplement. On appelle ces mots clefs « directives ». ASM6 vient avec sa documentation détaillée de chaque directive qu'il propose, ainsi que de ses différents paramètres.

Voici quelques exemples de ces directives :

- **EQU** permet de définir un symbole, pour donner un nom à un nombre. Par exemple, `PPUCTRL EQU $2000` nous permettra d'utiliser **PPUCTRL** à la place de $2000 pour parler du registre de contrôle du PPU, ce qui est plus pratique et plus lisible.
- **DC** permet d'inclure des données constantes directement, en donnant leur valeur :

```
donnees:  ; un label pour faire référence aux données
  DC.B  124   ; l'adresse "donnees" contiendra 124
  DC.W  $1234 ; l'adresse "donnees+1" contiendra $34
              ; et l'adresse "donnees+2" contiendra $12
  DC.B  'A'   ; l'adresse "donnees+3" contiendra 65 (le code ASCII de 'A')
```

C'est donc une sorte d'*opcode* générique, vous allez voir que l'on s'en sert beaucoup. **DC.B** (qui définit une donnée sur un octet) et **DC.W** (qui définit une donnée sur deux octets) sont généralement utilisés dans la partie code (la ROM), pas très loin du code qui se sert de ces données.

- **DS** fonctionne un peu de la même manière, mais en donnant juste la taille des données que l'on veut, ce qui est pratique pour réserver de la place pour une variable en RAM (Je donne un exemple tout de suite après).
- **ENUM** permet justement de définir une zone en RAM ou l'on placera nos données.

```
  ENUM $0300 ; les données suivantes seront accessibles à partir de l'adresse $0300
compteur: DS.B 1
position: DS.B 1
  ENDE ; fin de la zone
```

- **BASE** va nous permettre de faire un peu la même chose, mais pour le code. Vous vous souvenez que la ROM du CPU, qui contient le code doit commencer en $8000 ou en $C000 ? Hé bien, on commencera la partie code de notre programme avec un `BASE $C000`, ce qui permettra à l'assembleur de générer les bonnes adresses de sauts par exemple.
- Enfin, **ORG** nous permettra de placer nos vecteurs d'interruption à la fin de la ROM sans avoir à remplir à la main la zone entre la fin de notre code et $FFFA, avec un simple `ORG $FFFA`. Notons qu'à la différence de **BASE**, **ORG** remplit la mémoire avant lui (pour qu'il n'y ait pas de trou entre la fin de notre code et $FFFA dans notre cas).

L'assembleur ASM6 offre plein d'autres possibilités, comme la gestion des répétitions d'un bout de code ou encore les macros, mais cela dépasse le cadre de cet article. On utilisera ces possibilités avancées dans la suite de cette série.


*Figure 9 : hello world sur NES !*

## 3.3 La structure générale

Il ne nous reste plus qu'à écrire le code de notre programme.

On va réaliser un exemple simple qui affiche juste « Hello, world », avant de partir dans une boucle infinie nous laissant admirer le résultat (figure 9).

Vous trouverez bien évidemment l'intégralité du code sur le GitHub du magazine **[5]**, mais je vais en décrire les parties importantes.

Tout d'abord, comme notre code est aussi destiné aux émulateurs, il faut décrire le contenu et le type de cartouche de jeu qu'ils doivent émuler. Pour cela, avant le début de notre code, on devra placer un bloc de 16 octets servant d'en-tête. Ce bloc doit avoir la forme suivante :

```
DC.B "NES", $1a ; l'en-tête doit toujours commencer ainsi
DC.B 1          ; Le nombre de boîtiers de 16 Ko de ROM CPU (1 ou 2)
DC.B 1          ; Le nombre de boîtiers de 8 Ko de ROM PPU
DC.B 0          ; Le type de cartouche, ici on veut le plus simple
DS.B 9, $00     ; puis juste 9 zéros pour faire 16 en tout
```

Il existe plus d'une centaine de types de cartouches, mais la grande majorité des jeux utilise les formats de cartouches les plus simples. L'octet contenant ce type contient aussi l'agencement des écrans du PPU (horizontal ou vertical). Pour notre programme exemple, l'orientation importe peu, et on laisse donc cela à 0.

Comme notre programme consiste en un affichage puis une attente, la plus grande partie du code consistera à correctement initialiser le tout. Et la routine correspondante est évidemment RESET. Elle commence ainsi :

```
  BASE $C000
RESET:
  LDA #0      ; Remise à zéro
  STA PPUCTRL ;   du Contrôle du PPU
  STA PPUMASK ;   du Mask du PPU
  ...
```

On commence en effet toujours par désactiver le PPU, afin que les initialisations n'affichent pas des choses bizarres tant que l'on n'a pas fini. Et au moment du RESET, le PPU se réveille un peu en même temps que le CPU, les yeux encore tout collés et il lui faut un petit moment pour qu'il

retrouve ses esprits. Heureusement, dès le début, son registre **PPUSTATUS** est disponible et peut nous servir à attendre que le reste soit opérationnel (soit le temps d'une VBL, au moins).

```
;; On attend un peu que le PPU se réveille
  BIT PPUSTATUS ; La première lecture passe l'état à zéro
- BIT PPUSTATUS ; On boucle tant que le
  BPL -         ;    PPU n'est pas prêt (VBL suivante)
```

Note : L'assembleur ASM6 propose des labels locaux sous forme de « - » et de « + ». Le « **BPL -** » signifie ainsi « saut conditionnel au - le plus près qui est au-dessus ». Ça évite de devoir trouver des noms de label pour chaque boucle. Très pratique à l'usage.

Puis on peut passer à l'initialisation proprement dite, en commençant par mettre à 0 toute la RAM disponible pour le CPU avec le code suivant. Notez l'utilisation du mode d'adressage indexé par **X**, pour considérer les 2 Ko de RAM comme 8 tableaux de 256 octets.

```
;; Remise à zéro de toute la RAM
  LDA #0        ; Place 0 dans A
  TAX           ;   et dans X
- STA $0000,X   ; Efface l'adresse   0 + X
  STA $0100,X   ; Efface l'adresse 256 + X
  STA $0200,X   ; Efface l'adresse 512 + X
  STA $0300,X   ;   etc.
  STA $0400,X
  STA $0500,X
  STA $0600,X
  STA $0700,X
  INX           ; Incrémente X
  BNE -         ; et boucle tant que X ne revient pas à 0
  ...
```

Après avoir nettoyé la RAM CPU, on remplit la RAM du PPU avec nos données, comme on l'a vu précédemment :

```
  ;; Chargement du fond
  LDA PPUSTATUS  ; Resynchronisation
  LDA #$20       ;   On copie maintenant
  STA PPUADDR    ;     vers l'adresse $2000
  LDA #$00
  STA PPUADDR

  LDX #0
- LDA nametable,X    ; On charge les 256 premiers octets
  STA PPUDATA        ;   depuis notre nametable
  INX
  BNE -
  ...
```

Évidemment, d'autres boucles de copies ou de mise à zéro suivent, mais je vous laisse découvrir ça en détail par vous-même.

Et la routine RESET finit par réactiver le PPU avant de sauter à une boucle sans fin :

```
;; Avant de rebrancher le PPU
  LDA #%10010000 ; Réactivation, avec les tuiles de fond en $1000
  STA PPUCTRL
  LDA #%00011110 ; On veut montrer le fond au moins
  STA PPUMASK
  ...
```

La routine VBL (NMI) ne fait pas grand-chose non plus. Elle se contente de mettre à **1** la variable **vbl_flag** et d'incrémenter **vbl_cnt**, ce qui n'a pas une utilité réelle pour l'instant. Cependant, on notera l'obligation de sauvegarder la valeur du registre **A** (sur la pile ici), au cas où l'interruption vienne interrompre (!) le code de la boucle principale à un moment où le contenu de **A** est important. On notera également que cette routine d'interruption doit absolument se terminer par l'opcode **RTI** (ReTurn from Interrupt).

```
;; La routine VBL
VBL:
  PHA           ; On sauvegarde A sur la pile
  LDA #1        ; On indique à la partie principale
  STA vbl_flag  ;   que la VBL a eu lieu
  INC vbl_cnt   ; Et on incrémente le compteur de VBL
  PLA           ; Récupération de A
  RTI           ; Fin de routine d'interruption
```

À la fin de notre programme, on n'oubliera pas les trois vecteurs d'interruption :

```
;; Les vecteurs du 6502
  ORG $FFFA
  DC.W VBL     ; Appelé à chaque début d'image
  DC.W RESET   ; Appelé au lancement
  DC.W $00     ; Inutilisé
```

Puis on inclura le fichier de 8 Ko contenant les tuiles pour la ROM du PPU. C'est dans ces tuiles que l'on retrouvera les caractères qui sont affichés, on pourrait aussi mettre des graphismes plus rigolos :

```
  INCBIN "gfx.chr"   ; la ROM du PPU
```

Il existe plein de programmes pour créer ce fichier, décrits dans **[3]**.

Vous pouvez récupérer le fichier **hello-world.nes** déjà tout compilé sur le GitHub du magazine **[5]** et l'utiliser dans un émulateur comme FCEUX **[6]** ou Nestopia **[7]**, par exemple. Mais vous pouvez aussi récupérer le code source et l'assembler vous-même avec la ligne de commande suivante :

```
$ ./asm6 hello-world.asm hello-world.nes
```

Cette dernière façon de faire vous permettra de modifier le code et les données afin de mieux comprendre le fonctionnement et l'intérêt de chaque ligne, ce à quoi je vous encourage vivement ! Amusez-vous notamment à modifier la *nametable*, la palette et les attributs pour changer ce qui est affiché.

## 4. UN PROGRAMME PLUS INTÉRESSANT

Cela fait beaucoup de lignes de code pour pas grand-chose, me direz-vous. Certes, mais on peut rendre ce programme plus intéressant en ajoutant assez peu de lignes. Pour cela, nous allons utiliser un registre supplémentaire du PPU : **PPUSCROLL**, auquel on peut accéder depuis le CPU en écrivant à l'adresse $2005.

Ce registre est un peu particulier puisqu'il nécessite deux écritures consécutives pour qu'il soit vraiment modifié, un peu comme dans le cas de **PPUADDR**. La première écriture modifie le décalage de l'écran en X et la seconde, le décalage en Y. Dans mon exemple, je fais juste un *scrolling* horizontal, un décalage en X est donc suffisant, mais il faut tout de même écrire le décalage (de 0 pixels) en Y pour que tout soit bien pris en compte. Sans surprise, cela ressemble à ça (à mettre dans la VBL) :

```
  LDA offset     ; On charge notre décalage
  STA PPUSCROLL ;   qui devient la valeur de scrolling en X
  LDA #0         ; Et on met 0 pour la valeur
  STA PPUSCROLL ;   de scrolling en Y
```

La variable **offset** est mise à jour dans la boucle principale de cette manière :

```
;; Mise à jour de l'offset
  LDA direction  ; Si la direction vaut 1
  BNE a_gauche   ; C'est qu'on décale vers la gauche
  CLC            ; Sinon,
  LDA offset     ; On effectue une addition
  ADC #3         ;   de 3 pixels
  STA offset     ;   sur l'offset
  CMP #255       ; Et si on arrive à 255
  BNE mainloop
  INC direction  ; ... on change de direction
  JMP mainloop
a_gauche         ; Si on va à gauche,
  SEC            ; Le processus est le même
  LDA offset     ;   dans l'autre sens :
  SBC #3         ;   on soustrait 3
```

```
STA offset
BNE mainloop    ; Et si on est à 0
DEC direction   ; On rechange de direction
```

On notera l'utilisation d'une seconde variable, `direction` qui indique si on est en train d'aller vers la gauche ou vers la droite.

Je vous laisse le soin d'examiner le reste du programme source `scrolling.asm` afin de voir les (rares) changements qu'il peut y avoir avec le `hello-world.asm` vue précédemment. On peut voir le résultat de ce programme sur la figure 10. Encore une fois, je vous encourage à modifier ce programme pour changer l'affichage, la vitesse du *scrolling* ou bien d'autres choses encore.


*Figure 10 : Un logo qui défile sur NES !*

## 5. LA PROCHAINE FOIS

La prochaine fois, nous continuerons notre exploration des possibilités du PPU. Nous en profiterons pour ajouter des *sprites* (des éléments qui peuvent se déplacer sur le décor) et en ajoutant la gestion des joypads, on pourra réaliser notre premier mini jeu sur NES ! DO

## RÉFÉRENCES

[1] https://nesdev.com/

[2] https://wiki.nesdev.com/w/index.php/2A03

[3] https://wiki.nesdev.com/w/index.php/Tools

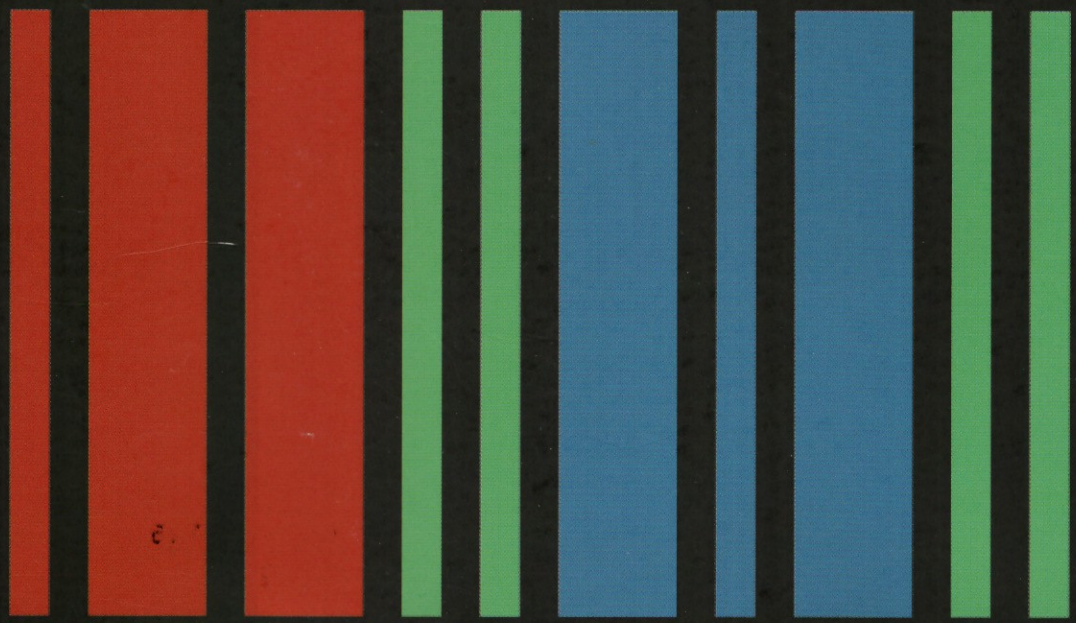
[4] http://3dscapture.com/NES/asm6.zip

[5] https://github.com/Hackable-magazine/Hackable34

[6] http://www.fceux.com/web/home.html

[7] http://nestopia.sourceforge.net/

# WIKI_



# _DATA

## LA BASE DE DONNÉES LIBRE

Participez à Wikidata sur www.wikidata.org
ou lors de nos ateliers tous les 3e vendredi du
mois au 40 rue de Cléry 75002 Paris
Toutes les infos sur wikimedia.fr

WIKIMEDIA
FRANCE